PetaPerl documentation

Experimental preview — draft documentation ***0.6.25***

PetaMem, s.r.o.

Oct 25, 2026

USER GUIDE

1 Introduction **3**
- 1.1 Goals . . . 3
- 1.2 Architecture Overview . . . 4
- 1.3 Quick Start . . . 5
- 1.4 Compatibility . . . 5
- 1.5 Platform Support . . . 5

2 Getting Started **7**
- 2.1 Installation . . . 7
- 2.2 First Script . . . 7
- 2.3 Command Line Options . . . 7
- 2.4 Notes . . . 7

3 CLI options **9**
- 3.1 Usage . . . 9
- 3.2 Perl-compatible options . . . 10
- 3.3 pperl-specific options . . . 10
- 3.4 Runtime options . . . 11
- 3.5 Regex JIT . . . 11
- 3.6 Module installation . . . 12
- 3.7 Daemon options . . . 12
- 3.8 Examples . . . 13
- 3.9 Module search paths: `-I`, `PERL5LIB`, `@INC` . . . 13
- 3.10 Environment variables . . . 14

4 How-To **15**
- 4.1 Guides . . . 15
- 4.2 Tutorials . . . 16
  - 4.2.1 PPerl Architecture . . . 16
    - JIT Compilation . . . 17
    - Auto-FFI (Peta::FFI) . . . 19
    - Native Modules . . . 29
    - Differences from Perl 5 . . . 32
  - 4.2.2 Concurrent Execution . . . 34
    - Parallel Execution . . . 34
    - Coroutines and event loops . . . 37
    - Threads . . . 41
  - 4.2.3 Regular expressions . . . 55
    - Who this is for . . . 56

How this guide is organised . . . 56
A first round-trip . . . 147
Conventions in the examples . . . 147
Related reference pages . . . 147
4.2.4 Performance . . . 148
Who this is for . . . 148
The one rule that survives everything . . . 148
How this guide is organised . . . 148
What this guide does not cover . . . 167
A note on numbers . . . 167
4.2.5 Object-oriented programming . . . 168
Which chapter you want . . . 168
When to use OO at all . . . 168
A note on scope . . . 169
4.2.6 Debugging Perl programs . . . 195
Which chapter do I want? . . . 196
The three layers of this guide . . . 196
Known gaps . . . 196
See also . . . 197
4.2.7 The Definitive Perl One-Liner Guide . . . 233
Which chapter do I want? . . . 233
The three layers of this guide . . . 233
Two examples before you start . . . 234
Read in this order on a first pass . . . 234
Bibliography . . . 234
4.2.8 Tying variables . . . 269
Which chapter you want . . . 270
How each chapter is built . . . 270
What ties cost . . . 270
Differences from upstream . . . 271
4.2.9 Coming from another language . . . 283
Which guide you want . . . 283
The one concept to read first . . . 284
A note on the runtime . . . 284
4.2.10 The Definitive Guide to 2D Graphics with Perl . . . 377
Which chapter do I want? . . . 378
The three arcs of this guide . . . 378
A note on the runtime . . . 378
Known gaps . . . 378
See also . . . 378
4.2.11 The Definitive Guide to 3D Graphics with Perl . . . 433
What this guide runs on . . . 434
Which chapter do I want? . . . 434
A note on the second dimension . . . 434
4.2.12 pack and unpack - a tutorial . . . 532
When you need pack and unpack . . . 532
How this tutorial is organised . . . 532
A first round-trip . . . 555
Conventions in the examples . . . 555
4.2.13 I/O streams - a tutorial . . . 555
What this chapter does not cover . . . 556

Where to go next . . . 556
Reference cross-links . . . 568
4.2.14 References - a tutorial . . . 568
Who this is for . . . 568
How this tutorial is organised . . . 568
Related reference pages . . . 587
4.2.15 Unicode - a tutorial . . . 587
The one idea . . . 587
The pipeline . . . 588
Chapters . . . 588
Conventions . . . 601
4.2.16 Boolean Logic for Perl Programmers - a tutorial . . . 601
Who this is for . . . 601
Scope . . . 601
How this tutorial is organised . . . 601
What you will be able to do after reading . . . 619
4.2.17 Interprocess communication - a tutorial . . . 620
Choosing a channel . . . 620
How this tutorial is organised . . . 620
What this tutorial does not cover . . . 634
4.2.18 Installing CPAN modules - a tutorial . . . 634
The one idea . . . 634
What you will need . . . 635
Chapters . . . 635
4.2.19 ipps, the interactive mode - a tutorial . . . 645
Starting it . . . 645
The one idea . . . 646
Chapters . . . 646

5 Reference **659**
5.1 Perl language . . . 659
5.1.1 Special variables . . . 659
Choose a family . . . 660
Variable namespaces . . . 697
The `English` module . . . 698
Localising special variables . . . 698
See also . . . 698
5.1.2 Built-in Functions . . . 699
A . . . 699
B . . . 699
C . . . 699
D . . . 700
E . . . 700
F . . . 701
G . . . 701
H . . . 702
I . . . 702
J . . . 702
K . . . 702
L . . . 703
M . . . 703

N . . . 703
O . . . 704
P . . . 704
Q . . . 704
R . . . 704
S . . . 705
T . . . 707
U . . . 707
V . . . 708
W . . . 708
Y . . . 708
__ (double-underscored) . . . 708
5.1.3 Operators . . . 1447
Choose a family . . . 1448
Cross-cutting concepts . . . 1485
See also . . . 1485
5.1.4 Data types . . . 1486
Choose a topic . . . 1486
Cross-cutting concepts . . . 1522
See also . . . 1523
5.1.5 Subroutines . . . 1523
Choose a topic . . . 1523
Defining a sub vs. calling a sub . . . 1559
The classic vs. modern split . . . 1560
See also . . . 1560
5.1.6 Locale handling . . . 1561
What a locale is . . . 1561
The `use locale` pragma . . . 1561
Locale categories . . . 1562
Setting the locale . . . 1562
Locale and Unicode . . . 1563
Locale leaks and safety . . . 1563
Parallel execution and locale . . . 1563
Environment variables . . . 1563
Differences from upstream . . . 1564
See also . . . 1564
5.1.7 Diagnostic messages . . . 1564
Reading an entry . . . 1565
Enabling and silencing warnings . . . 1565
A note on portability . . . 1565
The diagnostics . . . 1566
5.1.8 Security and taint mode . . . 1579
Taint mode . . . 1579
What gets tainted . . . 1580
Laundering and detecting tainted data . . . 1581
Cleaning up `%ENV` and `$ENV{PATH}` . . . 1581
Taint mode and `@INC` . . . 1582
Set-id scripts and the shebang race . . . 1582
Algorithmic complexity attacks . . . 1583
Differences from upstream . . . 1583
See also . . . 1583

5.2 Modules . . . 1584
5.2.1 AE . . . 1590
SYNOPSIS . . . 1590
DESCRIPTION . . . 1591
AUTHOR . . . 1593
5.2.2 AnyDBM_File . . . 1593
SYNOPSIS . . . 1593
DESCRIPTION . . . 1593
SEE ALSO . . . 1594
5.2.3 AnyEvent . . . 1594
SYNOPSIS . . . 1594
INTRODUCTION/TUTORIAL . . . 1595
SUPPORT . . . 1595
WHY YOU SHOULD USE THIS MODULE (OR NOT) . . . 1595
DESCRIPTION . . . 1596
WATCHERS . . . 1597
SUPPORTED EVENT LOOPS/BACKENDS . . . 1609
GLOBAL VARIABLES AND FUNCTIONS . . . 1610
WHAT TO DO IN A MODULE . . . 1613
WHAT TO DO IN THE MAIN PROGRAM . . . 1613
OTHER MODULES . . . 1614
SIMPLIFIED AE API . . . 1615
ERROR AND EXCEPTION HANDLING . . . 1615
ENVIRONMENT VARIABLES . . . 1616
SUPPLYING YOUR OWN EVENT MODEL INTERFACE . . . 1620
EXAMPLE PROGRAM . . . 1620
REAL-WORLD EXAMPLE . . . 1621
BENCHMARKS . . . 1623
SIGNALS . . . 1629
RECOMMENDED/OPTIONAL MODULES . . . 1630
FORK . . . 1631
SECURITY CONSIDERATIONS . . . 1632
BUGS . . . 1632
SEE ALSO . . . 1632
AUTHOR . . . 1633
Modules . . . 1633
5.2.4 App . . . 1741
App::Cpan . . . 1741
App::Prove . . . 1747
5.2.5 Archive . . . 1758
Archive::Tar . . . 1758
5.2.6 Attribute . . . 1778
Attribute::Handlers . . . 1778
5.2.7 AutoLoader . . . 1790
SYNOPSIS . . . 1790
DESCRIPTION . . . 1791
CAVEATS . . . 1793
SEE ALSO . . . 1793
AUTHOR . . . 1793
COPYRIGHT AND LICENSE . . . 1794
5.2.8 AutoSplit . . . 1795

SYNOPSIS . . . 1795
DESCRIPTION . . . 1795
DIAGNOSTICS . . . 1796
AUTHOR . . . 1797
COPYRIGHT AND LICENSE . . . 1797
5.2.9 B . . . 1798
Differences from upstream . . . 1799
Modules . . . 1799
Functions . . . 1801
5.2.10 Benchmark . . . 1857
SYNOPSIS . . . 1858
DESCRIPTION . . . 1859
Benchmark Object . . . 1862
NOTES . . . 1863
EXAMPLES . . . 1863
INHERITANCE . . . 1864
CAVEATS . . . 1864
SEE ALSO . . . 1864
AUTHORS . . . 1865
MODIFICATION HISTORY . . . 1865
5.2.11 CPAN . . . 1865
SYNOPSIS . . . 1865
DESCRIPTION . . . 1866
CONFIGURATION . . . 1875
PROGRAMMER'S INTERFACE . . . 1889
PREREQUISITES . . . 1898
UTILITIES . . . 1899
SECURITY . . . 1901
EXPORT . . . 1901
ENVIRONMENT . . . 1901
POPULATE AN INSTALLATION WITH LOTS OF MODULES . . . 1901
WORKING WITH CPAN.pm BEHIND FIREWALLS . . . 1902
FAQ . . . 1903
COMPATIBILITY . . . 1908
SECURITY ADVICE . . . 1909
BUGS . . . 1909
AUTHOR . . . 1909
LICENSE . . . 1909
TRANSLATIONS . . . 1909
SEE ALSO . . . 1909
Modules . . . 1909
5.2.12 Cairo . . . 1981
Objects and their lifetime . . . 1981
Differences from upstream . . . 1982
Modules . . . 1982
Functions . . . 1983
5.2.13 Capture . . . 2009
Capture::Tiny . . . 2009
5.2.14 Carp . . . 2017
SYNOPSIS . . . 2017
DESCRIPTION . . . 2017

GLOBAL VARIABLES . . . 2019
BUGS . . . 2021
SEE ALSO . . . 2021
CONTRIBUTING . . . 2022
AUTHOR . . . 2022
COPYRIGHT . . . 2022
LICENSE . . . 2022
5.2.15 Class . . . 2022
Class::Inspector . . . 2022
Class::Load . . . 2029
Class::MOP . . . 2030
Class::Struct . . . 2031
Class::XSAccessor . . . 2038
5.2.16 Clone . . . 2046
Synopsis . . . 2046
Differences from upstream . . . 2046
Functions . . . 2046
5.2.17 Compress . . . 2048
Compress::Raw . . . 2048
Compress::Zlib . . . 2057
5.2.18 Config . . . 2072
Differences from upstream . . . 2072
Modules . . . 2073
Functions . . . 2073
5.2.19 Coro . . . 2085
SYNOPSIS . . . 2085
DESCRIPTION . . . 2086
CORO THREAD LIFE CYCLE . . . 2086
GLOBAL VARIABLES . . . 2091
SIMPLE CORO CREATION . . . 2092
STATIC METHODS . . . 2093
CORO OBJECT METHODS . . . 2095
GLOBAL FUNCTIONS . . . 2100
HOW TO WAIT FOR A CALLBACK . . . 2101
BUGS/LIMITATIONS . . . 2102
WINDOWS PROCESS EMULATION . . . 2103
SEE ALSO . . . 2104
AUTHOR/SUPPORT/CONTACT . . . 2104
Modules . . . 2104
5.2.20 Crypt . . . 2139
Crypt::OpenSSL . . . 2139
5.2.21 Cwd . . . 2140
Cwd's Perl half . . . 2141
Synopsis . . . 2141
Differences from upstream . . . 2141
Functions . . . 2141
5.2.22 DB . . . 2148
SYNOPSIS . . . 2148
DESCRIPTION . . . 2149
BUGS . . . 2151
AUTHOR . . . 2151

5.2.23 DBD . . . 2152
DBD::DBM . . . 2152
DBD::File . . . 2167
DBD::Gofer . . . 2175
DBD::Mem . . . 2196
DBD::Proxy . . . 2198
DBD::SQLite . . . 2202
DBD::Sponge . . . 2210
5.2.24 DBI . . . 2212
SYNOPSIS . . . 2212
DESCRIPTION . . . 2214
THE DBI PACKAGE AND CLASS . . . 2222
METHODS COMMON TO ALL HANDLES . . . 2233
ATTRIBUTES COMMON TO ALL HANDLES . . . 2238
DBI DATABASE HANDLE OBJECTS . . . 2254
DBI STATEMENT HANDLE OBJECTS . . . 2280
FURTHER INFORMATION . . . 2300
TRACING . . . 2309
DBI ENVIRONMENT VARIABLES . . . 2314
WARNING AND ERROR MESSAGES . . . 2315
Pure-Perl DBI . . . 2316
SEE ALSO . . . 2316
AUTHORS . . . 2318
COPYRIGHT . . . 2318
SUPPORT / WARRANTY . . . 2319
ACKNOWLEDGEMENTS . . . 2319
CONTRIBUTING . . . 2320
TRANSLATIONS . . . 2321
OTHER RELATED WORK AND PERL MODULES . . . 2321
TODO . . . 2322
Modules . . . 2325
5.2.25 DBM_Filter . . . 2385
SYNOPSIS . . . 2386
DESCRIPTION . . . 2386
What is a DBM Filter? . . . 2387
METHODS . . . 2387
Writing a Filter . . . 2388
Filters Included . . . 2390
NOTES . . . 2390
EXAMPLE . . . 2391
SEE ALSO . . . 2391
AUTHOR . . . 2391
Modules . . . 2391
5.2.26 DB_File . . . 2394
Synopsis . . . 2395
Opening a database . . . 2395
Sorted keys and duplicates . . . 2396
Arrays . . . 2396
The cursor . . . 2397
Files . . . 2397
Constants . . . 2397

Differences from upstream . . . 2398
Functions . . . 2398
5.2.27 Data . . . 2411
Data::Dumper . . . 2412
Data::UUID . . . 2415
5.2.28 Devel . . . 2419
Devel::Cover . . . 2419
Devel::Peek . . . 2419
Devel::SelfStubber . . . 2424
Devel::Size . . . 2425
5.2.29 Digest . . . 2426
SYNOPSIS . . . 2426
DESCRIPTION . . . 2426
OO INTERFACE . . . 2427
Digest speed . . . 2429
SEE ALSO . . . 2430
AUTHOR . . . 2430
Modules . . . 2430
5.2.30 DirHandle . . . 2475
SYNOPSIS . . . 2475
DESCRIPTION . . . 2475
5.2.31 Dumpvalue . . . 2475
SYNOPSIS . . . 2476
DESCRIPTION . . . 2476
5.2.32 DynaLoader . . . 2478
SYNOPSIS . . . 2478
DESCRIPTION . . . 2479
AUTHOR . . . 2485
5.2.33 EV . . . 2485
Synopsis . . . 2485
Watchers . . . 2486
Priorities . . . 2486
Constants . . . 2486
Backends . . . 2487
Signals and fork . . . 2487
Differences from upstream . . . 2487
Modules . . . 2487
Functions . . . 2488
5.2.34 Encode . . . 2512
Differences from upstream . . . 2513
Modules . . . 2513
Functions . . . 2513
5.2.35 English . . . 2537
SYNOPSIS . . . 2537
DESCRIPTION . . . 2537
PERFORMANCE . . . 2537
5.2.36 Env . . . 2538
SYNOPSIS . . . 2538
DESCRIPTION . . . 2538
LIMITATIONS . . . 2539
AUTHOR . . . 2539

5.2.37 Errno . . . 2539
Synopsis . . . 2539
Differences from upstream . . . 2539
5.2.38 Event . . . 2539
Exports . . . 2540
Functions written in Perl . . . 2540
Priorities . . . 2541
Customization . . . 2541
Differences from upstream . . . 2541
Modules . . . 2541
Functions . . . 2542
5.2.39 Exporter . . . 2557
SYNOPSIS . . . 2557
DESCRIPTION . . . 2558
Advanced Features . . . 2559
Good Practices . . . 2563
SEE ALSO . . . 2565
LICENSE . . . 2566
Modules . . . 2566
5.2.40 ExtUtils . . . 2566
ExtUtils::CBuilder . . . 2567
ExtUtils::Command . . . 2571
ExtUtils::Embed . . . 2575
ExtUtils::HasCompiler . . . 2579
ExtUtils::Install . . . 2579
ExtUtils::Installed . . . 2583
ExtUtils::Liblist . . . 2586
ExtUtils::MM . . . 2590
ExtUtils::MM_Any . . . 2590
ExtUtils::MM_Unix . . . 2606
ExtUtils::MY . . . 2616
ExtUtils::MakeMaker . . . 2616
ExtUtils::Manifest . . . 2652
ExtUtils::Miniperl . . . 2658
ExtUtils::PL2Bat . . . 2658
ExtUtils::Packlist . . . 2660
ExtUtils::testlib . . . 2662
5.2.41 FFI . . . 2663
FFI::Build . . . 2663
FFI::CheckLib . . . 2687
FFI::Platypus . . . 2693
FFI::Probe . . . 2799
FFI::Temp . . . 2810
5.2.42 Fatal . . . 2811
SYNOPSIS . . . 2811
BEST PRACTICE . . . 2811
DESCRIPTION . . . 2811
DIAGNOSTICS . . . 2812
BUGS . . . 2812
AUTHOR . . . 2812
LICENSE . . . 2813

SEE ALSO . . . 2813
5.2.43 Fcntl . . . 2813
Open flags (sysopen) . . . 2813
File-descriptor control (fcntl) . . . 2813
Lock types (flock) . . . 2813
Seek whence . . . 2814
File mode bits (stat) . . . 2814
Mode test helpers . . . 2814
Performance . . . 2814
Differences from upstream . . . 2815
Functions . . . 2815
5.2.44 File . . . 2816
File::Basename . . . 2817
File::Compare . . . 2820
File::Copy . . . 2820
File::DosGlob . . . 2823
File::Fetch . . . 2824
File::Find . . . 2829
File::Glob . . . 2835
File::GlobMapper . . . 2839
File::Map . . . 2844
File::Path . . . 2855
File::ShareDir . . . 2866
File::Spec . . . 2871
File::Temp . . . 2895
File::Which . . . 2911
File::stat . . . 2915
5.2.45 FileCache . . . 2917
SYNOPSIS . . . 2917
DESCRIPTION . . . 2918
CAVEATS . . . 2918
BUGS . . . 2918
5.2.46 FileHandle . . . 2918
SYNOPSIS . . . 2919
DESCRIPTION . . . 2919
SEE ALSO . . . 2921
5.2.47 Filter . . . 2921
Filter::Simple . . . 2921
Filter::Util . . . 2931
Filter::cpp . . . 2940
Filter::decrypt . . . 2941
Filter::exec . . . 2942
Filter::m4 . . . 2943
Filter::sh . . . 2944
Filter::tee . . . 2945
5.2.48 FindBin . . . 2946
SYNOPSIS . . . 2946
DESCRIPTION . . . 2946
EXPORTABLE VARIABLES . . . 2946
KNOWN ISSUES . . . 2947
AUTHORS . . . 2947

COPYRIGHT 2947
5.2.49 GD 2947
The colour index 2948
GD::Image 2948
GD::Font - the built-in bitmap fonts 2949
What is native and what is Perl 2949
Differences from upstream 2949
Modules 2950
Functions 2950
5.2.50 GDBM_File 2966
Synopsis 2967
Opening a database 2967
Errors 2967
Options 2967
DBM filters 2968
Files 2968
Constants 2968
Differences from upstream 2968
Functions 2969
5.2.51 Getopt 2975
Getopt::Long 2975
Getopt::Std 2996
5.2.52 Guard 2998
Differences from upstream 2998
Functions 2998
5.2.53 HTML 2999
HTML::Entities 2999
HTML::Filter 3002
HTML::HeadParser 3003
HTML::LinkExtor 3006
HTML::Parser 3007
HTML::PullParser 3009
HTML::Tagset 3011
HTML::TokeParser 3015
5.2.54 HTTP 3018
HTTP::Parser 3018
HTTP::Tiny 3021
5.2.55 Hash 3031
Hash::Util 3032
5.2.56 I18N 3050
I18N::Collate 3050
I18N::LangTags 3051
I18N::Langinfo 3083
5.2.57 IO 3084
Differences from upstream 3085
Modules 3085
5.2.58 IPC 3385
IPC::Cmd 3385
IPC::Msg 3393
IPC::Open2 3395
IPC::Open3 3397

IPC::Semaphore . . . 3400
IPC::SharedMem . . . 3402
IPC::SysV . . . 3404
5.2.59 Imager . . . 3406
What is available . . . 3406
Drawing . . . 3407
Where Imager differs from the other raster modules . . . 3407
Four kinds of image . . . 3407
Colours . . . 3408
The channel mask . . . 3408
Errors . . . 3408
Files . . . 3408
Masked images . . . 3409
Fills . . . 3409
One image onto another . . . 3409
Channels . . . 3410
Scaling . . . 3410
Rotation and matrix transforms . . . 3410
Expressions per pixel . . . 3411
Filters . . . 3411
Paletted images from direct ones . . . 3411
Differences from upstream . . . 3412
Modules . . . 3413
Functions . . . 3414
5.2.60 JSON . . . 3460
JSON::PP . . . 3461
JSON::XS . . . 3472
5.2.61 LMDB_File . . . 3477
Synopsis . . . 3477
The tie interface . . . 3477
LMDB::Env . . . 3478
LMDB::Txn . . . 3478
LMDB_File: a database in a transaction . . . 3479
LMDB::Cursor . . . 3479
Errors . . . 3479
Constants . . . 3480
Differences from upstream . . . 3480
5.2.62 List . . . 3480
List::MoreUtils . . . 3480
List::Util . . . 3482
5.2.63 Locale . . . 3510
Locale::Maketext . . . 3510
5.2.64 MIME . . . 3536
MIME::Base64 . . . 3536
MIME::Charset . . . 3545
MIME::QuotedPrint . . . 3552
5.2.65 Math . . . 3555
Math::BigFloat . . . 3555
Math::BigInt . . . 3574
Math::BigRat . . . 3641
Math::Complex . . . 3660

Math::GMP . . . 3669
Math::Random . . . 3684
Math::Trig . . . 3685
5.2.66 Memoize . . . 3695
Synopsis . . . 3695
Differences from upstream . . . 3696
Modules . . . 3696
Functions . . . 3696
5.2.67 Module . . . 3703
Module::CoreList . . . 3703
Module::Load . . . 3709
Module::Loaded . . . 3717
Module::Metadata . . . 3718
5.2.68 Moose . . . 3724
Synopsis . . . 3724
Differences from upstream . . . 3724
5.2.69 NDBM_File . . . 3725
Synopsis . . . 3725
Opening a database . . . 3725
DBM filters . . . 3725
Files . . . 3725
Differences from upstream . . . 3726
Functions . . . 3726
5.2.70 NEXT . . . 3726
SYNOPSIS . . . 3726
DESCRIPTION . . . 3727
SEE ALSO . . . 3733
AUTHOR . . . 3733
BUGS AND IRRITATIONS . . . 3733
COPYRIGHT . . . 3733
5.2.71 Net . . . 3733
Net::Cmd . . . 3734
Net::Config . . . 3738
Net::Domain . . . 3742
Net::FTP . . . 3743
Net::NNTP . . . 3755
Net::Netrc . . . 3763
Net::POP3 . . . 3766
Net::Ping . . . 3770
Net::SMTP . . . 3779
Net::SSLeay . . . 3787
Net::Time . . . 3793
Net::hostent . . . 3795
Net::netent . . . 3796
Net::protoent . . . 3799
Net::servent . . . 3799
5.2.72 O . . . 3801
SYNOPSIS . . . 3801
DESCRIPTION . . . 3801
CONVENTIONS . . . 3801
IMPLEMENTATION . . . 3801

BUGS . . . 3802
AUTHOR . . . 3802
5.2.73 ODBM_File . . . 3802
Synopsis . . . 3802
Opening a database . . . 3802
DBM filters . . . 3803
Files . . . 3803
Differences from upstream . . . 3803
5.2.74 Opcode . . . 3803
Differences from upstream . . . 3804
Functions . . . 3804
5.2.75 PDL . . . 3806
Differences from upstream . . . 3807
Modules . . . 3807
Functions . . . 3808
5.2.76 POSIX . . . 3904
Differences from upstream . . . 3905
Modules . . . 3905
Functions . . . 3905
5.2.77 Package . . . 3946
Package::Stash . . . 3946
5.2.78 PadWalker . . . 3949
Differences from upstream . . . 3950
Functions . . . 3950
5.2.79 Params . . . 3958
Params::Check . . . 3958
Params::Util . . . 3964
5.2.80 Parse . . . 3967
Parse::CPAN . . . 3967
5.2.81 Perl . . . 3971
Perl::OSType . . . 3971
5.2.82 PerlIO . . . 3973
SYNOPSIS . . . 3973
DESCRIPTION . . . 3973
AUTHOR . . . 3978
SEE ALSO . . . 3978
Modules . . . 3979
5.2.83 Peta . . . 3985
Peta::Dict . . . 3985
Peta::FFI . . . 3986
Peta::ReadLine . . . 4035
Peta::Shell . . . 4047
Peta::WebGPU . . . 4061
Peta::XS . . . 4070
5.2.84 Pod . . . 4079
Pod::Checker . . . 4079
Pod::Escapes . . . 4087
Pod::Functions . . . 4090
Pod::Html . . . 4091
Pod::Man . . . 4095
Pod::ParseLink . . . 4108

Pod::Perldoc . . . 4109
Pod::Simple . . . 4121
Pod::Text . . . 4181
Pod::Usage . . . 4193
5.2.85 SDBM_File . . . 4202
Synopsis . . . 4202
Persistence . . . 4202
Size limits . . . 4203
Security . . . 4203
Differences from upstream . . . 4203
Functions . . . 4203
5.2.86 SDL2 . . . 4204
Two layers . . . 4204
Lifetime . . . 4204
Errors . . . 4205
Exports and constants . . . 4205
Modules . . . 4205
Functions . . . 4205
5.2.87 Safe . . . 4222
SYNOPSIS . . . 4222
DESCRIPTION . . . 4222
WARNING . . . 4223
METHODS . . . 4224
RISKS . . . 4227
AUTHOR . . . 4228
5.2.88 Scalar . . . 4228
Scalar::List . . . 4228
Scalar::Util . . . 4229
5.2.89 Search . . . 4236
Search::Dict . . . 4236
5.2.90 SelectSaver . . . 4237
SYNOPSIS . . . 4237
DESCRIPTION . . . 4237
5.2.91 SelfLoader . . . 4238
SYNOPSIS . . . 4238
DESCRIPTION . . . 4238
Multiple packages and fully qualified subroutine names . . . 4240
AUTHOR . . . 4241
COPYRIGHT AND LICENSE . . . 4241
5.2.92 Socket . . . 4242
What lives here . . . 4242
The dual-mode shortcuts . . . 4243
Differences from upstream . . . 4243
Functions . . . 4243
5.2.93 Storable . . . 4258
Synopsis . . . 4259
The functions, written in Perl . . . 4259
Differences from upstream . . . 4260
Functions . . . 4260
5.2.94 String . . . 4261
String::CRC32 . . . 4261

5.2.95 Sub 4262
Sub::Exporter 4262
Sub::Identify 4264
Sub::Name 4266
Sub::Util 4267
5.2.96 Symbol 4270
SYNOPSIS 4270
DESCRIPTION 4271
BUGS 4272
5.2.97 Sys 4272
Sys::Hostname 4272
Sys::Syslog 4272
5.2.98 TAP 4274
TAP::Base 4274
TAP::Formatter 4275
TAP::Harness 4282
TAP::Object 4293
TAP::Parser 4295
5.2.99 Template 4373
Template::Stash 4374
5.2.100 Term 4376
Term::ANSIColor 4376
Term::Cap 4389
Term::Complete 4393
Term::ReadKey 4394
Term::ReadLine 4399
Term::Table 4403
5.2.101 Test 4415
SYNOPSIS 4415
DESCRIPTION 4416
QUICK START GUIDE 4416
TEST TYPES 4420
ONFAIL 4420
BUGS and CAVEATS 4421
ENVIRONMENT 4422
NOTE 4423
SEE ALSO 4423
AUTHOR 4423
Modules 4423
5.2.102 Test2 4483
DESCRIPTION 4483
GETTING STARTED 4484
NAMESPACE LAYOUT 4484
SEE ALSO 4485
CONTACTING US 4486
SOURCE 4486
MAINTAINERS 4486
AUTHORS 4486
COPYRIGHT 4486
Modules 4486
5.2.103 Text 4889

Text::Abbrev . . . 4889
Text::Balanced . . . 4890
Text::CSV . . . 4911
Text::CSV_PP . . . 4963
Text::LevenshteinXS . . . 5015
Text::LineFold . . . 5016
Text::ParseWords . . . 5018
Text::Tabs . . . 5021
Text::Wrap . . . 5022
5.2.104 Thread . . . 5025
DEPRECATED . . . 5025
HISTORY . . . 5025
SYNOPSIS . . . 5026
DESCRIPTION . . . 5026
FUNCTIONS . . . 5026
METHODS . . . 5028
DEFUNCT . . . 5028
SEE ALSO . . . 5029
Modules . . . 5029
5.2.105 Tie . . . 5037
Tie::Array . . . 5038
Tie::File . . . 5040
Tie::Handle . . . 5051
Tie::Hash . . . 5053
Tie::Memoize . . . 5060
Tie::RefHash . . . 5061
Tie::Scalar . . . 5063
Tie::StdHandle . . . 5065
Tie::SubstrHash . . . 5065
5.2.106 Time . . . 5066
Time::HiRes . . . 5067
Time::Local . . . 5069
Time::Piece . . . 5073
Time::Seconds . . . 5075
Time::gmtime . . . 5076
Time::localtime . . . 5077
Time::tm . . . 5078
5.2.107 Types . . . 5078
Types::Serialiser . . . 5078
5.2.108 UNIVERSAL . . . 5082
SYNOPSIS . . . 5082
DESCRIPTION . . . 5083
WARNINGS . . . 5085
EXPORTS . . . 5085
5.2.109 URL . . . 5085
URL::Encode . . . 5085
5.2.110 Unicode . . . 5090
Unicode::Collate . . . 5091
Unicode::GCString . . . 5122
Unicode::LineBreak . . . 5124
Unicode::Normalize . . . 5127

Unicode::UCD . . . 5131
5.2.111 User . . . 5132
User::grent . . . 5132
User::pwent . . . 5133
5.2.112 XML . . . 5135
XML::Parser . . . 5135
5.2.113 XSLoader . . . 5154
VERSION . . . 5154
SYNOPSIS . . . 5155
DESCRIPTION . . . 5155
5.2.114 XString . . . 5155
Synopsis . . . 5155
Differences from upstream . . . 5156
Functions . . . 5156
5.2.115 YAML . . . 5157
YAML::LibYAML . . . 5157
YAML::PP . . . 5157
YAML::XS . . . 5217
5.2.116 _charnames . . . 5225
Differences from upstream . . . 5225
5.2.117 attributes . . . 5225
Synopsis . . . 5226
Functions . . . 5226
Differences from upstream . . . 5226
5.2.118 autodie . . . 5226
SYNOPSIS . . . 5226
DESCRIPTION . . . 5227
EXCEPTIONS . . . 5227
CATEGORIES . . . 5228
FUNCTION SPECIFIC NOTES . . . 5229
GOTCHAS . . . 5230
DIAGNOSTICS . . . 5231
Tips and Tricks . . . 5231
BUGS . . . 5231
FEEDBACK . . . 5232
AUTHOR . . . 5232
LICENSE . . . 5232
SEE ALSO . . . 5232
ACKNOWLEDGEMENTS . . . 5232
Modules . . . 5232
5.2.119 autouse . . . 5248
SYNOPSIS . . . 5248
DESCRIPTION . . . 5248
WARNING . . . 5249
AUTHOR . . . 5249
SEE ALSO . . . 5249
5.2.120 base . . . 5249
SYNOPSIS . . . 5249
DESCRIPTION . . . 5250
DIAGNOSTICS . . . 5250
HISTORY . . . 5251

CAVEATS . . . 5251
SEE ALSO . . . 5251
5.2.121 bigfloat . . . 5251
SYNOPSIS . . . 5251
DESCRIPTION . . . 5251
CAVEATS . . . 5256
EXAMPLES . . . 5257
BUGS . . . 5257
SUPPORT . . . 5257
LICENSE . . . 5258
SEE ALSO . . . 5258
AUTHORS . . . 5258
5.2.122 bigint . . . 5258
SYNOPSIS . . . 5258
DESCRIPTION . . . 5259
CAVEATS . . . 5264
EXAMPLES . . . 5265
BUGS . . . 5265
SUPPORT . . . 5266
LICENSE . . . 5266
SEE ALSO . . . 5266
AUTHORS . . . 5266
5.2.123 bignum . . . 5266
SYNOPSIS . . . 5266
DESCRIPTION . . . 5267
CAVEATS . . . 5273
EXAMPLES . . . 5275
BUGS . . . 5275
SUPPORT . . . 5275
LICENSE . . . 5275
SEE ALSO . . . 5276
AUTHORS . . . 5276
5.2.124 bigrat . . . 5276
SYNOPSIS . . . 5276
DESCRIPTION . . . 5276
CAVEATS . . . 5281
EXAMPLES . . . 5282
BUGS . . . 5282
SUPPORT . . . 5282
LICENSE . . . 5283
SEE ALSO . . . 5283
AUTHORS . . . 5283
5.2.125 blib . . . 5283
SYNOPSIS . . . 5283
DESCRIPTION . . . 5283
BUGS . . . 5283
AUTHOR . . . 5284
5.2.126 builtin . . . 5284
SYNOPSIS . . . 5284
DESCRIPTION . . . 5284
FUNCTIONS . . . 5286

SEE ALSO . . . 5292
5.2.127 bytes . . . 5292
NOTICE . . . 5292
SYNOPSIS . . . 5293
DESCRIPTION . . . 5293
LIMITATIONS . . . 5294
SEE ALSO . . . 5294
5.2.128 charnames . . . 5294
SYNOPSIS . . . 5294
DESCRIPTION . . . 5295
LOOSE MATCHES . . . 5296
ALIASES . . . 5297
CUSTOM ALIASES . . . 5297
charnames::string_vianame(*name*) . . . 5298
charnames::vianame(*name*) . . . 5298
charnames::viacode(*code*) . . . 5299
CUSTOM TRANSLATORS . . . 5300
BUGS . . . 5301
5.2.129 constant . . . 5301
SYNOPSIS . . . 5301
DESCRIPTION . . . 5301
NOTES . . . 5302
TECHNICAL NOTES . . . 5304
CAVEATS . . . 5304
SEE ALSO . . . 5304
BUGS . . . 5305
AUTHORS . . . 5305
COPYRIGHT & LICENSE . . . 5305
5.2.130 deprecate . . . 5305
SYNOPSIS . . . 5305
DESCRIPTION . . . 5305
EXPORT . . . 5306
SEE ALSO . . . 5306
AUTHOR . . . 5306
COPYRIGHT AND LICENSE . . . 5306
5.2.131 diagnostics . . . 5306
SYNOPSIS . . . 5306
DESCRIPTION . . . 5307
EXAMPLES . . . 5308
INTERNALS . . . 5309
BUGS . . . 5309
AUTHOR . . . 5309
5.2.132 encoding . . . 5309
WARNING . . . 5310
SYNOPSIS . . . 5310
DESCRIPTION . . . 5310
OPTIONS . . . 5312
CAVEATS . . . 5313
EXAMPLE - Greekperl . . . 5316
BUGS . . . 5316
HISTORY . . . 5317

SEE ALSO . . . 5317
Modules . . . 5317
5.2.133 experimental . . . 5320
VERSION . . . 5320
SYNOPSIS . . . 5320
DESCRIPTION . . . 5321
SEE ALSO . . . 5323
AUTHOR . . . 5323
COPYRIGHT AND LICENSE . . . 5323
5.2.134 feature . . . 5324
SYNOPSIS . . . 5324
DESCRIPTION . . . 5324
AVAILABLE FEATURES . . . 5325
FEATURE BUNDLES . . . 5332
IMPLICIT LOADING . . . 5335
CHECKING FEATURES . . . 5336
5.2.135 fields . . . 5337
SYNOPSIS . . . 5337
DESCRIPTION . . . 5338
SEE ALSO . . . 5339
5.2.136 filetest . . . 5339
SYNOPSIS . . . 5339
DESCRIPTION . . . 5339
5.2.137 if . . . 5341
SYNOPSIS . . . 5341
DESCRIPTION . . . 5341
BUGS . . . 5342
SEE ALSO . . . 5342
AUTHOR . . . 5342
COPYRIGHT AND LICENCE . . . 5342
5.2.138 integer . . . 5342
SYNOPSIS . . . 5343
DESCRIPTION . . . 5343
5.2.139 less . . . 5344
SYNOPSIS . . . 5344
DESCRIPTION . . . 5344
FOR MODULE AUTHORS . . . 5344
CAVEATS . . . 5345
5.2.140 lib . . . 5345
SYNOPSIS . . . 5345
DESCRIPTION . . . 5345
CAVEATS . . . 5346
NOTES . . . 5347
SEE ALSO . . . 5347
AUTHOR . . . 5347
COPYRIGHT AND LICENSE . . . 5347
5.2.141 locale . . . 5347
SYNOPSIS . . . 5347
DESCRIPTION . . . 5348
5.2.142 mro . . . 5348
Synopsis . . . 5349

next:: dispatch . . . 5349
Differences from upstream . . . 5349
Functions . . . 5349
5.2.143 ok . . . 5354
SYNOPSIS . . . 5354
DESCRIPTION . . . 5355
CC0 1.0 Universal . . . 5355
5.2.144 open . . . 5355
SYNOPSIS . . . 5355
DESCRIPTION . . . 5355
IMPLEMENTATION DETAILS . . . 5357
SEE ALSO . . . 5357
5.2.145 ops . . . 5357
SYNOPSIS . . . 5357
DESCRIPTION . . . 5357
SEE ALSO . . . 5357
5.2.146 overload . . . 5357
SYNOPSIS . . . 5357
DESCRIPTION . . . 5358
IMPLEMENTATION . . . 5371
COOKBOOK . . . 5371
AUTHOR . . . 5380
SEE ALSO . . . 5380
DIAGNOSTICS . . . 5380
BUGS AND PITFALLS . . . 5380
5.2.147 overloading . . . 5382
SYNOPSIS . . . 5382
DESCRIPTION . . . 5382
5.2.148 parent . . . 5382
SYNOPSIS . . . 5382
DESCRIPTION . . . 5383
HISTORY . . . 5384
CAVEATS . . . 5384
SEE ALSO . . . 5384
AUTHORS AND CONTRIBUTORS . . . 5384
MAINTAINER . . . 5384
LICENSE . . . 5384
5.2.149 perlfaq . . . 5384
VERSION . . . 5384
DESCRIPTION . . . 5384
TABLE OF CONTENTS . . . 5385
THE QUESTIONS . . . 5385
CREDITS . . . 5394
AUTHOR AND COPYRIGHT . . . 5394
5.2.150 re . . . 5394
Synopsis . . . 5394
What you can do with it . . . 5395
Differences from upstream . . . 5395
Functions . . . 5395
5.2.151 sigtrap . . . 5400
SYNOPSIS . . . 5400

DESCRIPTION . . . 5400
OPTIONS . . . 5401
EXAMPLES . . . 5402
5.2.152 sort . . . 5403
SYNOPSIS . . . 5403
DESCRIPTION . . . 5403
CAVEATS . . . 5403
5.2.153 source . . . 5404
source::encoding . . . 5404
5.2.154 stable . . . 5404
VERSION . . . 5404
SYNOPSIS . . . 5405
DESCRIPTION . . . 5405
SEE ALSO . . . 5406
AUTHOR . . . 5406
COPYRIGHT AND LICENSE . . . 5406
5.2.155 strict . . . 5406
SYNOPSIS . . . 5406
DESCRIPTION . . . 5406
HISTORY . . . 5408
5.2.156 subs . . . 5408
SYNOPSIS . . . 5408
DESCRIPTION . . . 5408
5.2.157 threads . . . 5408
VERSION . . . 5408
WARNING . . . 5409
SYNOPSIS . . . 5409
DESCRIPTION . . . 5410
EXITING A THREAD . . . 5414
THREAD STATE . . . 5415
THREAD CONTEXT . . . 5416
THREAD STACK SIZE . . . 5417
THREAD SIGNALLING . . . 5418
WARNINGS . . . 5420
ERRORS . . . 5420
BUGS AND LIMITATIONS . . . 5421
REQUIREMENTS . . . 5425
SEE ALSO . . . 5425
AUTHOR . . . 5425
LICENSE . . . 5425
ACKNOWLEDGEMENTS . . . 5425
Modules . . . 5425
5.2.158 utf8 . . . 5434
SYNOPSIS . . . 5434
DESCRIPTION . . . 5435
BUGS . . . 5438
SEE ALSO . . . 5438
5.2.159 vars . . . 5439
SYNOPSIS . . . 5439
DESCRIPTION . . . 5439
5.2.160 version . . . 5439

Synopsis . . . 5439
Operators . . . 5440
Functions . . . 5440
5.2.161 warnings . . . 5442
SYNOPSIS . . . 5442
DESCRIPTION . . . 5442
FUNCTIONS . . . 5455
Modules . . . 5457

6 Changelog **5459**
6.1 pperl 0.6.25 . . . 5459
6.2 pperl 0.6.24 . . . 5459
6.2.1 JIT and parallelization . . . 5459
6.2.2 Language and runtime . . . 5460
6.2.3 Modules . . . 5462
6.2.4 Command line . . . 5468
6.2.5 Documentation . . . 5470
6.3 pperl 0.6.23 . . . 5470
6.3.1 Modules . . . 5470
6.3.2 Command line . . . 5472
6.3.3 Documentation . . . 5474
6.4 pperl 0.6.22 . . . 5474
6.4.1 Language and runtime . . . 5475
6.4.2 JIT and parallelization . . . 5475
6.4.3 Modules . . . 5476
6.5 pperl 0.6.21 . . . 5480
6.5.1 Language and runtime . . . 5480
6.5.2 JIT and parallelization . . . 5480
6.5.3 Modules . . . 5482
6.5.4 Command line . . . 5485
6.5.5 Distribution . . . 5486
6.5.6 Documentation . . . 5486
6.6 pperl 0.6.20 . . . 5486
6.7 pperl 0.6.19 . . . 5487
6.7.1 Language and runtime . . . 5487
6.7.2 JIT and parallelization . . . 5487
6.7.3 Modules . . . 5489
6.7.4 Command line . . . 5491
6.7.5 Documentation . . . 5491
6.8 pperl 0.6.18 . . . 5492
6.9 pperl 0.6.17 . . . 5493
6.9.1 Language and runtime . . . 5493
6.9.2 JIT and parallelization . . . 5493
6.9.3 Modules . . . 5494
6.9.4 Documentation . . . 5495
6.10 pperl 0.6.16 . . . 5495
6.10.1 JIT and parallelization . . . 5495
6.10.2 Modules . . . 5497
6.10.3 Command line . . . 5497
6.10.4 Distribution . . . 5497
6.10.5 Documentation . . . 5497

6.11 pperl 0.6.15 . . . 5498
6.11.1 JIT and parallelization . . . 5498
6.11.2 Modules . . . 5499
6.12 pperl 0.6.14 . . . 5499
6.13 pperl 0.6.13 . . . 5500
6.13.1 JIT and parallelization . . . 5500
6.13.2 Modules . . . 5501
6.13.3 Documentation . . . 5501
6.14 pperl 0.6.12 . . . 5501
6.14.1 JIT and parallelization . . . 5501
6.14.2 Modules . . . 5502
6.14.3 Documentation . . . 5502
6.15 pperl 0.6.11 . . . 5502
6.16 pperl 0.6.9 . . . 5503
6.16.1 Language and runtime . . . 5503
6.16.2 JIT and parallelization . . . 5503
6.16.3 Modules . . . 5504
6.16.4 Command line . . . 5507
6.16.5 Documentation . . . 5508
6.17 pperl 0.6.8 . . . 5508
6.18 pperl 0.6.6 . . . 5509
6.18.1 Modules . . . 5509
6.18.2 Command line . . . 5510
6.18.3 Distribution . . . 5510
6.19 pperl 0.6.5 . . . 5510
6.19.1 JIT and parallelization . . . 5510
6.19.2 Modules . . . 5510
6.19.3 Command line . . . 5512
6.19.4 Documentation . . . 5512
6.20 pperl 0.6.3 . . . 5512
6.20.1 Language and runtime . . . 5512
6.20.2 JIT and parallelization . . . 5512
6.20.3 Modules . . . 5513
6.20.4 Command line . . . 5514
6.20.5 Documentation . . . 5514
6.21 pperl 0.6.2 . . . 5514
6.21.1 Language and runtime . . . 5514
6.21.2 JIT and parallelization . . . 5514
6.21.3 Modules . . . 5515
6.21.4 Command line . . . 5515
6.21.5 Documentation . . . 5516
6.22 pperl 0.6.1 . . . 5516
6.23 pperl 0.6.0 . . . 5516
6.23.1 Language and runtime . . . 5516
6.23.2 JIT and parallelization . . . 5516
6.23.3 Modules . . . 5517
6.23.4 Command line . . . 5518
6.23.5 Distribution . . . 5518
6.23.6 Documentation . . . 5518
6.24 pperl 0.5.6 . . . 5518
6.24.1 Language and runtime . . . 5519

6.24.2 Modules . . . 5519
6.24.3 Command line . . . 5520
6.24.4 Documentation . . . 5520

A Disclaimer & bug reports **5523**
A.1 PetaPerl is not upstream Perl . . . 5523
A.2 Bug reports . . . 5523
A.3 Intent of compatibility . . . 5524
A.4 Upstream's contributions . . . 5524
A.5 License . . . 5524

B License **5525**
B.1 PetaMem Research Preview License . . . 5525
B.1.1 1. Definitions . . . 5525
B.1.2 2. Grant . . . 5525
B.1.3 3. Attribution and Non-Appropriation . . . 5525
B.1.4 4. Reverse Engineering . . . 5525
B.1.5 5. Research Preview Status . . . 5525
B.1.6 6. No Warranty . . . 5526
B.1.7 7. Limitation of Liability . . . 5526
B.1.8 8. Termination . . . 5526
B.1.9 9. Reservation of Rights . . . 5526
B.1.10 10. Governing Law and Venue . . . 5526

C Glossary **5527**

Index **5529**

A next-generation Perl 5 runtime written in Rust, with JIT and auto-parallelization. This documentation covers installation, usage, the built-in module reference, and the internals of the interpreter.

This is the **pperl 0.6.25** documentation, built . (translated ≈ words, %)

Getting started First steps: install pperl, run your first script, see what's different from system perl.

Getting Started How-To Task-oriented material. **Guides** are long-form and multi-chapter (regex, OO, debugging, concurrent execution, pperl architecture). **Tutorials** are single-sitting (pack, I/O streams, references, Unicode).

How-To Reference Per-module reference for every built-in module; Perl language reference (variables, functions, operators, regex); the changelog of every version.

Reference Index & glossary Alphabetical index of every documented module, function, and term; short glossary of project vocabulary.

genindex

CHAPTER

ONE

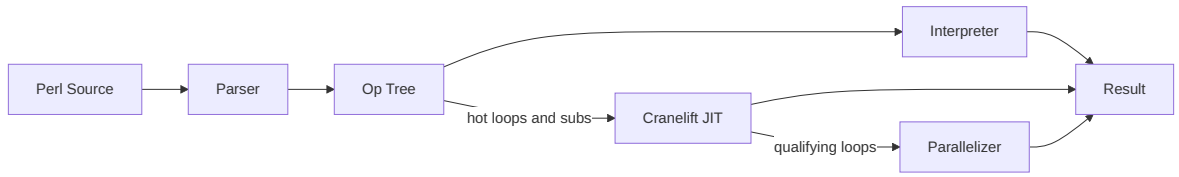
INTRODUCTION

pperl is a next-generation Perl 5 platform written in Rust, targeting Perl 5.44+ compatibility.

1.1 Goals

- **JIT Compilation** - Native code generation via Cranelift for hot loops and hot recursive subs, with byte-identical output to the interpreter
- **Auto-Parallelization** - Qualifying JIT-compiled `for` and `while` loops spread across CPU cores, only where the result is provably identical
- **Pure Perl Viability** - Fast enough that XS becomes optional

1.2 Architecture Overview



The parser and the interpreter follow perl5's own design: the same grammar, the same op tree, the same op-by-op execution. On top of that, the *JIT* compiles hot code to native machine code and declines whatever it cannot reproduce exactly, and the *parallelizer* spreads qualifying compiled loops across cores. Modules that perl5 implements in XS are built into pperl as *native modules*.

1.3 Quick Start

```
# Run a script
pperl script.pl

# Run one-liner
pperl -e 'print "Hello, World!\n"'

# Check syntax only
pperl -c script.pl
```

1.4 Compatibility

pperl aims for high compatibility with Perl 5.44+.

1.5 Platform Support

- **Linux only**, on the architectures Cranelift supports: x86-64, AArch64, RISC-V and IBM z/Architecture
- No Windows, macOS, or other platforms

CHAPTER

TWO

GETTING STARTED

2.1 Installation

Download binaries and find documentation at perl.petamem.com.

2.2 First Script

```
#!/usr/bin/env pperl
print "Hello from PetaPerl!\n";
```

2.3 Command Line Options

Option	Description
`-e 'code'`	Execute one-liner
`-E 'code'`	Like `-e`, with all optional features enabled (`say`, ...)
`-I dir`	Prepend `dir` to the module search path `@INC`
`-c`	Check syntax only
`-w`	Enable warnings
`-v`	Show version
`-V`	Show configuration, `%ENV` and `@INC`
`--no-jit`	Disable JIT compilation
`--no-parallel`	Disable auto-parallelization

See *CLI options* for the full option list.

2.4 Notes

- Defaults match perl: modern features (`say`, `state` and friends) are enabled by `use v5.44;` in scripts or `-E` for one-liners
- XS modules are not supported; native Rust reimplementations are provided for common modules
- Module search paths work as in perl: `-I` and the `PERL5LIB` environment variable are honored; `PPERL5LIB` additionally provides pperl-only paths (see *CLI options*)

CHAPTER

THREE

CLI OPTIONS

pperl accepts a small set of command-line flags. Most are `perl`-compatible; a handful are pperl-specific (JIT, parallelism). The authoritative list is whatever `pperl --help` prints; the tables below show those flags, extracted from the interpreter source and rendered here on every documentation build.

> **Note**
>
> The actual `pperl --help` output on the terminal is always in English - the binary does not localise help strings. Translated documentation shows the same flags with descriptions in the page's language so readers can study what each flag does; flag tokens (`-T`, `--threads=N`, ...) are always verbatim.

PetaPerl - Next-generation Perl 5 runtime

3.1 Usage

```
pperl [OPTIONS] [--] [SCRIPT [ARGS...]]
pperl -e 'code'
pperl -E 'code'
```

3.2 Perl-compatible options

Flag	Description
`-e 'code'`	Execute one-liner
`-E 'code'`	Like -e, but enable all optional features (say, ...)
`-Mmodule`	use module before executing (-MO=Deparse, -Mstrict)
`-mmodule`	use module () before executing (no imports)
`-Idirectory`	Prepend directory to @INC
`-c`	Check syntax only, don't execute
`-w`	Enable warnings
`-W`	Enable all warnings, even under 'no warnings'
`-X`	Disable all warnings, even under 'use warnings'
`-s`	Turn the program's -name[=value] arguments into variables
`-n`	Loop over input lines (no auto-print), like while(<>)
`-p`	Loop over input lines and auto-print, like a sed filter
`-a`	Autosplit each line into @F (with -n or -p)
`-F pattern`	Set the -a split pattern (implies -a)
`-l[octal]`	Auto-chomp input, set output record separator
`-0[octal]`	Set input record separator (-0 = NUL, -00 = paragraph)
`-i[ext]`	Edit files in place (optional backup extension)
`-T`	Taint mode (forced)
`-t`	Taint warnings
`-v`	Show version information
`-V`	Show configuration summary, %ENV and @INC
`-V:key`	Show the value of a Config key, e.g. -V:archname
`--shell [--norc]`	ipps, the interactive mode: type Perl, see the value, keep everything. Also entered by running pperl under the name ipps. –norc skips ~/.pperl/ipps/rc.pl
`-h, -?, --help`	Show this help message

3.3 pperl-specific options

Flag	Description
`--stats`	Show performance statistics (time, memory)

3.4 Runtime options

Flag	Description
`--no-jit`	Disable JIT compilation (on by default)
`--no-regex-jit`	Disable only the regex JIT, leaving the op JIT on
`--no-parallel`	Disable auto-parallelization (on by default)
`--threads=SPEC`	Parallel thread count. SPEC is one of:
`phys`	all physical cores (DEFAULT)
`log`	all logical cores (hyperthreads)
`phys-N`	physical minus N (min 1)
`log-N`	logical minus N (min 1)
`P%`	P percent of logical cores
`N`	an explicit count
`--parallel-thresh-old=N`	Minimum iterations to parallelize (default: 100)
`--jit-stats`	Print JIT summary to stderr at exit (see also PPERL_JIT_LOG)

3.5 Regex JIT

The regex JIT is separate from the op JIT and on by default. It compiles a pattern only after that pattern has matched against 64 KiB of subject, and only for the shapes it supports; anything else runs on the standard engine and is refused once, at pattern-compile time.

Flag	Description
`--no-r`	Disable the regex JIT, leaving the op JIT on. Costs nothing to leave on for workloads it helps, but a program whose patterns are ALL refused still pays a small per-match dispatch cost (~3%); turning it off removes that path completely. –no-jit disables it too, along with the op JIT.
`PPERL_`	The same switch as an environment variable, for wrapping a program you cannot pass flags to. PPERL_RXJIT=1 forces it ON even under –no-jit (a diagnostic, not a normal posture).
`PPERL_`	Report, per pattern, whether it was admitted, refused (with the reason) or compiled. Run this FIRST: it is what tells you whether a given workload is one of the ones worth disabling.

3.6 Module installation

Flag	Description
`--cp MODU ..`	Install CPAN distributions for pperl with the built-in CPAN client. Everything after this flag goes to it, as to the `cpan` command. Besides a module name it takes a git repository (URL ending in .git, optionally .git@REVISION), a directory (./DIR) or an archive file. Pure Perl only: pperl runs no XS. Modules built into the binary are reported as already installed, so they are never rebuilt.
`--cp --sy tem`	Install into the system overlay instead of the per-user tree ($PPERL_HOME, default ~/.pperl)

3.7 Daemon options

Flag	Description
`--dae- mon`	Run the fast-start daemon in the foreground
`--dae- mon=star`	Start the daemon in the background
`--dae- mon=stop`	Stop the running daemon for this binary
`--dae- mon=sta- tus`	Show daemon status and cached script templates
`--dae- mon-idle`	Daemon exits after SECS without a request (default: 900)
`--dae- mon-max-`	Cache at most N script templates (default: 100)
`--dae- mon-re- set`	Drop all cached script templates
`--dae- mon-re- set=FILE`	Drop the cached template for FILE only
`--via-da mon`	Route this run through the daemon. With a script file that means reusing its cached compiled image; the script itself stays a normal argument: pperl –via-daemon myscript.pl (PPERL_DAEMON=2 does the same without the flag)
`--via-da mon=trun`	Force the shared zygote: fork a ready interpreter but compile the script every run. For testing the daemon; it saves little. Same as PPERL_DAEMON=1
`--build-`	Print the binary identity that keys the daemon socket

3.8 Examples

```
pperl script.pl
```

Execute a Perl script

```
pperl -E 'say "Hello!"'
```

Execute a one-liner (-E enables say, state, etc.)

```
pperl -c script.pl
```

Check script syntax without executing

```
pperl script.pl arg1 arg2
```

Pass arguments to script (@ARGV)

For more information, see: https://perl.petamem.com/docs/eng/

Report bugs at: https://github.com/petamem/pperl-bugtracker/issues

3.9 Module search paths: -I, PERL5LIB, @INC

`-I dir` prepends `dir` to `@INC`, exactly like perl. Entries from `-I` and from the `PERL5LIB` environment variable (`PERLLIB` as the fallback; both ignored under taint mode) are additionally expanded: if the directory has versioned children (`5.44.99`, `5.40.0`, ...), those are added around the entry, nearest perl version first. This makes `local::lib`- and Carton-style trees built by a real perl work unmodified.

Two deliberate differences from perl:

- **No architecture subdirectories.** perl would also add `dir/5.44/x86_64-linux`-style children; pperl never does. It runs no compiled XS objects, so those directories hold nothing it could use, and skipping them avoids loading `.pm` wrappers that would die trying to bootstrap `.so` files.
- **Native modules win over every @INC entry.** Modules built into the pperl binary (see `PPERL_MODS` below) are served directly by `require`; a same-name `.pm` on disk is never loaded in front of them, no matter what `-I` says.

Two pperl-private search locations exist for parallel installations, so a module can be provided to pperl WITHOUT touching the shared perl trees:

- The `PPERL5LIB` environment variable: same syntax and expansion as `PERL5LIB`, honored immediately before it.
- The pperl overlay trees, probed before every host perl tree: `/usr/local/share/pperl/site_perl` (local overrides), `/usr/share/pperl/site_perl` and `/usr/share/pperl/vendor_perl` (packaged), each optionally with versioned children.

A `.pm` in either location outranks all host trees for pperl only - the host perl never looks there. Neither can outrank a module built into the binary.

The rest of `@INC` is built by probing the host's standard library trees (Arch, Debian and Fedora layouts) rather than baking one distribution's paths into the binary; `pperl -V`

prints the real result. Details and rationale: *differences from upstream* and the `@INC` reference in *program input*.

3.10 Environment variables

pperl sets several environment variables inside the process it runs, so scripts can introspect the running interpreter:

Variable	Meaning
`PPERL`	Version string (same as `pperl -v`)
`PPERL_FEATURES`	Comma-separated list of compiled-in Cargo features
`PPERL_MODS`	Comma-separated `Name/version` pairs for every native module
`PPERL_SHELL`	Shell integration flavor (no if `shell` feature off)
`PPERL_STATIC`	`yes` if the binary is musl-statically linked, else no

Run `pperl -V` to see all values for the current binary.

CHAPTER

FOUR

HOW-TO

Two kinds of task-oriented material live in this section. The distinction matters because it tells you how much time to put aside and what you'll walk away with.

Guides are long-form and multi-chapter. You read them the way you read a book - cover to cover, or chapter by chapter with each chapter building on the last. A guide covers a whole subject in depth. The regex guide, for instance, takes you from `m//` through Unicode properties and backtracking performance; the debugging guide covers everything from attaching `-d` to decoding production stack traces.

Tutorials are narrow and single-sitting. One tutorial, one topic, one specific question answered. You land on a tutorial with a task in mind and leave with it done. No larger arc, no required prior reading.

4.1 Guides

PPerl Architecture How pperl is built: the JIT, the FFI, native modules, and the concrete differences from upstream Perl 5.

PPerl Architecture Concurrent Execution Doing work in parallel: auto-parallelization, the ithreads reality, and how to choose between them.

Concurrent Execution Regular Expressions Perl regex from first principles through performance tuning.

Regular expressions Performance Making Perl run fast on pperl: measuring with the right tools, which classic idioms still pay off, and how to shape a loop so the JIT and parallelizer take it.

Performance Object-oriented Programming Modern `class`/`field`/`method` first, classical `bless` for maintainers, migration paths.

Object-oriented programming Debugging Finding and fixing bugs: the `-d` debugger, inspecting live state, production-path tools.

Debugging Perl programs One-Liners `pperl` at the shell: `-e`, implicit loops, progressive recipes, parameterised aliases, quoting gotchas.

The Definitive Perl One-Liner Guide Tying Variables Making a variable dispatch through a class: tied scalars, arrays, hashes, and filehandles, with a complete worked class for each.

Tying variables Coming from another language Perl by analogy for developers fluent in another language: each construct you already know mapped to its Perl coun-

terpart, with the gotcha the similarity hides. Python, PHP, Ruby, Lua, sed, awk, shell and Tcl.

Coming from another language 2D graphics Two-dimensional graphics in Perl, from the pixel up: the drawing algorithms, the raster and vector toolkits, reaching C libraries through pperl's native FFI, and two runnable capstones on SDL2 (a drawing app and a 2D game).

The Definitive Guide to 2D Graphics with Perl 3D graphics Three dimensions: the transform pipeline, meshes and model formats, a software rasteriser you write yourself, and the GPU through pperl's native `Peta::WebGPU` binding.

The Definitive Guide to 3D Graphics with Perl

4.2 Tutorials

Pack & Unpack Binary data layout: directives, endianness, network protocols, file formats.

pack and unpack - a tutorial Working with I/O Streams The full open story: modes, encoding layers, pipes, in-memory handles, error handling.

I/O streams - a tutorial Perl References Scalars, arrays, hashes, code, anonymous and weak references, nested structures.

References - a tutorial Unicode in Perl Text vs bytes, source encoding, PerlIO layers, Unicode regex, common pitfalls.

Unicode - a tutorial Boolean Logic for Perl Programmers Truthiness, all sixteen binary operators, De Morgan, NAND/NOR completeness, bitwise and regex applications.

Boolean Logic for Perl Programmers - a tutorial Interprocess Communication Runnable IPC cookbook: subprocesses, FIFOs, TCP and UDP sockets, socket pairs, System V semaphores and shared memory.

Interprocess communication - a tutorial Installing CPAN Modules `pperl --cpan`: where modules land, how pperl finds them, why the installer needs telling what pperl already has, and what the no-XS rule means in practice.

Installing CPAN modules - a tutorial The Interactive Mode `ipps`, the prompt that runs Perl: values as you would write them, variables that stay, Tab completion, commands to inspect, time and ask the JIT, and a start file for your own macros.

ipps, the interactive mode - a tutorial

4.2.1 PPerl Architecture

How pperl is built, what sets it apart from upstream Perl 5, and the subsystems a working programmer reaches for when they want more than "run this Perl script" - the JIT, the FFI, and the native-module layer.

This guide is descriptive, not a tutorial. Read it to understand *how* pperl does what it does and *why* certain things behave the way they do. Per-function behaviour lives in the *reference tree*.

JIT Compilation

pperl includes a JIT compiler built on Cranelift, the code generator used by Wasmtime. It compiles hot loops and hot recursive subs to native machine code, and hands qualifying loops to a parallelizer that spreads them across CPU cores. Both are automatic: no annotations, no pragmas, no configuration. Individual compiled shapes run 20x to 400x faster than perl5; across the whole project benchmark suite, including everything the JIT does not touch, pperl comes out about twice as fast.

Two properties define the design, and everything else follows from them:

1. **Exactness.** Compiled code produces byte-identical output to the interpreter, always. Not "close enough": the test harness compares pperl's output byte-for-byte against perl5 on every test, with the JIT and parallelizer on.
2. **Decline, never guess.** Anything the compiler cannot prove it can reproduce exactly, it declines. A declined loop runs on the interpreter at normal speed. There is no mode in which compiled code is "probably right".

What compiles

The compiler recognises whole constructs, not individual opcodes. As of mid-2026 the covered surface is:

- **Numeric loops**: `for` over integer ranges, `foreach` over one array, `while`/`until` including `while (@queue)` drain loops; nested loops; `if`/`else`, ternaries, `last`; integer and float arithmetic, comparisons, bitwise operations; `++`/`--` in both void and value positions.
- **Strings**: interpolation and concatenation, `sprintf` (`%d` forms), `length`, `index` with a constant needle, `uc`/`lc`, `chomp`, `reverse`, the `x` repeat, `substr` (read forms), string comparisons.
- **Arrays and hashes**: `$a[$i]` reads and writes, push/pop, list assignment, `scalar(@a)`, `$h{key}` reads and writes with constant or variable keys.
- **List functions**: `map` and `grep` with pure expression blocks, numeric `sort { $a <=> $b }` (and descending), `join` with a constant separator, `split` on a literal pattern.
- **Named sub calls**: calls to subs whose body is a `my (...) = @_;` prelude plus one pure numeric expression, including self-recursion. These compile as real native functions with register argument passing; a recursive call tree runs entirely in native code. Recursion compiles both inside loops and for a single top-level call like `fib(30)`.
- **Regex**: scalar `m//` matches (stored, branched, and capture-list forms) and `s/pat/replacement/` with a constant replacement, including `/g`. These run the real regex engine from compiled code, so `$&`, `$1`, `@-` and friends behave exactly as interpreted; what disappears is the per-operation interpreter dispatch around the engine.
- **Native-module calls**: the `Scalar::Util` operations that Perl compiles to single ops (`blessed`, `reftype`, `refaddr`, `weaken`, `unweaken`, `isweak`) compile directly; a curated set of pure XSUBs (`List::Util` `sum`/`sum0`/`min`/`max`/`product`,

`Scalar::Util` `looks_like_number` and `readonly`) runs from compiled loops as call-outs through the real `entersub`, with the callee re-verified at every loop entry.

The exactness machinery

The compiler attaches through Perl's own extension points: the peephole-optimizer hook discovers candidate loops after Perl's optimizer has finished, and the per-op function pointers of loop-entry and sub-call ops are redirected to trampolines. Everything below is driven from those trampolines.

Guards. At every loop entry the trampoline re-checks the world: variable types, magic and tie-ness, readonly-ness, aliasing between variables, whether a sub was redefined, whether `$/` still holds its default. Any mismatch means the interpreter runs this entry. After repeated failures the trampoline uninstalls itself and the loop costs nothing extra forever after.

Scratch state and abort/replay. Compiled code never writes anything observable until it completes. Scalars live in a private buffer; strings and arrays operate on scratch copies; hash writes are buffered. If anything leaves the provable domain mid-run (integer overflow where Perl would promote, a `substr` outside the string, a missing hash key, a pending signal, recursion depth at a cap), the run aborts: the scratch is discarded and the interpreter replays the whole loop from the start. Since nothing observable happened, the replay is exact, including any warnings Perl would have printed. Completion, by contrast, writes results back through Perl's own setter functions.

Two numeric modes. Loops compile in integer mode (checked 64-bit arithmetic; overflow aborts precisely where Perl promotes to a double) or float mode (IEEE doubles evaluated in Perl's own expression order, which is bit-exact with Perl's NV arithmetic). Values that might be integers in Perl carry exactness checks at the 2^53 boundary. Which mode runs is decided per entry from the actual types, and both variants can coexist for one loop.

Semantics that survive by construction. Perl's boolean false is the dual-natured `""`, which no numeric slot can spell - so stored booleans (comparison results, predicate returns) live in write-only slots that are written back as copies of Perl's own true/false values, exactly what the interpreter's assignment would store. The "Deep recursion" warning is reproduced by capping compiled recursion exactly where the warning would fire and letting the replay warn. Match variables after a loop show what the interpreter would show, because Perl itself restores match state at block exit.

Parallel execution

Loops with the right shape additionally parallelize: an integer range whose per-iteration work is independent except for sum-style reductions is chunked across a work-stealing thread pool. The gate is strict bit-exactness: integer reductions always qualify; float reductions only when reassociation provably cannot change the result. A loop that touches strings, arrays, hashes, or calls subs runs compiled but sequential. Trip-count thresholds keep small loops off the pool.

`--threads=N` sizes the pool; `--parallel-threshold=N` adjusts the minimum trip count; `--no-parallel` disables dispatch entirely.

Observing the JIT

- `--jit-stats` prints counters at exit: candidates found, loops compiled, entries run compiled, guard failures, deopts.
- `--no-jit` disables compilation. Comparing a run against `--no-jit` tells you whether the JIT was engaging at all.
- `PPERL_JIT_LOG=/path/file` appends one line per compiler event: `candidate ok`, `compile ok`, `compile skip reason=body-op:NAME` (the loop contained op NAME the compiler does not handle), `deopt reason=...` (a guard failed or a run aborted). The `reason=` vocabulary is the fastest way to learn why a specific loop is not compiling.

For test authors: the test harness understands a `# harness: jit-engage=compile` pragma that fails a test unless the JIT actually compiled something during it, preventing vacuous passes.

Limitations

- Perl-visible line numbers and `caller`-style introspection inside a compiled region reflect the loop entry, not the current statement. Debuggers and profilers see a compiled loop as one opaque unit.
- Native-module calls outside the compiled set (anything not in the pure allowlist, and any block-taking form like `first { }`) decline the loop.
- `s///e`, scalar `m//g`, patterns with embedded code blocks, ties, overloading, and tainting all decline.
- The first compiled entry of a loop costs roughly a millisecond of compilation; loops below the trip thresholds stay interpreted for that reason.

Numbers

Measured mid-2026 on the project benchmark suite (speedups are perl5 time divided by pperl time):

Shape	Speedup vs perl5
Mandelbrot 1000x1000, parallel	~400x
Sub calls (empty through 3-arg, and recursion)	20x to 60x
`fib(30)` as a single top-level call	~30x
Whole 97-benchmark suite, geometric mean	~2x

The whole-suite number includes benchmarks the JIT does not touch; the per-shape numbers are what compiled code delivers on its own ground.

Auto-FFI (Peta::FFI)

PetaPerl includes a built-in Foreign Function Interface that allows calling C library functions directly from Perl - without writing XS, without a C compiler, without any build step. This is a PetaPerl-exclusive feature with no perl5 equivalent.

Architecture

Any library the system can `dlopen` is reachable from pure Perl, at run time, with nothing compiled. There are two levels, and which one you use depends only on whether someone has already written a binding for the library you need.

- **Raw FFI** - the `Peta::FFI` module itself. Open a library, call any function in it by writing a type signature, declare its C object types as Perl classes, and use the memory and callback primitives that go with that. It works for every library, including one nobody has written a binding for.
- **A native binding** - a module presenting an ordinary Perl API that reaches the C library for you, with the signatures settled in one place and the C objects tied to the lifetime of Perl values. It is built on Peta::FFI's binding layer and is the later, faster and more comfortable form of the same reach. `Peta::FFI::Libc`, `Peta::FFI::UUID`, `Math::GMP` and `Cairo` are among the ones that ship. Where one covers your library, use it.

Raw FFI dispatches through **libffi**, so any signature you can spell works with no code generation and no build step. pperl also has *FFI::Platypus* built in, as a layer over Peta::FFI's engine: each of the two can express everything the other can, and code written for CPAN's FFI::Platypus runs unchanged. New code can use Peta::FFI directly.

Raw FFI

dlopen / call / dlclose

```
use Peta::FFI qw(dlopen call dlclose);

my $lib = dlopen("libm.so.6");
my $result = call($lib, "sqrt", "(d)d", 2.0);   # sqrt(2) = 1.4142..
↪.
print "sqrt(2) = $result\n";
dlclose($lib);
```

`call` parses the signature, looks the symbol up and prepares the call every time it runs, which suits a one-off call. The function may also be given by address instead of by name: a function pointer C handed out, such as an entry of a table of callbacks, has no name to look up.

Prepared functions: function / attach

A function you call more than a few times is better prepared once. `function` does the parsing, the lookup and the preparation once and returns a code reference; each call then pays only for converting its arguments and its result, roughly a quarter to a third of what `call` costs. `attach` does the same and installs the result as a named sub, in the calling package unless the name contains `::`

```
use Peta::FFI;

my $cairo   = Peta::FFI::dlopen("libcairo.so.2");
my $move_to = Peta::FFI::function($cairo, "cairo_move_to", "(odd)v
```

(continues on next page)

(continued from previous page)

```
 ↪");
$move_to->($cr, 10, 20);

package My::Zlib;
my $z = Peta::FFI::dlopen("libz.so.1");
Peta::FFI::attach($z, "zlibVersion", "()p");            #␣
 ↪My::Zlib::zlibVersion
Peta::FFI::attach($z, "crc32", "(Lpu32)L", as => "crc");
print zlibVersion(), "\n";
```

A prepared function keeps its library loaded for as long as the code reference or the sub lives, whatever `dlclose` is called on. A missing symbol or a bad signature croaks when the function is prepared, not on its first call.

Type Signature Format

The signature string `"(arg_types)return_type"` lists one type per position, optionally separated by blanks:

Code	C Type	Perl Mapping
`v`	`void`	return only; 1 is returned
`i`	`int`	integer
`l`	`long`	integer
`L`	`unsigned long` / `size_t`	unsigned integer
`i8 i16 i32 i64`	`int8_t` … `int64_t`	integer
`u8 u16 u32 u64`	`uint8_t` … `uint64_t`	unsigned integer
`b`	`bool`	true or false
`d`	`double`	float
`f`	`float`	float
`D`	`long double`	float (rounded to a double)
`cf cd`	`float` / `double complex`	`[re, im]`; a number is `[n, 0]`
`p`	`const char*`	string in; a returned string is copied
`s`	`char*` the caller frees	return only; copied, then freed
`P`	`void*` (mutable buffer)	output buffer; `\$buf` is written
`o`	opaque pointer	raw address as an integer
`<Class>`	a C object's pointer	an object of a declared handle type
`{Class}`	`struct` by value	an object of a declared record class
`*{Class}`	`struct *` / `union *`	the same object, by pointer
`*T`	`T *`	`\$scalar` in and out; `undef` is NULL
`T[N] T[]`	`T *` to an array	`\@array` in and out

Examples:

- `"(d)d"` - one double argument, returns double (e.g. `sqrt`)
- `"(p)L"` - one string argument, returns unsigned long (e.g. `strlen`)
- `"(ppi)i"` - two strings + int, returns int
- `"()i"` - no arguments, returns int (e.g. `getpid`)

- `"(o)i"` - one opaque handle, returns int (e.g. `sqlite3_close`)
- `"(u64 <My::Pool>)b"` - a 64-bit unsigned integer and a handle, returns bool
- `"(D)D"` - a long double in and out (e.g. `sqrtl`)
- `"(cd)d"` - a double complex in, a double out (e.g. `cabs`, which gives 5 for `[3, 4]`)

An unsigned type reads its argument as an unsigned integer, so `2**64-1` passes as itself rather than as -1, and a return above `2**63` comes back unsigned.

Pointers and arrays. `*T` is a pointer to one value and `T[N]` (a fixed count) or `T[]` (any length) a pointer to an array, for `T` a number type, `o`, `b` or `p` (so `p[]` is a `char **`). You pass a reference, the callee reads and writes through it, and afterwards the scalar or the array holds what C left there; `undef` passes NULL. As a return type, `*T` gives a reference to the value (`undef` for NULL) and an array type gives an array reference.

```
my $m = Peta::FFI::call($libm, "frexp", "(d*i)d", 40, \my $exp);   #␣
↪0.625, $exp = 6
Peta::FFI::call($libc, "qsort", "(i32[]LLo)v", \@v, scalar @v, 4,
↪$cmp);
```

`P` given a reference passes that scalar's own bytes, and after the call the scalar holds what C wrote, at the same length:

```
my $buf = "\0" x 64;
Peta::FFI::call($libc, "snprintf", "(PLp;i)i", \$buf, 64, "%d", 42);
```

Given a string or a number instead, `P` passes a zeroed scratch buffer of that length or that many bytes (at least 16), which nobody reads back.

A **variadic** function lists its named parameters, then a `;`, then the types this particular call passes through its `...`. `"(iL;o)i"` is `ioctl(int, unsigned long, ...)` passing one pointer, and `"(pi;u32)i"` is `open` with a mode.

`s` is for a C function that returns a string the caller must free. Perl gets a copy and the C string is freed at once, with libc's `free()` unless `function` or `attach` names the library's own free function with `free => "name"`. A library not written in C, a Rust one for instance, must name its own.

`o` is the code for every C API built on opaque structs, the `sqlite3 *` and `PGconn *` and `cairo_t *` kind. The address travels as an ordinary Perl integer, the runtime never dereferences it, and `undef` or `0` passes `NULL`. Never reach for `p` to carry such a pointer: `p` means "copy a string until the first NUL byte", which is both wrong and unsafe for anything that is not a string.

Typed handles

`o` leaves the pointer's lifetime to you. `handle_type` instead declares a Perl class for a C object type, together with the function that destroys it, and a signature can then name the type as `<Class>`:

```
Peta::FFI::handle_type("My::Schema", $lib, "schema_free");
my $compile = Peta::FFI::function($lib, "schema_compile", "(p)
↪<My::Schema>");
```

(continues on next page)

(continued from previous page)

```
my $schema = $compile->($schema_json) or die "invalid schema";
...                                   # schema_free runs when
 ↪$schema goes
```

- **As a result**, the returned pointer becomes an object of the class; a NULL pointer becomes `undef`, so a failed constructor reads as false.
- **As an argument**, an object of the class passes its pointer, and `undef` passes NULL. Anything else - another class's handle, a plain number, a released handle - croaks before C sees it.
- When the last reference to an object goes, the destructor runs with its pointer, exactly once. No `DESTROY` method is involved, so the class is free for your own methods and subclasses.

`Peta::FFI::release($schema)` runs the destructor early. The object stays in its class, and passing it to C afterwards croaks.

Records: C structs and unions

`record` declares a Perl class for a C struct, member by member in the order of the C declaration:

```
Peta::FFI::record("My::Point", x => "d", y => "d");
Peta::FFI::record("My::Rect",  origin => "{My::Point}", size => "
 ↪{My::Point}");

my $p = My::Point->new(x => 1.5, y => -2);
print $p->x, "\n";                                  # 1.5
$p->y(4);                                           # set

# struct Point point_add(struct Point, struct Point);
my $sum = Peta::FFI::call($lib, "point_add", "({My::Point}
 ↪{My::Point}){My::Point}", $p, $p);
# void point_scale(struct Point *, double);
Peta::FFI::call($lib, "point_scale", "(*{My::Point}d)v", $p, 2);
```

A member is an integer type, `f`, `d`, `b`, `o`, `p` (a `const char *` the record points to, read-only), a fixed array of numbers or `o` such as `u8[16]`, or another record by value, `{Class}`. The layout, each member's offset and the size and alignment of the whole, is the C compiler's for the platform, as libffi computes it, so you never write an offset.

`Class->new(member => value, ...)` makes a zeroed record and sets the members given. `$r->name` reads a member and `$r->name($value)` sets it; an array member also takes `($index)`, `($index, $value)` and an array reference. A member that is itself a record reads as a copy, an object of its class (`$rect->origin->x`), and is set from one. A record is a reference to the struct's bytes.

In a signature, `{Class}` passes or returns the struct by value, as `"(ii){Div}"` does for libc's `div`. `*{Class}` passes a pointer to the record's own bytes, which the callee may change in place; `undef` is NULL. Callbacks take and return records too: `closure("({My::Point})v", sub { ... })`.

`Peta::FFI::sizeof("{My::Point}")` and `alignof` give a type's size and alignment, for records and every other type.

`union` declares a C union the same way, with every member at offset 0: the union is as large as its largest member and as aligned as its strictest. SDL's event type is one:

```
Peta::FFI::record("Keysym", scancode => "i32", sym => "i32", mod =>
 ↪"u16", unused => "u32");
Peta::FFI::record("KeyEvent", type => "u32", timestamp => "u32",␣
 ↪windowID => "u32",
                  state => "u8", repeat => "u8", pad => "u8[2]",␣
 ↪keysym => "{Keysym}");
Peta::FFI::union("Event", type => "u32", key => "{KeyEvent}",␣
 ↪padding => "u8[56]");

my $ev = Event->new;                          # SDL_Event
while (Peta::FFI::call($sdl, "SDL_PollEvent", "(*{Event})i", $ev)) {
    print $ev->key->keysym->sym, "\n" if $ev->type == 0x300;   #␣
 ↪SDL_KEYDOWN
}
```

A union is passed by pointer, `*{Event}`, and is read and written whole by `fetch` and `store` and as a member of a record. Passing one by value, as `{Event}` or inside a struct passed by value, croaks: libffi has no union type, and a struct of the same size is passed differently on some platforms, which would make a wrong call rather than an error. Real C APIs nearly always pass unions by pointer.

Library variables

`symbol($lib, $name)` gives the address of a function or a variable in a library. `variable` binds a Perl scalar to a library's global variable: reading `$$var` reads the C variable at that moment, and assigning to it writes the variable.

```
my $optind = Peta::FFI::variable($libc, "optind", "i");
print $$optind, "\n";        # reads the C variable
$$optind = 1;                # writes it
```

The type is any that `store` can write; a `p` variable is read-only. The library stays loaded while the scalar lives.

errno

`Peta::FFI::errno()` returns errno as the last C call through Peta::FFI left it, read before Perl could change it. As in C, it means something only when the function reported a failure:

```
my $fd = Peta::FFI::call($libc, "open", "(pi;u32)i", $path, 0x40,␣
 ↪0644);
if ($fd < 0) {
    local $! = Peta::FFI::errno();
    die "open $path: $!";
}
```

Opaque pointers and memory

Most out-parameters need no memory handling of your own: `*o` takes a reference to a scalar the callee writes a pointer into.

```
Peta::FFI::call($lib, "sqlite3_open", "(p*o)i", ":memory:", \my
 ↪$db);
```

For memory that has to outlive a call, or that C hands you by address, there is the pointer toolbox; every address in it is a plain Perl integer, as with `o`. `alloc` and `free` obtain and release a zeroed block. `fetch($addr, $type)` and `store($addr, $type, $value)` read and write a value of any type at an address, records and arrays included:

```
my $n  = Peta::FFI::fetch($ptr, "i32");
my $xy = Peta::FFI::fetch($ptr, "d[2]");           # [1.5, -2]
my $pt = Peta::FFI::fetch($ptr, "{My::Point}");    # a My::Point␣
 ↪copy
```

For raw bytes, `peek` reads a count of them back as a Perl byte string and `peek_cstr` reads a NUL-terminated C string; `poke` writes bytes to an address. `pack_ptr` and `unpack_ptr` convert between an address and its machine-width byte representation.

Callbacks

C can call back into Perl. `closure` wraps a coderef as a real function pointer, using the same signature grammar as `call`, and returns its address for you to pass wherever the C API wants a function. The canonical shape is a comparator or a handler:

```
my $cmp = Peta::FFI::closure('(oo)i', sub {
    my ($pa, $pb) = @_;
    my $a = unpack "l", Peta::FFI::peek($pa, 4);
    my $b = unpack "l", Peta::FFI::peek($pb, 4);
    return $a <=> $b;
});

Peta::FFI::call($libc, "qsort", "(oLLo)v", $buf, $n, 4, $cmp);
Peta::FFI::closure_free($cmp);
```

Three rules go with it. The closure stays alive until `closure_free` is called with that address, so free it only once the C side has been told to stop using it; freeing an address twice is harmless, as `free(NULL)` is. A callback that dies does not propagate, because the error would have to unwind through the C caller's own stack frame: `$@` is set, the callback returns a zero of its declared return type, and you check `$@` after the C call that triggered it. And a closure may be called only on the interpreter's own thread, during a C call Perl made: a comparator, a progress handler, a visitor. A library that calls it from a thread of its own stops the program with a message saying so, because Perl cannot run there.

Events from a library's own threads

A library that produces events on threads of its own, such as a device monitor or a netlink listener, queues them instead and signals through a file descriptor that some are waiting. *Peta::FFI::EventSource* watches that descriptor and takes the events on Perl's thread. The library provides a function returning the descriptor, one returning the next event as an `s` string (NULL when the queue is empty), and a destructor declared with `handle_type` that stops its thread:

```
use Peta::FFI;
use Peta::FFI::EventSource;

Peta::FFI::handle_type('Example::Monitor', $lib, 'example_monitor_
↪free');
my $open = Peta::FFI::function($lib, 'example_monitor_open', '(p)
↪<Example::Monitor>');
my $fd   = Peta::FFI::function($lib, 'example_monitor_fd',   '(
↪<Example::Monitor>)i32');
my $next = Peta::FFI::function($lib, 'example_monitor_next', '(
↪<Example::Monitor>)s',
                               free => 'example_string_free');

my $source = Peta::FFI::EventSource->new($open->('block'), fd =>
↪$fd, next => $next);
my $guard  = $source->watch(sub { handle_event($_) for @_ });   #␣
↪AnyEvent
```

`->watch` uses AnyEvent. For EV or `select` there are `->fd` and `->drain`, and for loops that take filehandles (Mojo::IOLoop, IO::Select) there is `->fh`.

Library Discovery

```
use Peta::FFI qw(scan);

my $libs = scan();
# Returns hashref: { "libz.so.1" => "/usr/lib/libz.so.1", ... }

my $ours = scan("/opt/example/lib");
my $lib  = Peta::FFI::dlopen($ours->{"libexample.so"});
```

`scan` maps each library's file name to its path: every `lib*.so` and `lib*.so.*` file in the directories searched that this program can load. Linker scripts (`/usr/lib/libc.so` is one) and libraries of another architecture, such as 32-bit ones on a 64-bit system, are left out. A name found in more than one directory maps to the first one's file.

`scan(@dirs)` searches those directories, in that order, and no others. `scan()` searches the directories in `PETA_FFI_PATH` (colon-separated, like `PATH`) first, then where the dynamic linker looks for a bare name: `LD_LIBRARY_PATH`, the directories `/etc/ld.so.conf` names, and `/lib`, `/usr/lib` and their 64-bit forms.

Nothing else searches. `dlopen` with a bare name asks the dynamic linker, which never reads `PETA_FFI_PATH`; a library installed in a directory of its own is found by `scan` and

opened by the path it returns.

Native bindings

A native binding gives you a Perl API for one C library, with argument validation, return-value conversion, and error handling done for you. No type signatures, no addresses: you call functions and methods. Each C function's exact type is declared in the binding and called directly, which is faster than a prepared Peta::FFI function; the C objects free themselves when the Perl values holding them go away. A module loads even when its library is not installed, and the first call that needs the library dies with an install hint. A library too old to have a function the binding calls makes that call die, naming the C function.

Peta::FFI::Libc

Direct bindings to libc functions, grouped by category:

Process: `getpid`, `getppid`, `getuid`, `getgid`, `geteuid`, `getegid`

Strings: `strlen`, `strerror`

Environment: `getenv`, `setenv`, `unsetenv`

Math: `abs`, `labs`

System: `sleep`, `usleep`, `gethostname`, `uname`, `time`, `clock`

File operations: `access`, `unlink`, `rmdir`, `mkdir`, `chmod`, `chown`, plus the access-mode constants `F_OK`, `R_OK`, `W_OK`, `X_OK`

```
use Peta::FFI::Libc qw(getpid gethostname uname);

print "PID: ", getpid(), "\n";
print "Host: ", gethostname(), "\n";

my @info = uname();
print "OS: $info[0] $info[2] ($info[4])\n";
```

Peta::FFI::UUID

Bindings to libuuid (must be installed on the system):

```
use Peta::FFI::UUID qw(uuid_generate uuid_generate_random);

my $uuid = uuid_generate();           # e.g. "550e8400-e29b-41d4-
 ↪a716-446655440000"
my $rand = uuid_generate_random();    # random-based UUID
```

Functions: `uuid_generate`, `uuid_generate_random`, `uuid_generate_time`, `uuid_parse`, `uuid_is_null`, `uuid_unparse`.

If libuuid is not installed, `use Peta::FFI::UUID` dies with a diagnostic message.

Math::GMP

Arbitrary-precision integer arithmetic via libgmp, under the upstream `Math::GMP` name, so code written for the CPAN module runs unchanged:

```
use Math::GMP;

my $big = Math::GMP->new("123456789012345678901234567890");
my $sum = $big + 42;
print "$sum\n";
```

GMP integers are heap-allocated mpz_t values stored as blessed references. Operator overloading (`+`, `-`, `*`, `/`, `%`, `**`, `<=>`, `""`) works identically to the XS module.

If libgmp is not installed, `use Math::GMP` dies with a diagnostic message.

Cairo

The 2D vector graphics library, under the upstream `Cairo` name and with the upstream method signatures, so code written for CPAN `Cairo` runs unchanged with no CPAN distribution installed. It is the largest native binding that ships and the one to read if you want to see the pattern at full size: `Cairo::ImageSurface`, `Cairo::Context`, `Cairo::Path`, `Cairo::Pattern`, `Cairo::Matrix`, `Cairo::Font` and `Cairo::Region` each have their own reference page under `Cairo`.

```
use Cairo;

my $surface = Cairo::ImageSurface->create('argb32', 200, 200);
my $cr      = Cairo::Context->create($surface);
$cr->set_source_rgb(0, 0, 0);
$cr->arc(100, 100, 40, 0, 6.28318);
$cr->fill;
$surface->write_to_png('out.png');
```

If libcairo is not installed, the call site croaks with an install hint rather than the module failing to load. The *2D graphics guide* walks through how this binding is put together.

What raw FFI cannot do

Derive types from a header. Nothing reads your `.h` files. You read the C prototype and write the matching signature yourself, and you declare each record's and union's members yourself, in the order and with the types of the C declaration. A signature or a record that disagrees with the real declaration is undefined behaviour rather than an error you get told about.

When to Use FFI vs Native Modules

Scenario	Recommendation
Common CPAN module (List::Util, etc.)	The built-in native module
A library that a native binding covers	That binding (`Math::GMP`, `Cairo`, ...)
A system library that nothing wraps yet	Raw FFI
Frequent calls to the same function	`function` or `attach`
One-off experiment	`call`

`call` prepares the call every time it runs. A prepared function from `function` or `attach` pays only for converting its arguments and its result, roughly a quarter to a third of what `call` costs. A native binding calls the C function directly and is faster again; `Cairo`'s methods take less time per call than perl5's XS `Cairo`. Native modules such as List::Util, which implement the work itself, are fastest, at the speed of a built-in operator.

Native Modules

PetaPerl does not support XS (C extensions). Instead, performance-critical CPAN modules are reimplemented natively in Rust. These "native modules" provide the same Perl-level API as their XS counterparts but integrate directly with PetaPerl's runtime - enabling JIT compilation, auto-parallelization, and zero FFI overhead.

How It Works

When you write `use List::Util qw(sum min max)`, PetaPerl's module loader detects that `List::Util` has a native implementation and registers Rust function pointers directly. There is no compilation step, no `.so` loading, and no XS glue code.

Native functions are dispatched via `NativeFn` - a direct function pointer with O(1) call overhead, identical to built-in operators.

Available Native Modules

Core Utilities

Module	Functions	Notes
Scalar::Uti	blessed, reftype, refaddr, weaken, isweak, looks_like_number, ...	Full API
List::Util	sum, min, max, first, any, all, none, reduce, ...	MULTICALL optimized
Sub::Util	subname, set_subname	
Hash::Util	lock_keys, lock_hash, ...	

Digest / Crypto

Module	Functions	Notes
Digest::MD5	md5, md5_hex, md5_base64, OO interface	Full API
Digest::SHA	sha1, sha256, sha512, OO interface	Full API
MIME::Base64	encode_base64, decode_base64	
MIME::QuotedPrint	encode_qp, decode_qp	

File / System

Module	Functions	Notes
File::Basename	basename, dirname, fileparse	
File::Copy	copy, move	
File::Find	find, finddepth	
File::Glob	bsd_glob	
File::Path	make_path, remove_tree	
File::Spec	catdir, catfile, rel2abs, ...	
File::Temp	tempfile, tempdir	OO + functional
File::stat	stat (OO)	
Cwd	cwd, getcwd, abs_path	
Sys::Hostname	hostname	

I/O

Module	Functions	Notes
IO::File	OO file handle	
IO::Handle	OO handle base	
IO::Dir	OO directory handle	
IO::Pipe	OO pipe handle	
IO::Select	select wrapper	
IO::Socket	OO socket handle	
IO::Seekable	seek/tell mixin	
FileHandle	Legacy OO handle	
Socket	socket primitives	

Data / Encoding

Module	Functions	Notes
Data::Dumper	Dumper	
Storable	freeze, thaw, nstore, retrieve	
Encode	encode, decode, find_encoding	
JSON::PP	encode_json, decode_json	Via Pure Perl

Numeric / Math

Module	Functions	Notes
POSIX	floor, ceil, fmod, strtod, strftime, ...	Subset
Math::GMP	Arbitrary precision integers	Via Peta::FFI::GMP

Build / Config

Module	Functions	Notes
Config	%Config hash	Build configuration
Fcntl	O_RDONLY, O_WRONLY, ...	Constants auto-loaded
Errno	ENOENT, EACCES, ...	Constants auto-loaded

Introspection

Module	Functions	Notes
B	Compiler backend introspection	Minimal
PadWalker	peek_my, peek_our	
mro	get_linear_isa, set_mro	
version	Version object handling	Full OO API

Performance

Native modules run at the same speed as built-in operators since they share the same dispatch mechanism. For block-taking functions (first, any, all, reduce), PetaPerl uses MULTICALL optimization to avoid per-element subroutine call overhead.

Benchmarks (vs perl5 with XS):

Function	PetaPerl native	perl5 XS	Ratio
List::Util::sum	1.7x faster	baseline	
List::Util::min/max	2.9x faster	baseline	
List::Util::first	~1x	baseline	MULTICALL parity

Adding Native Modules

Native module documentation is extracted from the module's Rust source (`//!` module docs and `///` function docs) by `docs/sphinx/bin/20-reference` during the docs build.

Differences from Perl 5

PetaPerl targets full behavioral compatibility with Perl 5.44: same syntax, same pragmas, same runtime semantics, byte-identical output. The test harness compares every test's output against a real perl binary, so "works" below means "verified byte-exact". The differences that remain are deliberate design decisions, catalogued here.

Every claim on this page is re-verified against the current runtime when the page changes; historic limitations that no longer exist are removed rather than annotated.

Intentional Differences

No XS binary loading

PetaPerl never loads compiled extension objects (`.so` files). Instead, the three things XS is used for are served separately:

- **Speed**: the core XS module population (List::Util, Scalar::Util, Data::Dumper, Digest::*, Encode, Storable, POSIX, ...) ships as native modules inside the binary, and the JIT with auto-parallelization makes pure Perl itself competitive.
- **C libraries**: `Peta::FFI` calls into system libraries without any compilation (see the Cairo binding for a complete example).
- **Pure Perl fallbacks**: dists that ship one (most dual-life modules do) simply use it, on a faster interpreter.

A dist that strictly requires its own compiled XS object and offers no fallback will not run. `cpan install Some::XS::Module` is not a supported path and never will be.

Module loading and `@INC`

Three deliberate deviations, all consequences of pperl being one self-contained binary that runs no compiled XS objects:

- **Native modules have absolute priority.** A module built into the binary is served by `require` directly; a same-name `.pm` on disk can never shadow it, regardless of `-I` or `PERL5LIB`. Everything not built in resolves through `@INC` in normal first-hit-wins order.
- **Only pperl's own architecture directories.** pperl's archname is `x86_64-linux`, and the architecture directories of the trees you add are searched under that name, as perl searches its own: `<entry>/x86_64-linux` and `<entry>/<version>/x86_64-linux` of each `-I`, `PERL5LIB` and `PPERL5LIB` entry and of `~/.pperl/lib/perl5`, ahead of the entry. A host perl's own architecture children (`x86_64-linux-thread-multi` and the like) and its `archlib` are not searched: they hold that perl's compiled extensions. No `.so` is ever loaded from a searched directory either; a module whose compiled part is not built in dies with `Can't locate loadable object`, the message pure-Perl fallbacks test for. That is what lets library trees built by a real perl (`local::lib`, Carton) work unmodified.
- **Host library trees are probed, not baked.** `@INC` is constructed by checking the standard Arch, Debian and Fedora pure-perl tree locations for existence, and versioned children of those trees and of `-I`/`PERL5LIB` entries are ordered by nearness to pperl's own version: exact match first, then the same minor release descending,

then newer versions, then the plain directory, then older versions. A host perl of a different version is used gracefully as a last resort rather than rejected. `pperl -V` always shows the actual probed result.

Additionally, pperl-private search locations exist for parallel installations: the `PPERL5LIB` environment variable (honored before `PERL5LIB`, same rules), the per-user tree `pperl --cpan` installs into (`lib/perl5` under `$PPERL_HOME`, default `~/.pperl`) and the overlay trees under `/usr/local/share/pperl` and `/usr/share/pperl`, probed before every host tree. They allow shipping or overriding a `.pm` for pperl without touching the shared perl trees; the host perl never reads them, and nothing in them can outrank a built-in native module.

No interpreter threads

`use threads` is not supported (pperl is a non-threaded build, as are most distribution perls' recommended configurations). Perl-level concurrency options are `fork` and the automatic parallelization of qualifying loops, which uses native threads internally without exposing interpreter threading semantics.

Linux only

PetaPerl runs on Linux exclusively (any architecture supported by Cranelift: x86-64, AArch64, RISC-V, s390x). `$^O` is always `"linux"`.

Version identity

`$]` reports `5.044099` and `$^V` reports `v5.44.99`: a 5.44-class interpreter, above any released 5.44.x patch level. `use v5.44` and version checks against 5.44 or below behave exactly as on a real 5.44. `$Config{version}` reports `5.44.99`.

Modern features are NOT enabled by default: exactly like perl, `say`, `state` or signatures need `use feature ...` or a `use v5.XX` declaration. Scripts written for older perls run with their old semantics.

Command-line flags

Not supported:

Flag	Description
`-x`	Extract script from message
`-C`	Unicode handling of input and output

`-I`, `-M`/`-m`, `-n`/`-p`, `-l`, `-a`/`-F`, `-0`, `-i`, `-T`/`-t`, `-c`, `-d`, `-w`, `-e`/`-E` and `-V`/`-V:key` work as in perl. The one debugger variant that does not is `-dt`: it asks for thread support, and pperl is built without threads. pperl-specific flags (JIT, parallelism) are listed in *CLI options*.

Known Issues

Current, specific, and expected to shrink:

- **`charnames` emits redefinition warnings.** Loading `charnames` (or `\N{...}` names) prints `Subroutine re::is_regexp redefined` warnings before working correctly: the built-in `re` module pre-registers helpers that `_charnames.pm` also defines. Output is otherwise exact.
- **`B::Deparse` on subs with signatures.** The optree for signature parameters uses a newer internal representation than the deparser expects; deparsing such subs produces an "unexpected OP" note instead of the signature. Ordinary subs deparse exactly.
- **Embedding pperl in C hosts** (mod_perl-class, programs linking libperl) is out of scope until a concrete user appears.

4.2.2 Concurrent Execution

Doing work in parallel, or several things at once. There are three concurrency stories to know about, and they answer different questions:

- **Auto-parallelization** - `pperl`'s runtime detects parallelisable patterns in ordinary Perl and spreads them across cores without any user-level threading. This is the preferred answer for CPU-bound loops over large data.
- **Coroutines and event loops** - Coro, EV, Event and AnyEvent: many tasks that mostly wait - on sockets, timers, child processes - each written as straight-line code, taking turns in one interpreter. The answer for I/O-bound concurrency.
- **Classical ithreads (not implemented)** - Perl 5's `threads` and `threads::shared` model. pperl deliberately does not provide it: `use threads` fails at compile time. The chapter teaches the model so that you can read existing code, and most programs that used ithreads are better rewritten against the auto-parallel path.

Start with the *decision chapter* if you have an existing threaded program and are deciding how to port it; start with *parallel* if you are writing new code and want to understand when pperl speeds your loop up, and with *coroutines* if your program spends its time waiting.

Parallel Execution

PetaPerl automatically parallelizes eligible loops using Rayon, a work-stealing thread pool. Combined with JIT compilation, this enables dramatic speedups for compute-heavy workloads.

How It Works

When PetaPerl encounters a parallelizable loop, it:

1. **Analyzes the loop body** for side effects and shared mutable state
2. **Identifies reduction variables** (accumulators like `$sum += ...`)
3. **Distributes iterations** across threads using Rayon's work-stealing scheduler
4. **Combines results** using the detected reduction operations

Each thread gets its own copy of loop-local variables. Reduction variables are combined after all threads complete.

What Gets Parallelized

Parallel dispatch applies to JIT-compiled loops whose iterations are provably independent except for sum-style reductions:

- `for` over an integer range - the primary parallel shape. The range is chunked across threads; each thread accumulates private reduction copies that merge at the end.
- `while` loops whose condition is a counter bounded by a constant or read-only variable, with the counter incremented exactly once per iteration.

One gate is stricter than the side-effect analysis and deserves its own sentence: **bit-exactness**. Integer reductions always qualify. Floating-point reductions qualify only when reassociating the sum provably cannot change the result (for example when every contribution is integral and within the exact range of a double). An inexact float reduction runs compiled but sequential - pperl never trades determinism for cores.

Loops that touch strings, arrays, hashes, or call subs compile to native code but run on one core; see *Writing fast pperl* for how to split such loops.

CLI Control

```
# Default: parallelization enabled
pperl script.pl

# Disable parallelization
pperl --no-parallel script.pl

# Explicitly enable (default)
pperl --parallel script.pl

# Set thread count (default: number of CPU cores)
pperl --threads=4 script.pl

# Set minimum collection size for parallelization
pperl --parallel-threshold=1000 script.pl
```

The test harness runs with `--no-parallel` by default to ensure deterministic test output.

Performance

Mandelbrot Set (1000x1000, mid-2026)

Mode	Time	vs perl5
perl5	~12,500ms	1.0x
pperl JIT (sequential)	~160ms	~75x faster
pperl JIT + parallel (8 threads)	~30ms	~400x faster

Scaling

The work-stealing scheduler provides near-linear scaling for embarrassingly parallel workloads:

Threads	Mandelbrot 4000x4000	Scaling
1	baseline	1.0x
2	~50% time	~1.9x
4	~25% time	~3.8x
8	~13% time	~5.2x

Scaling is sub-linear due to memory bandwidth, cache effects, and reduction overhead.

Limitations

String Operations

String operations (`.=` concat, string building) are not parallelized. The JIT's string support uses extern calls back to the Rust runtime, which requires mutable access to shared state. When the JIT detects string variables in a loop, parallel dispatch is disabled.

Side-Effect Detection

The parallelization analyzer is conservative. Any of these disqualify a loop:

- I/O operations (`print`, open, file reads)
- Global variable writes
- Subroutine calls (unless proven pure)
- Regex operations with side effects (`s///`)

False negatives (missed parallelization opportunities) are safe - the loop simply runs sequentially. False positives (incorrect parallelization) would be bugs.

Determinism

Parallel execution may change the order of side effects. For this reason, parallelization is only applied when the analysis proves the loop body is free of observable side effects.

`map` and `grep` compile to sequential native loops (array state gates parallel dispatch off), so their output order is trivially the sequential one.

How Reduction Detection Works

The analyzer identifies reduction variables by scanning for accumulation patterns and subtracting reset patterns:

```
# Detected as reduction: $sum accumulates, never reset in loop
my $sum = 0;
for my $x (@data) {
```

(continues on next page)

(continued from previous page)

```
    $sum += $x;
}

# NOT a reduction: $temp is reset each iteration
for my $x (@data) {
    my $temp = $x * 2;  # reset (my declaration)
    $sum += $temp;       # $sum is still a reduction
}
```

The formula: `reductions = accumulations - resets`. This prevents false positives where a variable is both accumulated and reset within the loop body.

Coroutines and event loops

A **coroutine** is a thread of execution inside one interpreter: it has its own call stack, its own `$_` and `@_`, its own `local` values, and it runs until it chooses to let another one run. Perl's coroutine module is Coro; together with an event loop it lets a program wait on many sockets, timers and child processes at once while each task is written as plain straight-line code.

pperl runs Coro natively, with the event loops it is used with: EV, Event and AnyEvent, the layer that lets code run on any of them. Coro, EV and Event are part of the **native-max** module set; in a build without it, `use Coro` fails with `Can't locate Coro.pm in @INC`. AnyEvent, with its own pure-Perl loop, is in every standard build.

When to use them

Coroutines are cooperative: one runs at a time, on one CPU, and a switch happens only where the code asks for one. That makes them the right tool for work that mostly *waits* - network clients and servers, pipelines of I/O stages, many timers - and the wrong one for work that computes. For compute-bound loops use *auto-parallelisation*; for isolated subprocesses use `fork`.

Because nothing is preemptive, there are no data races: a variable cannot change under you between two statements unless one of them switches. There are no locks to take for ordinary data either, only coordination primitives for waiting on each other.

Starting and switching

`async` starts a coroutine; `cede` lets the others run; `join` waits for one to finish and returns its result:

```
use Coro;

my @workers = map {
    my $n = $_;
    async {
        for my $step (1 .. 3) {
            print "worker $n, step $step\n";
            cede;                          # let the others take a turn
```

(continues on next page)

(continued from previous page)

```
        }
        return $n * 10;
    };
} 1 .. 2;

print "result: ", $_->join, "\n" for @workers;
```

The two workers alternate step by step. The main program is itself a coroutine (`$Coro::main`); `$Coro::current` is always the one running. A coroutine ends when its block returns, with `terminate`, or when another one calls `$coro->cancel`. An uncaught `die` inside a coroutine ends the whole program, as it would in the main program.

`$coro->prio` sets a priority from `Coro::PRIO_MIN` to `Coro::PRIO_MAX` (default `PRIO_NORMAL`); higher priorities always run first.

Waiting for each other

Coro's modules cover the usual coordination patterns:

- `Coro::Channel` - a queue: `put` adds, `get` waits until there is something to take;
- `Coro::Semaphore` - a counting semaphore, with down/up and a scope-bound `guard`;
- `Coro::Signal` - wait until somebody sends a signal;
- rouse callbacks - turn any callback into a wait: `Coro::rouse_cb` makes a callback, `Coro::rouse_wait` blocks until it is called and returns its arguments.

A producer and a consumer connected by a bounded channel:

```
use Coro;
use Coro::Channel;

my $queue = Coro::Channel->new(4);      # holds at most 4 items

my $producer = async {
    $queue->put($_) for 1 .. 10;
    $queue->put(undef);                 # end marker
};

while (defined(my $item = $queue->get)) {
    print "got $item\n";
}
```

Blocking on I/O with an event loop

Coroutines pay off when a wait on a socket or a timer blocks only the coroutine that waits, not the program. That needs an event loop, and `Coro::AnyEvent` connects Coro to AnyEvent: whenever every coroutine is waiting, the loop runs until something is ready.

```
use Coro;
use Coro::AnyEvent;
use AnyEvent;

my @tasks = map {
    my ($name, $delay) = @$_;
    async {
        Coro::AnyEvent::sleep $delay;   # only this coroutine sleeps
        print "$name done after ${delay}s\n";
    };
} ["slow", 0.3], ["fast", 0.1];

$_->join for @tasks;
```

The same holds for files and sockets: `Coro::Handle` wraps a filehandle so that `readline`, `read` and `print` wait in the calling coroutine only, and `Coro::Socket` makes such handles for network connections. A server then handles each client in a coroutine of its own, written as if it were the only one.

Event loops

AnyEvent chooses its loop the first time it needs one. A loop the program has already loaded wins; otherwise it loads the best one available, which in a native-max build is `EV`, and falls back to its own pure-Perl loop elsewhere. Code written against AnyEvent, and Coro code using `Coro::AnyEvent`, `Coro::Handle` or `Coro::Socket`, runs unchanged on any of them.

EV and Event can also be used directly, with callbacks:

```
use EV;

my $count = 0;
my $tick = EV::timer 0.1, 0.1, sub {    # after 0.1s, then every 0.
↪1s
    print "tick ", ++$count, "\n";
    EV::break if $count == 3;
};

EV::run;                                # until EV::break
```

A watcher runs as long as the object `EV::timer` returned is alive; dropping the last reference stops it.

With coroutines, `Coro::EV` makes EV's loop and Coro's scheduler take turns, and adds waits of its own:

```
use Coro;
use Coro::EV;

async {
    Coro::EV::timer_once 0.2;           # this coroutine sleeps 0.2s
```

(continues on next page)

(continued from previous page)

```
    print "woke up\n";
};

EV::run;                                # returns once nothing is␣
↪left to wait for
```

`Coro::Event` does the same for Event, whose watchers block the coroutine that waits on them:

```
use Coro;
use Coro::Event;

async { Event::loop() };                # the event loop, in a␣
↪coroutine of its own

my $timer = Coro::Event->timer(interval => 0.1);
for my $tick (1 .. 3) {
    $timer->next;                       # waits for the next tick
    print "tick $tick\n";
}
```

A Coro::Event watcher starts when something first waits on it, so the loop has to run alongside the code that waits, not before it.

What each coroutine has of its own

Each coroutine has its own:

- call stack and lexical variables (a sub can be running in several coroutines at once, each with its own `my` variables);
- `$_`, `@_`, `$@`, `$/` and the current regex match (`$1` and friends);
- `local` values - a `local` in one coroutine is not seen by the others;
- `$SIG{__DIE__}` and `$SIG{__WARN__}`, and `%^H`.

Everything else is shared: package variables, `$,` and `$\`, open filehandles, and `$a` and `$b` - so two coroutines must not switch in the middle of a `sort` block that uses them in the same package.

C stacks

Most switches cost no memory beyond the coroutine's Perl-level stacks. A coroutine that switches away from *inside* a callback that the interpreter itself is running - a `sort` block, a tie method, a `DESTROY`, an overloaded operator - keeps a C stack of its own until it returns from there. Those stacks are 128 KiB each by default (`Coro::State::cctx_stacksize` sets the size, in pointers); code that recurses deeply inside such a callback, or through XS-level callbacks, may need more.

Porting from ithreads

A thread per connection, or a work crew fed from a `Thread::Queue`, becomes a coroutine per connection, or coroutines fed from a `Coro::Channel`, with blocking I/O through `Coro::Handle` or `Coro::AnyEvent`. The `:shared` attributes and every `lock` go away: nothing else runs between two statements of a coroutine unless one of them switches. If the threads were there to use several CPUs for computation, coroutines are not the replacement - see *Alternatives*.

See also

- Coro and its modules - the full interface; pperl runs Coro 6.57's own `.pm` files unchanged
- `Coro::State` - the engine under Coro
- `EV` and `Event` - the event loops, and `Coro::EV`, `Coro::Event` to run coroutines on them
- AnyEvent - the event-loop abstraction most Coro programs use
- `Guard` - run code when a scope ends, which Coro builds on
- *Alternatives* - choosing between coroutines, auto-parallelisation and `fork`

Threads

Perl 5's headline concurrency story is **interpreter threads** - *ithreads* for short - supplied by the core `threads` and `threads::shared` modules. Each ithread is a separate Perl interpreter running inside the same OS process, with its own pad, its own lexicals, and a copy of every package variable captured at spawn time. Data is shared between threads only when you opt in by attaching `:shared` to it.

This chapter teaches that model. The idiom, the vocabulary, and the pitfalls are the ones every Perl programmer who has touched concurrency has encountered, and you need to recognise them when reading existing Perl code.

pperl status - read this first

pperl **does not implement ithreads**. The `threads` module is not available, and without it the sharing primitives have no second interpreter to reach. At runtime:

- `use threads` fails at compile time.
- `threads::shared` is not part of pperl. Where a system perl5 provides it on `@INC` it loads, and `share` and `is_shared` do nothing.
- The `lock` built-in is a silent no-op, exactly as upstream Perl behaves when built without thread support. Code that sprinkles `lock` defensively for single-threaded compatibility runs unchanged.

That decision is deliberate. ithreads impose a per-interpreter data copy, constrain every core module's internals, and historically bred a long tail of subtle bugs around shared aggregates, signal delivery, and library thread-safety. pperl invests instead

in **runtime-level parallelism for pure Perl code** - JIT-compiled loop bodies dispatched across a Rayon work-stealing pool, with automatic reduction analysis and zero user-visible thread objects. See *Parallel Execution* for how that works and when it fires.

If you are porting an existing ithreaded program to pperl, read *Alternatives to ithreads* for the shapes that map cleanly - almost every `threads`-based program falls into one of three patterns, each with a simpler and faster pperl equivalent.

How this chapter is organised

ithreads basics

This chapter describes the perl5 ithreads model as it exists in upstream Perl 5.44. It is the vocabulary you need to read existing threaded Perl code. pperl does not run this code - see the landing page for the reason and *Alternatives* for the pperl-native equivalents.

What is a thread

A thread is a flow of control through a program with a single execution point. Every process has at least one thread - the *main* thread - and Perl's `threads` module lets a program spawn additional threads that run inside the same OS process.

Each ithread has **its own Perl interpreter**. When a new thread is created, the entire data state of the spawning thread is copied into the new one, like a fork but within the same process. After that moment the two interpreters are independent: modifying a variable in one does not affect the other, unless the variable is explicitly shared through `threads::shared`.

This distinguishes Perl ithreads from POSIX threads, Java threads, Win32 threads, and most other threading systems you may have met. Shared-everything is the default in those systems; shared-nothing is the default in Perl.

Checking for thread support

Thread support is a Perl build-time option. A program that needs threads should fail early if the interpreter was built without it:

```
use Config;
$Config{useithreads}
    or die "Recompile Perl with threads to run this program.\n";
```

Under pperl, `$Config{useithreads}` is **false** and the `threads` module does not load. The defensive `die` fires, which is the correct outcome - better than silently running the single-threaded code path on a program that needed concurrency.

Creating a thread

Load the `threads` module and call `threads->create`, passing a code reference to the subroutine the new thread should run:

```
use threads;
```

(continues on next page)

(continued from previous page)

```
my $thr = threads->create(\&worker);

sub worker {
    print "in the thread\n";
}
```

`create` takes a code reference - or an anonymous sub - and optional arguments passed to the subroutine. Control then continues in both the caller and the new thread. `new` is a synonym for `create`.

Passing arguments:

```
my $thr = threads->create(\&worker, 'first', 'second', 42);

sub worker {
    my @args = @_;
    # ...
}
```

Each argument is *copied* into the new thread. Modifications inside the thread are invisible to the caller, because the thread has its own interpreter state.

Joining a thread

A thread's subroutine can return a value. To wait for the thread to finish and collect its return, call `join`:

```
use threads;

my ($thr) = threads->create(\&worker);
my @result = $thr->join;
print "thread returned @result\n";

sub worker { return ('ok', 2, 'done'); }
```

`join` blocks until the thread exits, collects the return value, and performs the OS-level cleanup for the thread. The list-context assignment `my ($thr) = ...` is the idiom for spawning a thread whose return will be consumed as a list.

If several threads are outstanding, join them in turn:

```
my @thr = map { threads->create(\&worker, $_) } 1..4;
my @results = map { $_->join } @thr;
```

Detaching a thread

A *detached* thread runs until it finishes, then is cleaned up automatically. Its return value is discarded. Use `detach` when you do not care about the return and do not want to track the thread object:

```
my $thr = threads->create(\&background_work);
$thr->detach;
```

Once detached, a thread cannot be joined. A thread may also detach itself from inside its own body:

```
sub background_work {
    threads->detach;
    # ... long-running work ...
}
```

Process and thread termination

`die` or `exit` in **any** thread terminates the whole process - every other thread stops mid-execution, with no chance to finish. Perl also calls `exit` implicitly when the main thread falls off the end of the program, even if other threads are still running.

The canonical stumbling block:

```
use threads;

my $thr1 = threads->create(\&thrsub, 'one');
my $thr2 = threads->create(\&thrsub, 'two');

sub thrsub {
    my ($label) = @_;
    sleep 1;
    print "thread $label\n";
}
# Falls off the end. Perl warns "Perl exited with active threads:
# 2 running and unjoined" and aborts both threads.
```

Fix by joining before the main thread exits:

```
$thr1->join;
$thr2->join;
```

Three structural patterns

Most threaded programs fall into one of three shapes.

Boss/worker

A single *boss* thread collects or generates tasks and hands them to *worker* threads. The boss does little computation itself - its job is dispatch. Typical in GUI and server programs, where the main thread must stay responsive to events.

Work crew

Several threads run the same subroutine on different pieces of data. Closely maps to data-parallel problems: render tiles of an image, hash slices of a file, fold a large array. This is the pattern pperl's auto-parallelisation targets - see *Alternatives*.

Pipeline

Threads form a chain. Each one receives data, performs one step of processing, and forwards the result to the next. Handy for prime sieves and stream-processing workloads where each stage has its own state.

The three patterns are not exclusive; a non-trivial program uses several of them for different parts.

Thread identity

Every thread has a numeric **thread ID** (TID). The main thread has TID 0; each subsequent thread gets the next integer. From inside a thread, ask for its own object with `threads->self` and its own TID with `threads->tid`:

```
my $me = threads->self;
my $id = threads->tid;
```

`threads->list` returns every non-detached thread currently running. A common clean-up idiom at end of main:

```
foreach my $thr (threads->list) {
    $thr->join;
}
```

yield - usually a no-op

```
threads->yield;
```

Hints to the OS scheduler that this thread is willing to give up the CPU. It is a hint only. On most modern operating systems `yield` is a no-op, and well-behaved code does not depend on it.

pperl status

Every snippet on this page fails at compile time under pperl because `use threads` fails. The cleanest migration path is not to translate the snippets but to identify which of the three structural patterns the original uses, then follow the matching section of *Alternatives*.

Thread-safety of modules

An upstream Perl note worth repeating: a module is **not** thread-safe unless its documentation says so. That caveat applies to the ithreaded program's code, to every CPAN module it uses, and to the C libraries behind any XS modules.

pperl's auto-parallelisation sidesteps this problem entirely: the parallel dispatch is opt-in per loop, the analyser refuses to parallelise any body with side effects, and there is no user-visible shared heap to serialise. See *Parallel Execution*.

See also

- `lock` - the synchronisation primitive, unchanged in syntax under pperl but a no-op at runtime
- *Shared data* - `threads::shared`, semaphores, queues
- *Alternatives* - pperl-native concurrency patterns
- `fork` - process-level concurrency

Shared data

Under upstream Perl 5.44, sharing data between ithreads is explicit: variables are private by default, and become visible across threads only when the `:shared` attribute is applied or a value is passed through a thread-safe primitive. This chapter walks through the full upstream shared-data surface - `threads::shared`, `lock`, semaphores, queues, condition variables - and flags pperl's status at each section.

pperl status at a glance

None of the sharing primitives on this page crosses a thread boundary under pperl, because there is no second interpreter. The modules are not part of pperl; where a system perl5 provides them on `@INC` they load:

- `threads::shared`: `share` and `is_shared` do nothing.
- `:shared` is parsed but has no underlying interpreter to share to.
- `lock` is a silent no-op, as documented on its reference page.
- `Thread::Queue` and `Thread::Semaphore` work inside the one interpreter.

Reading this page is still worth your time if you maintain upstream Perl code or need to read existing ithreaded programs. For new concurrency on pperl, skip to *Alternatives*.

Shared and unshared variables

By default, every Perl variable in an ithread is private. When you spawn a thread, the child starts with a copy of the parent's data and goes its own way:

```
use threads;

my $private = 1;
```

(continues on next page)

(continued from previous page)

```
threads->create(sub { $private++ })->join;
print $private, "\n";   # 1 - parent untouched
```

To cross the thread boundary, mark the variable `:shared` and `use threads::shared`:

```
use threads;
use threads::shared;

my $shared :shared = 1;
threads->create(sub { $shared++ })->join;
print $shared, "\n";    # 2 - write survived into parent
```

For an aggregate, every element becomes shared automatically:

```
my @queue :shared;
my %counts :shared;
```

Only simple scalars and references to other shared variables may be stored into a shared aggregate. Storing a reference to an unshared variable dies - upstream refuses the assignment to protect the sharing boundary.

The `lock` built-in

`lock` places an advisory mutex on a shared datum for the rest of the enclosing block:

```
use threads;
use threads::shared;

my $total :shared = 0;

sub calc {
    while (my $result = compute_chunk()) {
        lock $total;            # block until mutex available
        $total += $result;
    }                            # mutex released here
}
```

Properties worth remembering:

- **Advisory only.** `lock` blocks other callers of `lock` on the same datum. A thread that reads or writes `$total` without calling `lock` is not serialised.
- **Block-scoped.** Release happens at block exit, not at the end of the `lock` statement. There is no `unlock`.
- **Recursive per thread.** The same thread may re-acquire a lock it already holds; other threads still block.
- **Works on aggregates.** `lock @queue`, `lock %h`, `lock &sub` are all valid.

Under pperl, `lock` accepts its argument and returns - no mutex, no blocking. See the `lock` reference page for the full behaviour and the *Differences from upstream* note.

Races

The canonical race:

```
use threads;
use threads::shared;

my $x :shared = 1;
my $t1 = threads->create(sub { my $foo = $x; $x = $foo + 1 });
my $t2 = threads->create(sub { my $bar = $x; $x = $bar + 1 });
$t1->join; $t2->join;
print $x, "\n";         # 2 or 3 - depends on scheduling
```

Two threads read `$x`, compute a new value, and write back. If both reads land before either write, both writes store the same value and one increment is lost. Even `$x++` on a shared scalar is not atomic.

The fix is to hold the lock across the read-modify-write:

```
{
    lock $x;
    $x++;
}
```

Under pperl this problem does not arise because auto-parallelisation does not expose shared mutable state to user code. The analyser declines to parallelise anything that writes a non-reduction global. See *Parallel Execution*, section *How reduction detection works*.

Deadlocks

Two threads, two locks, acquired in opposite order:

```
my $x :shared = 4;
my $y :shared = 'foo';

threads->create(sub { lock $x; sleep 1; lock $y })->join;
threads->create(sub { lock $y; sleep 1; lock $x })->join;
# Both threads wait forever.
```

The standard defences:

- **Fixed lock order.** Every thread that needs multiple locks acquires them in the same global order. Always `$x` before `$y`.
- **Short critical sections.** Hold a lock only for the work that must be serialised, not for anything else.
- **Fewer locks.** Protect the whole structure with one lock rather than one per field.

Queues - the preferred communication channel

`Thread::Queue` is a thread-safe FIFO. Producer threads enqueue, consumer threads dequeue, and the queue handles all synchronisation internally:

```
use threads;
use Thread::Queue;

my $q = Thread::Queue->new;
my $worker = threads->create(sub {
    while (defined(my $item = $q->dequeue)) {
        handle($item);
    }
});

$q->enqueue($_) for @tasks;
$q->enqueue(undef);     # sentinel: tell worker to stop
$worker->join;
```

dequeue blocks when the queue is empty. Enqueueing `undef` (or any agreed sentinel) is the conventional shutdown signal.

Queues replace most explicit `lock` / `cond_wait` patterns in modern Perl threaded code. They are harder to get wrong.

Under pperl the equivalent shape is a straight Perl array processed by a data-parallel loop, or a compiled `map`/`grep` - see *Alternatives*.

Semaphores

`Thread::Semaphore` is a counter with blocking down and up:

```
use threads;
use Thread::Semaphore;

my $sem = Thread::Semaphore->new(4);    # 4 concurrent I/O slots

sub do_io {
    $sem->down;
    # ... perform I/O; at most 4 threads here at once ...
    $sem->up;
}
```

With a starting count of 1, a semaphore behaves like a mutex but must be released explicitly - unlike `lock`, which is scope-released. Starting counts above 1 model pools of identical resources: a quota of concurrent open files, a number of allowed simultaneous HTTP calls, and so on.

Condition variables

`cond_wait` and `cond_signal` - paired with `lock` on a shared scalar - let one thread wait for another to signal a state change. They resemble POSIX `pthread_cond_wait` / `pthread_cond_signal`. For the overwhelming majority of cases a queue does the same job more simply; reach for condition variables only when a queue does not fit the data shape.

Process-scope side effects

Even with nothing shared at the Perl level, threads share the OS process. Anything that changes process state changes it for every thread:

- `chdir` - current working directory
- `chroot` - root directory, unrecoverable
- `umask` - default file mode mask
- `setuid` / `setgid` - effective user / group
- Signal dispositions - there is no per-thread signal mask in Perl

This is a classic source of "mysterious" cross-thread effects in otherwise shared-nothing code. Both upstream and pperl inherit this from the OS.

Differences from upstream

pperl has no `threads` module, so everything on this page runs in a single interpreter. `threads::shared`, `Thread::Queue` and `Thread::Semaphore` are not part of pperl; where a system perl5 provides them on `@INC` they load, the `:shared` attribute is accepted, and `cond_wait`, `cond_signal` and `lock` are no-ops. This matches upstream's behaviour under a build without thread support.

pperl exposes parallelism at a different layer - the JIT compiler dispatches eligible loop bodies across a Rayon thread pool, with automatic analysis of counter variables, reduction accumulators, and side effects. No user-visible synchronisation primitives are needed because the analyser refuses to parallelise any loop whose body is not trivially independent. See *Parallel Execution* for the details and *Alternatives* for the pperl-native patterns.

See also

- `lock` - reference page for the only shared-data primitive with a pperl compile-time parse
- *ithreads basics* - spawning, joining, detaching
- *Alternatives* - the pperl-native concurrency patterns that replace the primitives above
- *Parallel Execution* - how auto-parallelisation handles reductions and detects side effects
- `fork` - process-level concurrency with no shared memory

Alternatives to ithreads

pperl does not implement Perl's interpreter threads. That sounds like a loss until you look at *why* most programs reach for threads. Almost every upstream ithreaded program falls into one of three shapes:

- **Parallel compute over data** - "apply this function to each element of this large array, as fast as you can."
- **Parallel I/O** - "perform these N independent network or filesystem calls concurrently."
- **Background work with isolated state** - "run this task to the side, with no shared memory, and wait for the result."

pperl has a dedicated, better-performing answer for each.

Decision table

Your goal	Upstream tool	pperl-native answer
Parallelise a compute-heavy loop	`threads` + work-crew pattern	Auto-parallelisation (JIT + Rayon)
Transform a large list with `map` / `grep`	Work-crew thread pool	Compiled `map` / `grep`, or a data-parallel loop
Run I/O-bound tasks concurrently	`threads` + `Thread::Queue`	*Coroutines* with an event loop, or `fork` with pipes
Isolate a subtask with its own state	Detached thread	`fork`
Produce/consume pipeline	`threads` + `Thread::Queue` chain	`Coro::Channel` between *coroutines*, `fork` + pipes, or a sequential iterator
Synchronised counter	`:shared` + `lock`	Sequential accumulator + parallel reduction

Auto-parallelisation for compute loops

pperl's JIT detects loops whose bodies are free of I/O, global writes, and impure calls, then dispatches the loop body across a Rayon work-stealing pool. A scalar-accumulator like `$sum += ...` is recognised as a reduction and combined after all iterations finish.

```
my $sum = 0;
for my $i (1 .. 10_000_000) {
    $sum += $i * $i % 7;
}
print $sum, "\n";
```

Under pperl this compiles to a single Rayon-dispatched loop body. On a typical 8-core machine the speedup is in the 5-6x range over the sequential JIT, with no user-visible thread objects, no synchronisation code, and no shared state.

See *Parallel Execution* for the full story - what qualifies, how reduction detection works, and the CLI flags (`--no-parallel`, `--threads=N`, `--parallel-threshold=N`) that control the behaviour.

Porting a work-crew ithreaded loop typically means **deleting the threading code**. The sequential form is often the pperl-optimal form already.

Before:

```
use threads;
use threads::shared;

my $sum :shared = 0;
my @chunks = chunk_data(\@big);
my @workers = map {
    my $c = $_;
    threads->create(sub {
        my $local = 0;
        $local += process($_) for @$c;
        lock $sum; $sum += $local;
    });
} @chunks;
$_->join for @workers;
```

After:

```
my $sum = 0;
$sum += process($_) for @big;
```

The chunking, the per-thread accumulator, the lock - all gone. The JIT does the chunking. The reduction detector handles the accumulator. There is no shared variable to lock.

map and grep at native speed

For list-shaped transformations, the built-ins are already the natural idiom:

```
my @results  = map  { $_ * $k + 1 } @input;
my @filtered = grep { $_ % 7      } @input;
```

When the block is a pure expression, pperl compiles the whole map/grep to a native loop (see *Writing fast pperl*). This runs on one core - array state gates parallel dispatch off - but several times faster than the interpreted op machinery. When the work per element is a heavy NUMERIC computation, restructure as an index-range `for` loop with a reduction and the parallelizer can take it across cores.

Equivalent ithreaded code would involve a thread pool, an input queue, an output queue, a sentinel value to signal end-of-input, and per-thread accumulators.

fork - process-level concurrency

When work is **not** a pure compute loop - it involves I/O, subprocess management, or genuinely independent state - reach for `fork`. A forked child has:

- A full copy of the parent's memory, copy-on-write at the OS level.
- Its own file descriptor table (with shared underlying file descriptions).
- Complete isolation at the interpreter level - no shared Perl heap, ever.

```
my $pid = fork;
die "fork: $!" unless defined $pid;
if ($pid == 0) {
    # child
    exec 'processing-tool', @args
        or die "exec: $!";
}
# parent
waitpid $pid, 0;
```

Communication uses pipes, sockets, or the filesystem - the same inter-process primitives you would use between unrelated programs. That sounds heavier than in-process threading, and at the raw syscall level it is; but for program structure it is often simpler because there is no shared memory to guard.

Forking a work crew

The upstream `threads::shared` work-crew in the previous section translates to a fork-based equivalent when the work involves I/O:

```
my @pids;
for my $chunk (@chunks) {
    my $pid = fork // die "fork: $!";
    if ($pid == 0) {
        process_chunk($chunk);
        exit 0;
    }
    push @pids, $pid;
}
waitpid $_, 0 for @pids;
```

Results flow back through pipes, files, or a named collection point in `/tmp`. Whatever you would have used between separate programs.

Pipeline via fork and pipes

For pipeline-shaped workloads, Perl's open with a `|-` or `-|` form spawns a child with a pipe already attached:

```
open my $producer, '-|', 'find', '/data', '-type', 'f'
    or die "fork: $!";
```

(continues on next page)

(continued from previous page)

```
while (my $path = <$producer>) {
    chomp $path;
    # filter / process / forward
}
close $producer;
```

Each pipeline stage is its own process, scheduled by the OS, with no shared Perl state.

Isolated background work

The upstream pattern of `threads->create(sub { ... })->detach` - fire and forget - maps to a double-fork:

```
my $pid = fork // die "fork: $!";
if ($pid == 0) {
    # first child: fork again and exit immediately, orphaning
    # the grandchild to init so the parent does not have to wait
    my $grand = fork // die "fork: $!";
    exit 0 if $grand != 0;
    background_work();
    exit 0;
}
waitpid $pid, 0;    # reap the first child, not the grandchild
```

The grandchild runs to completion independently of the original program, no zombie is left behind, and state isolation is absolute.

Coroutines for work that waits

When the threads were there because each task spends most of its time *waiting* - on a socket, a timer, a child process - coroutines keep the thread-per-task structure without threads. Each task is a coroutine with its own call stack and lexicals, written as straight-line code; a wait blocks only that coroutine, and an event loop wakes it. Work crews fed from a `Thread::Queue` become coroutines fed from a `Coro::Channel`, and the `:shared` attributes and locks go away, since only one coroutine runs at a time. See *Coroutines and event loops*.

Choosing between them

- **Auto-parallelisation** wins for compute-bound loops over in-memory data: no process startup cost, no serialisation of results, JIT-compiled body.
- **Coroutines** win for many concurrent waits in one program - network clients and servers, timers, pipelines of I/O stages - with no copying and no locking, on one CPU.
- **fork** wins for subprocess work, for using several CPUs on work that is not a parallelisable loop, and for anything where the task should not inherit the parent's global state changes.

- **None of them** is the right answer for low-cost task dispatch in tight loops - the usual culprit there is a sequential loop that does not actually benefit from concurrency. Measure before adding any layer.

See also

- *Parallel Execution* - the auto-parallelisation chapter: what qualifies, CLI flags, reduction detection
- *Coroutines and event loops* - Coro, EV, Event and AnyEvent for I/O-bound concurrency
- *ithreads basics* - the upstream model for context on what these alternatives replace
- *Shared data* - the upstream sharing primitives, for reading existing ithreaded code
- `fork` - full reference for the process-level primitive
- `wait` - reap a child process
- `lock` - the no-op ithreads primitive under pperl
- *Reference* - threading support is a runtime concern
- **ithreads basics** - the perl5 model: spawning with `threads->create`, joining and detaching, passing parameters, returning values, the three structural patterns (boss/worker, work-crew, pipeline).
- **Shared data** - `threads::shared`, the `:shared` attribute, `lock`, semaphores, queues, race conditions, deadlocks. Every section flags pperl's divergence.
- **Alternatives** - pperl's auto-parallelisation, coroutines, fork-based concurrency with `fork`, and a decision table for picking the right tool.

Reference cross-links

- `lock` - the only ithreads primitive with a reference page. Its *Differences from upstream* section is the canonical statement of pperl's divergence.
- `fork` - process-level concurrency, fully supported under pperl.
- `wait` - reap a child process.
- *Parallel Execution* - pperl's auto-parallelisation.
- *Coroutines and event loops* - Coro, EV, Event and AnyEvent, for tasks that wait.
- *Reference* - threading support is a runtime-level topic.

4.2.3 Regular expressions

A regular expression (regexp, regex) is a pattern that decides whether a string has a given shape, or pulls pieces out of a string that does. Perl treats regexps as a first-class sublanguage: they appear wherever you match (`m//`), substitute (`s///`), quote a pattern (`qr//`), or split on a separator (`split`).

This guide is the definitive pperl reference for regex. Each chapter covers one topic in three layers: an overview that orients you, a precise reference that defines what the construct does, and the gory details - corner cases, performance pathologies, and the

cross-engine differences that surprise a Perl-trained reader. The same chapter serves the reader who needs a one-paragraph answer and the reader who wants to know exactly when a pattern will misbehave.

Who this is for

Readers who know Perl well enough to use scalars, arrays, and hashes, but treat regexps as something to copy from elsewhere and hope works. After reading, you will read an unfamiliar pattern and know what it will match - and when it will refuse, and why.

How this guide is organised

The chapters are meant to be read in order on a first pass, but each stands on its own for later reference.

Basics

The smallest useful regexp is a plain string. `"Hello World" =~ /World/` asks: does the string on the left contain the pattern on the right? It does, so the expression is true.

```
if ("Hello World" =~ /World/) {
    print "matched\n";
}
```

The `//` enclose the pattern. The `=~` operator binds the pattern to the string you want to test. Without a binding operator, Perl applies the pattern to `$_` instead.

The match operator

The long form is `m//`:

```
"Hello World" =~ m/World/;
"Hello World" =~ m!World!;    # alternate delimiters
"Hello World" =~ m{World};    # paired delimiters
```

`m` lets you pick any delimiter. That matters when the pattern itself contains the default delimiter `/` - compare

```
"/usr/bin/perl" =~ /\/usr\/bin\/perl/;  # leaning toothpick syndrome
"/usr/bin/perl" =~ m!/usr/bin/perl!;    # clearer
```

Paired delimiters (`{}`, `()`, `[]`, `<>`) nest, which is useful when the pattern contains the delimiter character.

Without `m`, the leading slash is required: `/pat/` only. With `m`, the leading `m` is required: `m{pat}`, not `{pat}`.

A small but useful corner: `m''` (single quotes as delimiters) makes the pattern *single-quotish* - no variable interpolation, no doublequote escapes. Useful when the pattern is meant to contain literal `$` or `@` and you don't want to escape them.

```
'price: $10' =~ m'\$\d+';      # $-as-anchor would be /$\d+/
'price: $10' =~ m'$10';        # literal '$10' - no interpolation
```

Other regex operators

`m//` is one of four:

Operator	Purpose
`m//`	match - does the pattern match the string?
`s///`	substitute - replace the match with something else
`qr//`	compile a pattern object for reuse
`tr///` (also `y///`)	character-by-character translation

`tr///` is *not* a regex operator despite living in the same neighbourhood - it does character-class translation, not pattern matching. It is mentioned for completeness; see `tr` for its semantics.

`m`, `s`, and `qr` all share the regex syntax covered in this guide. The differences are in what they do *with* a successful match.

Pattern reuse with `qr//`

`qr//` compiles a pattern once and produces a reusable object. Use it when the same pattern is matched repeatedly, especially in loops:

```
my $word = qr/\b[a-z]+\b/;

for my $line (@lines) {
    while ($line =~ /$word/g) {
        print "$&\n";
    }
}
```

The compiled `qr//` object is dropped into other patterns by interpolation. It also lets you build patterns out of named pieces, which becomes essential for any regex over about ten lines:

```
my $name   = qr/[A-Z][a-z]+/;
my $number = qr/\d+/;
my $entry  = qr/$name \s+ $number/x;

"Alice 42" =~ /^$entry$/;   # matches
```

The same compile-once benefit, expressed at the right level of abstraction.

Binding: `=~` and `!~`

`=~` asks “does it match?”. `!~` asks “does it fail to match?”.

```
$s = "Hello World";

print "yes\n" if $s =~ /World/;   # yes
print "no\n"  if $s !~ /planet/;  # no
```

`!~` is not a separate regexp construct - it is the negated binding. It is equivalent to `not ($s =~ /pat/)`.

Matching against $_

If you omit the binding, the match is against `$_`:

```
for ("cat", "dog", "bird") {
    print "has an 'o'\n" if /o/;   # implicit: $_ =~ /o/
}
```

This is idiomatic in `while (<>)` loops, inside `grep` and `map`, and inside `for` loops that set `$_`.

Case sensitivity and the default anchor

Matches are case-sensitive and unanchored:

```
"Hello" =~ /hello/;    # does not match - case differs
"Hello" =~ /ell/;      # matches - inside the string is fine
```

To match case-insensitively, append `/i`. To constrain the match to the start or end of the string, use anchors. Both are covered in their own chapters.

When a pattern could match at several positions, Perl tries from the left and takes the first one that works:

```
"That hat is red" =~ /hat/;   # matches 'hat' in 'That', not in 'hat
↪'
```

The "leftmost match wins" rule is fundamental and pre-empts every other match preference: a match at an earlier position is always better than a match at a later position, regardless of what choices either match makes internally. This is sometimes called the *bump-along property*: the engine tries to match at position 0; if it fails, it bumps to position 1; if it fails, position 2; and so on, returning the first success.

Metacharacters

Most characters in a pattern match themselves. These do not:

```
{ } [ ] ( ) ^ $ . | * + ? \
```

Two more are special only in specific contexts:

- `-` is a metacharacter only inside a character class, where it forms a range (`[a-z]`). Outside `[…]` it is literal.
- `#` is a metacharacter only under the `/x` flag, where it introduces a comment to end of line. Without `/x` it is literal.

Each metacharacter has a special meaning covered later. To match a literal copy of one, put a backslash in front:

```
"2+2=4" =~ /2+2/;    # fails - '+' is a quantifier, needs escaping
"2+2=4" =~ /2\+2/;   # matches

"end." =~ /end\./;   # matches a literal dot
"end." =~ /end./;    # also matches - but . matches any character,
                     # so this would also match "endx", "end ", etc.
```

The backslash itself is a metacharacter, so a literal backslash in a pattern needs `\\`:

```
'C:\WIN32' =~ /C:\\WIN/;    # matches
```

A metacharacter that has nothing special to do in its context reverts to matching itself. `}` only closes a `{...}` quantifier; outside that context it is a literal `}`. This is convenient but easy to misread; see *Strict mode* below.

Strict mode: `use re 'strict'`

`use re 'strict'` turns previously-tolerated regex sloppiness into compile-time errors. Use it when you want the regex compiler to flag patterns that *probably* mean something different from what you wrote:

```
use re 'strict';

/abc{,1/;        # error: unescaped '{' in non-quantifier context
/(?-p)/;         # error: useless negation of always-on flag
/[a-]/;          # error: dash at end of class
```

Strict mode is per-lexical-scope, so it can be turned on for regex-heavy modules without affecting other code. It is not default because it would break working older patterns; new code should consider it.

Escape sequences

Non-printing characters use the same escapes as in double-quoted strings:

Sequence	Matches
`\t`	tab
`\n`	newline
`\r`	carriage return
`\f`	form feed
`\a`	alert (bell)
`\e`	escape (`\x1B`)
`\0`	NUL byte
`\xHH`	byte with hex value HH
`\x{...}`	Unicode codepoint with hex value
`\o{...}`	octal codepoint
`\cX`	control-X
`\N{...}`	Unicode character by name

```
"1000\t2000" =~ /0\t2/;      # matches
"a\x{263a}b" =~ /\x{263a}/;  # matches U+263A, WHITE SMILING FACE
```

The full Unicode story is in the *unicode* chapter.

Variables in patterns

A pattern is (by default) interpolated like a double-quoted string, so variables are substituted before matching:

```
my $word = "house";
"housecat" =~ /$word/;       # matches
"housecat" =~ /${word}cat/;  # matches - braces disambiguate
```

To match a literal `$` or `@`, escape it:

```
'price: $10' =~ /\$10/;      # matches a literal dollar sign
```

If a user-supplied string will be interpolated into a pattern and you want its metacharacters treated literally, use `quotemeta` - or its in-pattern equivalent `\Q…\E`:

```
my $input = "1+1";
"1+1=2" =~ /\Q$input\E/;     # matches the literal string
```

Without `\Q…\E` the `+` would be read as a quantifier.

How a regex is read

Perl reads a regex pattern in four phases. Knowing the phases explains a few "why does that work?" questions:

1. **Phase A**: the parser identifies the delimiter and finds the end of the pattern. `(?#…)` comments are removed.
2. **Phase B**: the pattern is parsed as a *double-quotish string* - variables interpolate, escape sequences cook, `\Q…\E` translates to `quotemeta`-style escaping.
3. **Phase C**: under `/x` or `/xx`, unescaped whitespace and comments after `#` are stripped.
4. **Phase D**: the regex compiler reads the result and turns it into the engine's internal form.

The order matters because phases B and D operate on different representations. `\Q$dir\E` cooks at Phase B, before the regex compiler sees it - by Phase D, the variable's contents are already escaped, and the regex compiler sees a literal pattern. Conversely, `\U…\E` is interpreted by Phase B as a string-cook directive (uppercase the contents), which is almost certainly *not* what you want inside a regex. The convention is: `\Q` and `\E` for regex purposes; `\U`, `\L`, `\u`, `\l` only when you know what you are doing.

`(?#comment)` is removed before Phase B sees the pattern at all. A literal `#` inside `(?#…)` is fine. A literal `)` is not - the comment ends at the first `)`.

Earliest match wins - the bump-along

Friedl puts it precisely:

> The match that begins earliest wins.

Position 0 is tried first, then 1, then 2, and so on. The engine *never* prefers a later, longer, or "more aesthetic" match over an earlier one. This is so foundational it shapes every other rule in this guide:

- "Greedy quantifiers grab as much as possible" - *at the current starting position.* The grabbing happens after the bump-along has chosen the position.
- "Leftmost alternative wins" - *at the current starting position.* The alternative chosen affects what is captured but not where the match starts.
- Anchors like ^ constrain *what positions are legal*, not what is preferred among legal ones.

Say what you mean

Friedl's recurring point: vagueness in the regex causes both correctness problems and performance problems. The example he hammers:

```
/-?[0-9]*\.?[0-9]*/
```

Read as English: "an optional sign, optional digits, optional decimal, optional digits." Read as code: every part is optional, so the pattern *matches the empty string* - at the start of any input, before the engine has even looked. Apply it to "nothing here" and it matches at position 0, capturing the empty string, returning success.

The fix is to require at least one digit on at least one side of the decimal:

```
/-?[0-9]+(\.[0-9]*)?|-?\.[0-9]+/
```

Two alternatives, each requiring at least one digit. The pattern now matches what its author intended.

The lesson generalises. A pattern that *can* match the empty string usually *will* match the empty string somewhere. If your match has to mean something, write requirements that force it to mean something.

Substitution at a glance

Replacing text uses the `s///` operator, which takes a pattern and a replacement string:

```
my $x = "feed the cat";
$x =~ s/cat/dog/;            # $x is now "feed the dog"
```

Substitution is covered in depth in its own chapter; it is mentioned here so you can combine it with the facts above. Most everything that applies to `m//` patterns applies inside `s///` patterns too.

Where to go next

Literal matches get you surprisingly far, but every real regexp uses character classes, anchors, or quantifiers. Character classes come next - they let one position in the pattern accept any of several characters.

If you are reading the guide for a specific question, the chapters are independent and the *index* lists them all. Patterns that read fine but run for hours are covered in *performance* - when in doubt, the answer is usually there.

See also

- `m` - full reference for the match operator.
- `s` - full reference for substitution.
- `qr` - compile a pattern object.
- `quotemeta` - escape a string for safe pattern interpolation.
- The *character classes* chapter - what comes after literal matching.
- The *performance* chapter - what could go wrong.

Character classes

A character class matches exactly one character, chosen from a set you define. Where a literal `a` matches only `a`, the class `[abc]` matches any one of `a`, `b`, or `c`.

```
/cat/;               # matches 'cat'
/[bcr]at/;           # matches 'bat', 'cat', or 'rat'
/item[0123456789]/;  # matches 'item0' through 'item9'
```

The class still consumes one character of the string. `[bcr]at` never matches `at` (no letter present) and never matches `brat` (two letters where the class expects one).

Ranges

Inside `[...]`, a dash between two characters denotes a contiguous range in the underlying character set:

```
/[0-9]/;         # any ASCII digit
/[a-z]/;         # any ASCII lowercase letter
/[a-zA-Z]/;      # any ASCII letter
/[0-9a-fA-F]/;   # any hex digit
```

Ranges can be combined with individual characters:

```
/[0-9bx-z]aa/;   # matches '0aa'..'9aa', 'baa', 'xaa', 'yaa', 'zaa'
```

A dash that is first or last inside the class is literal:

```
/[-ab]/;         # matches '-', 'a', or 'b'
/[ab-]/;         # same
```

Negation

A caret ^ as the first character inside `[...]` inverts the class:

```
/[^a]/;          # any character except 'a'
/[^0-9]/;        # any non-digit
```

A caret elsewhere is literal:

```
/[a^]/;          # matches 'a' or '^'
```

A negated class still matches *one* character - `[^a]` does not match the empty string; it requires one non-a.

Special characters inside a class

Inside `[...]` the special set shrinks to - `]` `\` `^` `$` (and the pattern delimiter). The others - `.`, `*`, `+`, `?`, `(`, `)`, `{`, `}`, `|` - are literals in a class:

```
/[.+*]/;         # matches a literal '.', '+', or '*'
/[()]/;          # matches '(' or ')'
```

To match `]` inside the class, either escape it or put it first (after any leading ^):

```
/[\]]/;          # matches ']'
/[]ab]/;         # matches ']', 'a', or 'b'
```

`$` and `\` are slightly awkward because they interact with interpolation and escaping:

```
my $x = 'bcr';
/[$x]at/;        # matches 'bat', 'cat', or 'rat' - interpolated
/[\$x]at/;       # matches '$at' or 'xat' - '$' is literal
/[\\$x]at/;      # matches '\at' plus interpolation of $x
```

`\b` inside a character class means *backspace* (`\x08`), not "word boundary". Outside a class it is the word boundary assertion. This is the single most common dual-meaning trap in the regex syntax - see the *anchors and assertions* chapter for the boundary form.

Shorthand classes

Several common classes have shorthand names usable both inside and outside `[...]`:

Shorthand	Matches
`\d`	a digit
`\D`	a non-digit
`\w`	a word character (alphanumeric or _)
`\W`	a non-word character
`\s`	whitespace (space, tab, `\r`, `\n`, `\f`, more)
`\S`	non-whitespace
`\h`	horizontal whitespace (space, tab, Unicode)
`\H`	non-horizontal-whitespace
`\v`	vertical whitespace (`\n`, `\r`, `\f`, ...)
`\V`	non-vertical-whitespace
`\R`	linebreak: `\r\n`, `\n`, `\v`, `\f`, `\x{85}`, ...
`\N`	any character except `\n` (regardless of `/s`)

Under Unicode (the default), `\d`, `\w`, `\s` match more than just ASCII. `\d` matches any Unicode digit (Devanagari digits, Arabic-Indic digits, and many more), `\w` matches any letter in any script plus marks and connector punctuation, and `\s` adds Unicode space characters such as non-breaking space.

To restrict these to ASCII, add the `/a` modifier or use explicit ranges like `[0-9]` and `[A-Za-z_0-9]`.

```
"item0" =~ /\w\w\w\w\d/;      # matches
"abc\x{0660}" =~ /\w\w\w\d/; # matches: U+0660 is an Arabic-Indic␣
 ↪zero
"abc\x{0660}" =~ /\w\w\w\d/a;# does not match under /a
```

`\R` is the linebreak shorthand - it matches any of the recognised line-break sequences as one token. Useful for parsing text that may have CRLF, LF, or rarer line terminators interchangeably. Unlike a class, `\R` may match *two* characters (the CRLF case) and so cannot appear inside `[…]`.

`\N` (uppercase) means "any character except `\n`", and is *not* affected by the `/s` modifier. This is the dual meaning to be careful of: `\N{NAME}` (with brace) is a Unicode named-character escape (see the *unicode* chapter); bare `\N` is the non-newline class.

The period

`.` matches any single character except newline. Under the `/s` modifier (covered in the *modifiers* chapter), `.` also matches newline:

```
"a\nb" =~ /a.b/;        # does not match
"a\nb" =~ /a.b/s;       # matches
```

When you want "any character including newline" without `/s`, the classic idiom is `[\s\S]` (or `[\d\D]`):

```
"a\nb" =~ /a[\s\S]b/;   # matches without /s
```

The trick is that any character is either a whitespace or a non-whitespace; the class covers both.

Composing classes

You can mix shorthands, ranges, and individual characters inside one class:

```
/[\d\s]/;        # a digit or whitespace
/[A-Z\d_]/;      # uppercase letter, digit, or underscore
/[a-zA-Z\d]/;    # letter or digit (ASCII)
```

De Morgan's law matters: `[^\d\w]` is *not* `[\D\W]`. The first requires the character to be *both* non-digit and non-word. But every digit is a word character, so `[^\d\w]` simplifies to `[^\w]`, i.e. `\W`. Be careful when combining negated shorthands.

POSIX classes

POSIX character classes use the form `[:name:]` and only work inside `[…]`:

POSIX	Equivalent
`[:alpha:]`	alphabetic
`[:alnum:]`	alphanumeric
`[:digit:]`	digit (like `\d`)
`[:word:]`	word char (Perl extension)
`[:space:]`	whitespace (like `\s`)
`[:upper:]`	uppercase
`[:lower:]`	lowercase
`[:xdigit:]`	hex digit
`[:ascii:]`	0x00–0x7F
`[:cntrl:]`	control character
`[:graph:]`	printable, not space
`[:print:]`	printable, including space
`[:punct:]`	punctuation
`[:blank:]`	space or tab

Negate a POSIX class with ^ *inside* the colons:

```
/[[:^digit:]]/;   # same as \D
/[[:alpha:][:digit:]]/;  # letter or digit - equivalent to \w minus
↪'_'
```

POSIX classes follow the same Unicode-vs-ASCII rules as the shorthands: without `/a`, `[:alpha:]` is the Unicode alphabetic set.

POSIX also defines two related constructs that are rarely implemented:

- **Collating elements** `[.span-ll.]` - match a multi-character collation element as a single unit (e.g. Spanish `ll` historically).
- **Equivalence classes** `[[=n=]]` - match any character that is equivalent under the locale's collation rules (e.g. accented variants of n).

Perl recognises the syntax but treats both forms as the literal characters. In practice no portable script relies on these; they are documented for completeness.

Unicode properties

Unicode defines thousands of properties. The notation is `\p{Name}` for "has this property" and `\P{Name}` for "does not have this property":

```
/\p{Lu}/;              # any uppercase letter, any script
/\p{Greek}/;           # any character in the Greek script
/\p{Number}/;          # any numeric character
/\P{ASCII}/;           # any non-ASCII character
```

Short single-letter aliases exist for common properties and drop the braces: `\pL` is a letter, `\pN` a number, `\pP` punctuation. `\p{L}` is the same as `\pL`.

The *unicode* chapter covers properties in detail, including the compound form `\p{Name=Value}`, the `\X` grapheme cluster, and the charset modifiers `/a`, `/u`, `/l`, `/d`.

Extended bracketed classes - `(?[ ])`

The standard `[...]` syntax handles unions ("any of these characters") well but does not have set operations on classes. The extended form `(?[ ... ])` does:

Operator	Meaning
+ or |	union (the same characters either operand has)
&	intersection (in both)
-	difference (in left, not in right)
^	symmetric difference (in one but not both)
!	complement (everything except)

Whitespace inside `(?[...])` is ignored, so the operators read as arithmetic.

```
# Greek letters only:
/(?[ \p{Greek} & \p{Letter} ])/;

# Letters that are not Latin:
/(?[ \p{Letter} - \p{Latin} ])/;

# Hex digit, but not 'a' through 'f':
/(?[ [0-9A-Fa-f] - [a-f] ])/;
```

The construct is most useful when combining Unicode properties that overlap. Without it, the same expressions would require verbose lookaround or out-of-pattern logic.

`(?[...])` is itself a character class - it consumes one character and can be quantified:

```
/(?[ \p{Letter} & \p{ASCII} ])+ /x;   # ASCII letters
```

Caveat: `(?[...])` is its own little parser inside the regex parser. Inside it, only specific operators and operands are recognised. Mistakes produce specific compile-time errors, which in turn means *strict mode* (`use re 'strict'`) catches more bad class expressions when you use the extended form.

Negated class beats .*?

A common newcomer pattern: use .*? (non-greedy .) to match everything up to a delimiter, like <.*?>. The pattern works on the inputs you tested it on; on adversarial input it does not.

```
"<a> </a>"      =~ /<.*?>/;        # matches '<a>' - fine
"<a> <b>foo"    =~ /<.+?>foo/;     # matches '<a> <b>foo' - bad
```

In the second case the engine first matched <a>, then needed foo but found a space. Under that backtracking pressure the non-greedy .+? was forced to expand, gladly consuming the > of <a> and the space until foo lined up. The negated character class cannot give ground that way:

```
"<a> <b>foo" =~ /<[^>]+>foo/;      # matches '<b>foo' - [^>]+␣
↪refuses to cross '>'
```

Two reasons to prefer [^>]+ over .+? whenever the delimiter is a single character:

1. **Correctness**: the negated class is a hard barrier; the non-greedy form is a preference.
2. **Performance**: a negated class participates in the simple-repetition optimisation; .+? does not (the engine has to leave the inner loop on every iteration to test what follows). On long inputs this matters.

Worked example: an IP-address regex

The canonical "specificity vs. complexity" exercise. Five iterations, each fixing one category of vagueness:

1. Naive. "Four dot-separated digit groups."

```
/[0-9]*\.[0-9]*\.[0-9]*\.[0-9]*/;
```

Matches and then . . . . ? happily - every digit run is optional, so the pattern is satisfied by three dots and nothing else.

2. Require digits. Anchor the pattern.

```
/^[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+$/;
```

Matches 1234.5678.9101112.131415. Each group has digits but no upper bound on count.

3. Bound the digit count, badly.

```
/^\d{3}\.\d{3}\.\d{3}\.\d{3}$/;
```

Matches 192.168.001.001 but rejects 1.2.3.4 - leading zeros not always written.

4. Allow 1 to 3 digits.

```
/^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$/;
```

Now matches `1.2.3.4` and rejects `1234.5.6.7`. But it also matches `999.999.999.999` - beyond the 0–255 range.

5. Range-correct.

```
/^(?:25[0-5]|2[0-4]\d|[01]?\d\d?)\.
   (?:25[0-5]|2[0-4]\d|[01]?\d\d?)\.
   (?:25[0-5]|2[0-4]\d|[01]?\d\d?)\.
   (?:25[0-5]|2[0-4]\d|[01]?\d\d?)$/x;
```

Each group is one of: `25[0-5]` (250–255), `2[0-4]\d` (200–249), or `[01]?\d\d?` (0–199 in various forms). The pattern now matches *exactly* the strings that name a syntactically-valid IPv4 address.

The lesson is not "memorise this regex". It is the *progression*: each iteration tightened a specific kind of vagueness. The right regex for a problem is the one that admits exactly the right inputs and rejects everything else, and you only get there by asking what the *previous* regex actually allowed.

Cross-engine: shorthand classes

The `\d`, `\w`, `\s` shorthands are not portable. The *cross-engine* chapter has the full table; the relevant rows extracted:

Shorthand	Perl 5.44 (default)	PCRE	Emacs	POSIX BRE / ERE	RE2 / Go (default)
`\d`	ASCII (or Unicode under `/u`)	ASCI	NO (use `[0-9]` or `[[:digit:]]`)	NO	ASCII; Unicode under `(?u)`
`\w`	ASCII or Unicode	ASCI	yes (syntax-table-driver	NO	ASCII; Unicode under `(?u)`
`\s`	ASCII or Unicode	ASCI	yes	NO	ASCII; Unicode under `(?u)`
`\b` word boundary	yes	yes	yes (`\</\>` traditional)	NO	yes
`\h`, `\v`	yes	yes	NO	NO	NO

Two things to internalise:

1. POSIX BRE and ERE lack `\d`, `\w`, `\s` entirely. Portable shell scripts use `[0-9]`, `[[:alnum:]_]`, `[[:space:]]`.
2. Emacs has `\w` and `\s` but no `\d`. Emacs's `\s` syntax is *followed by* a syntax-class character (`\s-` for whitespace, `\sw` for word) - unique to Emacs.

POSIX bracket classes (`[[:digit:]]`, `[[:alpha:]]`, ...) are the universal portable spelling: every engine in the comparison recognises them.

A useful habit

Named and shorthand classes are almost always clearer than explicit ranges. `\d{4}-\d{2}-\d{2}` reads; `[0-9]{4}-[0-9]{2}-[0-9]{2}` needs a moment. Use the ranges only when you have a concrete reason - usually performance in a hot loop, or deliberately restricting to ASCII.

See also

- The *unicode* chapter - `\p{…}`, `\P{…}`, `\X`, the charset modifiers `/a`, `/u`, `/l`, `/d`.
- The *anchors and assertions* chapter - `\b` outside a class.
- The *cross-engine* chapter - full table of shorthand support across engines.
- `m` - the match operator.
- `qr` - compile a pattern for reuse.

Anchors and assertions

Anchors and assertions match *positions* rather than characters. They test where in the string you are, not what is there. A plain character class consumes one character; an anchor consumes none.

They are called *zero-width assertions* for that reason - they have width zero, but they assert a property must hold.

Start and end of string

^ matches at the start of the string. $ matches at the end, or just before a final newline.

```
"housekeeper" =~ /keeper/;    # matches - 'keeper' appears somewhere
"housekeeper" =~ /^keeper/;   # does not match - not at start
"housekeeper" =~ /keeper$/;   # matches - at end
"housekeeper\n" =~ /keeper$/; # matches - before the trailing
↪newline
```

Used together, ^…$ forces the pattern to account for the whole string:

```
"bert"    =~ /^bert$/;     # matches
"bertram" =~ /^bert$/;     # does not match
"dilbert" =~ /^bert$/;     # does not match
""        =~ /^$/;         # matches - empty string
```

For a literal string, `$s eq "bert"` is faster and clearer. ^…$ only earns its keep once the middle uses real regexp features.

Absolute versus line-relative anchors

Under the `/m` modifier, ^ and $ anchor at every line boundary inside the string, not just the outer ends:

```
my $x = "There once was a girl\nWho programmed in Perl\n";

$x =~ /^Who/;    # does not match - 'Who' is not at string start
$x =~ /^Who/m;   # matches - 'Who' is at start of second line
```

For the absolute ends *regardless of* `/m`, use the dedicated anchors:

- `\A` - absolute start of string.
- `\z` - absolute end of string.
- `\Z` - end of string, or just before a final newline. Similar to `$` but unaffected by `/m`.

```
$x =~ /^Who/m;    # matches, as above
$x =~ /\AWho/m;   # does not match - \A is string start, always

$x =~ /girl$/m;   # matches, end of first line
$x =~ /girl\Z/m;  # does not match - 'girl' is not at string end
$x =~ /Perl\Z/m;  # matches - end or before final newline
$x =~ /Perl\z/m;  # does not match - \z is strict end
```

Rule of thumb: `\A` and `\z` when you mean the whole string; ^ and `$` when you mean lines (usually with `/m`). `\A` and `\z` are also faster: the engine knows they constrain position, and skips bump-along retries that ^ permits.

The four newline modes can be summarised in a table. Choose the one matching what ^, `$`, and `.` should mean for your input:

Mode	^ matches at	$ matches at	. matches \n?
default	start of string only	end of string (and before final \n)	no
/m	start of every line	end of every line	no
/s	start of string only	end of string (and before final \n)	yes
/sm	start of every line	end of every line	yes

`\A`, `\Z`, `\z` are unaffected by either modifier.

Word boundaries

`\b` matches a position between a word character (`\w`) and a non-word character (`\W`), or between either and the edge of the string. It does not consume any character.

```
my $x = "Housecat catenates house and cat";

$x =~ /cat/;      # matches 'cat' inside 'Housecat'
$x =~ /\bcat/;    # matches 'cat' inside 'catenates'
$x =~ /cat\b/;    # matches 'cat' inside 'Housecat'
$x =~ /\bcat\b/;  # matches the standalone 'cat' at end
```

`\B` is the negation - a position that is *not* a word boundary.

`\b` outside `[…]` is a word boundary. `\b` *inside* `[…]` is the *backspace* character, `\x08`. The dual meaning is the single most common trap in the regex syntax. Always read `\b`'s

context before reading what it does.

\b and \W-starting items - the $3.75 gotcha

\b requires a transition between word and non-word characters. If both sides of the position are non-word characters, there is no boundary. This catches people building patterns by interpolation:

```
my $item = '$3.75';
my $regex = qr/\b\Q$item\E\b/;

"is $3.75 plus tax" =~ /$regex/;   # does NOT match
```

Why? After \Q$item\E cooks, the pattern is \b\$3\.75\b. The engine looks for a position immediately *before* $ where one side of the position is a word character. The character before $ in the input is a space (non-word), and $ itself is also non-word. There is no word/non-word transition; \b fails.

To anchor the start of $item regardless of its first character:

```
my $regex = qr/(?:^|\s)\Q$item\E(?:\s|$)/;
```

Use string-edge or whitespace anchors when the interpolated content might begin with a non-word character. \b is for contexts where you *know* the surrounded text is word-like.

Unicode word-boundary variants

Plain \b follows the \w/\W definition, which under Unicode covers letters, digits, marks, and connector punctuation. Real text - natural language with apostrophes, hyphens, numerics with commas - wants more nuance. Perl provides four semantic boundary variants:

Boundary	Meaning
\b{wb}	Unicode word boundary - handles don't, state-of-the-art
\b{sb}	sentence boundary
\b{lb}	line break (suitable for line wrapping)
\b{gcb}	grapheme-cluster boundary (same effect as \X)

```
"don't" =~ /.+?\b{wb}/x;   # matches whole word - apostrophe is␣
↪inside
"don't" =~ /.+?\b/x;       # stops at the apostrophe - plain \b␣
↪splits
```

For natural-language processing, \b{wb} and \b{sb} are almost always what you want. Plain \b is for ASCII identifier contexts.

`\G` - the last-match position

`\G` anchors at the position where the previous successful `/g` match ended on the same string. It is what makes tokenisation with `/g` in `while` loops robust.

```
my $s = "12abc34";
while ($s =~ /\G(\d+|[a-z]+)/gc) {
    print "got '$1'\n";
}
# prints: '12', 'abc', '34'
```

Without `\G`, the same pattern would re-scan from wherever it first matched, skipping over characters that didn't fit - quietly losing data. The `/gc` modifier preserves the position on failure; covered in the *modifiers* chapter.

Why `\G` matters: the synch-or-skip choice

Friedl's worked example. Find every five-digit ZIP code beginning with 44 in a run-together string:

```
my $s = "06192054410-44272-13901-44106-22134";
while ($s =~ /44(\d{3})/g) {
    print "got 44$1\n";
}
```

Without `\G`, the engine's bump-along is happy to find `44272` and `44106` correctly *and also* find `44272` from a different start position whenever a match earlier was rejected. The output is "right answer plus extra wrong ones".

With `\G` on every iteration, the engine cannot bump-along on failure - it must resume *exactly* where the last match ended. A failure terminates the loop:

```
while ($s =~ /\G\D*(44\d{3})/gc) {
    print "got $1\n";
}
```

`\G` effectively disables the bump-along, which is what synchronised tokenisation demands. The general rule: **if the correctness of your loop depends on every match being adjacent to the last, you need `\G`.**

`\G` reads the value `pos($string)` returns. Setting `pos` explicitly resets the anchor. On the first iteration of a `/g` loop (or any pattern not yet matched against this string), `\G` is equivalent to `\A`.

Lookahead

Lookahead checks what comes next without consuming it. Positive lookahead is `(?=…)`:

```
my $x = "I catch the housecat 'Tom-cat' with catnip";

$x =~ /cat(?=\s)/;   # matches 'cat' in 'housecat' - space follows
$x =~ /cat(?!\s)/;   # matches 'cat' in 'catch' - no space follows
```

`(?!…)` is negative lookahead: match only if the inner pattern does *not* apply. Zero-width, just like `^` and `$`.

```
"foobar" =~ /foo(?!bar)/;   # does not match
"foobaz" =~ /foo(?!bar)/;   # matches
```

Lookahead worked example: digits not followed by a period

A natural-sounding question that is harder than it looks: "match runs of digits not followed by a period."

The first try: `\d+(?!\.)`. Apply it to `OH 44272`:

```
"OH 44272" =~ /\d+(?!\.)/;   # MATCHES, matches '44272'
```

That looks right, but for the wrong reason. Greedy `\d+` first grabbed the entire digit run `44272`; the next character is end-of-string (not `.`), so `(?!\.)` succeeded straight away - the lookahead happened to hold without the engine ever needing to back off. The *intuitive* reading "the run of digits ends with something other than a period" is satisfied for the trailing-digits case, but the next example shows what happens when it doesn't.

Now apply it to `4423.45`:

```
"4423.45" =~ /\d+(?!\.)/;   # matches '442', not '4423'
```

The engine's first try: `\d+` matches `4423`. `(?!\.)` checks the next character: it is `.`, the negative lookahead fails. The engine backtracks: `\d+` matches `442`, the next character is `3`, not `.`, success. The pattern matches `442` - *not* the full digit run before the period. The greedy quantifier was willing to back off to make the lookahead succeed.

The fix: require the lookahead to also reject other digits, so the run cannot back off into "digit followed by digit".

```
"4423.45" =~ /\d+(?![\d.])/;   # matches '45' - correct
```

`\d+(?![\d.])` reads as "match a run of digits not followed by either another digit or a period". The greedy quantifier still expands maximally, but the lookahead now refuses to succeed in the middle of a digit run, so the only matching positions are the actual digit-run ends.

The lesson generalises: **a lookahead that excludes only one character can be defeated by a greedy quantifier backing off**. Always include the characters of the *atom* the quantifier matches in the lookahead's negation.

A related case: leading negative lookahead and the bump-along.

```
"cattle" =~ /(?!cat)\w+/;   # MATCHES, captures 'cattle'
```

The lookahead `(?!cat)` rejects position 0 (where `cat` would match). The engine bumps along: position 1 starts with `attle`, where `(?!cat)` succeeds, and `\w+` matches `attle`.

To force "first word that doesn't begin with `cat`", anchor the lookahead to a word boundary:

```
"cattle" =~ /\b(?!cat)\w+/;   # does not match
```

Now the lookahead applies at every word boundary, which in `cattle` is only position 0.

Lookbehind

Lookbehind checks what came before. Positive lookbehind is `(?<=…)`:

```
"cats and dogs" =~ /(?<=and )dogs/;   # matches 'dogs' after 'and '
```

Negative lookbehind is `(?<!…)`:

```
"prefix_foo suffix_foo" =~ /(?<!prefix_)foo/;
# matches 'foo' in 'suffix_foo'
```

Modern Perl supports variable-length lookbehind from 1 to 255 characters. Patterns like `(?<=cat|kitten)` are fine. Patterns where the lookbehind itself contains capturing groups produce an experimental warning (the captures' contents are not fully-defined when the lookbehind has variable length).

Because the lookbehind must finish before the current position, its maximum length is limited to 255 *characters* under default semantics. Under `/i`, a few characters fold to multi-character sequences (ß to `ss`), which counts the *expanded* length toward the 255 limit - so a 127-character lookbehind containing ß is fine, but 128 such characters is not.

A practical workaround for longer lookbehind: use `\K` (next section), which has no length limit.

`\K` - keep-left

`\K` is the lookbehind alternative. Everything matched *before* `\K` is excluded from `$&`, but is required to have matched. Pragmatically: "match this prefix, but don't include it in the output."

```
"feed the cat" =~ /the \Kcat/;   # matches; $& is 'cat'
```

Equivalent to `(?<=the )cat`, but:

- No length limit. `\K` works after any prefix.
- Often substantially faster than `(?<=…)`, because the engine does not need to look backward - it forgets the start of the match instead.
- Especially useful in substitution: `s/foo\Kbar/QUUX/` is cleaner than `s/(foo)bar/$1QUUX/`.

`\K` only forgets `$&` and `@-`/`@+[0]`; capture groups before `\K` are still set as captured. The construct may appear inside other lookarounds, though the behaviour there is described as "currently not well defined" - the conservative use is at the top level of a match or substitution.

Long-form lookaround aliases

Each lookaround construct has a verb-style spelling. Long-form aliases read more clearly in patterns that already use `(*VERB:…)` for backtracking control:

Standard form	Verb form (short)	Verb form (long)
`(?=…)`	`(*pla:…)`	`(*positive_lookahead:…)`
`(?!…)`	`(*nla:…)`	`(*negative_lookahead:…)`
`(?<=…)`	`(*plb:…)`	`(*positive_lookbehind:…)`
`(?<!…)`	`(*nlb:…)`	`(*negative_lookbehind:…)`

The verb forms are exactly equivalent to the standard forms. They are accepted; they are not idiomatic in handwritten Perl. You will see them in autogenerated patterns and PCRE2 ports.

Combining lookaround with split

Anchors and lookaround let `split` separate a string on invisible positions rather than visible characters:

```
my $str = "one two - --6-8";
my @toks = split / \s+            # whitespace
                 | (?<=\S) (?=-)  # non-space followed by '-'
                 | (?<=-)  (?=\S) # '-' followed by non-space
                 /x, $str;
# @toks = ("one", "two", "-", "-", "-", "6", "-", "8")
```

The second and third alternatives match *between* characters - no consumption. They are legal split separators because split is happy to split on a zero-width match.

Two zero-width assertions juxtaposed are AND-ed

A small but illuminating fact: when two lookarounds appear next to each other in a pattern, both must hold at the same position. This works as expected:

```
$x =~ /^(\D*)(?=\d)(?!123)/;
# Matches the leading non-digit run, but only if a digit follows
# AND that digit run does not begin with '123'.
```

Both `(?=\d)` and `(?!123)` apply at the same position; the engine treats their conjunction as the requirement. Without `(?=\d)`, the negative lookahead `(?!123)` is satisfiable by *any* non-123 continuation - including end-of-string or a non-digit, neither of which is what the author meant.

This generalises Friedl's framing: juxtaposition in a regex always means AND, except when written with `|`. `/ab/` means "a AND (then) b", just as `/^$/` means "start AND end" - except the AND is across positions for `ab` and at one position for the lookarounds.

Cross-engine: anchors and lookaround

The *cross-engine* chapter has the full table; the relevant rows extracted.

Anchors and newline handling

Concern	Perl 5.44	PCRE2	Emacs	POSIX BRE / ERE	RE2 / Go
^ start-of-string	yes	yes	start-of-buffer	yes	yes
^ after \n under multi-line	/m	(?m)	always (line-oriented)	not specified	(?m)
$ end-of-string	yes	yes	end-of-buffer	yes	yes
$ before final \n	yes	yes	no	varies	yes
\A / \z	yes	yes	\` / \'	no	yes
\Z	yes	yes	no	no	no (use \z)
\G	yes	yes	no	no	no

Lookaround and atomic constructs

Feature	Perl 5.44	PCRE2	Emacs	POSIX	RE2 / Go
Lookahead (?=…) / (?!…)	yes	yes	NO	NO	NO
Fixed-width lookbehind	yes	yes	NO	NO	NO
Variable-length lookbehind	yes (exp.)	yes	NO	NO	NO
\K	yes	yes	NO	NO	NO
Atomic group (?>…)	yes	yes	NO	NO	NO

Effectively, only PCRE2 (and other Perl-derived engines outside this table - Java, Python, .NET) supports lookaround. POSIX tools and RE2 / Go simply do not have it. Patterns that rely on lookaround do not port across this divide.

Summary

Anchor	Matches
`^`	start of string, or start of line under `/m`
`$`	end of string (before trailing `\n`), or end of line under `/m`
`\A`	absolute start of string
`\z`	absolute end of string
`\Z`	end of string or before trailing `\n`, ignores `/m`
`\b`	word boundary
`\B`	non-word-boundary
`\b{wb}`	Unicode word boundary
`\b{sb}`	sentence boundary
`\b{lb}`	line break
`\b{gcb}`	grapheme cluster boundary
`\G`	end of previous `/g` match (or `\A` if none)
`(?=…)`	positive lookahead
`(?!…)`	negative lookahead
`(?<=…)`	positive lookbehind
`(?<!…)`	negative lookbehind
`\K`	keep-left (forget the prefix from `$&`)

See also

- The *modifiers* chapter - `/m`, `/s`, `/g`, `/c`.
- The *unicode* chapter - `\b{wb}`, `\b{sb}`, `\b{lb}`, `\b{gcb}` and what they mean.
- The *performance* chapter - atomic groups `(?>…)`, special backtracking control verbs.
- The *cross-engine* chapter - full anchor and lookaround compatibility table.
- `split` - splitting at zero-width positions.
- `pos` - read or set the position `\G` anchors at.

Quantifiers

A quantifier says "this previous thing, N times." It attaches to a single character, a character class, or a group, and turns a single-character match into a match of many.

```
/a*/;       # zero or more 'a'
/\d+/;      # one or more digits
/colou?r/;  # 'color' or 'colour'
```

The basic quantifiers

Quantifier	Meaning
`a?`	0 or 1 times
`a*`	0 or more times
`a+`	1 or more times
`a{n}`	exactly n times
`a{n,m}`	at least n, at most m
`a{n,}`	at least n
`a{,m}`	at most m

`*`, `+`, and `?` are exact synonyms for `{0,}`, `{1,}`, and `{0,1}`. The brace forms are mostly used when `n` and `m` are specific small numbers; the bare forms read more naturally for "any number of these".

The bounds `n` and `m` cannot exceed an engine-built constant (2147483646 on a 64-bit perl 5.44 - the actual limit appears in the error if you exceed it). For "any sane number" this is plenty; patterns hitting the limit are usually generated, not handwritten.

Whitespace inside `{…}` is tolerated but not required. `a{2, 4}` and `a{2,4}` are identical.

```
/[a-z]+\s+\d*/;             # word, spaces, any number of digits
/y(es)?/i;                  # 'y', 'Y', 'yes', or 'YES'
$year =~ /^\d{2,4}$/;       # 2, 3, or 4 digits
$year =~ /^\d{4}$|^\d{2}$/;# better: exactly 2 or exactly 4
```

The quantifier applies to the atom immediately before it:

- `ab?` - a then optional b.
- `(ab)?` - optional ab.
- `[ab]?` - optional a or b.

Group or class, then quantify.

Greedy by default

Basic quantifiers are *greedy*: they grab as much as the string allows and only give back if the rest of the pattern cannot match otherwise. Consider:

```
my $x = "the cat in the hat";
$x =~ /^(.*)(cat)(.*)$/;
# $1 = 'the '
# $2 = 'cat'
# $3 = ' in the hat'
```

Here `.*` in the middle works as you'd expect - it stops at `cat` because `cat` is the only place where the rest of the pattern can match. Now compare:

```
$x =~ /^(.*)(at)(.*)$/;
# $1 = 'the cat in the h'
```

(continues on next page)

(continued from previous page)

```
# $2 = 'at'
# $3 = ''
```

There are *two* places where `at` occurs - at the end of `cat` and at the end of `hat`. The first `.*` grabs as much as possible while still leaving room for `at` somewhere, so it grabs up to the *last* `at`.

Greedy gives back only what it must

```
"about 24 characters long" =~ /^.*([0-9]+)/;
# $1 = '4'
```

The greedy `.*` first matched the whole string. To make `[0-9]+` match, it had to give back enough characters to expose at least one digit. It gave back exactly one, leaving 4 to capture. The intuition that `.*` "stops at the digits" is wrong - it grabs everything, then the engine asks how little it can return.

Greediness vs. matching

The engine does not always match what the greedy interpretation suggests:

```
"27.625" =~ /(\d\d)([1-9]?)(\d+)/;
# $1 = '62'
# $2 = ''
# $3 = '5'
```

The first try fails: `\d\d` can match 27, but the next character is `.`, so neither branch of `[1-9]?` lets `\d+` proceed. Earliest-position-wins keeps moving the start until `\d\d` lands on 62. There, `[1-9]?` greedily tried 5, but then `\d+` had no remaining digit - so it gave the 5 back and went empty, leaving 5 for `\d+`. Optional quantifiers prefer the *longer* alternative (taking the character) but will give it up if necessary; the ordering rule "earliest position wins, then longest at that position" only constrains the *whole match*, not its individual captures.

Principles of the match

With greedy quantifiers, the engine follows four principles in order:

1. **Earliest position wins.** The whole pattern is tried starting at position 0, then 1, then 2, until a match is found. The earliest starting position always wins.
2. **Leftmost alternative wins** (in `a|b|c`).
3. **Greedy quantifiers grab as much as possible** while still letting the rest of the pattern match.
4. **Leftmost greedy quantifier gets priority.** If two `.*` compete for the same characters, the first one takes them.

Examples:

```
my $x = "The programming republic of Perl";

$x =~ /^(.+)(e|r)(.*)$/;
# $1 = 'The programming republic of Pe'
# $2 = 'r'
# $3 = 'l'
# .+ is leftmost greedy; grabs everything it can while leaving
# room for (e|r) somewhere.

$x =~ /(m{1,2})(.*)$/;
# $1 = 'mm'    -- m{1,2} matches at first 'm' in 'programming',
# $2 = 'ing republic of Perl'
#                takes the maximum 2.

$x =~ /.*(m{1,2})(.*)$/;
# $1 = 'm'
# $2 = 'ing republic of Perl'
# .* grabs all the way to the last 'm', leaving only one 'm' for
# m{1,2}.
```

Non-greedy quantifiers

Appending ? to any quantifier flips it from *grab as much as possible* to *grab as little as possible*.

Non-greedy	Meaning
a??	0 or 1, prefer 0
a*?	0 or more, prefer fewer
a+?	1 or more, prefer fewer
a{n,m}?	n to m, prefer n

The overall match still has to succeed, so the engine will *expand* the non-greedy quantifier one step at a time until it does.

```
my $x = "The programming republic of Perl";

$x =~ /^(.+?)(e|r)(.*)$/;
# $1 = 'Th'
# $2 = 'e'
# $3 = ' programming republic of Perl'
# .+? grabs as little as possible while allowing the match.
```

Non-greedy is usually what you want when scanning between delimiters:

```
my $html = '<b>bold</b> and <i>italic</i>';
while ($html =~ /<(\w+)>(.*?)<\/\1>/g) {
    print "$1: $2\n";
}
```

(continues on next page)

(continued from previous page)

```
# b: bold
# i: italic
```

With greedy `.*` the match would swallow the first `</b>` and stop at the final `</i>`, mangling the capture.

When non-greedy loses to a negated class

A common idiom: `<.*?>` to match a tag. A common bug: the same pattern, applied to broken input, expands the non-greedy quantifier across content it shouldn't. The cleaner form uses a negated character class:

```
"<a> body </a>"  =~ /<.+?>/;        # matches '<a>'  - fine
"<a> <b>foo"     =~ /<.+?>foo/;     # matches '<a> <b>foo' - wrong
"<a> <b>foo"     =~ /<[^>]+>foo/;   # matches '<b>foo' - locked to␣
↪one tag
```

When a token after `>` is required, the non-greedy form *expands* under backtracking pressure: the engine first tried `<a>`, found no `foo`, and forced `.+?` to grow across the `>` and beyond. The negated class cannot give ground that way - `[^>]` is a hard barrier, so the engine bumps the start position to `<b>` instead of swallowing intervening `>`s.

A negated class is also faster: it participates in the engine's simple-repetition fast path, which the non-greedy form disables.

Use `.+?` only when no negated class can express the same requirement. Use `[^X]+` whenever the delimiter is a single character.

Possessive quantifiers

Adding `+` after a quantifier (not `?`) makes it *possessive*: greedy, but it refuses to give anything back on backtracking. Once it matches, those characters are out of play for the rest of the pattern.

Possessive	Meaning
`a?+`	0 or 1, do not give back
`a*+`	0 or more, do not give back
`a++`	1 or more, do not give back
`a{n,m}+`	n to m, do not give back

The payoff is speed - on patterns that *should* fail, possessive quantifiers fail *immediately* instead of exhaustively backtracking.

```
# Ordinary greedy: backtracks once per character on 'abc   ' when
# the second \w+ cannot match.
/^\w+\s+\w+$/;

# Possessive: once \w+ claims the word characters, it keeps them.
```

(continues on next page)

(continued from previous page)

```
# No backtracking, no quadratic blowup on pathological input.
/^\w++\s+\w+$/;
```

The classic `'aaaa' =~ /a++a/`:

```
'aaaa' =~ /a++a/;   # never matches
```

`a++` gobbles all four a's. The trailing `a` then has nothing to match. Because `a++` refuses to give back, the engine fails immediately. (The same pattern with greedy `a+a` matches `'aaaa'` after backtracking.)

A common idiom for matching quoted strings without backtracking catastrophe:

```
/"(?:[^"\\]++|\\.)*+"/;
```

Each iteration of the inner group either gobbles an unlimited run of non-quote, non-backslash characters (possessively), or a single escape. Neither alternative is willing to give back, so failure is fast. The full treatment of this pattern lives in the *performance* chapter.

Possessive equals atomic-group sugar

Possessive quantifiers are exact syntactic sugar for an atomic group around the quantified atom. The following are equivalent:

Possessive form	Atomic-group form
`PAT*+`	`(?>PAT*)`
`PAT++`	`(?>PAT+)`
`PAT?+`	`(?>PAT?)`
`PAT{n,m}+`	`(?>PAT{n,m})`

The possessive notation is shorter; the atomic-group notation generalises further (you can wrap arbitrary alternations, not just quantified atoms). The *groups and captures* chapter covers atomic groups in detail.

Illegal possessive-and-non-greedy combinations

Possessive and non-greedy are contradictory; Perl rejects the combinations. There is an equivalent legal form for each:

Illegal	Legal equivalent
`X??+`	`X{0}`
`X+?+`	`X{1}`
`X{n,m}?+`	`X{n}`

If you find yourself wanting "non-greedy *and* possessive", what you actually want is a fixed count.

Choosing the right quantifier

- **Greedy** - the default, and right most of the time.
- **Non-greedy** - when scanning between a start and end marker that can both appear legitimately multiple times.
- **Possessive** - when the match either works in exactly one way or fails, and speed on failure matters.

When in doubt, write it greedy and add a test that fails if you got it wrong.

Zero-width quantification

A quantifier on a zero-width assertion is almost always a mistake. `\b*` is syntactically legal but matches the empty string at any word boundary, which is not useful. The engine rejects some of these explicitly with a "matches null string many times" warning under `use warnings`; for the rest, the results are usually not what the author meant.

The same caveat applies to quantified groups whose inner content can match nothing. If `(\w*)*` reaches an iteration where `\w*` matched zero characters, the outer `*` would loop forever. Perl breaks this loop automatically - see the *performance* chapter on zero-length-match termination.

Worked examples from `perlre`'s backtracking section

Eight variants of "find the digits at the end of a string". Each fails or succeeds for a different reason; reading them as a group is a fast way to internalise greedy and non-greedy behaviour.

```
$_ = "I have 2 numbers: 53147";

# Pattern              $1 captured              $2 captured
/(.*)(\d*)/         => "I have 2 numbers: 53147"  ""
/(.*)(\d+)/         => "I have 2 numbers: 5314"   "7"
/(.*?)(\d*)/        => ""                          ""
/(.*?)(\d+)/        => "I have "                   "2"
/(.*)(\d+)$/        => "I have 2 numbers: 5314"   "7"
/(.*?)(\d+)$/       => "I have 2 numbers: "       "53147"
/(.*)\b(\d+)$/      => "I have 2 numbers: "       "53147"
/(.*\D)(\d+)$/      => "I have 2 numbers: "       "53147"
```

Reading each:

- `(.*)(\d*)` - greedy `.*` eats everything; `\d*` is satisfied by the empty string. Matches but neither capture is what the author wanted.
- `(.*)(\d+)` - `.*` gives back enough to expose at least one digit, but only the *last* digit of the run.
- `(.*?)(\d*)` - both non-greedy: prefer the empty match; both captures are empty.
- `(.*?)(\d+)` - first non-greedy, second greedy. `.*?` expands until `\d+` can grab 2. Stops at the first digit run.

- `(.*)(\d+)$` - anchored: `\d+` must be at end. Greedy `.*` gives back digits until the last one is at end-of-string; result is the same "last-digit only" oddity.
- `(.*?)(\d+)$` - non-greedy and anchored: `.*?` expands until `\d+$` can grab the *whole* trailing digit run. Correct.
- `(.*)\b(\d+)$` - anchored, with `\b` to force the boundary before the digit run. Equivalent.
- `(.*\D)(\d+)$` - explicit non-digit before the digit run. Equivalent.

The lessons:

1. The quantifier (`.*` vs `.*?`) controls *greediness*, but anchoring (`$`, `\b`, `\D`) is what controls *which run is matched*.
2. Greedy is rarely "wrong" by itself; the wrongness is a missing anchor.

Cross-engine notes

Quantifier syntax is one of the most-divergent surfaces between regex engines. The full table is in the *cross-engine* chapter; the headline:

- **POSIX BRE** treats bare `+` and `?` as *literal characters*. Use `\+` and `\?` for the metacharacter forms.
- **POSIX ERE** has `+`, `?`, `{m,n}` as metacharacters (Perl-shaped) but no possessive quantifiers and no atomic groups.
- **RE2 / Go** has `+`, `?`, `*`, `*?`, but no possessive quantifiers, no atomic groups, and no backreferences. Linear-time guarantee replaces the need for them.

See also

- The *performance* chapter - catastrophic backtracking, unrolling the loop, and where possessive quantifiers earn their keep.
- The *groups and captures* chapter - atomic groups `(?>…)`, the general form of "no backtracking inside this construct".
- The *character classes* chapter - `[^X]+` as a faster, safer alternative to `.*?`.
- The *cross-engine* chapter - quantifier availability across families.
- `m`, `s`, `qr` - operator references.

Groups and captures

Grouping does two things, which are easy to confuse: it turns a sequence of pattern elements into a single unit for quantification and alternation, and it saves what that unit matched for later use.

```
/house(cat|keeper)/;       # 'house' followed by 'cat' or 'keeper'
/(ab){3}/;                 # 'abababab'
/(\d{3})-(\d{4})/;         # capture two groups separated by '-'
```

Beyond the plain (…) capturing form, Perl provides a small zoo of parenthesised constructs: non-capturing groups for clean grouping, named captures for self-documenting patterns, atomic groups for controlling backtracking, recursive subpatterns for nested structures, conditional patterns for branching on prior captures, and the branch-reset construct for parallel alternatives. This chapter covers all of them.

Capturing groups: $1, $2, ...

Every pair of unescaped parentheses in a pattern opens a capturing group. After a successful match the matched text of the nth group is in $n:

```
if ($time =~ /(\d\d):(\d\d):(\d\d)/) {
    my ($hours, $minutes, $seconds) = ($1, $2, $3);
}
```

In list context, a match returns the list of captured strings directly:

```
my ($h, $m, $s) = $time =~ /(\d\d):(\d\d):(\d\d)/;
```

If the pattern fails, the list is empty - a useful idiom for “parse or give up”:

```
my ($h, $m, $s) = $time =~ /(\d\d):(\d\d):(\d\d)/
    or die "not a time: $time";
```

There is no upper limit on the number of capture groups. Groups are numbered with the leftmost open parenthesis as group 1, the next as group 2, and so on:

```
/(ab(cd|ef)((gi)|j))/
  1  2      34
```

$1 captures the outer group, $2 the first inner, $3 the next, $4 the innermost.

Captured the empty string vs. did not match at all

A capture group that did not participate in the match has $n *undefined*. A group that participated and matched an empty string has $n defined and equal to "". The distinction matters:

```
"aba" =~ / a (x)* b \g1 a /x;   # does NOT match
"aba" =~ / a (x)? b \g1 a /x;   # does NOT match
"aba" =~ / a (x*) b \g1 a /x;   # matches; $1 = ""
"aba" =~ / a (x?) b \g1 a /x;   # matches; $1 = ""
```

In the first two cases, the quantifier is *outside* the group, so the group itself never closed (the engine matched zero iterations - the group was not entered at all). The backreference \g1 therefore has no value to compare against and fails. In the second two cases, the quantifier is *inside* the group, so the group ran exactly once and captured an empty string; \g1 matches the same empty string at that position.

The lesson: when a group is optional, put the quantifier inside if you want the empty-match-counts-as-match semantics.

Always check captures with `defined`, not for truth - an empty capture is defined but false under string semantics:

```
if ("x" =~ /(a)?(x)/) {
    print "1 is $1\n" if defined $1;   # $1 is undef here
    print "2 is $2\n" if defined $2;
}
```

Failed matches do not reset capture variables

If a match fails, `$1`, `$2`, ... keep their previous values from the last *successful* match in the same scope. This is a feature: it lets you write a series of more specific patterns and refer to the best match's captures afterwards.

```
"foo" =~ /(\w+)/ and "" =~ /(\d+)/;   # second fails
print $1;   # "foo" - first match's capture survives
```

It is also a common source of confusion. Always check the match's return value before reading the capture variables.

Non-capturing groups: `(?:…)`

If you only need the grouping for quantification or alternation, and don't want the capture, use `(?:…)`:

```
/(?:ab){3}/;            # 'ababab', no capture
/(?:\d+\.)*\d+/;        # a dotted decimal, no captures at all
```

Non-capturing groups are a small speed win and a larger clarity win. They signal "this grouping is for syntax, not data". They also prevent renumbering of the capturing groups you do care about:

```
# match a number - $1 = whole, $2 = optional exponent value
/([+-]?\ *(?:\d+(?:\.\d*)?|\.\d+)(?:[eE]([+-]?\d+))?)/;
```

Without the `(?:…)` wrappings, `$2`, `$3`, `$4` would all be set and the intended `$2` (the exponent) would shift to `$5`.

The scoped form `(?flags:…)` (e.g. `(?i:cat)`, `(?xms:…)`) attaches modifiers to the inner pattern only and is also non-capturing - see the *modifiers* chapter.

`split` also benefits from `(?:…)`. `split /(?:\s+)/` separates on runs of whitespace without inserting the separators into the output; `split /(\s+)/` leaves them in alternating positions.

Named captures

`(?<name>…)` or `(?'name'…)` names a group. Its match is accessible through the hash `%+`:

```
if ("2026-04-23" =~ /(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})/)
 ↪{
    print "year = $+{year}\n";    # 2026
    print "month = $+{month}\n";  # 04
    print "day = $+{day}\n";      # 23
}
```

Named groups also populate `$1`, `$2`, ... in the usual left-to-right order, so code that uses both conventions works. Inside the pattern, reference a named group with `\k<name>` (or any of the brace/quote forms below):

```
/(?<quote>["'])(.*?)\k<quote>/;   # same quote at start and end
```

Names follow Perl-identifier rules (`[_A-Za-z][_A-Za-z0-9]*`); they cannot begin with a digit and cannot contain hyphens.

If two distinct groups share a name, `$+{name}` refers to the *leftmost defined* group in the match. This is unusual to want *outside* a branch reset (`(?|...)`), where shared names are idiomatic.

`(?P<name>...)` - Python/PCRE compatibility

For programmers porting from Python's `re` module or PCRE, Perl accepts the Python-style spellings:

Python form	Perl equivalent
`(?P<NAME>...)`	`(?<NAME>...)`
`(?P=NAME)`	`\k<NAME>`
`(?P>NAME)`	`(?&NAME)`

The Python forms work but are not idiomatic Perl; prefer the native forms in new code.

Backreferences

A backreference in a pattern demands that a later position match the *same text* an earlier group captured - not the same pattern, the same actual characters.

Form	Refers to
`\1` ... `\9`	first through ninth capturing group (legacy form)
`\g1`, `\g2`	numbered capture; equivalent to `\1`, `\2`
`\g{1}`	brace form; required when digits would otherwise follow
`\g-1`	most-recently-opened capturing group (relative)
`\g{-2}`	second-most-recently-opened group
`\k<name>`	named capture
`\k'name'`	same, single-quote form
`\k{name}`	same, brace form (allows surrounding spaces)
`\g{name}`	named, alternative spelling

Examples:

```
# Match a three-letter word followed by a space and the same word.
"the the other day" =~ /\b(\w{3})\s\1\b/;   # $1 eq 'the'

# Match a four-letter, three-letter, two-letter, or one-letter
# run followed by itself.
/^(\w{1,4})\1$/;    # 'beriberi', 'booboo', 'coco', 'mama', 'papa'
```

`\g{…}` versus `\1` for disambiguation

Use `\g{…}` (or `\k<…>`) when digits or octal-looking characters follow the reference, to avoid ambiguity:

```
/(\d)abc\g{1}23/;    # the '1' refers to group 1, '23' is literal
/(\d)abc\123/;       # '\123' is octal 0x53 ('S'), not group 1
```

Perl's rule for the bare numbered form: `\1` through `\9` always mean backreferences. `\10`, `\11`, … mean a backreference *only if* that many capture groups have opened earlier in the pattern; otherwise they are octal character literals. This is why `\g{...}` is the safer form when the pattern is built by concatenation.

If you use the brace form, optional surrounding spaces are permitted - `\g{ -1 }` and `\k{ name }` are valid.

Relative backreferences

`\g-1`, `\g{-1}` refer to the *immediately previous* capturing group; `\g{-2}` refers to the one before that; and so on. The distance is counted by *opened* parens, including unclosed ones. This matters when a pattern fragment is interpolated inside another:

```
my $pair = '([a-z])(\d)\g{-1}\g{-2}';   # a11a, g22g, x33x, ...

# Embed it: outer group shifts numbering by 1, but relative
# backreferences still work:
"code=e99e" =~ /^(\w+)=$pair$/;   # matches
```

Without relative backreferences, this would require knowing how many groups precede `$pair` at every interpolation site.

Named and relative references make long patterns robust against copy-paste.

Atomic groups: `(?>…)`

`(?>…)` is a non-capturing group whose contents, once matched, are *committed*. The engine cannot backtrack into the group on failure - only past it as a whole.

```
"aaab" =~ /a*ab/;       # matches: a* gives back one 'a'
"aaab" =~ /(?>a*)ab/;   # does not match: a* takes all, refuses to␣
↪give
```

The full treatment lives in the *performance* chapter. The construct is documented here because its syntax is a parenthesised group; structurally it belongs in the captures chapter alongside `(?:…)`.

Possessive quantifiers (`*+`, `++`, `?+`, `{n,m}+`) are exact syntactic sugar for `(?>…)` around the quantified atom. The following are equivalent:

Possessive	Atomic-group form
`PAT*+`	`(?>PAT*)`
`PAT++`	`(?>PAT+)`
`PAT?+`	`(?>PAT?)`
`PAT{n,m}+`	`(?>PAT{n,m})`

The long-form spelling `(*atomic:…)` is also accepted.

Recursive subpatterns

Perl regexps can refer back to a capture group as if it were a subroutine call. The construct re-runs the captured pattern at the current position. This makes genuinely recursive structures matchable - balanced parentheses, S-expressions, nested brackets - without falling out of the regex DSL.

Form	Recurses to
`(?R)`	the entire pattern
`(?0)`	same as `(?R)`
`(?1)`	group 1
`(?2)`	group 2 (and so on)
`(?-1)`	most-recently-opened group (relative)
`(?+1)`	next group to be opened (forward relative)
`(?&NAME)`	named group
`(?P>NAME)`	same as `(?&NAME)` (Python-compatible)

Note: relative recursion counts unclosed groups, just as relative backreferences do. `(?-1)` always means the most recently opened group whether or not it has closed.

A balanced-paren matcher:

```
my $bal = qr/
    (?(DEFINE)
        (?<paren>
            \(                  # opening paren
            (?:
                [^()]++         # non-paren run, possessive
              | (?&paren)       # or a balanced sub-group, recursively
            )*+
            \)                  # closing paren
        )
    )
```

(continues on next page)

(continued from previous page)

```
    (?&paren)
/x;

"((a)(b(c)))" =~ /^$bal$/;   # matches
```

The `(?(DEFINE)...)` block declares a named subpattern without matching anything itself; the `(?&paren)` after it invokes that subpattern, and the recursive `(?&paren)` inside the body is what gives the unbounded depth. `(?R)` would not work here - it recurses the *whole* enclosing pattern, including the `^` and `$` anchors that get added at the call site, which forces every level of recursion to demand start- and end-of-string.

A more nuanced recursive pattern: match a function `foo(...)` where the argument may itself contain balanced parens.

```
my $re = qr/(             # group 1: full function call
              foo
              (           # group 2: parens with content
                \(
                  (       # group 3: contents of parens
                    (?:
                       (?> [^()]+ )   # non-paren without␣
↪backtracking
                     |
                       (?2)           # recurse to group 2
                    )*
                  )
                \)
              )
            )/x;

'foo(bar(baz)+baz(bop))' =~ /$re/ and
    print "1: $1\n2: $2\n3: $3\n";
# 1: foo(bar(baz)+baz(bop))
# 2: (bar(baz)+baz(bop))
# 3: bar(baz)+baz(bop)
```

Capture state inside recursion

When a group recurses into itself, captures set inside the recursion are *not* visible to the caller after the recursion returns. The recursive call has its own capture state, which is discarded on return. This is why most recursive patterns wrap a secondary capture group around the recursive call when the matched text is needed:

```
/(?<NAME>(?&NAME_PAT))(?<ADDR>(?&ADDRESS_PAT))
 (?(DEFINE)
   (?<NAME_PAT>....)
   (?<ADDRESS_PAT>....)
 )/x
```

Here `$+{NAME}` is the outer capture; `$+{NAME_PAT}` is undefined because it lived only

inside the recursion.

The recursion-depth ceiling is compiled into the engine. Patterns that recurse without consuming input fail immediately rather than running forever - the engine detects the cycle.

`(DEFINE)` blocks

The `(?(DEFINE)…)` block holds named subpatterns that never run on their own - they are only invoked through `(?&NAME)`. This is how to write a regex that looks like a small grammar:

```
my $email = qr/
    \A (?&LOCAL) @ (?&DOMAIN) \z
    (?(DEFINE)
        (?<LOCAL>   [\w.+-]+ )
        (?<DOMAIN>  (?&LABEL) (?: \. (?&LABEL) )+ )
        (?<LABEL>   [a-zA-Z0-9] (?: [a-zA-Z0-9-]* [a-zA-Z0-9] )? )
    )
/x;
```

The DEFINE block at the end keeps the named subpatterns in one place; the top of the pattern reads as a clean specification.

Two cautions:

- Subpatterns in a DEFINE block *count toward* the absolute and relative numbering of capture groups in the surrounding pattern. Name everything in the DEFINE so you don't have to count.
- The optimiser is less effective on DEFINE-style patterns than on the equivalent inlined form. They run, but not always at top speed.

Conditional patterns: `(?(cond)yes|no)`

A conditional pattern picks between two sub-patterns based on a runtime test:

- `(?(N)YES|NO)` - match YES if group N matched something, else NO.
- `(?(<NAME>)YES|NO)` - same, but by name.
- `(?(?=LOOK)YES|NO)` - match YES if the lookahead succeeds.
- `(?(?{CODE})YES|NO)` - match YES if the code block returns true.
- `(?(R)YES|NO)` - match YES if currently inside a recursion.
- `(?(R1)YES|NO)` - match YES if recursing through group 1.
- `(?(R&NAME)YES|NO)` - match YES if recursing through named group.

The NO branch is optional; missing NO is treated as "always match".

Worked example: optionally-parenthesised text. If the input opens with `(`, require a closing `)`. Otherwise allow no parens:

```
m{ ( \( )?            # optional opening paren, group 1
   [^()]+             # body (no parens)
   (?(1) \) )         # if group 1 matched, require closing paren
}x;
```

Without conditional patterns, this would need an alternation covering both shapes; the conditional version exposes the intent.

Worked example: balanced grammar with named recursion. Useful inside a DEFINE block for context-sensitive validation:

```
qr/
    (?<expr>
        (?<atom>  [a-z]+ | \( (?&expr) \) )
        (?: \s* [+*] \s* (?&atom) )*
    )
/x
```

Conditionals are the part of regex that most resembles a real programming language; reach for them when an alternation of mutually-exclusive shapes would be unwieldy.

Branch reset: `(?|…)`

Inside `(?|…)`, every alternative branch starts numbering its captures at the same slot. After the group, numbering resumes one past the maximum across all branches. The full treatment is in the *alternation* chapter; the capture-numbering implications belong here.

```
# Before  ---------------branch-reset----------- after
/ ( a )  (?| x ( y ) z | (p (q) r) | (t) u (v) ) ( z ) /x
# 1            2         2  3        2     3     4
```

After this pattern matches:

- `$1` is always the leading `a`.
- `$2` is `y`, `(p q r)`, or `t` depending on branch.
- `$3` is undef from branch 1, `q` from branch 2, `v` from branch 3.
- `$4` is the trailing `z`.

If you use named captures inside `(?|…)`, use the *same names in the same order* in every branch:

```
/(?|  (?<a> x ) (?<b> y )
   |  (?<a> z ) (?<b> w )) /x;
```

Mixing names across branches works (Perl resolves to the leftmost defined name) but produces surprising results: every name references the same slot, so `$+{a}` and `$+{b}` may have the same value across different branches.

Position arrays: @- and @+

After a successful match, `@-` and `@+` hold the start and end offsets of the whole match and of each capture group:

- `$-[0]`, `$+[0]` - offsets of the whole match.
- `$-[n]`, `$+[n]` - offsets of the nth capture, or undef if the group did not participate.

```
my $s = "Mmm...donut, thought Homer";
if ($s =~ /^(Mmm|Yech)\.\.\.(donut|peas)/) {
    for my $i (1 .. $#-) {
        printf "Match %d: %s at (%d,%d)\n",
               $i,
               substr($s, $-[$i], $+[$i] - $-[$i]),
               $-[$i], $+[$i];
    }
}
# Match 1: Mmm at (0,3)
# Match 2: donut at (6,11)
```

Offsets are often easier than substrings when you need to modify the original string at the matched position.

@{^CAPTURE} - captures as an array

Perl exposes all numbered captures as a single array `@{^CAPTURE}`, indexed from 0 (where index 0 is $1, index 1 is $2, ...):

```
$string =~ /$pattern/ and my @captured = @{^CAPTURE};
```

This is convenient when the number of captures is variable or unknown - code that takes captures from a user-supplied pattern no longer has to count `(` to know how many $1/$2/... to read.

Subscripting requires the demarcated-curly form:

```
print "${^CAPTURE[0]}";    # equivalent to $1
```

`%{^CAPTURE_ALL}` and `%{^CAPTURE_NAMES}` exist for named-capture introspection - see `perlvar`.

Prematch, match, postmatch

Perl sets three special scalars after each match that expose the surrounding text:

- `$`` - everything before the match (the *pre-match*).
- `$&` - the match itself.
- `$'` - everything after the match (the *post-match*).

```
"the cat caught the mouse" =~ /cat/;
# $`   = 'the '
```

(continues on next page)

(continued from previous page)

```
# $&   = 'cat'
# $'   = ' caught the mouse'
```

The named variants `${^PREMATCH}`, `${^MATCH}`, `${^POSTMATCH}` are equivalent. Both forms are zero-cost; use whichever reads better.

`$+` and `$^N`

`$+` holds the match of the highest-numbered capture group that succeeded.

`$^N` holds the match of the most-recently-closed capture group - the rightmost `)` that completed. This is exactly what you want inside a `(?{...})` code block to access the latest capture without counting parens:

```
$_ = "The brown fox jumps over the lazy dog";
/the (\S+)(?{ $color = $^N }) (\S+)(?{ $animal = $^N })/i;
print "color = $color, animal = $animal\n";
# color = brown, animal = fox
```

See the *performance* chapter for the full embedded-code story.

POSIX subexpression maximisation - what other engines do

Some engines (any conformant POSIX NFA, plus engine implementations designed to mimic POSIX, plus most DFA hybrids) follow a different rule for which match wins when multiple are possible: *longest-leftmost*, with the constraint that *each subexpression captures the longest substring it can*.

For the pattern `(to|top)(o|polo)?(gical|o?logical)` against `topological`:

- Perl (and PCRE2, and any Traditional NFA) tries alternatives left-to-right. `(to|top)` matches `to`; the engine pushes on. The next character is `p`, so `(o|polo)` cannot take `o`; it takes `polo`. `(gical|o?logical)` matches `gical`. Done: `to` `polo` `gical`.
- POSIX semantics requires the *overall* match be longest, and among matches of equal length, each group's capture to be as long as it can be, from left to right. The first group takes `top`; that leaves `ological`, so the result is `top` `o` `logical`.

Modern engines essentially never use POSIX-NFA semantics, but some DFA-based tools (`grep`, `awk`) approximate it. The *cross-engine* chapter has the full table.

Cross-engine: captures and backreferences

This is the one feature where engines diverge most. The full table is in the *cross-engine* chapter; the relevant rows extracted:

Feature	Perl 5.44	PCRE2	Emacs	POSIX BRE	POSIX ERE	RE2/Go
Numbered captures	yes	yes	yes	yes	strict no, GNU yes	yes (no backrefs)
Backreferences in pattern (\1, ...)	yes	yes	yes	yes (\1–\9)	strict no	NO
Named captures (?<name>...)	yes	yes	NO	NO	NO	yes
Branch reset (?|...)	yes	yes	NO	NO	NO	NO
@{^CAPTURE} array	yes	NO	NO	NO	NO	NO (different API)
Recursive subpatterns (?R) etc.	yes	yes	NO	NO	NO	NO
Conditional patterns (?(c)y|n)	yes	yes	NO	NO	NO	NO

The single most important takeaway: **RE2 / Go regexp does not support backreferences at all**. This is not an oversight; it is the price RE2 pays for guaranteed linear-time matching. Patterns that need backreferences are not regular languages and cannot be matched by a DFA in linear time.

If you are porting Perl regexps to a Go service, audit them for backreferences before assuming the conversion is mechanical. Patterns like `/(?<a>\w+) and \k<a>/` simply cannot be expressed in RE2.

See also

- The *alternation* chapter - branch reset and the alternative-numbering implications of (?|...).
- The *performance* chapter - atomic groups, recursive patterns in worked examples, embedded code.
- The *modifiers* chapter - /n to suppress all captures.
- perlvar - %+, %-, $&, $^N, @{^CAPTURE} and friends.

Alternation

Alternation is the | operator. It picks between two or more sub-patterns at the same position.

```
"cats and dogs" =~ /cat|dog|bird/;    # matches 'cat'
"cats and dogs" =~ /dog|cat|bird/;    # matches 'cat'
```

The order of the alternatives does not change *where* the overall pattern matches. The engine still honours "earliest position wins" - both patterns above match at position 0 because that's the earliest position where any alternative can match.

Leftmost alternative wins at a given position

Within a single starting position, alternatives are tried left to right and the first one that succeeds is used:

```
"cats" =~ /c|ca|cat|cats/;     # matches 'c' - first alternative
↪wins
"cats" =~ /cats|cat|ca|c/;     # matches 'cats' - first wins, longer
```

If one alternative is a prefix of another and you want the longer match, put it first. The engine does not look past the first successful alternative at the current position.

This is *Traditional NFA* behaviour, and it is what Perl, PCRE2, Python, and most modern engines implement. POSIX-conformant engines (notably some `awk` and `grep` implementations) follow the *longest-leftmost* rule instead - they would match `cats` regardless of alternative order. The *cross-engine* chapter has the comparison.

Friedl puts it succinctly:

> Greedy alternation is non-greedy in a Traditional NFA.

The pattern `tour|to|tournament` against `three tournaments won` matches `tour`, not `tournament`. The first alternative succeeds and the engine commits to it; longer alternatives further along the list are never tried.

Implications for ordering

An alternation that is part of a larger pattern can have its performance and correctness shaped by the order of alternatives:

- **Specificity first.** When you want the longest of several prefixes, put the longest first. `/web|website|websites/` matches `web` even on input `website`; `/websites|website|web/` matches `websites`.
- **Common case first.** Alternation is tried left-to-right; if 90% of your inputs hit alternative 3, the engine wastes time on alternatives 1 and 2 every time. Reorder by frequency.
- **Sibling captures.** When the alternatives capture, the order affects which `$n` is set - see *Alternation and capturing* below.

Combining-pieces formal rule

`perlre`'s "Combining RE Pieces" gives the precise statement underlying "leftmost wins". For two pattern pieces S and T:

> When S can match, it is a better match than when only T can match.

That is the formal version of the rule. "Better" means the engine prefers it. For two S matches, the same internal ordering applies (greediness rules within the alternative); likewise for T matches. Across alternatives, *the existence of a successful S match excludes consideration of T*.

This is why S|T cannot be reordered by the engine on its own: Perl does not search for the *best* alternative across the disjunction - it commits to S whenever S succeeds.

Grouping vs. alternation precedence

| has very low precedence. It splits the pattern at the *outermost* level containing it:

```
/ab|cd/;       # 'ab' OR 'cd'
/^ab|cd$/;     # '^ab' OR 'cd$' - probably not what you meant!
/^(ab|cd)$/;   # '^' + ('ab' or 'cd') + '$' - what you meant
```

To constrain alternation to part of a pattern, wrap it in a group. Non-capturing `(?:…)` is preferred unless you need the capture:

```
/house(?:cat|keeper)/;      # 'housecat' or 'housekeeper'
/house(cat|keeper)/;        # same, but $1 will be 'cat' or 'keeper'
```

The group creates a local scope for |. Outside the group | resumes its top-level role:

```
/^(?:foo|bar|baz)$|^xyz$/;   # ('foo'/'bar'/'baz') or 'xyz'
```

Empty alternatives

An empty alternative matches the empty string - a useful trick for “this or nothing”:

```
/house(cat|)/;          # 'housecat' or 'house'
/(19|20|)\d\d/;         # '19xx', '20xx', or just 'xx'
```

Modern style prefers `(?:…)?` over `(?:…|)`; they are equivalent, but the ? form is clearer:

```
/house(?:cat)?/;        # same as house(cat|), no capture
```

Watch for the backtracking cost when an empty alternative is combined with a quantifier - the engine can re-explore the same position many times. See the *performance* chapter on zero-length-match termination.

Cross-engine note

Strict POSIX, `lex`, and most older `awk` implementations *disallow* empty alternatives - `(this|that|)` is a syntax error. Perl 5.44, PCRE2, Rust’s `regex` crate, Python `re`, and modern engine implementations accept them. If a pattern needs to be portable to older POSIX tools, write `(?:this|that)?` instead - it expresses the same idea and is portable to the engines that reject an empty alternative.

Alternation inside character classes

Character classes are almost always what you want when alternating between single characters. `/a|b|c/` and `/[abc]/` match the same strings, but `[abc]` is faster, terser, and clearer:

```
/a|b|c/;      # works, but verbose
/[abc]/;      # use this
```

Alternation is for alternatives longer than one character (or ones that are themselves patterns). When each alternative is a single character, reach for a class. The engine

treats a class as a *single* atomic choice; an alternation as N choices to be explored in order.

Common-prefix factoring

A pattern like `/this|that|then|those/` defeats the engine's fixed-string-check optimisation: there is no literal prefix the engine can scan for cheaply. Refactoring to expose the common prefix turns the alternation into something the optimiser can work with:

```
/this|that|then|those/;       # no common prefix visible
/th(?:is|at|en|ose)/;         # common prefix 'th' exposed
```

The two patterns match the same strings, and the second is materially faster on large inputs. The engine scans for `th` using Boyer-Moore, then runs the small alternation only at candidate positions.

The general technique:

1. Find the longest literal prefix common to all alternatives.
2. Lift it outside the alternation.
3. Wrap the remainder in `(?:…)` so the alternation stays localised.

For lists of words with a few different prefixes, you can apply the rewrite recursively. For very long lists, see the *performance* chapter's section on matching many strings - once the list is in the thousands, a specialised matcher beats any alternation.

Alternation and capturing

Only one alternative inside a group can match at a time, so the group captures the matching alternative:

```
if ("bert" =~ /(cat|dog|bert|ernie)/) {
    print "matched $1\n";        # matched bert
}
```

Sibling groups outside the alternation retain their normal numbering:

```
/^(\w+):\s*(yes|no|maybe)$/;
# $1 = the key, $2 = the verdict
```

Inside a nested alternation, the groups are numbered left to right by opening paren, even across branches:

```
/(a)|(b)/;
# On match of 'a': $1 = 'a', $2 undef
# On match of 'b': $1 undef,  $2 = 'b'
```

Check with `defined $n`, not truth - an empty capture is different from an absent capture.

Branch reset: `(?|…)`

Parallel-capture patterns are the usual reason you pick up `(?|…)`. Inside `(?|…)`, every branch starts numbering its captures at the same slot. After the group, numbering resumes at one past the maximum across all branches.

```
# Without (?|…): need to know which branch matched.
if ($time =~ /(\d\d|\d):(\d\d)|(\d\d)(\d\d)/) {
    my ($h, $m) = ($1, $2);
    ($h, $m) = ($3, $4) unless defined $h;
}

# With (?|…): $1 and $2 come from whichever branch matched.
if ($time =~ /(?|(\d\d|\d):(\d\d)|(\d\d)(\d\d))/) {
    my ($h, $m) = ($1, $2);
}
```

With a trailing fixed piece:

```
if ($time =~ /(?|(\d\d|\d):(\d\d)|(\d\d)(\d\d))\s+([A-Z]{3})/) {
    # $1 = hours, $2 = minutes, $3 = zone (numbered after the group)
    print "hour=$1 minute=$2 zone=$3\n";
}
```

Rules inside `(?|…)`:

- Each branch independently numbers its capturing groups from the current group count.
- After the group, the outer numbering continues at one higher than the maximum count reached in any branch.
- Named groups keep their names; you can repeat a name across branches. Use the *same names in the same order* in every branch, or surprises ensue (see the *groups and captures* chapter).

Branch reset is the cleanest way to express "parse X in one of several equivalent formats, then reach for the same variables afterwards."

Alternation in `split`

`split` takes a regexp pattern, so alternation works there too:

```
my @words = split /\s+|-/, "one-two three  four-five";
# ('one', 'two', 'three', 'four', 'five')
```

If the separator pattern contains capturing groups, split includes the captured text in the output list - often surprising. Use `(?:…)` unless you want that:

```
split /(?:\s+|-)/, "a-b c";   # ('a', 'b', 'c')
split /(\s+|-)/,   "a-b c";   # ('a', '-', 'b', ' ', 'c')
```

The capturing form is occasionally what you want - preserving the exact separators between fields. Mostly you want non-capturing.

Summary

- `|` separates alternatives; leftmost that matches at the current position wins.
- Wrap alternations in `(?:...)` to localise them.
- Prefer character classes over single-character alternations.
- Lift common prefixes outside the alternation when you want the engine's literal-scan optimisation to fire.
- `(?|...)` resets capture numbering across branches - use it when the branches capture the same conceptual fields.

See also

- The *groups and captures* chapter - capture numbering inside `(?|...)`, and the rest of the named-capture machinery.
- The *performance* chapter - ordering by likelihood, common-prefix factoring, and matching many strings.
- The *cross-engine* chapter - alternation syntax differences across engine families.
- `split` - alternation as separator syntax.

Modifiers

Modifiers change how a pattern is matched without changing the pattern itself. They appear after the closing delimiter of a match, substitution, or `qr//`:

```
"Hello" =~ /hello/i;        # case-insensitive match
$x =~ s/foo/bar/g;          # global substitution
my $re = qr/\d+/i;          # compiled pattern with /i
```

The same modifiers can be embedded inside a pattern with `(?flags)` or `(?flags:...)`, localising their effect to part of the pattern.

The full modifier set

Modifier	Effect
`/i`	case-insensitive matching
`/m`	multi-line: `^` and `$` match at every `\n`
`/s`	single-line: `.` matches `\n` too
`/x`	extended: ignore whitespace and `#` comments
`/xx`	extended-extended: also ignore whitespace inside `[…]`
`/g`	global: match as many times as possible
`/c`	do not reset `pos` on failure (with `/g`)
`/r`	`s///r` returns the result instead of modifying
`/e`	`s///e` evaluates the replacement as Perl code
`/ee`	`s///ee` evaluates, then evaluates the result too
`/n`	non-capturing: `(…)` act like `(?:…)`
`/p`	no-op; accepted for compat
`/o`	compile the pattern once (rarely needed; prefer `qr//`)
`/a`	`\d`, `\w`, `\s` restricted to ASCII
`/aa`	`/a`, plus `/i` does not cross ASCII/non-ASCII boundary
`/u`	Unicode semantics regardless of `use utf8`
`/l`	use current locale
`/d`	dual-mode semantics - avoid in new code

The first eight are the everyday modifiers; the charset modifiers `/a`, `/u`, `/l`, `/d` are covered in detail in the *unicode* chapter and summarised again here.

`/i` - case-insensitive

```
"Hello" =~ /hello/i;          # matches
"HELLO" =~ /[A-Z]/i;          # matches - class is insensitive too
"Grüße" =~ /GRÜSSE/i;         # matches under Unicode semantics
```

Unicode case folding includes mappings like ß → `ss`, German Eszett casefolding, and so on. The full table lives in the Unicode standard; for ASCII, `/i` does what you expect.

`/i` carries a negligible performance penalty for ASCII. The implementation folds case at *compile time* - the pattern carries the case-folded character set, the input is read once. Unicode case-folding is more involved (sequences like ß ↔ `ss`), but the extra work is bounded and rarely shows up in practice.

`/m` - multi-line

Changes where `^` and `$` match. Without `/m`, they match only at the outer ends of the string. With `/m`, they match at every embedded newline too.

```
my $x = "first\nsecond\nthird";

$x =~ /^second/;    # does not match
$x =~ /^second/m;   # matches - second is at start of a line
```

(continues on next page)

(continued from previous page)

```
$x =~ /first$/m;    # matches
$x =~ /third$/m;    # matches
```

`\A`, `\z`, `\Z` remain absolute string anchors even under `/m` - see the *anchors and assertions* chapter.

`/s` - single-line

Makes `.` match newline characters too.

```
my $x = "a\nb";
$x =~ /a.b/;        # does not match - . does not cross \n
$x =~ /a.b/s;       # matches
```

`/m` and `/s` are independent. They can both be used on the same match. Despite the names, they do not conflict:

```
$x =~ /^a.b$/sm;    # . matches newline AND ^,$ are line-aware
```

`/x` - extended pattern

Ignores literal whitespace and lets # introduce end-of-line comments. Crucial for any pattern more than a line long.

Before `/x`:

```
/^[+-]?\d+(\.\d*)?([eE][+-]?\d+)?$/;
```

After:

```
/^
    [+-]?           # optional sign
    \d+             # integer part
    (\.\d*)?        # optional fraction
    ([eE][+-]?\d+)? # optional exponent
 $/x;
```

Whitespace in the *pattern* is ignored; whitespace you want to match becomes `\s`, `\ ` , or `[ ]`:

```
/\w+ \s+ \w+/x;      # three tokens: word, space, word (literal␣
↪spaces ignored)
/\w+\s+\w+/x;        # equivalent
/key:[ ]value/x;     # literal space via bracket class
/key:\ value/x;      # literal space via backslash
```

starts a comment that ends at the next newline. To match a literal # under `/x`, escape it or put it in a class.

The "Pattern White Space" set under `/x` follows Unicode UAX#31: SPACE, CHARACTER TABULATION, LINE FEED, LINE TABULATION, FORM FEED, CARRIAGE RETURN,

NEXT LINE, PARAGRAPH SEPARATOR, LINE SEPARATOR. In practice you only meet ASCII space, tab, and newline.

/xx - extended in classes too

Inside `[...]` whitespace is *not* ignored under plain `/x` - `[ab c]` matches a, b, ' ', or c. To also ignore whitespace inside classes, use `/xx`:

```
/[ab c]/xx;          # matches 'a', 'b', or 'c' - space ignored
/[ab\ c]/xx;         # the \ is needed to match a literal space
/[ ! @ ]/xx;         # matches '!' or '@' - visible spacing
```

`/xx` is a superset of `/x`. It is convenient for character classes laid out for readability, especially with shorthand classes:

```
/[ \d \s \-_,. ]+/xx;   # digits, whitespace, common punctuation
```

`/xx` was added in Perl 5.26. Patterns built before that read fine; new code can reach for it freely.

/g - global

In scalar context, keeps a position in the string (`pos $x`) and advances each time the pattern matches, allowing iteration:

```
my $x = "cat dog house";
while ($x =~ /(\w+)/g) {
    print "$1 at ", pos($x), "\n";
}
# cat at 3
# dog at 7
# house at 13
```

In list context, returns all matches at once:

```
my @words = $x =~ /(\w+)/g;   # ('cat', 'dog', 'house')
```

If the pattern contains no captures, list context returns the whole matched text for each match:

```
my @digits = "abc123def456" =~ /\d+/g;   # ('123', '456')
```

With multiple captures, each iteration returns the tuple in order:

```
my @pairs = "a=1,b=2,c=3" =~ /(\w)=(\d)/g;
# ('a', '1', 'b', '2', 'c', '3')
```

`/c` - preserve position on failure

By default, a failed `/g` match resets `pos` to undef. Under `/gc`, `pos` stays at its previous value - crucial for hand-rolled lexers:

```
my $s = "123abc";
while (1) {
    if ($s =~ /\G(\d+)/gc)    { print "num $1\n"; next; }
    if ($s =~ /\G([a-z]+)/gc) { print "word $1\n"; next; }
    last;   # nothing matched; exit
}
```

See the *anchors and assertions* chapter for `\G`.

`/r` - non-destructive substitution

`s///` normally modifies the target string and returns the count. `s///r` leaves the target alone and returns the result:

```
my $name = "  Alice  ";
my $trimmed = $name =~ s/^\s+|\s+$//gr;
# $name is still "  Alice  "
# $trimmed is "Alice"
```

Enables substitution chains without intermediate variables:

```
my $clean = $input
    =~ s/\s+/ /gr          # collapse runs of whitespace
    =~ s/^ | $//gr         # trim ends
    =~ s/[^\x00-\x7f]//gr; # drop non-ASCII
```

Each `s///r` returns the transformed string, which the next one receives.

`/e` - evaluate the replacement

The replacement half of `s///e` is Perl code, not a double-quoted string. The return value of the code replaces the match:

```
my $x = "numbers: 1 2 3 4";
$x =~ s/(\d+)/$1 * 2/ge;
# $x is now "numbers: 2 4 6 8"
```

`/ee` evaluates twice: the code returns a string, which is then evaluated as Perl again. Rarely useful and easy to misuse - only reach for it when you are sure. The *substitution* chapter has worked examples.

`/n` - non-capturing

Makes every ordinary `(...)` behave like `(?:...)`. Useful when a long pattern has parentheses purely for grouping and you want to keep `$1`, `$2`, ... unset:

```
"hello" =~ /(hi|hello)/n;    # matches, but $1 is not set
```

Named captures `(?<name>…)` still capture under `/n`. Added in Perl 5.22.

/p - no-op

`/p` is a silent no-op. `${^PREMATCH}`, `${^MATCH}`, `${^POSTMATCH}` are always available. Accepted for backward compatibility, but contributes nothing in new code.

/o - compile once

`/o` disables re-compilation of a pattern that interpolates variables. Compiled patterns are also cached automatically, and `qr//` gives explicit control. `/o` is rarely the right tool; `qr//` is clearer and composes.

Charset modifiers - /a, /aa, /u, /l, /d

These control how the shorthand classes (`\d`, `\w`, `\s`), case folding, and the POSIX class set behave under Unicode and locale considerations. Brief summary here; the full treatment is in the *unicode* chapter.

Modifier	Behaviour
`/u`	full Unicode semantics. Default under `use v5.12+`.
`/a`	restrict `\d`, `\w`, `\s`, `[[:…:]]` to ASCII.
`/aa`	`/a` plus `/i` does not cross ASCII/non-ASCII.
`/l`	follow the current `use locale` POSIX locale.
`/d`	dual-mode semantics. Avoid in new code.

`/a`, `/u`, `/l`, `/d` are mutually exclusive - at most one can be in effect at a time. `/aa` is a refinement of `/a`. They are *set-once* modifiers: an inner `(?d:…)` cannot un-set an outer `/u`.

The everyday choice in modern code: `use v5.12` (or later) selects `/u` automatically, which is what you want for Unicode-correct text. Add `/a` only when the pattern must reject non-ASCII characters - typically because the input is known to be ASCII-only and you want to enforce that.

Inline modifiers

`(?flags)` turns flags on for the rest of the enclosing group (or pattern):

```
/(?i)yes/;          # case-insensitive, same as /yes/i
```

`(?flags:…)` scopes the flags to the inner group only and is non-capturing:

```
/Answer: ((?i)yes)/;   # only the 'yes' is case-insensitive
/Answer: ((?i:yes))/;  # clearer: scope is the group's contents
```

`(?-flags)` turns flags off. They can be combined:

```
/(?i-m:pattern)/;   # turn on /i, turn off /m, within this group
```

Inline modifiers are the right tool when different parts of a long pattern need different modifiers. They are also useful in patterns built by interpolation, where the embedded `(?i)` travels with the pattern fragment.

Caret form: `(?^…)`

`(?^flags)` is shorthand for "reset all flags, then apply these". The expansion is `d-imnsx` followed by the listed flags. So `(?^x:…)` means "default flags except `/x` is on, regardless of what flags were in scope outside".

The caret form is what Perl uses when stringifying compiled patterns; you may see it in error messages or `qr//` output. In hand-written code it is occasionally useful when interpolating a pattern fragment that should not inherit modifiers from its surroundings:

```
my $strict = qr/(?^:foo|bar)/;   # always default flags
# ... no matter how this is interpolated.
```

A negative flag is not legal after the caret (it would be redundant - the caret already cleared everything).

Mutually-exclusive flags

`/a` and `/aa` override each other (last one wins); same for `/x` and `/xx`. They are not additive. `(?xx-x:…)` turns *all* `x` behaviour off, not "subtract one `x` from two".

The charset family `/a`, `/d`, `/l`, `/u` is mutually exclusive; specifying one un-specifies the others. They cannot be turned *off* with a leading `-`, only switched between. So `(?-d:…)` is a fatal error; `(?dl:…)` is also fatal (two charset flags together).

`/p` is special: its presence anywhere in a pattern has a global effect (and that effect is a no-op).

Order of modifiers

On a match or substitution, the order of trailing modifiers is not significant. `/mgis` and `/sgmi` are identical. Pick a house style and stick to it.

How a regex is read - the four phases

For modifiers and inline forms to work, you have to understand that Perl reads a regex in *phases*. The phases:

1. **Phase A**: parser identifies the delimiter, finds the end of the pattern. `(?#…)` comments are removed here.
2. **Phase B**: pattern is parsed as a *double-quotish string*. Variables interpolate, escape sequences cook, `\Q…\E` translates to `quotemeta`-style escaping.
3. **Phase C**: under `/x` or `/xx`, unescaped whitespace and #-comments are stripped.
4. **Phase D**: the regex compiler reads the cooked, stripped pattern and turns it into the engine's internal form.

The order matters because phases B and D operate on different representations:

- `\Q$dir\E` cooks at Phase B, before the regex compiler sees it. By Phase D the variable's contents have already been metaquoted; the regex compiler sees a literal pattern.
- `\U...\E` is interpreted by Phase B as a string-cook directive (uppercase the contents), which is almost certainly *not* what you want inside a regex. Use `\Q` and `\E` for regex purposes; `\U` only when you want the pattern's *literal text* to be uppercased.
- `(?#comment)` is removed before Phase B sees the pattern at all. A literal # inside `(?#...)` is fine. A literal `)` is not - the comment ends at the first `)`.

The `/x` modifier applies at Phase C, between interpolation and compilation. That means whitespace introduced by an interpolated variable is *not* stripped by `/x`:

```
my $sub = " a b ";       # contains literal spaces
"axb" =~ / x $sub x /x;  # the spaces in $sub are NOT stripped
```

If you want the interpolated content to obey `/x`, you have to strip its whitespace at the source.

use re 'strict'

`use re 'strict'` raises a number of normally-tolerated regex sloppiness conditions to compile-time errors. It is per-lexical-scope. In strict mode:

- A `{` in a non-quantifier position is an error, not a literal.
- `[a-]` (dash at end of class) is an error.
- A useless negation of an always-on flag is an error.
- A `(?-d)` (trying to turn off `/d`) is an error rather than a warning.
- Unmatched `[` in a string is an error.

Strict mode is recommended for new regex-heavy code, especially generated patterns where small typos go unnoticed in a normal build. It is not default because it would break working older patterns.

```
use re 'strict';

/abc{,1/;           # error in strict; warning otherwise
/[a-]/;             # error in strict
```

See also

- The *unicode* chapter - `/a`, `/aa`, `/u`, `/l`, `/d` in detail, plus what they do to character classes and case folding.
- The *substitution* chapter - `/e`, `/r`, `/g`, and `/c` in their substitution-specific roles.
- The *basics* chapter - the four-phase parsing model in its first appearance.
- The *anchors and assertions* chapter - `\A`, `\z`, `\Z` interactions with `/m`.
- The *performance* chapter - `/o` discussion and why `qr//` superseded it.

- `m`, `s`, `qr` - operator references.

Substitution

`s///` is the search-and-replace operator. Its general form is:

```
$target =~ s/pattern/replacement/modifiers;
```

It finds the first position where *pattern* matches in *target*, replaces that match with *replacement*, and returns the number of substitutions made. With `/g`, it does that for every non-overlapping match.

```
my $x = "Time to feed the cat!";
$x =~ s/cat/hacker/;          # $x is now "Time to feed the hacker!"
```

If the pattern does not match, the target is unchanged and `s///` returns a false value.

Operating on $_

Like `m//`, `s///` defaults to `$_` when no `=~` is given:

```
for (@lines) {
    s/\s+$//;       # trim trailing whitespace from each line
}
```

Counting substitutions

The return value of `s///` in scalar context is the number of replacements:

```
my $count = ($text =~ s/\btodo\b/done/gi);
print "cleared $count todos\n";
```

In boolean context, zero means no match.

The replacement string

The replacement is a double-quoted string, so variables and escape sequences interpolate:

```
my $suffix = ".bak";
$file =~ s/$/$suffix/;    # append .bak to the name

$x =~ s/\t/    /g;        # each tab becomes four spaces
```

Inside the replacement, `$1`, `$2`, `$&`, `$+` and friends refer to what the *current* match just captured:

```
my $y = "'quoted words'";
$y =~ s/^'(.*)'$/$1/;     # strip surrounding single quotes
# $y is now "quoted words"
```

Named captures work the same way through `$+{name}`:

```
"2026-04-23" =~ s/(?<y>\d{4})-(?<m>\d{2})-(?<d>\d{2})/$+{d}\/$+{m}\/
 ↪$+{y}/;
# result: "23/04/2026"
```

Escaping in the replacement

The replacement is interpreted as a double-quoted string, so the double-quote rules apply to *it*, independently of the pattern. Metacharacters like `*`, `+`, `?`, `$` retain their double-quoted meanings - only `$` and `@` trigger interpolation:

```
$x =~ s/(\w+)/[$1]/g;      # wrap every word in brackets
$x =~ s/\$/USD/g;          # replace literal '$' with 'USD'
```

Backslash escape sequences in the replacement follow the double-quote rules - `\t`, `\n`, `\x{…}` all work. The special case-modification escapes `\l`, `\u`, `\L`, `\U`, `\E`, `\Q` apply in the replacement too:

```
$x =~ s/(\w+)/\u$1/g;      # capitalise the first letter of each word
$x =~ s/(\w+)/\U$1\E/g;    # uppercase whole word
```

Warning: \1 versus $1 in the replacement

A long-grandfathered idiom: writing `\1`, `\2`, … in the replacement to mean "the first capture", "the second capture":

```
$pattern =~ s/(\W)/\\\1/g;      # works, but is a trap
```

Perl accepts `\1` through `\9` in the replacement of `s///` for historical compatibility with `sed`. **Use $1 through $9 instead.** The reason is that the replacement is a double-quoted string, where `\1` *normally* means a control-A character. The substitution operator's special-case rule kludges that double-quote interpretation to mean "first capture", but the edges are sharp.

Specifically: combining `\1` with `/e` re-introduces the double-quote meaning. Under `/e`, the replacement is *Perl code*; in Perl code, `"\1"` is control-A:

```
s/(\d+)/ \1 + 1 /eg;    # warning: \1 better written as $1
```

A second sharp edge: `\1` followed by digit characters is ambiguous. `\1000` could mean "first capture, then literal `000`", or it could mean "octal `\100` (@), then a literal `0`" - and you cannot disambiguate by writing `\{1}000`:

```
s/(\d+)/\1000/;      # ambiguous, best avoided
s/(\d+)/${1}000/;    # unambiguous
```

The rule for new code: **always use $1 in the replacement, never \1**. The `\1` form is grandfathered, not recommended.

/g - replace all

Without /g, s/// replaces only the first match. With /g, it replaces every non-overlapping match from left to right:

```
my $x = "I batted 4 for 4";
$x =~ s/4/four/;          # "I batted four for 4"
$x = "I batted 4 for 4";
$x =~ s/4/four/g;         # "I batted four for four"
```

/g in substitution is unrelated to /g in matching. Here it just means "do this again and again until the pattern stops matching."

Zero-length matches and s///g

A pattern that can match the empty string presents a problem under /g: every position would match again forever. Perl breaks this loop with a specific rule: **after a zero-length match, the next attempt at the same position is forbidden to also be zero-length**. The engine takes the second-best match instead.

The cleanest demonstration:

```
$_ = 'bar';
s/\w??/<$&>/g;
# Result: <><b><><a><><r><>
```

The non-greedy \w?? prefers the empty match. After producing one, the next iteration is forbidden from being zero-length too, so the engine takes the second-best match - a single \w character. Then another empty match, then another character, and so on, alternating until the string is consumed.

The pattern: zero-length matches alternate with one-character matches, and the substitution does not loop forever.

A useful corollary: s/(\d{3})/$1,/g does not insert a comma after every set of three digits *correctly* - it inserts after the first three from the left, then the next three, regardless of where you wanted the commas. To insert thousands separators, match from the right or use 1 while:

```
my $n = "1234567";
1 while $n =~ s/^(\d+)(\d{3})/$1,$2/;
# n is now "1,234,567"
```

The 1 while repeats until the substitution returns false (no more matches), which is exactly what we want for "every position, after settling".

/e - evaluate the replacement

Under /e, the replacement is Perl code, not a string. The code runs for every match and its return value replaces the match.

```
my $x = "numbers: 1 2 3 4";
$x =~ s/(\d+)/$1 * 2/ge;      # "numbers: 2 4 6 8"
```

The match variables $1, $2, ... are visible inside the code block.

Practical uses:

```
# Uppercase the first letter of every sentence.
$text =~ s/(^|\.\s+)(\w)/$1\U$2/g;

# Hex-encode non-ASCII bytes.
$bytes =~ s/([\x80-\xff])/sprintf "\\x%02x", ord $1/ge;

# Apply a lookup table.
my %subst = (red => 0xff0000, green => 0x00ff00, blue => 0x0000ff);
$css =~ s/\b(red|green|blue)\b/sprintf "#%06x", $subst{$1}/ge;
```

/ee - double-eval

`/ee` evaluates *twice* - first as code, then the result is itself evaluated as Perl. The use case is dynamic variable lookup by name:

```
our $greeting = "hello";
my $s = "greeting";
$s =~ s/(\w+)/"\$$1"/ee;          # first: '$greeting'; second:
 ↪'hello'
# $s is now "hello"
```

The replacement is a Perl expression - the explicit double-quotes make it produce the *string* `"$greeting"`. The first `e` evaluates that expression to the string `$greeting`. The second `e` treats that string as Perl, looking up the variable `$greeting` and yielding `hello`. The match position is replaced with the value `hello`.

The use case is narrow and the security implications are substantial: any input that reaches an `/ee` substitution is effectively executed as Perl. Treat `/ee` like `eval` - never on user-supplied data.

/r - non-destructive substitution

`/r` returns the new string instead of modifying the target. The target is untouched:

```
my $orig = "I like dogs";
my $new = $orig =~ s/dogs/cats/r;
# $orig is still "I like dogs"
# $new is "I like cats"
```

If the pattern does not match, `/r` returns the original string unchanged:

```
my $x = "I like dogs";
my $y = $x =~ s/elephants/cougars/r;
# $y eq "I like dogs"
```

The big win is chaining:

```
my $slug = $title
    =~ s/[^\w\s-]//gr          # drop punctuation
    =~ s/\s+/-/gr              # collapse spaces to dash
    =~ s/--+/-/gr;             # collapse runs of dashes
```

Each arrow returns a new string, which the next `s///r` receives.

`\K` in substitution

`\K` (covered in detail in the *anchors and assertions* chapter) shines in substitution. The pattern matches a prefix that establishes context, then `\K` excludes the prefix from the replacement target:

```
# Without \K - the prefix has to be re-output:
$_ =~ s/(foo)bar/$1/g;

# With \K - the prefix is matched but not part of $&:
$_ =~ s/foo\Kbar//g;
```

Both replace `foobar` with `foo`. The `\K` form is cleaner: no capture is needed, and the substitution pattern is shorter and faster.

`\K` works similarly in any substitution where a fixed prefix should match but not be replaced. It is the substitution-side twin of lookbehind.

Substitution and `pos()`

Successful `s///g` advances `pos($string)` past each match. After a zero-length match, the next attempt at the same position may not be zero-length, which is what breaks the loop. After substitution, `pos` is reset.

A subtle interaction: `s///g` will not re-scan a region that has just been matched. After each successful replacement the match position advances past the *matched* text - the length of the replacement does not matter. So:

```
my $x = "aaa";
$x =~ s/a/AA/g;
# $x is "AAAAAA" - three a's, each replaced with AA
# It is NOT an infinite loop: pos() moves past each matched 'a',
# so the just-inserted 'A's are never re-scanned.
```

This is sometimes surprising for substitutions whose replacement contains characters that the pattern would also match. The rule is “advance past the matched text, never re-scan it”.

Alternate delimiters

Both halves of `s///` can use matched brackets; the delimiters do not have to be the same:

```
s{pattern}{replacement}g;
s<pattern><replacement>g;
```

(continues on next page)

(continued from previous page)

```
s[pattern][replacement]g;
s(pattern)[replacement]g;    # legal, but avoid mixing for clarity
```

Any printable punctuation works in a single-character form:

```
s!/usr/local/!/opt/!g;
s#pattern#replacement#;
```

Useful when the pattern or replacement contains `/`.

Single-quoted substitution

`s'pattern'replacement'` treats both halves as single-quoted - no variable interpolation, no backslash escapes apart from `\'` and `\\`:

```
s'@users'@admins'g;     # literal '@users' becomes literal '@admins'
```

Rarely needed; mention it for completeness.

Combining with `/m`, `/s`, `/x`

Substitution modifiers compose with match modifiers, so you can get the full toolbox:

```
# Trim each line.
$x =~ s/^\s+|\s+$//mg;

# Collapse blank lines.
$x =~ s/\n{2,}/\n\n/g;

# Replace C-style comments across newlines.
$src =~ s{/\*.*?\*/}{}gs;   # /s lets . cross newlines
```

With `/x` the pattern is readable, with `/s` it spans newlines, with `/g` it repeats.

A real example

Turn a block of English into a slug - lowercase, hyphens instead of spaces, drop non-letters, collapse and trim:

```
sub slug {
    my ($s) = @_;
    return lc($s)
        =~ s/[^\w\s-]//gr     # remove punctuation
        =~ s/\s+/-/gr          # spaces to hyphens
        =~ s/-+/-/gr           # collapse hyphens
        =~ s/^-|-$//gr;        # trim leading/trailing hyphens
}

print slug("Hello, World! -- Regex Tutorial");
# prints: hello-world-regex-tutorial
```

Friedl's framing - "anything that isn't required will always be considered successful"

A pattern's optional pieces will always succeed if they can match nothing. This applies inside substitution as forcefully as in matching:

```
my $x = "no horizontal rule here";
$x =~ s/-*/<HR>/;
# Result: "<HR>no horizontal rule here"
```

The pattern `-*` matches zero or more dashes - and zero is always available, at the start of the string. `<HR>` is inserted at position 0. The pattern has succeeded by the engine's definition; it has failed by yours.

The fix is to require at least one dash:

```
$x =~ s/-+/<HR>/;       # no match; $x unchanged
```

The slogan: **always consider what will happen if there is no match**. Optional pieces are not free; they always succeed by matching nothing somewhere.

Things that look like substitution but aren't

`tr///` (also written `y///`) does character-by-character transliteration, not regexp substitution. It accepts two lists of characters; each occurrence of the nth character in the first list becomes the nth character in the second list. No regexp features apply:

```
my $x = "Hello";
$x =~ tr/A-Za-z/a-zA-Z/;     # swap case
(my $hex = $bytes) =~ tr/\x00-\xff//d;   # delete all bytes -␣
↪pointless
```

`tr///` is faster than `s///` for character-level work because it is not a regexp engine. See `tr` for full semantics.

See also

- `s` - substitution operator reference.
- `tr` - character transliteration; not substitution but often confused for it.
- `quotemeta` - escape a string for safe interpolation into a pattern.
- The *modifiers* chapter - `/g`, `/e`, `/r`, `/ee`, and how they interact.
- The *anchors and assertions* chapter - `\K` for "match this prefix but don't replace it".
- The *performance* chapter - `s///g` zero-length match termination in detail.

Unicode

Perl's strings are Unicode strings. Every code point from `\x{0000}` through `\x{10FFFF}` is a valid character, regardless of the underlying byte encoding. Regexps operate on characters, not bytes - so `\w`, `\d`, `\s`, `.`, and `\X` all understand the Unicode world.

This chapter covers how to write Unicode characters in a pattern, how to match by Unicode property, and how the *charset modifiers* `/a`, `/u`, `/l`, `/d` tune the default character-class behaviour. It closes with *Script Runs*, the mechanism for rejecting strings that mix scripts when they shouldn't.

Writing Unicode in a pattern

Four ways to denote a specific code point:

Form	Example	Notes
`\xHH`	`\x41` (A)	Two hex digits, 0–255
`\x{…}`	`\x{263a}` (☺)	Any code point
`\o{…}`	`\o{101}` (A)	Octal
`\N{name}`	`\N{GREEK SMALL LETTER SIGMA}`	Unicode character name

```
/\x{263a}/;                  # WHITE SMILING FACE
/\N{GREEK SMALL LETTER SIGMA}/;  # σ
/\N{U+03C3}/;                # same codepoint via U+ form
```

Whitespace and underscores inside `\N{…}` and `\p{…}` are ignored, so you can write `\N{GREEK SMALL LETTER SIGMA}` or `\N{greek_small_letter_sigma}`.

Inside a character class, the same escapes work:

```
/[\x{0370}-\x{03ff}]/;       # any Greek block character
/[\N{SECTION SIGN}\N{PILCROW SIGN}]/;
```

`\N{NAME}` (named character) is *not* the same as `\N` (any character except `\n` - see the *character classes* chapter). The presence or absence of the brace is what selects.

Matching by property: `\p{…}` and `\P{…}`

Unicode classifies every code point by dozens of properties. The most useful in everyday patterns:

Property	Matches
`\p{L}`	any letter
`\p{Ll}`	lowercase letter
`\p{Lu}`	uppercase letter
`\p{Lt}`	titlecase letter
`\p{N}`	any numeric character
`\p{Nd}`	decimal digit
`\p{P}`	punctuation
`\p{M}`	mark (combining character)
`\p{S}`	symbol
`\p{Z}`	separator (including spaces)
`\p{C}`	control, format, unassigned, private-use
`\p{ASCII}`	code point 0–127
`\p{White_Space}`	any Unicode whitespace

`\P{X}` is the negation of `\p{X}` - matches any code point *not* in property X.

Short single-letter properties drop the braces:

```
/\pL/;    # letter - same as \p{L}
/\pN/;    # number
/\pM/;    # mark
```

Compound properties

Some properties have multiple values. The compound form `\p{Property=Value}` (or `\p{Property:Value}`) spells them out:

```
/\p{General_Category=Uppercase_Letter}/;    # same as \p{Lu}
/\p{Block=Greek_and_Coptic}/;
/\p{Numeric_Type=Decimal}/;
/\p{Bidi_Class=R}/;                          # right-to-left␣
↪characters
```

Case, spaces, and underscores are ignored in the property name and value, so `\p{gc=lu}` is the same as `\p{General_Category=Uppercase_Letter}`.

For the full list of properties and values, see `perluniprops` - that list is generated from Unicode data and is not duplicated here.

Matching by script

Scripts (writing systems) are properties too:

```
/\p{Latin}/;
/\p{Greek}/;
/\p{Cyrillic}/;
/\p{Arabic}/;
/\p{Han}/;            # Chinese, Japanese kanji, Korean hanja
/\p{Hiragana}/;
/\p{Katakana}/;
/\p{Hangul}/;
/\p{Script=Greek}/;   # compound form, same meaning
```

For linguistic classification, prefer `\p{Script_Extensions=…}` - it handles code points used by multiple scripts more sensibly than `\p{Script=…}`. Perl's `\p{Greek}` shorthand actually uses `Script_Extensions` under the hood.

Grapheme clusters: `\X`

A *grapheme cluster* is what users perceive as a single character. A Danish å can be one code point (`U+00E5`) or two (`a` followed by combining ring `U+030A`). Both look identical on screen. `\X` matches either - one atomic user-visible character.

```
my $s = "a\x{030A}";     # a + combining ring
length($s);              # 2  - code-point count
```

(continues on next page)

(continued from previous page)

```
$s =~ /./;                # matches just the 'a'
$s =~ /\X/;               # matches the whole cluster
```

Use `\X` when iterating visibly across user-facing text: cursor movement, column counting, truncation, anywhere "characters" means "what the user sees".

Extended bracketed character classes - `(?[…])`

The standard `[…]` form takes unions of characters; `(?[…])` adds set-algebra operators that combine Unicode properties:

```
# Letters that are Greek:
/(?[ \p{Letter} & \p{Greek} ])/;

# Letters that are not Latin:
/(?[ \p{Letter} - \p{Latin} ])/;

# Hex digits without 'a' through 'f':
/(?[ [0-9A-Fa-f] - [a-f] ])/;
```

Operators: `+` (or `|`) for union, `&` for intersection, `-` for difference, `^` for symmetric difference, `!` for complement. The construct is itself a character class - it matches one character - and quantifies normally. The *character classes* chapter covers it in full.

The charset modifiers

The flags `/a`, `/u`, `/l`, `/d` control how the shorthand classes (`\d`, `\w`, `\s`) and case folding behave. They are orthogonal to `/i`, `/m`, `/s`, `/x`. At most one can be in effect at a time; `/aa` is a refinement of `/a`.

`/u` - Unicode semantics

`\d`, `\w`, `\s`, `\b`, case folding, and `[[:alpha:]]` use full Unicode definitions. This is the default when `use v5.12` or later is in effect.

```
"item\x{0660}" =~ /\w+\d/u;   # matches - U+0660 is an Arabic-Indic␣
↪zero
"naïve" =~ /\w+/u;            # matches the whole word
```

Modern code should use `/u`. `use v5.12+` selects it automatically.

`/a` - ASCII-only classes

Restricts `\d`, `\w`, `\s`, `[[:…:]]` to the ASCII range. Useful when you *want* English-like text and no more:

```
"item\x{0660}" =~ /\w+\d/a;   # does not match - \d is [0-9] under /
↪a
"item0"        =~ /\w+\d/a;   # matches
```

/aa - strict ASCII

/aa (doubled) additionally prevents case-insensitive matches crossing the ASCII/non-ASCII boundary. Under plain /ia, K could match \x{212A} (KELVIN SIGN) because the Kelvin sign casefolds to k. Under /iaa it does not:

```
"K"        =~ /k/i;      # matches
"\x{212A}" =~ /k/i;      # matches - Kelvin sign casefolds to 'k'
"\x{212A}" =~ /k/iaa;    # does NOT match - ASCII-only i
```

/aa is the strict form when the input is genuinely ASCII-only and you want the engine to refuse Unicode matches even under /i.

/l - locale semantics

\d, \w, \s, case folding, and [[:alpha:]] follow the current POSIX locale. Only sensible after use locale. Rarely what you want - locale behaviour is implementation-specific and harder to reason about than /a or /u. Provided for legacy code that genuinely needs locale-dependent matching.

/d - dual-mode semantics

Under /d, \d, \w, \s depend on whether the target string is flagged as UTF-8 internally. This is the source of most "works on my machine" Unicode regexp bugs. **Avoid /d in new code.** The default under use v5.12 is /u.

If you inherit code where /d matters, the failure mode is: the same input string produces different match results depending on whether some upstream operation flagged the string UTF-8. The fix is use feature 'unicode_strings' (or use v5.12+), which makes the rules independent of the UTF-8 flag.

Which modifier is in effect?

Resolution order:

1. Explicit modifier on the regex (/a, /u, /l, /d).
2. use re '/u' (or other) in the lexical scope.
3. use locale selects /l.
4. use feature 'unicode_strings' (and use v5.12+) selects /u.
5. Otherwise, /d (the dual-mode fallback; avoid).

Programs should use use v5.12; (or later) at the top of each file; this gives you /u, which is what you want.

Pragma interactions

```
use v5.12;                          # implicit feature 'unicode_
↪strings'
use feature 'unicode_strings';       # explicit
```

(continues on next page)

(continued from previous page)

```
use re '/u';                          # set default modifier for this␣
↪scope
```

`use feature 'unicode_strings'` (included in `use v5.12`) makes all strings Unicode for character-class purposes, regardless of whether they are marked UTF-8 internally. Turn it on at the top of every file.

Case folding

Under `/iu` (the default), case-insensitive matching uses the full Unicode case-folding tables:

```
"Grüße" =~ /grüsse/iu;          # matches - ß folds to 'ss'
"Σίγμα" =~ /σίγμα/iu;            # matches - upper Σ ↔ lower σ
"İstanbul" =~ /istanbul/iu;    # matches - dotted I folds
```

Under `/ia`, fold only ASCII letters. Under `/iaa`, plus the Kelvin/K rule above.

The folding happens at compile time - pperl does not lowercase the input string at match time.

The \b{…} word-boundary variants

Beyond plain `\b`, Perl offers semantic variants that handle real-world text better:

Boundary	Meaning
`\b{wb}`	Unicode word boundary - handles `don't`, `state-of-the-art`
`\b{sb}`	sentence boundary
`\b{lb}`	line break (suitable for line wrapping)
`\b{gcb}`	grapheme cluster boundary (same effect as `\X`)

```
"don't" =~ /.+?\b{wb}/x;       # matches whole word - apostrophe is␣
↪inside
"don't" =~ /.+?\b/x;           # stops at the apostrophe - plain \b␣
↪splits
```

For natural-language processing, `\b{wb}` and `\b{sb}` are almost always what you want. Plain `\b` is for ASCII identifier contexts.

The *anchors and assertions* chapter has the broader boundary discussion; this chapter is the home for the Unicode-aware variants because their definitions are Unicode-driven.

Script runs

Different writing systems have visually-similar characters. The Latin a, the Cyrillic а (U+0430), and the Greek α (U+03B1) look indistinguishable. Used together - typically in a URL or identifier - they enable a *homograph attack*:

paypal.com

Could be all Latin (legitimate). Could be a Cyrillic a embedded in otherwise-Latin text (a phishing attack pointing to a different domain). The browser's URL renders identically; the bytes do not.

A *script run* is a sequence of characters all from the same Unicode script (per `Script_Extensions` and Unicode UTS 39). Most legitimate words are script runs; mixed-script strings are rare outside specific multilingual contexts and almost always suspicious.

Perl provides two constructs, each with a short spelling, to require that a matched sub-pattern be a script run:

Form	Meaning
`(*script_run:PAT)` / `(*sr:PAT)`	match PAT, then check it is a script run
`(*atomic_script_run:PAT)` / `(*asr:PAT)`	same, but atomic (faster, less backtracking)

```
# Match a domain label only if it is a script run:
$label =~ /(*sr: \w+ )/x or warn "mixed-script label";

# Atomic version: faster on adversarial input.
$label =~ /(*asr: \w+ )/x or warn "mixed-script label";
```

The atomic form is what you want in most production code - it prevents the script-run check from being defeated by catastrophic backtracking on a malicious input. `(*sr:…)` is the non-atomic equivalent (`(*sr:(?>PAT))` is the same as `(*asr:PAT)`).

Script run rules

A sequence is a script run if and only if all of:

1. **No code point in the sequence has the `Script_Extensions` property `Unknown`.** This excludes private-use and surrogate code points; you cannot smuggle them through.
2. **All characters come from the Common script, the Inherited script, and at most one other script.**
3. **All decimal digits come from the same set of ten.** Many scripts have their own digits (Arabic-Indic, Devanagari, etc.); a string mixing 1 (ASCII) with ١ (Arabic-Indic one) is *not* a script run, even if other characters are compatible.

Three pseudo-scripts get special handling:

- **Common** - punctuation, ASCII digits 0–9, mathematical symbols, emoji, full-width digits. These can appear in any script run *except* that all decimal digits in the sequence must still come from the same set of ten.
- **Inherited** - combining marks (accents, diacritics). These attach to the previous character and inherit its script.
- **Unknown** - unassigned code points and similar. A single-character string of unknown is allowed; a longer string containing one is not.

The relaxation for Common and Inherited is what allows real-world text to be a script run despite punctuation and combining marks. The strict-equality rule for digits is what defeats homograph attacks that mix digits from look-alike scripts.

When script runs help

- **URL parsing**: refuse domain labels that are not script runs.
- **Identifier validation**: programming-language identifiers should normally be script runs.
- **Form input sanitisation**: real names are script runs; mixed-script names are usually attacks or test data.

Script runs are a 2018-and-later feature. They are what Perl's regex engine adds to a problem the rest of the engine cannot solve directly: a property of the *whole* match, not of any single character.

Cross-engine: Unicode coverage

Feature	Perl 5.44	PCRE2	Emacs (limited)	POSIX	RE2 / Go
`\p{General_Category}`	yes	yes	partial	no	yes
`\p{Script}`	yes	yes	partial	no	yes
`\p{Property=Value}`	yes	yes	partial	no	yes
`\X` grapheme cluster	yes	yes	no	no	yes (`(?-X:...)` is something else)
`\b{wb}`, `\b{sb}`	yes	no	no	no	no
Script Runs `(*sr:…)`	yes	no	no	no	no
Default to Unicode `\d/\w`	under `/u`	no	no	no	no (Unicode under `(?u)`)

Perl is a leader here: `\b{wb}` and Script Runs are essentially unique to Perl among the comparison set. RE2 / Go and PCRE2 both default to ASCII for `\d`, `\w`, `\s`; RE2 enables Unicode under the `(?u)` flag. PCRE2 has property support but lacks the boundary variants and script runs. Emacs has property support that varies by build.

The full table is in the *cross-engine* chapter.

Encoding vs. character

A Perl string is always a sequence of code points. The bytes on disk are the I/O layer's concern - use `use utf8` for source-code literals, `:encoding(UTF-8)` on filehandles, and `decode`/`encode` from `Encode` when crossing the boundary. Patterns never operate on the encoded bytes; they always operate on code points.

```
use utf8;            # source code is UTF-8
use feature 'unicode_strings';
use open ':std', ':encoding(UTF-8)';  # STDIN/STDOUT/STDERR

while (my $line = <>) {
    $line =~ /\p{Lu}/ and print "has uppercase letter\n";
}
```

Get the I/O right once, and patterns just work on characters for the rest of the program.

See also

- The *character classes* chapter - non-Unicode class basics, `(?[…])`.
- The *anchors and assertions* chapter - `\b`, `\B`, plus the `\b{…}` family in their boundary role.
- The *modifiers* chapter - `/i`, `/x`, `/m`, `/s`, and how they combine with charset modifiers.
- The *cross-engine* chapter - Unicode coverage across regex engines.
- `quotemeta` - Unicode-aware metacharacter escaping.

Performance

Most regexps run in roughly linear time over the string. A well-written pattern that matches is fast; a well-written pattern that fails is also fast. The trouble comes from patterns that *could have* matched in many different ways - the engine explores those possibilities one after another, and pathological patterns can explore exponentially many.

This chapter is about recognising that trouble, avoiding it, and when you cannot avoid it, damping it with possessive quantifiers, atomic groups, and careful structure. The second half is about the machinery the engine quietly applies on your behalf, and how to write patterns that let those optimisations fire.

Perl uses a *Traditional NFA* - an engine that walks the pattern under the direction of the input rather than the other way around, saving its place at every choice point so it can resume after a dead end. That property is what makes captures, backreferences, and lookaround possible; it is also what makes the worst case exponential. Everything in this chapter is a corollary of that one fact.

How backtracking works

A regexp engine walks the string and the pattern in lockstep. When it reaches a choice - an alternation, a quantifier, an optional group - it makes a decision, remembers the choice on a stack, and continues. If a later position in the pattern fails, it pops the most recent choice, tries the next available decision, and resumes. This is *backtracking*.

Every `?`, `*`, `+`, `{…}`, and `|` is a potential backtrack point. A short string and few choice points: fast. A long string, nested choice points, and no match: exponential.

A worked example from `perlre`. To pull the word following `foo` out of `Food is on the foo table.`:

```
$_ = "Food is on the foo table.";
if (/\b(foo)\s+(\w+)/i) {
    print "$2 follows $1.\n";
}
# table follows foo.
```

The engine first matches `\b(foo)` against `Food`, loading `$1` with `Foo`. It then looks for `\s+` and finds `d` - failure. It backtracks, advances the start position, and continues searching until it finds the literal `foo` followed by whitespace, then a word.

A more revealing case. The pattern says "everything between `foo` and `bar`":

```
$_ = "The food is under the bar in the barn.";
if (/foo(.*)bar/) {
    print "got <$1>\n";
}
# got <d is under the bar in the >
```

Greedy `.*` ran all the way to the end of the string, then crawled back left until `bar` could finally match - at the *last* `bar`, not the first. Use `.*?` to stop at the first.

`perlre` has a longer version of this with eight variants - most wrong, all instructive - under `Backtracking`. The lesson is the same in each: a greedy quantifier always overshoots first; the correct *shape* (which anchors? non-greedy or negated class?) is what fixes the match, not adjusting the quantifier in place.

The state-saving cost

Friedl's mechanical model is worth keeping in mind. For an NFA, the *saved-state count* is roughly the number of characters that quantifier-governed subexpressions matched. `[0-9]+` against `1234` saves four states - one per digit, marking each as a "match could end here" point. When the engine eventually fails further along the pattern, it pops those four states one at a time, trying to find a way to make the match succeed.

A subtle detail: the `+` quantifier *requires* its first match, so it doesn't save a state for "match nothing here". `*` does, because zero matches are allowed. That asymmetry shows up in benchmark microcomparisons of `+` vs `*` patterns; on inputs where the quantifier really must match at least one thing, `+` is slightly faster because it has one fewer state to manage on failure.

Catastrophic backtracking

Consider this pattern trying to match a long string of `a`'s:

```
"aaaaaaaaaaaaaaaaaaaa" =~ /(a+)+b/;
```

The pattern cannot match - there is no `b`. But the engine has to *prove* it. Each `a` in the string can be assigned to the inner `+` or the outer `+` in many ways. For a 20-character string that is millions of combinations; for a 30-character string, billions. The engine tries them all.

The general shape to watch for: **nested indeterminate quantifiers on overlapping classes.**

```
/(a+)+/;              # dangerous
/(a|b)*c/;            # dangerous on a+b strings without trailing c
/(\w+)*\s/;           # dangerous on a run of word chars with no␣
↪space
/([^"]|\\.)*"/;       # dangerous on long unterminated strings
```

Every one of these has the same shape: an outer quantifier on a group whose inner content is itself indeterminate. Friedl's canonical scary example, the doublequoted-string regex `"(\\.|[^"\\]+)*"` applied where it cannot match, is documented to take roughly 3.2×10^{29} backtracks before giving up - many universe-lifetimes if the engine really did try them all. Perl does not try them all: its super-linear cache remembers where an iteration has already failed, which turns the exponential search into a quadratic one. Quadratic is still a trap. On perl 5.44 that pattern rejects an unterminated string of 1,000 characters in about 0.01 s, of 10,000 in about 0.8 s, and of 100,000 in about 80 s.

The disease is the same in every case. For each extra character the input grows, the number of ways the engine can apportion characters between the inner and outer quantifier *doubles*.

The cure is *determinism*: write the regex so that for any input there is exactly one way the engine can divide the work between inner and outer quantifier. The next sections cover the three techniques that achieve this.

Possessive quantifiers

A possessive quantifier (`++`, `*+`, `?+`, `{n,m}+`) is greedy *and* never gives back. It tells the engine: "once I've matched these characters, they are mine; do not revisit this decision on failure."

```
/(a++)+b/;    # at most one backtrack point
/(\w++)*\s/;  # bounded - \w++ takes all word chars in one shot
```

The trade-off: if a later part of the pattern needs those characters back, the match simply fails. Possessive is correct when there really is no other way for the match to succeed at that prefix.

Possessive quantifiers are syntactic sugar for an atomic group around the quantified atom. The following are equivalent:

Possessive form	Atomic-group form
`PAT*+`	`(?>PAT*)`
`PAT++`	`(?>PAT+)`
`PAT?+`	`(?>PAT?)`
`PAT{n,m}+`	`(?>PAT{n,m})`

The possessive notation is shorter; the atomic-group notation generalises further (you can wrap arbitrary alternations, not just quantified atoms).

A consequence of "possessive equals atomic": some combinations make no sense and Perl rejects them.

Illegal form	Equivalent legal form
`X??+`	`X{0}`
`X+?+`	`X{1}`
`X{n,m}?+`	`X{n}`

Non-greedy and possessive are contradictory. Use `X{0}`, `X{1}`, or `X{n}` if you really wanted to match exactly that many.

Atomic groups

`(?>…)` is a non-capturing group whose contents, once matched, are committed. Like a possessive quantifier, no backtracking happens inside it. The difference is that atomic groups wrap *arbitrary* patterns, not just quantified atoms.

```
"aaab"  =~ /a*ab/;       # matches - the a* gives back one 'a'
"aaab"  =~ /(?>a*)ab/;   # does not match - a* takes all, none to␣
↪give
```

Atomic groups are the most general tool for controlling backtracking. Use them around a block of pattern that should either fully match or not participate.

A classic application: balanced parentheses, fast on bad input.

```
# Unbounded: dangerous on unbalanced input
/ \( ( [^()]+ | \([^()]*\) )+ \) /x;

# With atomic group: fast failure on bad input
/ \( ( (?> [^()]+ ) | \([^()]*\) )+ \) /x;
```

The inner `(?>[^()]+)` says: gobble as many non-paren characters as possible and don't come back. That removes the ambiguity that makes nested `+` expensive.

`(?>…)` does not disable backtracking *outside* it. The engine can still backtrack past the construct as a whole, just not into it. So `((?>a*)|(?>b*))ar` against `bar` still matches: the outer alternation can backtrack between branches.

Nested atomic groups are *not* no-ops. Each extra layer further constrains what the inner pieces can give up. `(?>a[bc]*c)` matches `abc`; `(?>a(?>[bc]*)c)` does not, because the inner `(?>[bc]*)` ate the trailing `c` and refuses to release it.

The long-form spelling `(*atomic:…)` is also accepted; it reads better in pattern code that already uses verbs like `(*positive_lookahead:…)`.

Matching quoted strings - the canonical case

The "match a quoted string with backslash escapes" problem is the one everyone gets wrong on the first try.

Naive:

```
/"(?:[^"\\]|\\.)*"/;
```

This one is correct, and it is not the dangerous kind: its two alternatives cannot match the same character, so there is only one way to divide the input, and a failed match fails in linear time. The trap is a `+` inside the group, `/"(?:[^"\\]+|\\.)*"/`: a run of `a`'s can now be split between iterations in many ways, and perl takes about 0.6 s instead of 0.002 s to reject an unterminated string of 10,000 characters. The safe form:

```
/"(?:[^"\\]++|\\.)*+"/;
```

The inner `++` takes all non-quote, non-backslash characters in one bite. The outer `*+` refuses to give back across iterations. A failed match fails in linear time. The atomic-group equivalent is identical in behaviour:

```
/" (?> (?: (?> [^"\\]+ ) | \\. )* ) "/x;
```

- and demonstrates why the possessive form is preferable when it applies. It's terser without losing anything.

Unrolling the loop

The single most useful technique for replacing a non-deterministic pattern with a deterministic one. Friedl named it; engines have internalised it; every working regexp programmer should know it.

The pattern shape:

```
opening normal* (special normal*)* closing
```

- where `normal` and `special` are subexpressions that consume disjoint sets of characters. The construction guarantees there is exactly one way the engine can split the input between them, so no exponential exploration ever begins.

Three rules, all mandatory:

1. **`special` and `normal` cannot start a match at the same position.** If both can match the same character, the engine has freedom to choose; that freedom is exactly what causes exponential blowup.
2. **`special` cannot match the empty string.** A `special` that matches nothingness is invisible to the engine and removes the checkpoint that makes the construction deterministic.
3. **One application of `special` cannot be coverable by multiple applications of `special`.** If `special` is `\s+` and the input has three consecutive spaces, the engine could match them as one `special`, three `specials`, or various combinations - exponential blowup is back.

Worked example: doublequoted strings

Naive form, prone to catastrophic backtracking:

```
/"(?:\\.|[^"\\])*"/;
```

Unrolled form, no nested ambiguity:

```
/"[^"\\]*(\\.[^"\\]*)*"/;
```

`[^"\\]*` is the `normal*` - it eats every non-special character. `\\.` is the `special` - it consumes a backslash plus exactly the following character. The two cannot overlap (`normal*` cannot contain `\\`; `special` cannot match a non-`\\` character), and `special` always consumes exactly two characters.

The unrolled form runs in linear time on any input - matching or not. The naive form is exponential on inputs with many backslashes and no closing quote.

The same problem with possessive quantifiers also runs in linear time:

```
/"[^"\\]*+(?:\\.[^"\\]*+)*+"/;
```

Pick whichever the rest of your team reads more easily. Both work.

Worked example: C comments

The "match a C comment, surviving asterisks inside" problem. Naive comment match `\/\*.*?\*\/` works but is slow on long comments because `.*?` has to expand one character at a time.

Unrolled:

```
/\/\*[^*]*\*+([^/*][^*]*\*+)*\//;
```

Reading from left to right:

- `\/\*` - opening `/*`.
- `[^*]*` - `normal*`, zero or more non-asterisk characters.
- `\*+` - at least one asterisk.
- `([^/*][^*]*\*+)*` - zero or more iterations of: a non-slash, non-asterisk character, then more non-asterisks, then more asterisks. (The non-slash bit is what stops the run: an asterisk followed by a slash is the terminator we want.)
- `\/` - closing slash.

The construction makes every iteration of the outer `*` consume at least one character that no other iteration could have consumed. Result: linear time, no catastrophic-backtracking pattern.

The pedagogical value of working through this is in the *thought process*, not memorising the final regex. The asterisks-inside complication forces the extra `[^/*]` term; recognising why is what teaches you to spot the next case.

Worked example: the freeflowing regex

When you have multiple "skip me" subexpressions (strings, comments, constants in a parser-like context), build a single regex with all of them in alternation and an `OTHER+` clause for the bulk of the data:

```
my $DOUBLE = qr/"[^"\\]*(?:\\.[^"\\]*)*"/;
my $SINGLE = qr/'[^'\\]*(?:\\.[^'\\]*)*'/;
my $COMMENT = qr/\/\*[^*]*\*+(?:[^\/*][^*]*\*+)*\//;
my $LINECOM = qr/\/\/[^\n]*/;
my $OTHER  = qr/[^"'\/]+/;

while ($source =~ / $DOUBLE | $SINGLE | $COMMENT | $LINECOM | $OTHER
 ↪/xg) {
    ...
}
```

The `$OTHER` arm consumes the bulk of the input in long runs, which means the engine spends almost all its time inside one linear-time loop, not on the bump-along trying alternatives at every uninteresting character. The technique is decisive on parser-like workloads.

Reducing choice

If your pattern works but is slow, look for spurious choice:

- **Anchor where you can.** `^pat` and `pat$` let the engine reject most starting positions immediately. `\A` is even better than `^` when you mean "start of string" - the engine can refuse to retry at every position rather than only at line starts.
- **Replace alternation of single characters with a class.** `/a|b|c/` is slow compared to `/[abc]/`; the class is a single choice, the alternation is three.
- **Put the common alternative first.** Alternation is tried left-to-right, and Perl stops at the first success. If 90% of your inputs hit alternative 3, reorder.
- **Expose common prefixes.** `/this|that|them/` defeats the fixed-string optimisation; `/th(?:is|at|em)/` hands the engine the literal `th` to scan for first.
- **Prefer character classes to `..`** `[^\n]` is cheaper than `.` because the engine knows it will never match across lines. Same for `[^"]` instead of `.*?` between quotes.
- **Negated character classes beat `.*?`.** The non-greedy form has to leave the inner loop on every iteration to test what follows; the negated class participates in the simple-repetition fast path. `<([^>]+)>` is several times faster than `<(.+?)>` and also more correct: the negated class *cannot* cross a `>` even under backtracking pressure.

Compile once, match many

Every time a pattern with interpolated variables is used, Perl checks whether the text of the pattern changed. If it has not, the compiled form is reused. For patterns that really are fixed, the overhead is small but nonzero.

`qr` compiles a pattern once and produces a reusable object:

```
my $word = qr/\b[a-z]+\b/;
for my $text (@corpus) {
    while ($text =~ /$word/g) {
```

(continues on next page)

(continued from previous page)

```
        ...
    }
}
```

Using `qr` in a hot loop saves the recompile-or-check step and makes the code clearer. It also lets you build complex patterns out of named pieces, which is decisive for any regex over about ten lines.

The `/o` modifier ("compile once") was the Perl 4 mechanism for this. It still works but is rarely the right tool today: `qr//` is clearer, scopes better, and composes.

Avoiding capture overhead

Capturing groups cost a little - the engine has to track the start and end positions, and the surrounding optimisations (simple repetition in particular) may be disabled. If you don't use the capture, don't ask for it.

```
# Capturing even though $1 is never read:
/(\d{4})-(\d{2})-(\d{2})/;

# Non-capturing:
/(?:\d{4})-(?:\d{2})-(?:\d{2})/;
```

The `/n` modifier turns every `(…)` into `(?:…)` without editing the pattern - useful for long patterns. Named captures still capture under `/n`.

Even non-capturing parens may suppress the simple-repetition optimisation in some engines, so `(?:[a-z])+` is occasionally slower than `[a-z]+`. Drop unnecessary grouping; flatten patterns where you can without hurting readability.

Recursive patterns

Perl regexps support recursion - a pattern can refer back to one of its own groups. The construct is `(?R)` (or `(?0)`) for the whole pattern, `(?N)` for group N, `(?-N)` and `(?+N)` for relative references, and `(?&NAME)` for named groups.

A recursive pattern for a balanced parenthesis, any depth:

```
my $bal = qr/
    (?(DEFINE)
        (?<paren>
            \(                  # opening paren
            (?:
                [^()]++         # run of non-paren, possessive
              | (?&paren)       # or a balanced sub-group,␣
↪recursively
            )*+
            \)                  # closing paren
        )
    )
    (?&paren)
```

(continues on next page)

(continued from previous page)

```
/x;

"((a)(b(c)))" =~ /^$bal$/;   # matches
```

A named subpattern declared in a `(?(DEFINE)...)` block recurses into itself, not into the whole enclosing pattern. Using `(?R)` here would force every level of recursion to re-apply the call site's anchors (^ and $), which is rarely what you want.

Recursion makes genuinely recursive structures matchable without falling out of the regex DSL. It is also a strong signal that you might be reaching for the wrong tool. A real parser is often clearer; reach for recursive regexps when you want a one-line match-or-fail decision, not a parse tree.

There is a recursion-depth limit compiled into the engine (it can be raised by rebuilding from source). Patterns that recurse without consuming input fail immediately rather than running forever - the engine detects the cycle.

The interaction with capture groups deserves attention. When a group recurses into itself, captures set inside the recursion are *not* visible to the caller after the recursion returns. This is the practical reason most recursive patterns use possessive quantifiers and atomic groups for the inner content: capturing it serves no purpose because the data won't be there when the outer match finishes.

Embedded code

`(?{ code })` runs Perl code at the point the engine reaches it. The code's return value is set as `$^R`. Useful for instrumentation and for assertions whose condition is more naturally written in Perl than in regex syntax:

```
my $tries = 0;
"aaaaab" =~ /a*(?{ $tries++ })b/;
print "engine tried $tries times\n";
```

Two important properties:

- The block is *backtracking-safe* if you assign to `local` variables. `local $cnt = $cnt + 1` undoes itself when the surrounding match backtracks past the block. A bare assignment does not.
- The block sees the same lexical scope it was compiled in. A `qr//` containing `(?{ ... })` closes over the lexicals in scope at compile time, *not* at match time. This is the same rule that governs closures everywhere else in Perl.

`$^N` is particularly useful inside `(?{...})` - it holds the text matched by the most-recently-closed capture group, so you can save matches without counting parentheses:

```
$_ = "The brown fox jumps over the lazy dog";
/the (\S+)(?{ $color = $^N }) (\S+)(?{ $animal = $^N })/i;
print "color = $color, animal = $animal\n";
# color = brown, animal = fox
```

The big cost: `(?{ ... })` *globally disables* certain optimisations in the pattern. Patterns containing it can run much slower than patterns without it, even on inputs where the code block runs zero times.

`(*{ code })` is the *optimistic* form, introduced in Perl 5.37.7. Same syntax, same behaviour, except it does not disable optimisations. The trade-off: the engine may execute the block fewer times than the strict semantics of `(?{...})` would require. On a failing match it might not run at all. Use `(?{...})` when you depend on side effects firing in a documented order; use `(*{...})` when you don't.

`(??{ code })` is the *postponed regex*: the code's return value is treated as a pattern, compiled if it is a string (or used directly if it is a `qr//` object), and matched at that position. It enables genuinely runtime-shaped patterns:

```
my $inner = '(.)\1';
"ABBA" =~ /^(.)(??{ $inner })\1/;
print $1;   # 'A'
```

The inner pattern has its own capture state - captures inside `(??{...})` are not visible to the outer pattern. For balanced parens or other recursive structures, `(?R)`/`(?N)` is more efficient and also clearer.

User-supplied input must never reach `(?{...})` or `(??{...})`. Perl forbids it by default; `use re 'eval'` removes the safety catch and you should not.

Embedded-code execution frequency

The exact number of times `(?{...})` and `(??{...})` execute during a match is *unspecified*. The engine guarantees only:

- On a successful match, blocks on the accepting path execute in left-to-right order, the appropriate number of times for the quantifier they sit under.
- Blocks may execute zero, one, or many times during failed matches, depending on what optimisations the engine applies.

For example, in `"aaabcdeeeee" =~ /a(?{print "a"})b(?{print "b"})cde/`, the precise number of a's and b's printed is unspecified for *failed* matches. For a successful match each block runs at least once on the accepting path; if b was printed, an a was necessarily printed at least once before it.

Quantified blocks behave intuitively on success:

```
"good" =~ /g(?:o(?{ print "o" }))*d/;
# prints o twice
```

Code blocks intended to count, log, or trace must use the `(?{...})` form (deterministic on success); blocks that are purely advisory may use `(*{...})` (and accept the lower call count in exchange for not breaking optimisations).

Special backtracking control verbs

Perl provides verbs that direct the engine more precisely than ordinary atomic groups can. They all use the form `(*VERB)` or `(*VERB:NAME)`. They are zero-width (they consume no input) and only do something when *backtracked into* on failure.

The verbs interact with two package variables: `$REGERROR` and `$REGMARK`. After a match involving any verb that takes a `NAME`:

- `$REGMARK` is set to the name of the last `(*MARK:NAME)` executed, on success.
- `$REGERROR` is set to the name of the verb that caused the failure, on failure.

These are package globals (not magic variables; not localised automatically); use `local` if you need scoping.

`(*PRUNE)` and `(*PRUNE:NAME)`

Prunes the backtracking tree at the current point. After `A(*PRUNE)B`: if `B` fails, the engine does not backtrack into `A`. The match fails *at the current starting position*; the engine then advances to the next one as usual. To stop the whole match for good, use `(*COMMIT)`.

```
'aaab' =~ /a+b?(?{ print "$&\n"; $count++ })(*FAIL)/;
# Prints aaab, aaa, aa, a, aab, aa, a, ab, a - 9 times.

$count = 0;
'aaab' =~ /a+b?(*PRUNE)(?{ print "$&\n"; $count++ })(*FAIL)/;
# Prints aaab, aab, ab - 3 times.
```

`(*PRUNE)` is a more powerful `(?>...)`: it controls backtracking across pattern segments rather than within a fixed sub-pattern.

`(*SKIP)` and `(*SKIP:NAME)`

Like `(*PRUNE)`, plus: signals that the text leading up to the verb cannot be part of *any* match. The engine advances the start position to where the verb fired and retries from there:

```
'aaabaaab' =~ /a+b?(*SKIP)(?{ print "$&\n"; $count++ })(*FAIL)/;
# Prints aaab, aaab - 2 times.
```

If `(*SKIP:NAME)` fires and a `(*MARK:NAME)` was set earlier, the new start position is the mark's position, not the verb's. This is the only sane way to do "if branch X failed, advance past where branch X started" in a single pattern.

`(*MARK:NAME)`

A named pseudo-checkpoint. Does nothing on its own - but `(*SKIP:NAME)` uses it as a target, and `$REGMARK` after a successful match holds the most recently executed mark name. The shorthand `(*:NAME)` is identical to `(*MARK:NAME)`.

This lets a pattern record which alternative branch matched without using a separate capture group per branch:

```
/(?:foo(*MARK:F)|bar(*MARK:B)|baz(*MARK:Z))/;
# After a successful match, $REGMARK is "F", "B", or "Z".
```

The optimiser handles MARK/SKIP more efficiently than `/(?:(foo)|(bar)|(baz))/` because it does not have to track three parallel capture states.

(*THEN) and (*THEN:NAME)

Inside an alternation: on failure of the rest of the pattern, backtrack to the next alternative of the *innermost enclosing group with alternations*, skipping branches above that level. Outside an alternation, equivalent to `(*PRUNE)`. Loosely: "this branch and only this branch; if it fails, try the next sibling branch."

```
/(COND1 (*THEN) ACT1 | COND2 (*THEN) ACT2 | COND3 (*THEN) ACT3)/x;
```

Reads like an if/elsif/else inside the regex.

(*COMMIT) and (*COMMIT:arg)

The strongest brake. Like `(*SKIP)`, but instead of advancing the start position, the engine *gives up entirely*. No further attempts to find a match anywhere in the string.

```
'aaabaaab' =~ /a+b?(*COMMIT)(?{ print "$&\n"; $count++ })(*FAIL)/;
# Prints aaab - 1 time.
```

Use when a partial match has consumed enough to prove that no match exists anywhere later either.

(*FAIL) / (*F)

Always fails. Equivalent to `(?!)` but more readable. Useful only combined with `(?{...})`: when you want to count or log every attempted match without ever accepting one. `(?!)` is internally optimised into `(*FAIL)`.

(*ACCEPT) and (*ACCEPT:arg)

The opposite of `(*FAIL)`. Always succeeds, ending the match immediately. Inside nested patterns or recursion, only the innermost pattern is ended. Capture groups that have opened but not yet closed are marked as ending where `(*ACCEPT)` fires:

```
'AB' =~ /(A (A|B(*ACCEPT)|C) D)(E)/x;
# matches; $1 is "AB", $2 is "B", $3 is undef
```

`(*ACCEPT)` is most useful inside `(??{...})` or `(?(condition)...)` constructs where you want a fast-path success without finishing the pattern.

Zero-length-match termination

A repeated pattern that can match the empty string is a problem. Trivially:

```
'foo' =~ m{ ( o? )* }x;
```

Each iteration of the outer * matches the empty string, the position does not advance, the next iteration matches empty again - infinite loop in waiting. Perl breaks this loop in two different ways depending on which kind of repetition is involved.

Lower-level loops - the quantifiers *, +, {n,m} - are *interrupted* when the engine detects that an iteration matched zero-length. The construct

```
(?: NON_ZERO_LENGTH | ZERO_LENGTH )*
```

is silently treated as

```
(?: NON_ZERO_LENGTH )* (?: ZERO_LENGTH )?
```

The zero-length branch can run, but only *once*, after the non-zero branch is done.

Higher-level loops - the /g modifier on m// and s///, and the split operator - preserve a flag across iterations recording whether the previous match was zero-length. After a zero-length match the next attempt is forbidden to *also* be zero-length; the engine takes the second-best match instead. The canonical demonstration:

```
$_ = 'bar';
s/\w??/<$&>/g;
# Result: <><b><><a><><r><>
```

The non-greedy \w?? prefers an empty match. After producing one, the next iteration is forbidden from being zero-length too, so the engine takes the *second*-best match - a single \w character. Then another empty match, then another character, and so on, alternating until the string is consumed.

The state of "previous match was zero-length" is associated with the matched string and is reset by any assignment to pos. Zero-length matches at the end of the previous match are also ignored during split, which is why split // produces individual characters and not a trailing empty fragment.

Combining patterns: better and worse

perlre's "Combining RE Pieces" section gives a precise account of which match the engine prefers when a pattern admits more than one. The rules are mechanical, and worth knowing for the cases where intuition fails.

For two pattern pieces concatenated, ST:

- If S matches A better than A', then AB is better than A'B' (the leftmost piece's preference dominates).
- If S matches both A and A' equally, T's preference decides.

For alternation, S|T:

- `S` matching is always better than only `T` matching (left-to-right choice - Traditional NFA semantics).

For greedy and non-greedy quantifiers:

- `S{n,m}` is equivalent to trying `S{m}`, then `S{m-1}`, ..., then `S{n}` (greedy: maximum first).
- `S{n,m}?` is equivalent to trying `S{n}`, then `S{n+1}`, ..., then `S{m}` (non-greedy: minimum first).

For atomic groups, `(?>S)`:

- *Only the best match for* `S` is considered; backtracking is forbidden inside.

For lookaround, `(?=S)` / `(?<=S)`:

- Only the best match for `S` is considered (the same as atomic). This matters only if `S` has capturing parens whose contents are used outside.

The above rules describe *which match is chosen at a fixed starting position*. One additional rule covers position selection: **a match at an earlier position is always better than a match at a later position** - the bump-along property is more important than any preference within a single position.

Engine internals - what the optimiser does for you

A regex engine's compiler runs a small army of analyses on the pattern before matching begins. Knowing them lets you write patterns that *enable* them, rather than ones that defeat them.

First-character discrimination

If the pattern must begin with one of a small set of characters (`am|pm` ⇒ `[ap]`), the bump-along can scan the input for one of those characters using a fast inner loop, invoking the full engine only at candidate positions.

A pattern starting with `.`, or with a top-level alternation the engine cannot see through, defeats this. Modern engines (PCRE2, RE2, the Rust regex crate) go far beyond first-character discrimination - Boyer-Moore on prefix literals, byte-level SIMD scanning, Aho-Corasick fallback for many literal alternatives. Perl 5.44 does the simpler form well; the more aggressive techniques are why a benchmark may show RE2 outperforming Perl on inputs dominated by literal scanning.

Fixed-string check

If the pattern requires some literal substring (e.g. `Subject:` in `^Subject: (Re: )?(.*)`), the engine uses Boyer-Moore to skip to candidate positions cheaply. A naive pattern like `/this|that|these|those/` defeats this; rewriting as `/th(?:is|at|ese|ose)/` exposes the common prefix `th` for the optimiser to find.

Simple-repetition fast path

`x*`, `[a-f]+`, `.?` and other "single atom, simple quantifier" constructs are dispatched to a fast inner loop that bypasses the general engine machinery. Wrapping the atom in *capturing* parens disables this. Even non-capturing parens may, depending on the exact construct.

Length cognizance

If the pattern requires at least N characters, the engine can refuse to attempt matches in the last N positions of the string, and refuse to attempt matches at all in strings shorter than N.

Anchor implication

A pattern starting with `^` is attempted only once (or once per logical line under `/m`). Less obviously, a pattern starting with `.*` followed by an anchorable subpattern *can* be implicitly anchored, because by the time `.*` has consumed the whole string, no later starting position can possibly succeed. Modern engines recognise this; do not write `.*` at the start unless you mean it.

Compile caching

Compiling a regex to its internal form is non-trivial. Perl caches compiled patterns aggressively - patterns embedded in code are compiled once when the surrounding sub is compiled, and patterns assembled from interpolated variables are recompiled only when the resulting text actually changes. `qr//` makes this explicit. The `/o` modifier was the older mechanism; `qr//` has superseded it in idiom.

Matching many strings efficiently

If you have a fixed list of strings to match and the list is stable, building one big alternation is often the simplest win:

```
my @words = qw(the quick brown fox);
my $any = join '|', map quotemeta, @words;
my $re = qr/\b(?:$any)\b/;
for my $line (@lines) {
    next unless $line =~ /$re/;
    ...
}
```

`quotemeta` protects each word from containing accidental metacharacters. The `(?:…)` is non-capturing - unnecessary captures would cost more than they pay. If the list is large (thousands of strings), a specialised module that builds an Aho-Corasick or trie matcher will beat a giant alternation; the built-in engine compiles the alternation to an NFA that is fast but not asymptotically optimal for that workload.

Measuring

Claims about regexp speed are cheap; measurements are not. `Benchmark::timethese` makes it easy to compare two forms on real data:

```
use Benchmark qw(timethese);

my $text = "..." x 10000;

timethese(1000, {
    greedy     => sub { $text =~ /"(?:[^"\\]|\\.)*"/ },
    possessive => sub { $text =~ /"(?:[^"\\]++|\\.)*+"/ },
});
```

Always profile before optimising. "Obviously slow" patterns often aren't; "obviously fast" ones sometimes are pathological on real input. Friedl's framing remains the best summary: the fastest program is the one that finishes first.

Rules of thumb

- Write the pattern that reads best; profile.
- If matching is slow, look for nested quantifiers on overlapping classes.
- Fix them with possessive quantifiers, atomic groups, or unrolling.
- Anchor with `^`, `$`, `\A`, `\z`, or `\G` whenever the pattern can only legitimately start or end in those positions.
- Prefer classes to alternations of single characters.
- Prefer negated classes to non-greedy quantifiers when they describe the same set.
- Use `qr//` for patterns that run in a loop.
- Use `(?:…)` for groups you don't capture, or `/n` to suppress captures wholesale.
- Use `(*{...})` instead of `(?{...})` when the code block is advisory.
- When a real parser would be clearer than a pattern, write the parser.

See also

- The *quantifiers* chapter - greedy, non-greedy, possessive forms.
- The *groups and captures* chapter - atomic groups, recursive subpatterns, conditional patterns.
- The *alternation* chapter - ordering, leftmost alternative wins.
- The *substitution* chapter - `/g` and zero-length match termination in substitution.
- `qr` - precompile patterns for repeated use.
- `pos` - read or set the `/g` match position; also resets the zero-length-match flag.

Cross-engine comparison

Perl regexps are not the only kind. A reader who knows Perl well will, sooner or later, meet a regex written for another tool - `sed`, `awk`, an Emacs Lisp buffer search, a Go service, an embedded PCRE2 library - and be expected to read it. The syntactic differences are usually small, the semantic differences are sometimes substantial, and the architectural differences (which inputs make the engine catastrophically slow) can be decisive.

This chapter is the single page that answers "what does X look like elsewhere?" for the five engine families a Perl-trained reader is most likely to encounter. It is not a porting tutorial; it is the comparison reference.

Engines covered

Five families, picked because each is widespread in its niche *and* materially different from Perl. Engines that are widespread but largely Perl-shaped (Java's `java.util.regex`, Python's `re`, ECMAScript, .NET) are out of scope; the differences are minor and a Perl reader can read them on sight.

Engine	Where you meet it	Family
PCRE2	nginx, PHP `preg_*`, command-line `pcre2grep`	Perl-extended
Emacs	Emacs Lisp regex search, `M-x replace-regexp`	Its own family
POSIX BRE	`sed` (default), `grep` (default), `ed`	POSIX Basic
POSIX ERE	`awk`, `egrep` / `grep -E`, `sed -E` / `sed -r`	POSIX Extended
RE2 / Go `regexp`	Go's standard library, Cloud Bigtable, code search	Linear-time hybrid

Versions anchored against in this chapter: PCRE2 10.44, Emacs 30.x, POSIX 1003.1-2024, Go 1.22 (RE2 tracks Go's regexp engine). The values in the tables below were verified against these versions; they will drift over time, and engine-specific updates should re-verify before relying on a row.

What's not in scope

The following engines are deliberately omitted from the tables. Each is widespread enough that you may meet it; in each case, the differences from Perl are small enough that a Perl reader can read the regex with little adjustment.

- **Java `java.util.regex`** - Perl-derived; differences are flag spellings and a few escape conventions.
- **Python `re`** - Perl-derived; biggest delta is variable-length lookbehind support arrived later than in Perl.
- **ECMAScript** (JavaScript) - Perl-derived; lookbehind only since ES2018, no possessive quantifiers, sticky flag `y` instead of `\G`. Modern V8 / SpiderMonkey converge on Perl's feature set.
- **.NET regex** - Perl-shaped, plus *balanced groups* (`(?<name-pop>…)`), which are unique. Niche enough to skip.

- **Rust `regex`** - RE2 lineage; the story is essentially Go's with a slightly different feature surface (it has fixed-width lookbehind, RE2 does not).
- **Vim** - different magic-mode semantics; Vim-internal.
- **Boost.Regex** - C++ niche; closely tracks PCRE2.

The five families, in a paragraph each

PCRE2

Perl-Compatible Regular Expressions, version 2 (PCRE2 replaced the original PCRE library in 2015). The reference implementation for "Perl regex outside of Perl". Reads almost identically to Perl 5.44 patterns and supports nearly the same feature set: backreferences, lookaround, atomic groups, possessive quantifiers, recursive subpatterns, named captures, conditional patterns, callouts. PCRE2 also has a JIT compiler that is sometimes faster than perl's.

Where PCRE2 differs from Perl 5.44:

- `(?{...})` and `(??{...})` - Perl runs arbitrary Perl code; PCRE2 has *callouts* (`(?C)` / `(?Cn)`) which call back into the host application. Not a security difference; just a different interface.
- A handful of flag spellings differ (`PCRE2_CASELESS` vs `/i`, etc.) at the API level, not in pattern syntax.
- Recursive subpattern atomicity: `(?R)` is *atomic* in PCRE2 by default; Perl allows backtracking into a recursed group.
- `(*VERB)` set is largely the same, with PCRE2 adding `(*UTF)`, `(*UCP)`, and other startup options.

For most patterns, "looks the same" is correct.

Emacs

GNU Emacs's regex engine is its own family. On the surface it looks BRE-shaped - `\(...\)` for grouping, `\{m,n\}` for counts - but inside it has Perl-lite features (non-greedy quantifiers, word boundaries) and Emacs-specific features (syntax classes `\sw`, category classes `\cC`) that no other engine has.

Notable Emacs-specific constructs:

- **Buffer anchors**: `\`` (backtick) is start-of-buffer, `\'` is end-of-buffer. Closer to Perl's `\A` / `\z` than to `^` / `$`.
- **Syntax classes**: `\sw` matches a "word-syntax" character as defined by the current buffer's syntax table; `\s-` matches whitespace-syntax. This is mode-dependent - the same regex matches different characters in C-mode vs Lisp-mode.
- **Category classes**: `\cC` matches a character in category C. Used heavily for CJK text classification.
- **No** `\d` shorthand - Emacs requires `[0-9]` or `[[:digit:]]`. (`\w`, `\s`, `\b` exist, but `\d` does not.)
- **No** lookahead, lookbehind, or `\K`.

If you read Emacs Lisp in 2026 you will see `\\(\\)` patterns constantly - that is the doubled escape required because Emacs Lisp string literals consume one level of backslash before the regex parser sees the rest.

POSIX BRE - `sed`, `grep`, `ed`

POSIX Basic Regular Expressions. The engine you get from `sed`, `grep`, and `ed` *with no flags*. The most surprising thing for a Perl reader: the metacharacter set is *smaller*. Bare `+`, `?`, `{m,n}`, `(`, `)`, and `|` are *literal characters*. The metacharacter forms are the backslashed variants: `\+`, `\?`, `\{m,n\}`, `\(`, `\)`, `\|` (the last is a GNU extension; strict POSIX BRE has no alternation at all).

```
# POSIX BRE, finds the literal text "a+b":
echo 'a+b' | grep 'a+b'

# To match "one or more a", under BRE:
echo 'aaab' | grep 'a\+b'
```

Backreferences (`\1`–`\9`) work in BRE (this is older than POSIX ERE's no-backreferences rule). POSIX classes `[[:digit:]]` etc. work. `\<` and `\>` (GNU extension) provide word boundaries. Shortened character escapes like `\d`, `\s`, `\w` are *not* supported.

`grep` and `sed` ship the same engine but flag-default to different families: GNU `grep -P` invokes PCRE2, `grep -E` is ERE, default `grep` is BRE. `sed -E` is ERE, default `sed` is BRE.

POSIX ERE - `awk`, `egrep`, `sed -E`

POSIX Extended Regular Expressions, the engine `awk` defaults to and the one selected by `egrep` (now `grep -E`) and `sed -E` / `sed -r`. Bare `+`, `?`, `{m,n}`, `(...)`, and `|` are metacharacters here; this is the family closest to "what a Perl-trained programmer expects" among the POSIX-conformant set.

What strict POSIX ERE does *not* have, by spec:

- **Backreferences in the pattern** (`\1`, `\2`, ...). They are legal in `sed` substitution replacements but not in match patterns. GNU extensions add them; portable scripts should not rely on this.
- Lookaround.
- Possessive quantifiers, atomic groups.
- `\d`, `\s`, `\w`, `\b` (use `[0-9]`, `[[:space:]]`, etc.).

The most common surprise: an ERE `\(` is a *literal opening paren*. Backslashing punctuation is sometimes a metacharacter and sometimes a no-op, with the rule being the opposite of BRE. Memorise: BRE backslash *enables* metacharacters; ERE backslash *disables* them.

RE2 / Go `regexp`

Google's RE2 is a fundamentally different architecture. RE2 constructs an NFA at compile time and uses a *lazy DFA* (constructing DFA states on demand and caching them) at match time. The result is *guaranteed linear time* in the input length, no matter the pattern.

The trade-off: RE2 cannot support features that take regular languages out of the regular-language family. Specifically:

- **No backreferences in patterns.** `\1`, `\g{1}`, `\k<name>` - none. A pattern like `(.)\1` simply cannot be compiled.
- **No general lookaround.** RE2 supports anchors and `\b` (zero-width assertions), but not `(?=...)`, `(?<=...)`, `(?!...)`, or `(?<!...)`.
- **No atomic groups, no possessive quantifiers, no recursive subpatterns, no embedded code.**

What RE2 *does* support, and what makes it pleasant for the common case:

- Named captures with the Python-shaped syntax `(?P<name>...)`.
- ASCII by default for `\d`, `\w`, `\s`; Unicode mode enabled with `(?u)`.
- The `(?flags)` and `(?flags:...)` modifier syntaxes.
- A swap-default-greediness flag `(?U)` - under it, `*` is non-greedy and `*?` is greedy. The opposite of Perl.

If you have ever written a Perl regex that catastrophically backtracked, RE2 is what you write next. The cost is the missing features above; the upside is the engine cannot be DoSed by a malicious input.

Go's standard `regexp` package wraps RE2 and adds a few Go-isms, none of which change the fundamentals.

Feature comparison tables

The remainder of this chapter is row-major: rows are features, columns are engines. This layout optimises for the "I know one engine, what does X look like in another?" use case. Cell values are concise and may simplify edge cases - verify against the engine's own reference for production code.

Quantifiers, grouping, alternation

Feature	Perl 5.44	PCRE2	Emacs	POSIX BRE	POSIX ERE	RE2 / Go
* zero-or-more	meta	meta	meta	meta	meta	meta
+ one-or-more	meta	meta	meta	literal (use \+)	meta	meta
? zero-or-one	meta	meta	meta	literal (use \?)	meta	meta
{m,n} interval	meta	meta	\{m,n\}	\{m,n\}	meta	meta
| alternation	meta	meta	meta	| (GNU)	meta	meta
(...) grouping	meta	meta	\(...\)	\(...\)	meta	meta
(?:...) non-capturing	yes	yes	no	no	no	yes
Possessive *+, ++, ?+	yes	yes	no	no	no	no
Non-greedy *?, +?	yes	yes	yes	no	no	yes
Atomic group (?>...)	yes	yes	no	no	no	no

The single highest-value row for a Perl reader meeting BRE: + and ? are *literal*. A regex like `colou?r` reads as "matches `colou?r` exactly" under BRE. The rule is consistent - BRE's metacharacter set is small; ERE expanded it; Perl expanded it further.

Character shorthand classes

What does `\d` match? It depends. The columns below show what character set each engine treats as digits, word characters, and whitespace under default settings.

Shorthand	Perl 5.44 (default)	Perl 5.44 + `/u`	PCRE2 (default)	Emacs	POSIX BRE	POSIX ERE	RE2 / Go (default)
`\d`	ASCII (or Unicode under `/u`)	Unicode	ASCII	NO	NO	NO	ASCII; Unicode under `(?u)`
`\w`	ASCII or Unicode	Unicode	ASCII	yes (syntax-table	NO	NO	ASCII; Unicode under `(?u)`
`\s`	ASCII or Unicode	Unicode	ASCII	yes	NO	NO	ASCII; Unicode under `(?u)`
`\b` word boundary	yes	yes	yes	yes (`\<`/`\>` traditional)	NO	NO	yes
`\h` horizontal WS	yes	yes	yes	NO	NO	NO	NO
`\v` vertical WS	yes	yes	yes	NO	NO	NO	NO

Two surprises here for a Perl reader:

1. POSIX BRE and ERE both lack `\d`, `\w`, `\s`. The portable forms are `[0-9]`, `[[:alnum:]_]`, `[[:space:]]`. Tools that accept `\d` (GNU `grep`, `awk` in some implementations) do so as a non-portable extension.
2. Emacs has `\w` and `\s` but no `\d`. The `\s` character is followed by a syntax-class character - `\s-` for whitespace, `\sw` for word - which is unique to Emacs.

POSIX bracket classes like `[[:digit:]]` and `[[:alpha:]]` work in every engine in the table; they are the portable spelling.

Anchors and newline handling

Feature	Perl 5.44	PCRE2	Emacs	POSIX BRE / ERE	RE2 / Go
^ start-of-string	yes	yes	start-of-buffer	yes	yes
^ after \n under multi-line	/m	(?m)	always (line-oriented)	not specified	(?m)
$ end-of-string	yes	yes	end-of-buffer	yes	yes
$ before final \n	yes	yes	no	varies	yes
. matches \n	/s	(?s)	no	no	(?s)
\A / \z	yes	yes	\` / \'	no	yes (\A, \z)
\Z (before optional final \n)	yes	yes	no	no	no (use \z)
\G (last-match position)	yes	yes	no	no	no

Emacs uses \` and \' for the buffer anchors, where most modern engines use \A and \z. The Emacs spelling is older; it predates POSIX. Mixing up ^ and \A matters in line-oriented Emacs code more than in Perl, because Emacs matches are often invoked in contexts that scan an entire buffer.

\G is Perl-specific (and PCRE2). RE2 / Go uses a different API on the *match object* (a Loc method that returns the next position) rather than embedding the position in the pattern.

Lookaround and atomic constructs

Feature	Perl 5.44	PCRE2	Emacs	POSIX	RE2 / Go
Lookahead (?=…) / (?!…)	yes	yes	NO	NO	NO
Fixed-width lookbehind (?<=…)	yes	yes	NO	NO	NO
Variable-length lookbehind	yes (exp.)	yes	NO	NO	NO
\K keep-left	yes	yes	NO	NO	NO
Atomic group (?>…)	yes	yes	NO	NO	NO
Possessive quantifiers	yes	yes	NO	NO	NO

Effectively: only PCRE2 (and other Perl-derived engines outside this table - Java, Python, .NET) supports lookaround. POSIX tools and RE2 / Go simply do not have it. Patterns that need lookaround do not port across this divide.

Captures and backreferences

Feature	Perl 5.44	PCRE2	Emac	POSIX BRE	POSIX ERE	RE2 / Go
Numbered captures	yes	yes	yes	yes	strict no, GNU yes	yes
Backreferences in pattern (\1–)	yes (1–99)	yes	yes	yes (\1–\9)	strict no, GNU yes	NO
Named captures (?<name>…)	yes	yes	NO	NO	NO	yes ((?P<name>…))
Branch reset (?|…)	yes	yes	NO	NO	NO	NO
Recursive subpatterns (?R)	yes	yes	NO	NO	NO	NO
Conditional patterns (?(c)y|n)	yes	yes	NO	NO	NO	NO
(?{…}) embedded code	yes	callouts	NO	NO	NO	NO
@{^CAPTURE} array	yes	NO	NO	NO	NO	NO (different API)

The single most important takeaway: **RE2 / Go regexp does not support backreferences**. Patterns like /(.)\1/ cannot be compiled. This is not an engine bug; it is the price RE2 pays for guaranteed linear time. Patterns containing backreferences are not regular languages and a true DFA cannot match them in linear time.

If you are porting Perl regexps to a Go service, the audit question is "does this pattern use backrefs?" - if so, the conversion is not mechanical.

Engine-architecture summary

Property	PCRE2 / Perl 5.44	Emacs	POSIX BRE / ERE (typical impl)	RE2 / Go
Algorithm	Traditional NFA + JIT (PCRE2)	Traditional NFA	DFA (GNU) / NFA (others)	Hybrid NFA → lazy DFA
Worst-case time	Exponential	Exponential	Linear (DFA)	Linear (guaranteed)
Backtracking model	yes	yes	DFA: no	no
Catastrophic backtracking risk	yes	yes	no (DFA)	no
Backreferences supported	yes	yes	varies	NO

POSIX BRE/ERE is implementation-dependent: GNU `grep` and `awk` use a hybrid DFA-with-NFA-fallback that is fast and not subject to catastrophic backtracking; older traditional implementations (historical `awk`, traditional `grep` on some BSDs) use a backtracking NFA. The portable assumption is "linear time, but verify if your input may be

hostile".

Selecting an engine when you have the choice

If you are writing the program (not reading code that already chose), the engine you pick is mostly determined by the language you are programming in. The exceptions are when the same library ships multiple engines, and when an external regex tool is your choice:

- **A Perl program**: Perl's built-in. No reason to reach outside.
- **A C program embedding regex**: PCRE2. The standard library's `regex.h` is POSIX BRE/ERE; PCRE2 is what you want for Perl-like behaviour.
- **A Go service handling untrusted input**: `regexp` (RE2). The linear-time guarantee is the headline feature; the missing backreferences and lookaround are usually acceptable.
- **A Rust program**: the `regex` crate. RE2-lineage; same linear-time guarantee.
- **Shell scripting**: prefer `grep -E` (ERE) and `sed -E` (ERE) over the BRE defaults for Perl-readability. Use `grep -P` (PCRE2) only when ERE genuinely is not enough.
- **An Emacs Lisp function**: Emacs's. Otherwise leave Emacs.

See also

- The *character classes* chapter - the shorthand-by-engine row appears in its L3 sidebar.
- The *anchors and assertions* chapter - the lookaround and anchor rows appear in its L3 sidebar.
- The *groups and captures* chapter - the captures-and-backreferences row appears in its L3 sidebar.
- The *performance* chapter - Traditional NFA model and catastrophic backtracking, the architectural reason RE2 exists.
- **Basics** - `m//`, `s///`, the binding operators `=~` and `!~`, what counts as a metacharacter, how to escape, the four-phase parsing model.
- **Character classes** - bracketed classes `[…]`, negated classes, shorthand `\d \w \s`, POSIX classes, the extended bracketed form `(?[…])`, the IP-address worked example.
- **Anchors and assertions** - `^`, `$`, `\b`, `\A`, `\z`, `\G`, `\K`, lookahead, lookbehind, the `\b{…}` Unicode boundaries, juxtaposition-as-AND.
- **Quantifiers** - `*`, `+`, `?`, `{n,m}`, greedy vs. non-greedy vs. possessive, the principles of the match.
- **Groups and captures** - `(...)`, `(?:...)`, named captures, backreferences, atomic groups, recursive subpatterns, conditional patterns, branch reset.
- **Alternation** - `|`, precedence, common-prefix factoring, empty alternatives, branch reset.

- **Modifiers** - `/i`, `/m`, `/s`, `/x`, `/xx`, `/g`, `/c`, `/r`, `/e`, `/n`, `/p`, `/o`, `/a`, `/aa`, `/u`, `/l`, `/d`, inline forms `(?i…)`, scoped forms `(?i:…)`, the four-phase parsing model, `use re 'strict'`.
- **Substitution** - `s///` in depth: the replacement string, `/e`, `/ee`, `/r`, `\K` in substitution, zero-length match termination.
- **Unicode** - `\p{…}`, `\P{…}`, `\X`, scripts, the charset modifiers, case folding, script runs, encoding vs. character.
- **Performance** - backtracking, catastrophic patterns, unrolling the loop, atomic groups, possessive quantifiers, recursive patterns, embedded code, special backtracking control verbs, internal optimisations the engine applies on your behalf.
- **Cross-engine** - comparison against PCRE2, Emacs, POSIX BRE, POSIX ERE, and RE2 / Go.

A first round-trip

The shortest useful regexp program - find three-letter words repeated back to back, separated by a single space:

```
my $text = "I said the the other day";
if ($text =~ /\b(\w{3})\s\1\b/) {
    print "Repeated: $1\n";        # Repeated: the
}
```

- `\b` is a word boundary - the pattern only fires at word edges.
- `(\w{3})` captures exactly three word characters into `$1`.
- `\s` is one whitespace character between the two copies.
- `\1` is the backreference: match the same three characters again.

Every more complicated pattern in the guide is layered on that idea - anchor where you need to, describe what you want, capture what you want back.

Conventions in the examples

- Example output appears as an inline `#  …` comment beside the expression that produces it.
- Examples assume default modifiers unless they show `/i`, `/x`, etc.
- Where a pattern is shown on its own (no =~), treat it as a fragment the surrounding text will combine with a string.
- When a Unicode character is needed, examples use `\x{263a}` or `\N{GREEK SMALL LETTER SIGMA}` so the rendered text matches the source.

Related reference pages

The guide is a teaching reference; these are the operator references the guide refers to throughout.

- `m` - the match operator.

- `s` - substitution.
- `qr` - compile a pattern for reuse.
- `split` - split a string on a regexp.
- `pos` - read or set the `/g` match position.
- `quotemeta` - escape metacharacters in a string.
- `tr` - character-by-character translation. Often confused with regexp substitution but unrelated in semantics.

4.2.4 Performance

How to make Perl run fast on pperl, and - just as important - how to tell whether it already does. This guide is the actionable layer: where the classic Perl performance advice still holds, where pperl's JIT and auto-parallelizer have quietly made it irrelevant, and where pperl rewards code shaped in ways upstream Perl never cared about.

It is deliberately not a tour of the machinery. The JIT, the parallel scheduler, the native-module dispatch and the FFI each have their own chapter in the *architecture guide*; this guide tells you what to *write* so they fire, and points at those chapters for the *why*.

Who this is for

A working Perl programmer with a program that is too slow, or one they want to keep fast. You know how to write the code; you want to know which habits pay off here, which are now dead weight, and how to measure the difference without guessing.

You do not need to understand Cranelift, Rayon, or the interpreter's inner loop. Where a behaviour depends on them, this guide states the behaviour and links the chapter that explains it.

The one rule that survives everything

Measure first. Every other rule in this guide is conditional on a measurement you have not yet taken. Perl performance intuition, honed on the upstream interpreter, mispredicts on pperl in both directions: code you expect to be slow is sometimes compiled to native machine code and runs faster than C; code you expect the JIT to rescue sometimes falls back to the interpreter because one string operation crept into a numeric loop.

The profiler does not lie and your intuition, ported from another runtime, does. Start at *Measuring*.

How this guide is organised

Measuring

You cannot tune what you have not measured, and on pperl the usual measuring tools split into three groups: the ones that work unchanged, the ones that work because they are plain Perl running on a faster interpreter, and the one everybody reaches for that does not work at all. This chapter sorts them out and shows how to read what the working ones tell you.

The tool that does not work: Devel::NYTProf

`Devel::NYTProf` is the standard line-and-subroutine profiler on upstream Perl. It is an XS module - it hooks the interpreter through a C extension. pperl ships no XS layer, so `Devel::NYTProf` does not load. Neither does any other XS-based profiler (`Devel::DProf`, `Devel::Profiler`'s XS paths).

This is not a gap to be worked around with a pure-Perl reimplementation of NYTProf's output format. What pperl offers instead is visibility into the layer that actually decides performance - the JIT - plus full transparency to the platform profiler. Use those.

Where the time goes: the JIT log

pperl does not currently ship a line-or-sub profiler of its own (the one it once had belonged to a retired execution engine and never saw the current runtime). What it does ship is precise visibility into the optimization layer, which for pperl-specific tuning is usually the question you are actually asking: is my hot loop compiled, and if not, why not?

- `--jit-stats` prints counters at exit: candidates found, loops compiled, entries run compiled, guard failures, deopts.
- `PPERL_JIT_LOG=/path/file` appends one line per compiler event. The `compile skip reason=...` lines name the exact operation that kept a loop interpreted.

For symbol-level CPU attribution, use the platform profiler on a release build - `perf record ./target/release/pperl script.pl` then `perf report`. pperl is a native binary; perf sees straight through it, including JIT-compiled frames.

Whole-program timing: the benchmark runner

When the question is "how fast is pperl on this, versus system perl, with and without JIT and parallelism", the project benchmark runner answers it directly. `bench/run-perlbench` always runs `/usr/bin/perl` as the baseline (normalised to 100) and compares pperl in up to four configurations.

```
./bench/run-perlbench                # no JIT, no parallel
./bench/run-perlbench +J             # add the JIT-enabled run
./bench/run-perlbench +P             # add the parallel run
./bench/run-perlbench +J+P           # all four combinations
./bench/run-perlbench -t arith       # only benchmarks matching
↪"arith"
```

Every runner shares one iteration count, derived from the slowest runner's empty-loop rate, so the reported ratios compare like with like. The `+J` / `+P` columns are exactly the comparison you want when deciding whether a piece of code benefits from the JIT or the parallelizer: if `+J` is no faster than the plain run, the loop is not being compiled, and *Writing fast pperl* explains why and what to change.

Plain-Perl tools that work: Benchmark and Time::HiRes

The classic in-program measuring tools are plain Perl. They run on pperl through the normal module path - `Benchmark` is reachable on the standard include path and executes as Perl on the pperl interpreter; `Time::HiRes` is provided natively. Both behave exactly as a Perl programmer expects.

`Benchmark` for A/B comparison of two ways to write the same routine:

```
use Benchmark qw(cmpthese);

my @data = (1 .. 100_000);

cmpthese(-3, {            # run each for ~3 CPU-seconds
    grep_count => sub { my $n = grep { $_ % 2 == 0 } @data },
    loop_count => sub { my $n = 0; $_ % 2 or $n++ for @data },
});
```

`Time::HiRes` for ad-hoc wall-clock timing of a single block:

```
use Time::HiRes qw(time);

my $t0 = time;
do_the_work();
printf "took %.3f s\n", time - $t0;
```

A caution that matters more on pperl than on upstream: `Benchmark` drives your code by calling a code reference over and over, and that per-call dispatch is Benchmark's own overhead, not your code's. A loop *inside* the callback still JIT-compiles (loops compile wherever they live, file scope or sub body), but for small bodies the harness call cost dominates and the comparison understates the real speed. When the thing you are timing is a loop that should JIT, prefer `bench/run-perlbench` or a `Time::HiRes` span around the loop in situ (see *JIT compilation*).

Turning the optimisers off to isolate a cause

Two flags let you attribute a speedup - or a slowdown - to the right layer:

```
pperl --no-jit script.pl          # interpreter only, no native
↪compilation
pperl --no-parallel script.pl     # single-threaded, no Rayon
↪dispatch
```

Run a program three ways - default, `--no-jit`, `--no-parallel` - and the differences tell you which optimiser is carrying the work. If `--no-jit` barely changes the time, the JIT was not engaging on this code, and the loop shape is the thing to look at. If `--no-parallel` barely changes it, the parallelizer declined the loop - usually because of a side effect it could not rule out (see *Parallel execution*).

A measuring discipline

1. Time the program default versus `--no-jit` versus `--no-parallel`. The differences tell you which layer carries the work.
2. If a hot routine is a loop you expected to be compiled, confirm with `run-perlbench +J` or by timing it `--no-jit` versus default. A loop that does not speed up under the JIT is shaped wrong, not slow.
3. Only now change code. Re-measure the same way. Keep the change if the number moved and the program still produces the same output.

Performance claims are cheap; the JIT log and the benchmark ratio are not. Trust the measurement.

See also

- *Writing fast pperl* - once you have found the hot loop, how to shape it so the JIT and parallelizer take it.
- *Idioms* - which classic optimisations are still worth applying before you reach for the measuring tools.
- *JIT compilation* - what the JIT compiles and why interpreted op timings disappear for compiled loops.
- *Parallel execution* - what the `+P` column and `--no-parallel` flag are exercising.
- *Regex performance* - if the measurement points at a regexp, the cause and cure live there, not here.

Idioms

Perl carries decades of performance folklore - habits that paid off on the upstream interpreter and got passed down as rules. Some of that advice is timeless. Some of it is now actively pointless on pperl, because the JIT or the compile-time folder already does the work. And a few new habits matter here that no upstream Perl programmer ever needed.

This chapter triages the classic idioms into three buckets:

- **Still valid** - the advice holds; the cost it avoids is real on pperl too.
- **Made moot** - pperl's compiler or JIT already eliminates the cost, so the contortion buys nothing and only hurts readability.
- **New** - relevant *because* of how pperl compiles, with no upstream precedent.

A blanket caveat runs through the whole "made moot" bucket: the JIT only takes a loop when the loop is *shaped* for it. A loop heavy with subroutine calls, regexp matches, or string building is not compiled and runs on the interpreter - where the classic advice re-applies in full. So "the JIT handles it" is true *for loops the JIT handles*. *Writing fast pperl* is the companion chapter on making sure your loop is one of them.

Still valid

These survive the move to pperl unchanged. The cost they avoid is paid on the interpreter, and most hot code touches the interpreter somewhere.

- **Hoist invariant work out of loops.** Computing the same value every iteration is wasted work whether the loop is interpreted or compiled. The JIT does not hoist arbitrary expressions for you; move the invariant out yourself.
- **Avoid needless copies of large structures.** Passing a big array by value, or slicing it just to count, copies. Pass a reference, use an aliased loop variable, or work in place. pperl's value model has the same copy costs Perl always had.
- **Precompile regexps used in a loop with `qr`.** The regex engine and its compile cache work as on upstream; a pattern built fresh each iteration pays to recompile. This is regex territory - see *regex performance*.
- **Choose the right data structure.** A hash lookup beats a linear scan at the same sizes it always did. Algorithmic complexity is unaffected by any runtime; an $O(n^2)$ loop is slow even fully JIT'd.
- **Read files in the right-sized chunks.** Line-by-line versus slurp-and-split is an I/O question, and I/O is not JIT-compiled. The upstream trade-offs carry over directly.
- **Prefer built-in list operations to hand-rolled loops where the built-in is native.** `sum`, `min`, `max`, `first` from `List::Util` run at built-in speed with no XS layer (see *native modules*). A hand-written accumulator loop *may* JIT to the same speed, but the native built-in gets you there without depending on the loop shape.

Made moot

These were real wins on upstream Perl. On pperl the compiler or JIT already removes the cost, so applying the idiom by hand buys nothing and usually costs clarity.

Hand-inlining hot arithmetic

The old habit of unrolling a tight arithmetic loop, or manually inlining a small arithmetic helper to dodge call overhead, targets a cost the JIT removes. A numeric `for` or `while` loop over integer or float accumulators is compiled to native machine code, with the loop variables held in CPU registers - see *JIT compilation*. Hand-unrolling such a loop makes it longer and harder to read without beating the compiler.

```
# Don't hand-unroll this. The JIT compiles the natural form to
# native code with $i and $sum in registers.
my $sum = 0;
for my $i (1 .. 1_000_000) {
    $sum += $i * $i;
}
```

The contortion only pays off if the loop does *not* JIT - and if it doesn't, the fix is to make it JIT (remove the call or string op that disqualified it), not to unroll it by hand.

DEBUG-constant logging guards

The idiom of guarding expensive logging behind a compile-time false constant -

```
use constant DEBUG => 0;
# ...
warn "state: @stuff\n" if DEBUG;
```

- costs exactly nothing on pperl, which is the whole point of the idiom, and pperl delivers it completely. When `DEBUG` is a false constant, the compiler folds the `if DEBUG` guard at compile time: the disabled statement collapses to a no-op, with no condition test and no branch left in the compiled program. The interpolation `"@stuff"` is never built. You can leave guarded logging in hot paths with a clear conscience; a false `DEBUG` constant erases it.

This is the idiom working *as designed* - keep writing it. What is moot is any further cleverness on top: you do not need to comment out the logging, move it behind a sub, or strip it for production. The constant does it.

Folding constant expressions yourself

Precomputing a constant arithmetic result, or a constant call to a pure built-in, and storing it in a variable to "avoid recomputing it" buys nothing - the compiler folds constant expressions and constant-argument pure built-ins (`sin`, `cos`, `sqrt`, `chr`, `ord`, and the like) at compile time, as long as the call is in a valid domain. `my $two_pi = 2 * 3.14159265358979;` and `my $root = sqrt(2);` are computed once, at compile time, whether or not you "help" by hoisting them.

The one nuance: folding is skipped for arguments outside a function's domain (`sqrt` of a negative, `log` of zero), so those fall through to run time and croak there exactly as they should. You never need to fold a constant by hand to gain speed.

Stripping dead constant-condition loops

A `while (0) { ... }` or a block guarded by a constant-false condition is eliminated at compile time - the body is removed, not merely skipped at run time. (A C-style `for (; 0; ...)` is handled slightly differently: the body is never entered, though the loop scaffolding op may remain.) Conditionally-compiled-out code behind a false constant carries no run-time weight; you do not need to physically delete it to make the program faster.

New

These have no upstream analogue. They matter because of how pperl compiles and parallelizes, and getting them wrong silently forfeits the largest speedups pperl offers.

Keep numeric hot loops numeric

The single most valuable new habit. The JIT compiles a numeric loop and, when parallelization is enabled (the default) and the loop is a clean reduction over a known range, dispatches it across threads - the difference between a 76× and a 431× speedup on the Mandelbrot benchmark (see *parallel execution*).

A string operation inside that loop forfeits the parallel half. When the JIT sees a string variable in the loop - a `.=` concat target, a string accumulator - it demotes the loop to a *sequential* compiled loop: still native code, but no longer spread across threads.

```
# Numeric reduction - JIT-compiled and eligible for parallel
 ↪dispatch.
my $total = 0;
for my $i (0 .. $n) {
    $total += compute($i);      # (only if compute() itself stays
 ↪numeric/inlinable)
}

# A string built in the same loop demotes it to sequential JIT -
# the parallel dispatch is gated off the moment a string var
 ↪appears.
my $log = "";
for my $i (0 .. $n) {
    $total += $i * $i;
    $log .= "$i ";              # forfeits parallel dispatch for this
 ↪loop
}
```

If you need both the numeric reduction and the string accumulation, split them into two loops: the numeric one parallelizes, the string one runs as a sequential compiled loop. Mixing them costs you the parallel speedup on the numeric work for no gain on the string work.

Shape reductions as accumulate-never-reset

The parallelizer recognises a reduction variable by seeing it accumulated and never reset inside the loop body (the analysis subtracts reset patterns from accumulation patterns; a variable that is both is not treated as a reduction). Writing your accumulator in the plain `$sum += ...` form, declared *before* the loop and never re-initialised inside it, is what lets the loop be claimed for parallel execution.

```
my $sum = 0;                    # declared outside - a reduction
for my $x (@data) {
    $sum += weight($x);         # accumulate, never reset →
 ↪parallelizable
}
```

Re-initialising the accumulator inside the loop, or interleaving a reset, defeats the recognition and the loop runs sequentially. This is covered in full under *reduction detection*.

Let the loop JIT - don't hide it in a sub

The JIT compiles *loops*, not subroutine bodies. A hot numeric loop written at file scope or inside a larger routine compiles fine; the same arithmetic refactored so each iteration is a subroutine call does not - the call frame is interpreter territory, and the per-call overhead is exactly what the JIT cannot remove. When a loop is hot and numeric, keep its body inline rather than factoring each iteration into a sub for tidiness. Factor for clarity elsewhere; keep the inner loop flat.

See also

- *Writing fast pperl* - the positive program: how to shape a loop so the JIT and parallelizer take it.
- *Measuring* - confirm an idiom actually moved the number before and after applying it.
- *JIT compilation* - what compiles, why hand-inlining arithmetic is redundant.
- *Parallel execution* - the reduction analysis and the side-effect gate behind the "new" idioms.
- *Sorting* - the sort-specific idioms (Schwartzian, Guttman-Rosler) get their own chapter.

Sorting

`sort` is where a surprising amount of a program's time hides, because the cost is not the sort - it is the comparator, called O(n log n) times. A comparator that does real work per call multiplies that work by the log of the list size. This chapter is about recognising when the comparator is the bottleneck and the two transforms that fix it.

The reference page for `sort` defines the operator, the comparator contract, and the `$a`/`$b` mechanics. This chapter assumes you know those and focuses on speed.

The comparator is the cost

`sort` itself is a stable mergesort - fast, and not something you can improve from Perl. What you control is how much work happens *inside* each comparison. Two facts set the budget:

- The comparator runs O(n log n) times. For a 100,000-element list that is well over a million calls.
- On pperl, the comparator block is a callback, and callbacks run on the interpreter. The JIT compiles loops, not `sort` comparator bodies (see *JIT compilation*), so you cannot rely on the JIT to rescue an expensive comparator the way it rescues a numeric loop. The comparator's cost is paid in full, every call.

That second fact is the pperl-specific reason the transforms below matter as much here as on upstream Perl - arguably more, because the JIT's speedups elsewhere make an un-transformed sort stand out more sharply in a profile.

The diagnosis is always the same: if *Measuring* shows time concentrated in a comparator, the key is being recomputed on every comparison. Compute it once instead.

The Schwartzian transform

When the sort key is expensive to derive from each element - a substring extraction, a field lookup, a length, a computation - the naive sort recomputes it on every comparison:

```
# Slow: the key (-M $_) is computed twice per comparison, O(n log n)␣
↪times.
my @sorted = sort { -M $a <=> -M $b } @files;
```

The Schwartzian transform computes each key exactly once, sorts on the precomputed key, then discards it - a `map`, a `sort`, a `map`:

```
my @sorted =
    map  { $_->[1] }                        # 3. strip the key, keep␣
↪the element
    sort { $a->[0] <=> $b->[0] }            # 2. sort on the␣
↪precomputed key
    map  { [ -M $_, $_ ] }                  # 1. compute the key once␣
↪per element
    @files;
```

Read it bottom to top: the lower `map` builds a `[key, element]` pair for each element (n key computations), the `sort` compares only the cheap precomputed `$_->[0]` (no key computation in the comparator), and the upper `map` throws the key away. The expensive key is computed n times instead of O(n log n) times.

Use it whenever the key costs more than a numeric or string compare to derive. For a 100,000-element list sorted by a key that takes a microsecond to compute, the naive form spends seconds in key computation alone; the transform spends a tenth of a second.

The pieces are `map` and `sort` - the `sort` reference page shows the same transform as a worked example.

The Guttman-Rosler transform

The Schwartzian transform sorts arrayrefs, which costs an arrayref allocation and a dereference per comparison. When the key can be encoded as a fixed-width *string* prefix, the Guttman-Rosler transform (GRT) goes further: it packs the key and the element into a single string, sorts those strings with the default string `sort` (no comparator block at all), then strips the prefix.

```
my @sorted =
    map  { substr($_, 8) }                          # 3. strip the␣
↪8-char key prefix
    sort                                            # 2. plain␣
↪string sort, no comparator
    map  { pack('N', $_->{priority}) . $_->{name} } # 1. fixed-width␣
↪key + payload
    @records;
```

The win has two parts. First, the key is precomputed once, as in the Schwartzian. Second - and this is the GRT's distinct advantage - the `sort` has *no comparator block*, so there is no per-comparison callback at all. Default string `sort` compares the packed strings directly, which avoids the callback cost entirely. Removing the comparator callback matters more on pperl precisely because that callback would run on the interpreter.

The cost of the GRT is that the key must be encodable so that a bytewise string comparison produces the order you want:

- Integers must be packed big-endian (`pack 'N'` / `pack 'Q>'`) so that byte order matches numeric order, and offset to be non-negative (string comparison has no sign bit).

- Floats and signed numbers need bias encoding to sort correctly as bytes - fiddly enough that the Schwartzian is usually the better choice unless the sort is genuinely hot.
- Every key must be the *same width*, or a short key's payload bleeds into the comparison.

Reach for the GRT when the sort is hot, the key is an unsigned integer or a string, and the Schwartzian still shows up in the profile. For anything else, the Schwartzian is simpler and almost always fast enough.

Cheaper comparators without a transform

Not every slow sort needs a transform. Sometimes the comparator is just written more expensively than it needs to be.

- **Compare numbers with <=>, strings with cmp.** Using `cmp` on numbers stringifies both sides every comparison; using `<=>` on strings is a numification trap. Match the operator to the data.
- **Order tiebreakers cheapest-first.** A multi-key comparator with `||` stops at the first non-zero result. Put the discriminating, cheap key first so most comparisons never reach the expensive one:

```
sort { $a->{rank} <=> $b->{rank}        # cheap, decides most
↪pairs
    || $a->{name} cmp $b->{name} }      # only when ranks tie
       @rows;
```

- **Sort once, reverse separately.** To sort descending, sorting ascending and calling `reverse` is often clearer and no slower than inverting the comparator, and it keeps the comparator in its cheapest form.

Decision order

1. Profile. If *Measuring* does not put the time in the sort, leave it alone.
2. If the comparator recomputes a key, apply the Schwartzian transform. This fixes the great majority of slow sorts.
3. If the sort is still hot and the key is an unsigned integer or a string, consider the Guttman-Rosler transform to drop the comparator callback entirely.
4. Re-measure after each change.

See also

- `sort` - the operator reference: comparator contract, `$a`/`$b`, stability.
- `map` - the transform half of both the Schwartzian and Guttman-Rosler forms.
- `reverse` - descending sorts without inverting the comparator.
- *Measuring* - confirm the comparator is the cost before transforming it.
- *JIT compilation* - why a `sort` comparator block is not JIT-compiled, so its cost is paid in full.

Writing fast pperl

The other chapters in this guide are about removing cost. This one is about earning the speedups pperl offers that no other Perl has: the JIT that compiles hot loops to native code, and the parallelizer that spreads a qualifying loop across CPU cores. Both are automatic - you do not call them, annotate for them, or configure them. But they only fire on code *shaped* the right way, and a single misplaced operation can shut either one off silently.

This chapter is the positive program: write the loop like this, and the compiler takes it. The *JIT* and *parallel execution* chapters explain the machinery; this one tells you what to type.

The shape the JIT wants

The JIT targets **loops** - `for` and `while` - whose bodies are arithmetic and comparison over numeric variables. When it recognises that shape, it compiles the body to native machine code and holds the loop variables in CPU registers instead of on the interpreter stack. That is where the order-of-magnitude speedups come from.

A loop the JIT compiles cleanly:

```
my $sum = 0;
for my $i (1 .. 1_000_000) {
    $sum += $i * $i;
}
```

Integer and floating-point arithmetic (+, -, *, /), comparisons (<, >, <=, >=, ==), and control flow (`if`, `last`) inside the loop all compile. Nested loops compile too, several levels deep, each level connected to the next. The natural, idiomatic form is the fast form - you do not unroll, hoist into temporaries, or otherwise distort it.

What pushes a loop back onto the interpreter

The JIT declines a loop whose body does work it cannot prove it can reproduce exactly. When it declines, the loop runs on the interpreter - at interpreter speed, which is still fast for non-numeric work, but without the native-code multiplier. The coverage is broad (arithmetic, strings, array and hash elements, push/pop, map/grep, numeric `sort`, `join`/`split`, regex matches and constant substitutions, calls to small pure subs, and the common `Scalar::Util`/`List::Util` operations - `blessed`, `reftype`, `weaken`, `sum`, `min`, `max` and friends), so the practical disqualifiers are fewer than you might expect:

- **Calls to subs with side effects.** A call compiles when the sub's body is a `my (...) = @_;` prelude plus one pure numeric expression - including recursion. A sub that touches globals, does I/O, uses `return`, or takes references to `@_` declines the loop. Keep hot helpers small and pure and the call costs almost nothing; factoring work OUT of a sub is no longer the advice.
- **I/O.** A `print`, a file read, a system call - I/O-bound work does not benefit from native compilation and is not compiled.
- **Regex beyond the compiled forms.** Scalar matches and `s/pat/constant/` (including `/g`) compile; `s///e`, scalar `m//g`, and patterns with embedded code do not.

See *regex performance* for tuning the engine's own work.

- **Magic in any form.** Tied variables, overloaded objects, `local`, taint mode.

None of these is an error; the loop simply runs the interpreted way. But if a loop you expected to be fast is not, run it `--no-jit` versus default (*Measuring*): if the times match, the JIT was already declining it. Set `PPERL_JIT_LOG=/tmp/jit.log` and read the `compile skip reason=...` lines to see exactly which operation is responsible (*JIT compilation*).

Keep strings out of numeric loops

The JIT *does* handle one string operation - `.=` concat-assign, and clearing a string with `$x = ""` - through calls back into the runtime. So a string-building loop still compiles. But this is the one case where the shape of the loop changes *which optimisers* apply, and it is the highest-value rule in this chapter.

When the JIT sees a string variable in a loop, the loop compiles as a **sequential** native loop - but parallel dispatch is gated off. The parallelizer needs the loop body free of the shared-mutable-state that string building requires, so the moment a string variable appears, the loop is no longer spread across threads. It still runs as compiled native code; it just runs on one core.

The practical consequence: a numeric reduction that *would* parallelize loses that parallelism if you build a string in the same loop.

```
# Parallelizable: pure numeric reduction.
my $total = 0;
for my $i (0 .. $n) {
    $total += $i * $i;
}

# Same arithmetic, but the string build demotes it to a
# single-core compiled loop - the parallel speedup is forfeited.
my $report = "";
my $total  = 0;
for my $i (0 .. $n) {
    $total  += $i * $i;
    $report .= "$i ";
}
```

If you need both, split the work into two loops. The numeric loop parallelizes; the string loop runs as a sequential compiled loop. Mixing them costs the parallel speedup on the numeric half and gains nothing on the string half.

The shape the parallelizer wants

When parallelization is enabled - it is on by default; `--no-parallel` turns it off - a JIT-compiled loop is dispatched across threads if the analysis can prove it safe. Two conditions carry the most weight:

A reduction shaped as accumulate-never-reset. The parallelizer recognises a reduction variable by seeing it accumulated and never reset inside the loop. Declare the ac-

cumulator before the loop, use the plain += form, and never re-initialise it inside the body:

```
my $sum = 0;                    # declared outside the loop
for my $x (@data) {
    $sum += score($x);          # accumulate only → recognised as a␣
↪reduction
}
```

Re-initialising `$sum` inside the loop, or interleaving a reset, defeats the recognition (the analysis subtracts reset patterns from accumulation patterns) and the loop runs sequentially. Full detail in *reduction detection*.

A loop body free of observable side effects. The analyzer is deliberately conservative. Any of these disqualifies a loop from parallel dispatch:

- I/O (`print`, open, file reads)
- Writes to package/global variables
- Subroutine calls not proven pure
- Side-effecting regex operations (`s///`)

A missed parallelization is safe - the loop just runs sequentially. An *incorrect* one would be a bug, so the analysis errs toward declining. That conservatism is why keeping the loop body small, local, and free of globals is what gets it parallelized.

map and grep compile too

`map` and `grep` with pure expression blocks compile to native loops:

```
my @out      = map  { $_ * $k + 1 } @data;
my @filtered = grep { $_ % 7 }      @data;
```

The block must be a side-effect-free expression over `$_`, loop variables, and constants; mutating `$_` (which writes through to the source array) or producing a boolean as the map VALUE declines. Like all array work, compiled map/grep runs sequentially - array state gates parallel dispatch off - but the native-loop speedup over the interpreted op machinery is itself several-fold.

When the work is not a loop

Not every program is a numeric loop, and the JIT and parallelizer have nothing to offer code that is call-bound, I/O-bound, or string-shaped. For that code the rest of this guide applies in full:

- Pick native built-ins over hand-rolled loops where they exist - `List::Util` runs at built-in speed.
- Apply the classic still-valid idioms from *Idioms*: hoist invariants, avoid copies, precompile regexps.
- Transform expensive sorts (*Sorting*).

- Call C through the *FFI* only when the call frequency is low - the per-call marshalling cost makes it the wrong tool inside a hot inner loop.

A checklist for a hot loop

1. Is the body expressible work - arithmetic, strings, element access, compiled regex forms, pure sub calls? If yes, the JIT compiles it. If not, `PPERL_JIT_LOG` names the disqualifying operation - address that first.
2. Is there a string operation in the loop? If so, and the loop is otherwise a numeric reduction, split the string work out so the numeric loop can parallelize.
3. Is the accumulator declared outside the loop and only accumulated, never reset? That is what makes it a recognised reduction.
4. Is the body free of I/O, global writes, and impure calls? That is what lets the loop parallelize.
5. Measure default versus `--no-jit` versus `--no-parallel` (*Measuring*) to confirm which optimiser is engaging.

See also

- *JIT compilation* - the compilation pipeline, the variable-type model, the caching.
- *Parallel execution* - the reduction analysis, the side-effect gate, the thread controls.
- *Measuring* - how to confirm the JIT and parallelizer are actually engaging on your code.
- *Idioms* - which classic optimisations still matter and which the compiler already handles.
- *Sorting* - the one common hot path the JIT does not reach.

Fast start-up: the daemon

Every `pperl` invocation normally pays the full start-up bill: process creation, interpreter initialisation, and - the expensive part for real programs - compiling your script and every module it `use`s. The daemon lets repeated invocations of the same script skip that bill: a resident process keeps a fully compiled image of the script, and each new run is a fork of that image that goes straight to execution.

The feature is strictly **opt-in**, per invocation. Nothing changes for plain `pperl script.pl`, and whenever the daemon cannot serve a request exactly, pperl silently falls back to a normal cold run. Correctness never depends on the daemon; only speed does.

Quick start

```
pperl --daemon=start                    # start the daemon (background)

pperl --via-daemon report.pl            # first run: compiles, caches
pperl --via-daemon report.pl            # every further run: fork + run
```

(continues on next page)

(continued from previous page)

```
pperl --daemon=status                  # what is cached, what is live
pperl --daemon=stop                    # shut it down
```

`PPERL_DAEMON=2 pperl report.pl` is the same request without the flag, which is the more convenient form when the command line is not yours to change.

The first opted-in run of a script builds its *template*: the daemon compiles the script (including all `used` modules and `BEGIN` blocks) and keeps the result resident. Every later run of the same script forks the template and only executes.

When it pays

The saving is your script's compile time: parsing plus module loading plus `BEGIN` work. For a script that loads a handful of modules this is routinely the dominant part of start-up, and a template run can be several times faster end to end. For a trivial script that loads nothing, the saving roughly equals the daemon's own overhead and the result is a wash - never a meaningful loss.

As everywhere in this guide: measure your own script, on your own machine, before drawing conclusions - see *Measuring*.

A worked example

Here is a script of the kind that benefits: it does almost no work, but it loads seven modules before it can start.

```
#!/usr/bin/env pperl
use strict;
use warnings;
use Getopt::Long;
use List::Util qw(sum max);
use POSIX qw(strftime);
use File::Spec;
use Time::HiRes qw(time);
use Data::Dumper;

# The contract, honoured in one line: INIT runs per invocation, so a
# run from a template gets its own start time and its own random␣
 ↪seed.
INIT { $^T = time; srand() }

my $top = 3;
GetOptions('top=i' => \$top) or die "usage: report.pl [--top N]\n";

my %sales = (north => [10, 42, 7], south => [3, 19, 88], west => [5,␣
 ↪5, 5]);
my @ranked = sort { sum(@{$sales{$b}}) <=> sum(@{$sales{$a}}) } keys␣
 ↪%sales;

printf "%s  top %d of %d regions\n",
```

(continues on next page)

(continued from previous page)

```
    strftime('%F %T', localtime), $top, scalar @ranked;
printf "  %-6s total %3d  peak %3d\n",
    $_, sum(@{$sales{$_}}), max(@{$sales{$_}})
    for @ranked[0 .. $top - 1];
```

Start a daemon, then time the same twenty runs with and without it:

```
pperl --daemon=start

time (for i in $(seq 20); do pperl report.pl >/dev/null; done)

pperl --via-daemon report.pl >/dev/null        # builds the template
time (for i in $(seq 20); do pperl --via-daemon report.pl >/dev/
↪null; done)
```

On the machine this page was written on, that is 0.47 s against 0.08 s: about 24 ms per cold run, about 4 ms per run from the template. Nothing in the program changed; what disappeared is the compile of the script and of the seven modules, which the first run paid once.

Two things about those numbers are worth internalising. The saving is a fixed amount of *start-up* time, not a percentage of your program: a script that then works for a minute saves the same 20 ms as this one. And the saving scales with what you `use`, not with how long the script is - the modules are almost always the expensive part.

Checking whether a run was served

Because a decline is silent, “it got faster” is the wrong way to tell whether the daemon is doing anything. Ask the daemon:

```
pperl --daemon=status
```

```
build 0.6.8-10307.2a84282.6a670d70.10d8b48
uptime 42s
served 21
live 0
templates 1
template /home/you/report.pl pid 38109 served 21 age 39s ready
```

`served` counts requests the daemon handled; a run that fell back cold never appears there. `templates` lists one line per compiled script, with the process holding the image, how often it has been forked, and whether it is `ready` or still `parsing`.

For a single run, ask the program itself. A served run is a child of the daemon (or of its template), never of your shell:

```
pperl -e 'print getppid, "\n"'                 # your shell
pperl --via-daemon -e 'print getppid, "\n"'    # the daemon
```

If both print the same pid, the request was declined and pperl ran the program itself - which is always correct, just not faster.

What running from a template means

A template run executes your script's *run phase* only. Compilation already happened, once, when the template was built. pperl guarantees that the following are fresh and correct for every run:

- `@ARGV`, `%ENV`, the current directory, umask
- standard input, output and error (your redirections and pipes)
- the process id `$$`
- exit codes, and death by signal (a killed script kills the invoking `pperl` the same way)
- the `__DATA__` handle - each run reads it from the start, independently of other runs

What is **not** re-done is anything your code did at compile time. `BEGIN` blocks, and module code executed at `use` time, ran once when the template was built; their side effects are part of the image every run inherits. Consequences you own when you opt a script in:

- `$^T` (the script start time) and the random-seed state date from template creation. If your script cares, re-stamp them at the start of the run phase:

```
INIT { $^T = time; srand(); }
```

- Values read from `%ENV` *at compile time* (inside `BEGIN` or at `use` time) were the values of the run that built the template. Read configuration at run time, or in an `INIT` block, if it must track the invoking environment.
- File handles, sockets or database connections opened at compile time are shared with every run. Open connections at run time - the same rule long established for preloading application servers.

If a script cannot satisfy this contract, simply do not opt it in; `PPERL_DAEMON=2` is per invocation, not global.

Editing the script, and the cache

The daemon notices when the script file itself changes and rebuilds the template automatically on the next run - editing your script never serves you a stale image.

Edits to *modules* the script loads are not detected automatically. After changing a module, drop the affected template by hand:

```
pperl --daemon-reset=report.pl   # forget this script's template
pperl --daemon-reset             # forget all templates
```

Templates are a bounded cache: at most `--daemon-max-scripts` of them (default 100). When the limit is reached, the least recently used template is dropped to make room.

`--daemon-idle` (default 900 seconds) is a single value that two independent clocks use. A *template* exits when it has served nothing for that long, releasing its memory while the daemon keeps running and its other templates stay cached. The *daemon's* own clock only starts once nothing of its own is left alive - no request in flight, no template - so it never shuts down out from under a cached script. With the defaults, a script you stop

using loses its template about fifteen minutes later, and a daemon with nothing left to hold exits about fifteen minutes after that.

Neither exit costs you anything but time: with the template gone the next run compiles the script again, and with the daemon gone the run is an ordinary cold start. A daemon is never restarted for you, so if a machine runs opted-in scripts around the clock, either raise `--daemon-idle` or run the daemon as a service (below).

When pperl falls back to a cold run

The daemon refuses - and the client falls back to a normal run, silently and correctly - whenever it could not reproduce the invocation exactly:

- no daemon is running for this pperl binary
- environment variables that shape interpreter start-up differ from the daemon's (`PERL_HASH_SEED`, `PERL_PERTURB_KEYS`, `PERL_RAND_SEED`, `PERL_INTERNAL_RAND_SEED`, `PERL_UNICODE`, `LANG`, any `LC_*`, any `MALLOC_*`)
- the command-line flags differ from the ones the template was built with (a template stores one configuration)
- the template is still being built by a concurrent first run
- the invocation is a `-e` one-liner (templates cache script files)
- `--interactive` or `--stats` is in effect

Opting a script in without typing it every time

`PPERL_DAEMON=2` in front of every invocation is fine for a measurement and tedious for daily use. Two ways to make the opt-in part of the script or the command instead.

In the shebang, which travels with the script and makes the preload-safety promise visible to whoever reads it next:

```
#!/usr/bin/env -S pperl --via-daemon
```

Or as a shell function, when the script is not yours to edit:

```
report() { pperl --via-daemon "$HOME/bin/report.pl" "$@"; }
```

`--via-daemon` and `PPERL_DAEMON=2` are the same request and share one template, so a script with the shebang above still hits the cache when you run it once by hand with the environment variable, and the other way round.

The flag names an intent, not a boundary: given a script file it uses that script's template, and the script itself stays an ordinary argument. `pperl --via-daemon=report.pl` is not a thing - that value slot takes `trunk`, nothing else (see below), and pperl will tell you so.

Running the daemon as a service

`pperl --daemon` runs in the foreground and logs to standard error, which is what a service manager wants. A systemd user unit:

```
# ~/.config/systemd/user/pperl-daemon.service
[Unit]
Description=pperl start-up daemon

[Service]
Type=simple
ExecStart=/usr/local/bin/pperl --daemon --daemon-idle=86400
Restart=on-failure

[Install]
WantedBy=default.target
```

```
systemctl --user enable --now pperl-daemon.service
```

A long `--daemon-idle` is the point of running it this way: the unit is meant to be there when the next invocation arrives, whenever that is. The value must be a positive number of seconds; there is no "never".

One thing to remember after upgrading pperl: the daemon is keyed to the binary it was started from, so the new binary will not talk to the old daemon. Nothing breaks - runs are simply cold again until you `systemctl --user restart pperl-daemon.service`.

Security and identity

The daemon serves exactly one user: its socket lives in a directory with owner-only permissions and every connection's peer user id is verified. It also serves exactly one *build* of pperl: the socket is keyed to the binary's identity (`pperl --build-id` prints it), so after a pperl upgrade the old daemon is simply never contacted again and exits on its idle timeout.

Limitations

- Job control is approximate: the running script is a child of the daemon, not of your shell. Signals you send to `pperl` (including `Ctrl-C`) are forwarded to the script, but `Ctrl-Z` suspends only the foreground `pperl`, not the script process.
- One template per script path, holding one flag configuration. Invoking the same script with different flags falls back to a cold run rather than rebuilding the template.
- A one-liner cannot be cached. A template is keyed on a script file, which is also what tells pperl the cached image has gone stale; `-e` and `-E` have neither, so they only ever get the reduced mode below. This is not a `-E` limitation - `-E` is `-e` with features enabled and takes exactly the same path.
- `--via-daemon=trunk` (or `PPERL_DAEMON=1`) is that reduced mode: it forks a ready interpreter but still compiles the script on every run. It saves almost nothing, and

it exists so the daemon's own tests can exercise that boundary. There is no reason to ask for it in normal use.

- **Measuring** - the tools that actually work under pperl. The upstream XS profilers (`Devel::NYTProf`) do not load; the JIT log, the platform profiler, and the `bench/run-perlbench` runner do. How to read their output and how to A/B two versions of a routine.
- **Idioms** - a triage of the classic Perl performance idioms. Each is tagged: still valid, made moot by the JIT or parallelizer, or newly relevant because of how pperl compiles. The idioms that were folklore on upstream Perl are not all folklore here.
- **Sorting** - `sort` in depth: the Schwartzian transform, the Guttman-Rosler transform, `sort` subroutine cost, and when the comparator is the bottleneck.
- **Writing fast pperl** - the positive program: how to structure a hot loop so the JIT compiles it, how to shape a reduction so the parallelizer claims it, and the one mistake - mixing string work into a numeric loop - that silently disables both.
- **Fast start-up: the daemon** - opt-in start-up acceleration for repeatedly invoked scripts: a resident daemon keeps each script's compiled image and every run forks it, skipping parsing and module loading. What it guarantees per run, the preload-safety contract your script signs, and when pperl silently falls back to a normal cold start.

What this guide does not cover

The mechanisms live elsewhere and this guide does not repeat them:

- *JIT compilation* - what gets compiled, the caching model, the variable-type analysis.
- *Parallel execution* - the reduction analysis, the side-effect gate, the CLI thread controls.
- *Regex performance* - backtracking, catastrophic patterns, the engine's optimiser. Regex speed is its own discipline; nothing in this guide duplicates it.
- *Native modules* - why `List::Util` and friends run at built-in speed with no XS layer.
- *Auto-FFI* - calling C without XS, and the per-call marshalling cost that makes it the wrong tool for a hot inner loop.

A note on numbers

Concrete speedups in this guide are reproduced from the architecture and concurrency chapters, which carry the measured figures and the benchmark that produced them. Where this guide states a speedup without a number, the qualitative behaviour is established but the exact figure depends on your machine, your data, and your build - run `bench/run-perlbench` and trust your own measurement over any printed ratio.

4.2.5 Object-oriented programming

Perl has two object systems living side by side: the **modern `class`/`field`/`method`** syntax introduced in 5.38 (codename *Corinna*), and the **classical `bless`/`@ISA`** model that predates it by three decades. pperl supports both, and both appear in real code you will read and maintain.

The short version:

- For **new code**, use `class`. The syntax is first-class, the constructor is generated for you, fields are genuinely private, and single inheritance has a clean declarative form.
- For **legacy code**, understand `bless`, `@ISA`, and the `$self = shift` convention. You will meet them in every pre-5.38 codebase, every mainline CPAN distribution, and much of core Perl itself.
- **Moose, Moo, Class::Accessor, Object::Pad** are the CPAN ecosystems that papered over the classical model's rough edges while the core feature was still being designed. They are a historical chapter; new code reaches for core `class` instead.

This how-to tree covers both systems end-to-end. Read the chapters in order on a first pass, or jump to the one that matches the code in front of you.

Which chapter you want

- *Modern classes* - `class`, `field`, `method`, `ADJUST`, `:param`, `:reader`. The recommended way to write OO in pperl.
- *Classical OO* - `bless`, `@ISA`, `sub new`, accessors by hand, `SUPER::`. The model you will maintain in existing code.
- *Inheritance and method resolution* - how method lookup works, `isa`, `can`, `DOES`, MRO, and the pitfalls of multiple inheritance.
- *Roles and delegation* - composition patterns for behaviour shared across unrelated classes.
- *Migrating from classical to modern* - a mechanical recipe for converting a `bless`-based class to the `class` feature, and the places where the translation is not mechanical.

When to use OO at all

OO is a tool, not a style. Reach for it when:

- The problem has a small vocabulary of nouns (user, session, request, record) and a larger vocabulary of verbs acting on each.
- A piece of data carries invariants that should be enforced at one point in the code, not at every call site.
- You expect the implementation behind a set of operations to change while the operations themselves stay stable.
- You are modelling a category with variants and want polymorphism to pick the right behaviour.

Do not reach for OO when:

- The program is small, linear, and has no reusable nouns.
- A plain hash of hashes or a closure is enough.
- The only reason to write a class is "it felt more professional." A twelve-line script does not need a class.

A note on scope

This tree is a tutorial, not a reference. The exhaustive syntax tables live in the per-keyword reference pages:

- `class`
- `field`
- `method`
- `bless`
- `ref`
- `isa`

Link into them whenever you need the full signature surface; this tree gives you the working-programmer overview.

Modern classes

The `class` feature is the recommended way to write object-oriented code in pperl. It replaces the hand-rolled constructor, the manual accessor generation, the hash-slot bookkeeping, and the `use parent` dance with a single declarative syntax. You describe *what* the class is; the runtime arranges *how* it is implemented.

This chapter walks through the feature as a working programmer would meet it: first a minimum class, then fields and attributes, then methods, then construction-time hooks, then inheritance. Each section is short on purpose; when you want the full surface of a keyword, follow the cross-link into its reference page.

Enabling the feature

The feature is flagged experimental in Perl 5.44 and pperl mirrors that status. Every file that uses it opens with:

```
use v5.38;
use feature 'class';
no warnings 'experimental::class';
```

The `use v5.38` line already enables the `class` feature on its own, but being explicit makes the dependency visible to anyone reading the file.

A minimal class

```
use v5.38;
use feature 'class';
no warnings 'experimental::class';

class Point {
    field $x :param;
    field $y :param;

    method distance_from_origin {
        return sqrt($x ** 2 + $y ** 2);
    }
}

my $p = Point->new(x => 3, y => 4);
say $p->distance_from_origin;                # 5
```

Three things to notice:

- There is no `sub new`. The runtime generates the constructor from the `field` declarations. A user-written `sub new` inside a `class` body is a compile-time error on purpose - the generated one knows how to wire up `:param`, run initialisers, and trigger `ADJUST` blocks in the right order.
- Fields are referenced by their *declared name* inside methods, not through `$self->{x}`. The instance storage is genuinely private; no code outside the class body can poke at it.
- The `method` keyword takes care of `$self`. You never write `my $self = shift`.

Fields

A `field` is per-instance storage. Every object of the class gets its own slot; every method and every `ADJUST` block sees the field as if it were a lexical.

```
class Counter {
    field $value = 0;
    field $step  = 1;

    method tick  { $value += $step }
    method value { $value }
}
```

Fields accept scalars, arrays, and hashes:

```
class Registry {
    field $name;
    field @items;
    field %seen;
}
```

Attributes on fields

Field attributes take the declarative work off your hands.

- **`:param`** - bind this field to a constructor argument. If the caller omits it, the field is either left unset or takes the declared default.
- **`:param(alt)`** - bind to a different constructor name.
- **`:reader`** - generate a read-only accessor with the field's name minus the sigil.
- **`:reader(custom)`** - generate a reader under a different name.
- **`:writer`** - generate a setter (`set_FIELDNAME`).

```
class User {
    field $name  :param :reader;
    field $email :param :reader :writer;
    field $role  :param :reader = 'guest';
}

my $u = User->new(name => 'Ada', email => 'ada@example.org');
say $u->name;                                   # Ada
$u->set_email('ada@analytical.org');
say $u->role;                                   # guest
```

Default values

Defaults use one of three operators, each with a different rule:

- **`=`** - apply when the caller *omitted* the parameter entirely.
- **`//=`** - also apply when the caller passed `undef`.
- **`||=`** - also apply when the caller passed any false value.

```
class Window {
    field $title :param         = 'untitled';
    field $width :param         //= 640;
    field $shown :param         ||= 1;
}
```

Pick `=` unless you have a concrete reason to allow the stronger forms. Silent promotion of `undef` or `0` to a default is a bug magnet.

Fields are not lexicals in the usual sense

Fields look like lexicals inside a method body, but they have tighter rules:

- You cannot `our` a field, you cannot `local` one, and you cannot take a reference to one and export it.
- Fields are private to the class. Subclasses do **not** inherit a parent's fields; if a subclass needs a parent's state, it asks for it through a method.

- A field's initialiser runs before `$self` is bound. Inside `field $x = EXPR`, you can use earlier fields by name and you can use `__CLASS__`, but you cannot use `$self`. Anything that needs the fully constructed instance belongs in an `ADJUST` block.

Methods

A `method` is a subroutine that auto-binds `$self` and reads the enclosing class's fields by name.

```
class File {
    field $path :param :reader;

    method slurp {
        open my $fh, '<', $path or die "open $path: $!";
        local $/;
        return <$fh>;
    }

    method rename ($new) {
        rename $path, $new or die "rename: $!";
        $path = $new;
    }
}
```

Methods use signatures automatically - `use feature 'signatures'` is implied inside a class body. `$self` is the method's implicit first argument and does **not** appear in the signature.

Private methods

A method declared with `my` is lexical to the class body and is invoked through the `->&` operator:

```
class Safe {
    field $secret :param;

    my method check ($n) { $n == $secret }

    method unlock ($n) {
        return $self->&check($n) ? 'open' : 'denied';
    }
}
```

`my method` is the closest pperl has to a hard-private method; no external caller can reach it because the name does not live in the package at all.

Construction hooks: ADJUST

An `ADJUST` block runs after field initialisation, with `$self` already bound. It is the place for:

- Post-construction validation.

- Derived fields that depend on other fields.
- Opening a handle or connecting to a resource whose lifetime matches the object.

```
class TempFile {
    field $prefix :param = 'tmp';
    field $path;
    field $fh;

    ADJUST {
        $path = "/tmp/$prefix-$$-" . int(rand 1_000_000);
        open $fh, '>', $path or die "open $path: $!";
    }

    method write ($data) { print {$fh} $data }
    method path { $path }
}
```

Multiple `ADJUST` blocks in one class run in declaration order. In an inheritance chain, the parent's `ADJUST` blocks run before the child's.

Inheritance

A class inherits from at most one parent using the `:isa` attribute:

```
class Shape {
    field $colour :param :reader = 'black';
    method area { 0 }
}

class Circle :isa(Shape) {
    field $radius :param :reader;
    method area { 3.14159265 * $radius ** 2 }
}
```

Only **methods** are inherited. Fields are per-class storage. A subclass that needs the parent's state asks through a reader:

```
class Labelled :isa(Shape) {
    field $label :param :reader;
    method describe { "$label (" . $self->colour . ')' }
}
```

Multiple inheritance is not supported. `:isa(A, B)` is a syntax error, and a class cannot carry more than one `:isa` attribute. The *inheritance chapter* explains why, and covers the role-based alternative.

Calling up the chain

`SUPER::` still works, exactly as it does for `bless`-based classes:

```
class Dog :isa(Animal) {
    method introduce {
        $self->SUPER::introduce;
        say "and I can fetch";
    }
}
```

__CLASS__

Inside a field initialiser or a method body, `__CLASS__` is the class currently being constructed. In a subclass, `__CLASS__` refers to the subclass even inside an inherited method or an inherited field initialiser. Use it for factory-style hooks:

```
class Base {
    sub DEFAULT_X { 10 }
    field $x = __CLASS__->DEFAULT_X;
    method x { $x }
}

class Tuned :isa(Base) {
    sub DEFAULT_X { 99 }
}

say Tuned->new->x;                          # 99
```

Interoperation with classical OO

A `class`-declared class is a regular package. Code that uses `ref`, the `isa` operator, `UNIVERSAL::isa`, and `UNIVERSAL::can` keeps working:

```
my $c = Circle->new(radius => 2);
say ref $c;                                 # Circle
say $c isa Shape ? 'yes' : 'no';            # yes
say $c->can('area') ? 'yes' : 'no';         # yes
```

A classical class can subclass a modern class and vice versa, as long as the classical side goes through methods (never through `$self->{fieldname}`) - see the *migration chapter* for the full interop story.

Edge cases worth knowing

- **Do not write `sub new`.** The constructor is generated. A user-written `new` inside a `class` body is rejected at compile time.
- **Treat `@ISA` as read-only** for `class`-declared classes. Runtime manipulation is undefined behaviour.
- **`$self` is read-only** inside a method. Assigning to `$self` is a runtime error.
- **`package` cannot reopen a `class`.** Once a namespace has been declared with `class`, a later `package` statement for the same name is a compile-time error, and vice

versa.

- **Statement-form scope.** `class Foo;` consumes the rest of the enclosing block, typically the rest of the file. A subsequent `class` or `package` ends the body; there is no way to close a statement-form class early.

Where to go next

- *Inheritance and method resolution* - the full story of how method lookup works across `:isa` chains and across mixed classical/modern hierarchies.
- *Roles and delegation* - the recommended answer to "I want to share behaviour across unrelated classes."
- *Migrating from classical to modern* - if you are staring at a `bless`-based class and wondering what the equivalent `class` block looks like.

Classical OO

Before the `class` feature landed in 5.38, Perl's OO was built from three primitives: a package, a reference, and the `bless` function. Those three primitives are still present - the modern `class` keyword sits on top of the same machinery - and every pre-5.38 codebase, plus most of CPAN, still uses them directly.

This chapter is for reading and maintaining that code. If you are writing a new class, start with the *modern class chapter* instead.

The three primitives

1. **A package** defines a namespace. Any package can be a class; the difference is purely how it is used.
2. **A reference** holds instance data. The convention is a hash reference, but any reference type works.
3. **`bless`** tags the referent with a class name, so that method dispatch through the reference looks up subs in that package.

That is the whole model. Everything else - constructors, accessors, inheritance, roles - is a convention built on top.

A minimal class

```
package File;

sub new {
    my ($class, %args) = @_;
    my $self = {
        path    => $args{path},
        content => $args{content},
    };
    return bless $self, $class;
}
```

(continues on next page)

(continued from previous page)

```
sub path    { $_[0]{path} }
sub content { $_[0]{content} }

sub print_info {
    my $self = shift;
    print "This file is at $self->{path}\n";
}

1;
```

Key conventions, each load-bearing:

- **`sub new`** is the constructor. It is an ordinary method; the language has no constructor keyword. By convention it is named `new` and returns the blessed reference.
- **`my ($class, %args) = @_`** unpacks the invocant first, then the named arguments. A class-method call `File->new(path => ...)` passes the class name as the first argument.
- **`bless $self, $class`** attaches the class tag. Using `$class` (the actual invocant) rather than a hard-coded `'File'` is what makes the constructor *inheritance-safe*: a subclass inheriting `new` will bless into the subclass's name.
- **`return` is usually omitted** after `bless` because `bless` returns its first argument. `bless $self, $class` as the last expression in the sub is idiomatic. We write it explicitly here for readability.

Accessor methods by hand

The model gives you no accessors; you write them. The terse form exploits the fact that `$_[0]` is the invocant:

```
sub path    { $_[0]{path} }
sub content { $_[0]{content} }
```

The named-parameter form is clearer and what you should write when the accessor does anything non-trivial:

```
sub path {
    my $self = shift;
    return $self->{path};
}
```

A read/write accessor:

```
sub name {
    my $self = shift;
    $self->{name} = shift if @_;
    return $self->{name};
}
```

This is the boilerplate the *modern class feature* replaces with `:reader` and `:writer`.

Why hash references

Hash references dominate because they let you add and rename fields without changing the layout, and because named keys survive a `Data::Dumper` or `JSON::encode` round-trip. Array references are faster and more compact but fragile: renumbering a slot breaks every access site.

Hash-backed instances have one serious drawback: no encapsulation. Any caller can reach into the object's internals with `$obj->{path}`. The classical model has no tool to stop them - self-discipline and code review are the only defences. This is the single biggest reason to prefer the *modern class feature* for new code.

Inheritance with `@ISA`

Inheritance is a package-level list named `@ISA`. Method dispatch walks it when the current package does not define the method.

```
package File::MP3;
our @ISA = ('File');

sub print_info {
    my $self = shift;
    $self->SUPER::print_info;
    print "Title: $self->{title}\n";
}

1;
```

`SUPER::` means "look for this method starting one step up the ISA chain from the package the current method was *compiled in*" - not from the object's class. That subtlety matters in diamond inheritance; the *inheritance chapter* covers it.

Declaring the parent - four ways

There are four spellings of the same idea. Prefer the third for new code you are still maintaining in the classical style.

```
# 1. Raw @ISA assignment.
package Child;
our @ISA = ('Parent');

# 2. use base.
package Child;
use base 'Parent';          # deprecated in modern Perl

# 3. use parent.
package Child;
use parent 'Parent';        # current recommendation for classical

# 4. @ISA with a require.
package Child;
```

(continues on next page)

(continued from previous page)

```
require Parent;
our @ISA = ('Parent');
```

`use parent` handles the load of the parent module and the `@ISA` push in one line. `use base` is an older variant that also imports `Exporter`-style fields; avoid it.

The `$self` invocant

Every method is a subroutine whose first argument is either the class name (for class methods like `new`) or the object (for instance methods). The convention is to shift it into a variable named `$self`:

```
sub method {
    my $self = shift;
    ...
}
```

Or unpack it together with other arguments:

```
sub method {
    my ($self, $arg1, $arg2) = @_;
    ...
}
```

There is no runtime enforcement. If you forget to shift `$self`, your first user-supplied argument silently becomes `$self` and chaos ensues. Static analysers catch this; the language does not.

Destructors

A `DESTROY` method, if defined, runs when the object's refcount falls to zero. Use it to release external resources:

```
sub DESTROY {
    my $self = shift;
    close $self->{fh} if defined $self->{fh};
}
```

Pitfalls that bite every project at least once:

- `DESTROY` runs at unpredictable times. Do not rely on it for correctness-critical cleanup - explicit `$obj->close` beats implicit destruction.
- `DESTROY` can see a partially constructed object if `new` croaked mid-way. Defensive `defined` checks are cheap insurance.
- During global destruction (`END` time), dependencies may already be gone. A `DESTROY` that walks references can fail with "Attempt to free unreferenced scalar" or similar. Wrap risky cleanup in `eval { ... }`.

AUTOLOAD

If a method call finds no matching sub in the class or its ancestors, Perl looks for an AUTOLOAD sub. The called method's name appears in our $AUTOLOAD:

```
our $AUTOLOAD;
sub AUTOLOAD {
    my $self = shift;
    my $name = $AUTOLOAD;
    $name =~ s/.*::/;
    return if $name eq 'DESTROY';       # critical
    ...
}
```

The DESTROY guard is mandatory: without it, object destruction triggers AUTOLOAD with $name = 'DESTROY' and you either misbehave or crash.

AUTOLOAD is powerful and easy to abuse. The cases it earns its keep:

- Lazy accessor generation - synthesise an accessor on first call, then install it as a real sub via typeglob assignment so the next call is direct.
- Forwarding - delegate unknown methods to a contained object (but see *Roles and delegation* for a cleaner approach).

Cases where it hurts more than it helps: pretending to be a full-blown MOP, catching typos as silent no-ops, implementing "classes" whose entire surface is AUTOLOAD.

A worked example

A classical Counter class with read/write accessor and inheritance:

```
package Counter;

sub new {
    my ($class, %args) = @_;
    my $self = {
        value => $args{value} // 0,
        step  => $args{step}  // 1,
    };
    return bless $self, $class;
}

sub value {
    my $self = shift;
    $self->{value} = shift if @_;
    return $self->{value};
}

sub step { $_[0]{step} }

sub tick {
    my $self = shift;
```

(continues on next page)

(continued from previous page)

```
    $self->{value} += $self->{step};
}

1;
```

A subclass that doubles the step:

```
package Counter::Double;
use parent 'Counter';

sub new {
    my ($class, %args) = @_;
    $args{step} //= 2;
    return $class->SUPER::new(%args);
}

1;
```

Compared with the *modern class* version, the classical form is about three times as many lines and every one of them is a place to get it wrong.

When classical OO is the right call

Despite everything, there are legitimate reasons to stick with `bless`-based classes:

- **You are maintaining existing code.** Rewriting a working classical class into `class` form is a migration project, not a free lunch - see the *migration chapter*.
- **You need to support older Perl.** If your code must run on Perl before 5.38, the `class` feature is unavailable.
- **You need array-backed or code-backed objects.** `class` only creates hash-backed instances. If you are building a performance-critical structure with array-reference instances, classical `bless` is still the answer.
- **You are doing something the `class` feature deliberately forbids** - reopening the package from outside, multiple inheritance, or deep metaprogramming via `@ISA` manipulation.

A historical note on CPAN OO systems

Because the classical model leaves so much to convention, CPAN grew an ecosystem of modules that layer declarative syntax on top:

- **Moose** - feature-complete OO system with a type miniature, roles, method modifiers, a full introspection API, and a large extension ecosystem. Heavy at load time; the dominant choice for large applications built between 2007 and roughly 2020.
- **Moo** - a subset of Moose's API without the introspection layer. Lighter, pure-Perl, API-compatible enough to let a Moo class and a Moose class share an inheritance tree.

- **Class::Accessor** - generates simple accessors and `new`. Minimal, pure-Perl, no roles.
- **Class::Tiny** - the smallest of the accessor-generator modules. All accessors are read/write, no type checks.
- **Object::Pad** - the experimental prototype that became the core `class` feature. Still installable on CPAN for code that needs the feature on older Perls.
- **Role::Tiny** - role composition for projects that do not use Moose but still want `does`-style sharing.

These systems are **historical context** in pperl. New code in pperl should use the core `class` feature, which covers the ground that Moose/Moo pioneered without the load-time and dependency cost. When you inherit a codebase that uses one of them, learn just enough of its surface to stay productive and plan the migration with the *migration chapter*.

Inheritance and method resolution

Method lookup in Perl is simple in the single-inheritance case and genuinely subtle once diamonds or mixed class systems enter the picture. This chapter covers the full story: the MRO algorithms, `SUPER::`, the `isa` operator, `can`, `DOES`, and the interaction between classical and modern classes in one hierarchy.

The lookup algorithm

When you write `$obj->foo(@args)`, Perl does the following:

1. Determine the class. For an object, it is the class the referent was blessed into; for a class-method call, it is the invocant itself.
2. Walk the class's **method resolution order** (MRO) in sequence. For each package in the order, look for a sub named `foo`.
3. The first match wins. If no match is found in any class on the MRO, walk again looking for `AUTOLOAD`.
4. If no `AUTOLOAD` is found either, throw `Can't locate object method "foo" via package "Class"`.

The MRO is what the class's `@ISA` chain linearises to. The linearisation rule depends on which MRO is in effect.

The two MROs: `dfs` and `c3`

Perl ships two MRO algorithms:

- **`dfs`** - depth-first search of `@ISA`. The default. Walk each parent, and each parent's parents, left-to-right, diving fully into each subtree before moving on.
- **`c3`** - the C3 linearisation used by Python, Raku, and many others. Guarantees monotonicity: a subclass never appears later than one of its ancestors in the lookup order.

For single inheritance the two are identical: the list walks straight up the chain.

For multiple inheritance they diverge. Consider the diamond:

```
    A
   / \
  B   C
   \ /
    D
```

- **`dfs`**: `D, B, A, C, A` (collapsed to `D, B, A, C`). A method on `C` that overrides `A` is **shadowed** by `A` because `A` is reached through `B` first.
- **`c3`**: `D, B, C, A`. C's override is visible.

C3 is almost always what you want under multiple inheritance. Opt in per-class:

```
use mro 'c3';
package Diamond::D;
our @ISA = ('Diamond::B', 'Diamond::C');
```

For classes declared with `class`, multiple inheritance is not supported at all, so the MRO choice is purely a linear walk and the default `dfs` is fine.

SUPER::

`$self->SUPER::foo(@args)` calls `foo` in one of the parent classes of *the package the current method was compiled in*. That last phrase is the one everyone gets wrong.

```
package Animal;
sub speak { 'some sound' }

package Dog;
our @ISA = ('Animal');
sub speak { 'woof' }

package Puppy;
our @ISA = ('Dog');
sub speak {
    my $self = shift;
    return $self->SUPER::speak;
}
```

`Puppy::speak` calls `SUPER::speak`. Perl looks up from **Puppy** (the compilation package), not from `ref($self)`. The search starts in Puppy's `@ISA`, so it finds `Dog::speak` and returns `'woof'`.

This matters when methods are inherited and the child calls `SUPER::` from within. Because `SUPER::` is anchored to the compile-time package, a method inherited from `Dog` and invoked on a `Puppy` object will still call `Animal::` via its `SUPER::`, not `Dog`. Usually that is the right behaviour; when it is not, reach for explicit chain navigation with `next::method`, covered below.

`next::method` - the MRO-aware SUPER

`mro::next::method` (exposed as `$self->next::method(@args)`) walks the live MRO from the current method's *actual* call position, not from the compile-time package:

```
use mro 'c3';

sub speak {
    my $self = shift;
    return $self->next::method;
}
```

Under C3 with a diamond, `next::method` is what you want; `SUPER::` can skip a sibling branch because it only consults the left-most ancestor.

The `isa` operator

The `isa` operator tests class membership without the footgun of `UNIVERSAL::isa` called as a function:

```
if ($obj isa 'File::MP3') { ... }
if ($obj isa File::MP3)   { ... }     # bareword form
```

Both `bless`-based and `class`-declared classes answer `isa` correctly, and subclass relationships flow through `:isa` and `@ISA` identically.

Three rules worth remembering:

- `isa` returns false (not an error) when given a non-object. `42 isa Foo` is false. `undef isa Foo` is false.
- `isa` respects the live MRO at the moment of the call. If code pushes onto `@ISA` at runtime, `isa` sees it. (This is another reason pperl rejects `@ISA` manipulation for `class`-declared classes - the guarantees would become muddy.)
- Prefer the `isa` operator over `UNIVERSAL::isa($obj, 'Class')`. The operator short-circuits on non-objects instead of croaking; the function-call form is also defeated by overloaded objects.

`can`

`$obj->can('method')` returns a code reference if the object has a method by that name, `undef` otherwise:

```
if (my $code = $obj->can('save')) {
    $code->($obj, @args);
}
```

Uses:

- Feature-detection before calling: avoids the `Can't locate object method` error.
- Dispatching to a method by name from a string held in a variable - more readable than the `$obj->$name` form when the method might not exist.

Non-uses:

- Do **not** use `can` to decide whether an object implements a conceptual role. A class can define a `save` method that does something completely unrelated to persistence. For role membership, see *roles and delegation*.

DOES

`$obj->DOES('RoleOrClass')` is the role-aware counterpart to `isa`. By default `DOES` delegates to `isa`, so for classes without explicit role machinery the two behave the same. Role systems (classical Moose, Moo, and the future core role feature) override `DOES` to also return true for roles the class consumes.

```
if ($obj->DOES('Printable')) { ... }
```

In pperl, `DOES` is the right method to check when you want to ask "does this object implement the contract?" rather than "does this object inherit from this class?"

Mixed classical / modern hierarchies

`class`-declared and `bless`-based classes interoperate in both directions, subject to a handful of rules.

A modern class inheriting from a classical class

```
package Shape;                              # classical
sub new {
    my ($class, %args) = @_;
    return bless { colour => $args{colour} // 'black' }, $class;
}
sub colour { $_[0]{colour} }

use feature 'class';
class Circle :isa(Shape) {                  # modern inherits classical
    field $radius :param :reader;
    method area { 3.14159265 * $radius ** 2 }
}
```

This works. The modern class's generated constructor delegates to the parent's new, and methods inherited from the classical parent appear on the modern child. The parent's hash-slot state lives *alongside* the child's field storage.

Caveat: the modern child cannot reach into the parent's hash. `$self->{colour}` is not legal inside a method of `Circle`; you must call `$self->colour`.

A classical class inheriting from a modern class

```
use feature 'class';
class Animal {
    field $name :param :reader;
```

(continues on next page)

(continued from previous page)

```
    method speak { 'some sound' }
}

package Dog;                                # classical inherits modern
our @ISA = ('Animal');
sub new {
    my ($class, %args) = @_;
    return $class->SUPER::new(%args);
}
sub speak { 'woof' }
```

This also works - the modern parent's generated constructor is inherited, and `SUPER::new` resolves to it. The classical child must not try to mutate the parent's fields directly; fields are private to the class that declared them.

What fails in mixed hierarchies

- **Reaching `$self->{fieldname}` across the boundary.** Modern fields are not hash slots, and classical hash slots are not visible as fields. Always cross the boundary through a method.
- **Multiple inheritance into a modern class.** The `:isa` attribute only takes one target. If you need to mix behaviour from multiple places, see *roles and delegation*.
- **Runtime `@ISA` manipulation on a modern class.** Undefined behaviour. Treat `@ISA` as read-only for any class declared with `class`.

Why multiple inheritance usually hurts

Every language with multiple inheritance eventually discovers the same pathologies. Method lookup becomes order-dependent; diamond problems force you to think about linearisation; the "I inherited twice from the same root" case tangles initialisation. Roles - composition of behaviour without an inheritance relationship - solve the underlying sharing problem without the pathologies.

In pperl the `class` feature forbids multiple inheritance deliberately. Classical code that uses it should be migrating towards a role model; the *roles chapter* is the next stop.

Further reading

- `class` - full syntax of modern class declarations including `:isa`.
- `bless` - the primitive under classical inheritance.
- `ref` - reports a blessed reference's class name.
- `isa` - the preferred class-membership operator.
- *Roles and delegation* - the composition-based alternative to multiple inheritance.
- *Migration* - walking an existing hierarchy from classical to modern.

Roles and delegation

Inheritance answers "what is this object?" Roles and delegation answer a different question: "what can this object do?" When two unrelated classes need to share behaviour, forcing them under a common parent is the wrong tool. Composition - giving each class the behaviour it needs without pretending they are related - is the right one.

This chapter covers the three composition patterns that pperl code uses in practice:

1. **Mixin-style sharing** via a tiny helper package.
2. **Role composition** via `Role::Tiny` or a Moose-family system in legacy code.
3. **Delegation** - forwarding methods to a contained object.

The core `class` feature does not yet ship a role primitive. That is a deliberate staging decision upstream: the `class` feature landed first, roles are a separate proposal. Until the core role feature ships, the patterns in this chapter are the working answers.

The problem

Imagine a `Radio` class and a `Computer` class. Both have an on/off switch. They are not related - a radio is not a computer, and neither specialises a hypothetical `Machine`. A `HasOnOffSwitch` parent class would be an artificial grouping whose only purpose is reuse of two methods.

Three solutions, in increasing order of ambition:

- Copy the `turn_on` / `turn_off` methods into each class. Tolerable for two classes, painful for ten.
- Extract them into a helper package, use one of the sharing mechanisms below.
- Use a role system with formal composition semantics.

Pattern 1: manual mixin

The smallest possible "share these methods" mechanism is a plain package that installs subroutines into the consuming class at `use` time:

```
package HasOnOffSwitch;

sub import {
    my $target = caller;
    no strict 'refs';
    *{"${target}::turn_on"}  = sub { $_[0]->{is_on} = 1 };
    *{"${target}::turn_off"} = sub { $_[0]->{is_on} = 0 };
    *{"${target}::is_on"}    = sub { $_[0]->{is_on} };
}

1;
```

Used:

```
package Radio;
use HasOnOffSwitch;

sub new { bless { is_on => 0 }, shift }
```

This works. It is also brittle: it reaches into the consuming class's hash directly (`$_[0]->{is_on}`), which is exactly the encapsulation violation that classical hash-backed OO is famous for. It breaks if the consumer is a `class`-declared class, because fields are not hash slots.

Use this pattern only for very small shared behaviour in a classical codebase. For anything larger, use a real role system.

Pattern 2: `Role::Tiny`

`Role::Tiny` is the lightweight role system that works with plain classical classes, with `Moo`, and with `Moose` (on the consuming side). It is the closest thing to a portable role primitive in the CPAN ecosystem.

```
package HasOnOffSwitch;
use Role::Tiny;

sub turn_on  { $_[0]->is_on(1) }
sub turn_off { $_[0]->is_on(0) }

requires 'is_on';       # the consumer must provide this accessor

1;
```

A consumer picks the role up with `with`:

```
package Radio;
use Role::Tiny::With;
with 'HasOnOffSwitch';

sub new    { bless { is_on => 0 }, shift }
sub is_on  { @_ > 1 ? $_[0]{is_on} = $_[1] : $_[0]{is_on} }
```

What `Role::Tiny` gives you over the manual mixin:

- **Requirements.** `requires 'is_on'` fails at composition time if the consumer does not supply `is_on`. The manual mixin fails silently at the first method call.
- **Conflict detection.** Consuming two roles that both define the same method is a composition-time error, not silent last-definition-wins.
- **Method modifiers** - `before`, `after`, `around` - if you want them.

`Role::Tiny` is **not** the same thing as Moose's role system. Its reach is intentionally narrow. If the codebase already uses Moose, use `Moose::Role`; if it uses Moo, use `Moo::Role`.

Pattern 3: Moose-family roles

Legacy codebases built on Moose or Moo use `Moose::Role` and `Moo::Role` respectively. The surface looks like the role system the core `class` feature will eventually gain:

```
package HasOnOffSwitch;
use Moose::Role;

has is_on => (is => 'rw', isa => 'Bool', default => 0);

sub turn_on  { $_[0]->is_on(1) }
sub turn_off { $_[0]->is_on(0) }

1;

package Radio;
use Moose;
with 'HasOnOffSwitch';

1;
```

In pperl, Moose still works. New code should not reach for it - the runtime cost and dependency weight are both large, and the core `class` feature now covers the declarative side of what Moose was built for. When the core role feature lands, migrating from `Moose::Role` to the core role will be the natural next step.

Pattern 4: delegation

Delegation says "I don't do this myself, but I hold something that does." Instead of inheriting behaviour, contain an object that provides it, and forward methods to it.

The manual form:

```
use feature 'class';

class Player {
    field $radio :param;

    method turn_on  { $radio->turn_on }
    method turn_off { $radio->turn_off }
    method is_on    { $radio->is_on }
}
```

Each forwarded method is three lines of mechanical plumbing. For two or three methods that is fine; for a wide surface, reach for `Class::Method::Delegate` or a Moose `handles` accessor:

```
package Player;
use Moose;

has radio => (
    is      => 'ro',
```

(continues on next page)

(continued from previous page)

```
    handles => ['turn_on', 'turn_off', 'is_on'],
);
```

`handles` generates the forwarders at class-build time.

When to delegate vs inherit

- **Delegate** when the contained object is a *collaborator* - something the owner *has* rather than *is*. A `Player` has a `Radio`; a `Player` is not a kind of `Radio`.
- **Inherit** when the child genuinely *is* a specialised parent. A `Circle` is a kind of `Shape`; a `File::MP3` is a kind of `File`.

If the answer to "is-a or has-a?" is not immediate, the relation is probably has-a. Defaulting to composition is almost always the less wrong call - you can refactor composition into inheritance with less pain than the reverse.

Delegation with lazy construction

A common pattern: the collaborator is expensive to build, so build it on first access. With the modern `class` feature:

```
use feature 'class';

class Logger {
    field $path :param;
    field $fh;

    method fh {
        unless ($fh) {
            open $fh, '>>', $path or die "open $path: $!";
        }
        return $fh;
    }

    method log ($msg) {
        print {$self->fh} $msg, "\n";
    }
}
```

`fh` is a reader that also handles the one-shot construction. Callers see a plain method; the lazy cache is private.

What not to do

- **Multiple inheritance to share two methods.** You pay the full MRO cost for a tiny win, and the next person to inherit from your class inherits the MRO pain too. Use a role or a mixin.
- **`AUTOLOAD` as a delegator.** Convenient in the five-minute demo, impossible to introspect, and it eats typos. Write explicit forwarders or use `handles`.

- **Roles that carry state.** A role that defines both methods and attributes is really a mini-class. The cleanest role defines behaviour that the consuming class's state supports; the role requires the accessors it needs and trusts the consumer to provide them. The manual-mixin example in pattern 1 got this wrong by reaching into `$_[0]->{is_on}` directly.
- **Deep delegation chains.** If `$player->radio->volume->level` appears repeatedly, add a `volume_level` accessor on `Player`. Train-wreck method chains are a composition smell.

Further reading

- *Modern classes* - the base layer that composition sits on top of.
- *Inheritance and method resolution* - when the answer really *is* inheritance.
- *Migration* - how role-based legacy code translates into the core `class` model (spoiler: mostly unchanged; roles migrate last).
- `class` - reference for the modern class syntax.
- `isa` - class-membership test; pair with `DOES` for role membership.

Migrating from classical to modern

If you are maintaining a `bless`-based codebase and wondering what the equivalent `class` block looks like, this chapter is the recipe. The translation is mostly mechanical, but the places where it is not mechanical are where real design decisions surface - and where the migration pays for itself.

Before starting, read the *modern class chapter* for the syntax you are translating *into*, and skim the *classical OO chapter* for any constructs the existing code uses that you may not remember in detail.

When to migrate

Migration is not free. Do it when you get real value:

- The class is under **active development**. You are about to add fields, add methods, or change the constructor contract. Doing that work against the `class` feature is less error-prone than doing it against hand-rolled hash slots.
- The class has **unclear encapsulation**. Callers reach into `$obj->{fieldname}` directly, and you want to stop that without rewriting every call site at once. The `class` feature makes the violation a compile-time error the moment you switch.
- The class sits on top of **Moose or Moo**, and you are paying the load-time cost without using the dependency-heavy features. The modern `class` is a straightforward replacement for most Moose/Moo classes that only use `has` and `with`.

Do **not** migrate a class that is working and frozen. A stable classical class that will not see new code for another five years is exactly as good as its modern equivalent, and the migration itself will introduce bugs that the stable class does not have.

The mechanical translation

Take this classical class:

```
package Counter;

sub new {
    my ($class, %args) = @_;
    my $self = {
        value => $args{value} // 0,
        step  => $args{step}  // 1,
    };
    die "step must be positive" if $self->{step} <= 0;
    return bless $self, $class;
}

sub value {
    my $self = shift;
    $self->{value} = shift if @_;
    return $self->{value};
}

sub step { $_[0]{step} }

sub tick {
    my $self = shift;
    $self->{value} += $self->{step};
}

1;
```

The modern form:

```
use v5.38;
use feature 'class';
no warnings 'experimental::class';

class Counter {
    field $value :param :reader :writer = 0;
    field $step  :param :reader         = 1;

    ADJUST {
        die "step must be positive" if $step <= 0;
    }

    method tick { $value += $step }
}

1;
```

The mapping, piece by piece:

Classical construct	Modern construct
`package Counter;`	`class Counter { ... }`
`sub new { ... bless ... }`	*(deleted - generated)*
Constructor argument defaulting with `//`	`= DEFAULT` or `:param ... = DEFAULT`
`$args{foo}` stored in `$self->{foo}`	`field $foo :param`
Read-only accessor	`:reader`
Read/write accessor	`:reader :writer`
Validation at the end of new	ADJUST block
`$self->{foo}` inside a method	`$foo` (the field name)
`my $self = shift`	*(deleted - implicit)*
`use parent 'Base';` / `our @ISA = ('Base')`	`class Sub :isa(Base) { ... }`
`$self->SUPER::foo(@args)`	unchanged - still works

The non-mechanical parts

Everything above is rote translation. The places where you need to think are all variants of "the classical code was doing something the `class` feature does not do the same way."

Multiple inheritance

`class` does not support `:isa(A, B)`. If the classical class has `@ISA = ('Parent1', 'Parent2')`, you have three options:

1. **Collapse to single inheritance.** Usually the second parent is a mixin or role in disguise. Migrate it into a *role* and consume it with `with`.
2. **Collapse to composition.** If the second parent is a collaborator rather than an ancestor, migrate to delegation instead of inheritance.
3. **Do not migrate the class.** If the multiple inheritance is genuinely load-bearing and neither of the above fits, the class is a poor fit for the modern feature. Leave it classical until the core role feature lands.

In practice option 1 or 2 applies to about 95% of the multiple-inheritance cases in real code.

Direct hash access from callers

If external callers do `$counter->{value}` rather than `$counter->value`, migrating the class will break them. The compiler error is clean and local - the error message names the site - but the migration is no longer a drop-in.

Mitigation strategy:

1. **Audit first.** Grep for `$obj_name->{` patterns across the codebase. The count is usually small.
2. **Add an accessor** to the classical class that covers every accessed field.
3. **Replace** `$obj->{fieldname}` call sites with `$obj->fieldname()` in a dedicated commit.

4. **Then migrate** the class to modern form.

Splitting the refactor into two phases keeps each commit small and bisectable.

AUTOLOAD

`AUTOLOAD` in a classical class has no modern equivalent. The `class` feature does not let you catch missing method calls.

If the `AUTOLOAD` is doing one of the common patterns, translate it:

- **Lazy accessor synthesis** - declare the accessors explicitly as fields with `:reader` or `:writer`. The `class` feature's code generation replaces what `AUTOLOAD` was doing.
- **Delegation to a contained object** - replace with explicit forwarder methods (see *roles and delegation*) or with a Moose-style `handles` accessor if the codebase can depend on Moose.
- **A generic proxy** - keep the class classical. The `class` feature is not the right tool.

DESTROY

`class` does not introduce a first-class destructor form. If the classical class has a `DESTROY`, write it as a plain method inside the `class` block - it is invoked through the same refcount hook:

```
class Logger {
    field $fh;

    ADJUST { ... }

    sub DESTROY {
        my $self = shift;
        close $self->{fh} if $self->{fh};
    }
}
```

Note the wart: inside `DESTROY` you cannot name the field by its declared name, because `sub` bodies do not see field bindings. Either use a regular method that `DESTROY` delegates to, or accept the slight asymmetry.

A cleaner pattern when the lifetime is under your control is to add an explicit `close` method and call it deterministically, leaving `DESTROY` as a belt-and-braces backup.

Non-hash instances

If the classical class does `bless \@self, $class` or `bless \$self, $class` - an array-backed or scalar-backed instance - the modern feature has no equivalent. Hash-like instance layout is baked in. Keep these classes classical.

Tied hashes / overloading

`use overload` and `tie` integrations work unchanged in a `class` block. Nothing special to do.

Migrating Moose / Moo classes

The translation from Moose to `class` is cleaner than from classical because Moose is already declarative. The rough map:

Moose / Moo	Modern `class`
`use Moose;` / `use Moo;`	`use feature 'class';` plus block
`has foo => (is => 'ro', ...)`	`field $foo :param :reader`
`has foo => (is => 'rw', ...)`	`field $foo :param :reader :writer`
`has foo => (default => sub { ... })`	`field $foo = EXPR` (or `ADJUST`)
`extends 'Parent'`	`class Sub :isa(Parent) { ... }`
`BUILD { ... }`	`ADJUST { ... }`
`with 'Role::Name'`	*(wait for core roles)*
`has foo => (isa => 'Int', ...)`	*(no direct equivalent)*
`method_modifiers` (before/after/around)	*(no direct equivalent)*

The gap items - type constraints, method modifiers, roles - are what keeps some Moose codebases on Moose for now. If your Moose use is limited to `has`, `extends`, and `BUILD`, the migration is straightforward. If it depends heavily on Moose's type system or method modifiers, the migration is a project, not a commit.

Testing the migration

A class migration that changes no external behaviour should pass the existing test suite unchanged. Steps:

1. Ensure the class has tests covering its public API. If it does not, write them **before** migrating - the tests are your migration safety net.
2. Migrate the class.
3. Run the tests. A clean pass is the completion signal.
4. Grep the codebase for `$obj->{...}` reaches into the migrated class's instances. Fix any hits.
5. Remove any now-dead `use parent`, `use base`, explicit `sub new`, or accessor boilerplate the old form needed.

If step 3 fails, the failures usually fall into a short list:

- **Direct hash access** in a method that the old form tolerated and the new form rejects. Replace with field references.
- **A constructor argument that was silently accepted.** The generated constructor is strict about `:param` names. Either declare the parameter or change the call site.

- **A field initialiser that referenced `$self`.** Field initialisers run before `$self` is bound. Move the dependent logic into an `ADJUST` block.

What you get out of it

The classical-to-modern migration is unusual in that the result is both shorter and more correct:

- **Shorter.** The `Counter` example above drops from ~20 lines to ~10, and the removed lines are all boilerplate.
- **Private fields.** External code can no longer reach in. The encapsulation you used to defend in review is now defended by the compiler.
- **No forgotten `bless`.** The generated constructor is always inheritance-safe; you cannot ship a constructor that accidentally blesses into the wrong class.
- **No forgotten `$self = shift`.** The implicit bind cannot go missing.
- **A named feature flag.** `no warnings 'experimental::class'` documents at the top of the file what the class depends on.

Further reading

- *Modern classes* - reference for the syntax you are migrating into.
- *Classical OO* - reference for the syntax you are migrating out of.
- *Inheritance and method resolution* - the mixed-hierarchy chapter, useful while the codebase is half-migrated.
- *Roles and delegation* - where multiple inheritance ends up in the migrated code.
- `class`, `field`, `method` - the reference pages for the three keywords at the heart of the migration.
- `bless`, `ref`, `isa` - the primitives that underpin both old and new forms and that keep working across mixed hierarchies.

4.2.6 Debugging Perl programs

Perl ships with an unusually complete debugging surface: a built-in statement-by-statement debugger, a regex-engine tracer, a documented API for writing alternative debuggers, and a CPAN ecosystem of profilers, leak finders, post-mortem tools, and IDE integrations. pperl's reimplementation preserves the interpreter's hook points - `$^P`, `DB::DB` and `DB::sub` fire as they do in perl5 - so perl5's own debugger, `perl5db.pl`, runs unchanged where a system perl5 provides it on `@INC`. `Devel::Cover` measures coverage, its compiled part built in (see *coverage*). Other tools that are XS extensions, such as `Devel::NYTProf`, do not load.

`pperl -d script.pl` starts the debugger, and so do `-de0`, `-d:Module` and a `#!/usr/bin/perl -d` first line; a session is the same as perl5's. The one exception is `-dt`: it asks `perl5db.pl` for thread support, and on pperl's threadless build that dies with `This Perl not built to support threads`.

Which chapter do I want?

Symptom	Start here
Program misbehaves and I want to inspect it live	*interactive-debugger*
Program crashes with a cryptic message	*exceptions*
I want to sprinkle prints and dumps	*print-and-die*
I want to turn more mistakes into compile errors	*preflight*
I need to stop on a specific line or condition	*breakpoints*
I need to see what a data structure actually holds	*inspecting-state*
I want to see every line / sub as it executes	*tracing*
I want breakpoints in VS Code / Neovim	*ide-dap*
I want to know which code my tests never run	*coverage*

The three layers of this guide

- **Orientation** - this page.
- **Working programmer** - the chapters listed above. Each is self-contained; reading a single one is enough to solve the problem it names.

Known gaps

Debugging surfaces where the ecosystem itself is thin, or where pperl's implementation has not reached parity with upstream. Named here so readers do not waste time on them.

- **Regex tracer output format.** pperl's regex engine is a native reimplementation; `use re 'debug'` produces a trace but node names and field formats are not byte-identical to perl5's. Consumers that parse the trace (few exist) will need adjustment.
- **`B::*` introspection modules.** pperl provides `B` and `O` natively. `B::Deparse` and `B::Concise` are pure Perl on top of them: where a system perl5 provides them on `@INC` they run under pperl and agree with perl5 on ordinary code. Less-used members of the family reach further into the op graph, where pperl's representation differs, and may still need adjustment.
- **XS-dependent introspection modules.** `Devel::Peek`, `Devel::FindRef`, `Devel::Gladiator`, `Devel::Refcount`, `Devel::Size`, `Devel::MAT::Dumper`, `Devel::Leak` family - all reach the SV through XS (C-level macros against the SV pointer). pperl mirrors perl5's SV layout faithfully, but XS loading is not supported; these modules therefore need native (Rust) reimplementations that expose the same Perl-level API. Pure-Perl `Devel::Peek`-workalike functionality is already provided natively by the built-in `Devel::Peek`; the others are in progress.
- **Enbugger on Perl $\geqslant$ 5.38.** Last release 2012; works anecdotally on modern perls but has no active maintainer. The `gdb-inject-perl` path is more reliable.
- **Async provenance.** No Perl equivalent to Node's `AsyncLocalStorage`: a `Future` that fails in a distant callback carries no automatic record of who scheduled it.
- **Sentry breadcrumbs.** `Sentry::Raven` is functional but maintenance-only; no first-class breadcrumb support. The OpenTelemetry Perl SDK is the forward path

for new installations.

- **`Perl::LanguageServer` attach-to-running-process.** Launch is supported; post-hoc attach is not.

See also

- `perlfunc/die`, `perlfunc/warn`, `perlfunc/eval`, `perlfunc/caller` - the language primitives the debugger and the stack-trace tools are built on.
- *PPerl Architecture guide* - how pperl is built and why the debugger surface works as it does.

Preflight: catching bugs before runtime

This chapter covers the pragmas, compile-time checks, and runtime hygiene settings that prevent most bugs from reaching the debugger at all. A clean preflight is the shortest path to a short debugging session.

Readers who already `use v5.36` or later run under strict and warnings by default and can skip straight to the diagnostics and autoflush sections. The rest of this chapter assumes you might be maintaining older code where neither is implicit.

Strict and warnings

Since Perl 5.36, `use v5.36;` (and any declaration of a higher version) implicitly enables `use strict;` and `use warnings;`. Modern scripts do not write those lines - the version declaration covers both:

```
use v5.40;
```

`use strict` refuses three classes of sloppy code at compile time: undeclared variables (`strict vars`), symbolic references (`strict refs`), and barewords used as strings (`strict subs`). `use warnings` turns on runtime diagnostics for uninitialised reads, lossy numeric conversions, filehandles used after close, and several dozen other sharp edges.

Legacy code that uses neither still benefits from both: add them, fix the symbols they complain about, commit.

Narrow opt-outs remain useful:

```
no strict 'refs';                  # genuine symbolic deref
no warnings 'uninitialized';       # known-safe defaulting
no warnings 'experimental::try';   # on 5.36, 5.38 - see below
```

Never blanket-disable: `no warnings;` with no category list is almost always wrong.

Promote warnings to fatal for a region:

```
local $SIG{__WARN__} = sub { die $_[0] };
```

Useful at program top during development; the first warning becomes a stack-trace-producing crash.

use diagnostics

Expands each warning or fatal message into the full paragraph from `perldiag`:

```
use diagnostics;
```

The output is verbose - too verbose for production - but during debugging it explains what a terse message like "Odd number of elements in hash assignment" actually means and how to fix it.

Post-hoc version: pipe a program's stderr through the `splain` command to annotate warnings after the fact.

Compile-time checks

`perl -c script.pl` parses the script and runs every `BEGIN` / `use`, then stops before the first runtime statement. A clean `-c` means the file and its dependencies compile; it does not mean the program runs correctly.

```
perl -c script.pl
script.pl syntax OK
```

Combine with `-w` to enable warnings globally for the compile phase and any compile-time code:

```
perl -cw script.pl
```

`perl -MO=Deparse script.pl` prints what perl actually parsed - precedence misparses, constant-folded branches, `for` rewritten as `while`. Use it to verify that perl saw what you meant:

```
perl -MO=Deparse script.pl
```

`perl -MO=Deparse,-p` forces full parenthesisation and makes precedence explicit.

`perl -MO=Concise script.pl` prints a compact op-tree dump; mostly for readers doing deep optimisation work.

autodie: make failure throw

Built-ins that signal failure via false return values (`open`, `close`, `chdir`, `unlink`) are easy to forget to check. `autodie` replaces them with versions that throw:

```
use autodie qw(open close chdir);

open my $fh, '<', $path;     # now dies on failure; no `or die` needed
```

`autodie` is lexical: only calls in its scope are affected. Narrow the opt-out for a specific call site:

```
{
    no autodie 'open';
```

(continues on next page)

(continued from previous page)

```
    open my $fh, '<', $path or warn "cannot read $path: $!";
}
```

Prefer `autodie` to the retired `Fatal` module, which did not cover functions returning counts.

Unbuffered output

When diagnostic output vanishes before a crash, or arrives interleaved oddly with `warn` messages, the cause is almost always block-buffered stdout. Set `$|` at the top of the script:

```
$| = 1;
```

Every `print` now flushes immediately. The cost is a small I/O-throughput hit; the gain is that what you see on the terminal matches what the program has actually done, in order.

For a specific filehandle without changing the selected one:

```
use IO::Handle;
$log_fh->autoflush(1);
```

Common early diagnostics and what they mean

Diagnostic	Meaning
`Global symbol "$x" requires explicit package name`	Strict is on and `$x` was never declared; add `my`.
`Odd number of elements in hash assignment`	A list assigned to a hash has odd count; a `qw(...)` or string with an embedded space produced two values where one was expected.
`Use of uninitialized value in ...`	An `undef` reached an operator. Usually a typo in a variable name or a missing return.
`Name "main::x" used only once: possible typo`	A package variable was referenced exactly once. Almost always a typo or a missing `my`.
`Can't locate Foo.pm in @INC`	A `use` or `require` could not find the module. Check spelling, `@INC`, `PERL5LIB`.
`Reference found where even-sized list expected`	`my %h = { ... }` - braces build a hashref; use parens.

Find out more

- *print-and-die* - when preflight is not enough and you need structured runtime diagnostics.
- *exceptions* - turning failure paths into typed exceptions the caller can match on.

Print debugging, warn, and the Carp family

This chapter covers non-interactive diagnostics: inserted prints, variable dumps, stack traces from inside the program. These techniques remain the fastest path to understanding "what does my code actually see here?" for bugs that are cheap to reproduce.

Readers who already know about `print` and `warn` will find here the structured variants that earn their keep: the Carp caller-reporting family, `Data::Dumper` and its modern replacements, and the stack-trace-everywhere one-liners.

warn vs print - pick the right stream

`warn` writes to STDERR; `print` defaults to STDOUT. Use `warn` for diagnostics: it keeps program output clean, survives STDOUT redirection, and is the stream loggers already capture.

```
warn "*** foo=[$foo] bar=[@bar]\n";
```

Wrap values in literal brackets so trailing whitespace and newlines are visible in the output. `"foo=[$foo]"` reveals `foo=[ ]` where `"foo=$foo"` just shows `foo= .`

Without a trailing `\n`, `warn` (and `die`) append `at FILE line N.` from the caller's perspective. With `\n`, they do not. Rule of thumb: append `\n` for user-facing messages; omit it for developer diagnostics where you want location.

Cheap location tags from the compiler:

```
warn "reached $0 " . __FILE__ . ':' . __LINE__ . "\n";
warn "in @{[__SUB__->name]}\n";          # 5.16+
```

The Carp family - report from the caller

When a library function fails, pointing at the library's own source is almost never useful. `Carp`'s family reports from the caller's frame:

Function	Reports from	Fatal?	Stack trace?
`carp`	caller	no	no
`croak`	caller	yes	no
`cluck`	caller	no	full
`confess`	caller	yes	full

```
use Carp;

sub open_config {
    my $path = shift // croak "open_config: missing path";
    open my $fh, '<', $path or croak "cannot read $path: $!";
    return $fh;
}
```

Library code should use `croak`/`carp`, not `die`/`warn` - the error message points at the caller's line, which is where the fix goes.

Stack traces without touching the code

`Carp::Always` installs `$SIG{__DIE__}` and `$SIG{__WARN__}` to route through `Carp::confess` / `Carp::cluck`. Run a misbehaving script with:

```
perl -MCarp::Always script.pl
```

Every `die` and `warn` produces a full stack. No source changes. This is the first thing to reach for when a program dies with a useless one-line message.

`Devel::Confess` is a drop-in equivalent with fewer edge cases around overloaded objects:

```
perl -d:Confess script.pl
```

Both modules add cost to every `warn` and `die`; use them while investigating, not in steady production.

For a permanent in-code install:

```
use Carp;
$SIG{__DIE__}  = sub { return if $^S; Carp::confess(@_) };
$SIG{__WARN__} = sub { Carp::cluck(@_) };
```

`return if $^S;` - "return if we are inside `eval`". Without this guard every caught exception pays the trace cost. See *exceptions* for the full `$SIG{__DIE__}` pattern.

Dumping data structures

Pass a **reference** to every dumper; dumping a bare hash or array prints its flat-list form and loses the structure.

`Data::Dumper` - universal, core, eval-roundtrippable

```
use Data::Dumper;

$Data::Dumper::Sortkeys = 1;
$Data::Dumper::Indent   = 1;
$Data::Dumper::Terse    = 1;
$Data::Dumper::Deepcopy = 1;

warn Dumper(\%config);
```

- `Sortkeys` makes output diff-stable.
- `Indent=1` is the readable setting (0 = flat, 2 = default, 3 = with array indices).
- `Terse=1` suppresses `$VAR1 =` prefixes when you do not need round-trip-ability.
- `Deepcopy=1` disables the "seen this ref before" `$VAR1->{...}` tracking - prints the data, not a graph reconstruction.

`Data::Dumper::Concise` is a drop-in with sensible defaults pre-applied; the DBIx::Class / Moose community standard.

`Data::Printer` - coloured, modern, readable

```
use DDP;
p %config;                      # not a reference - DDP handles both
```

`Data::Printer` prints better-looking output than `Data::Dumper`, colours by type, handles objects specially (class, attributes, overloads), and does not try to round-trip. Prefer it when the goal is reading the dump, not re-ingesting it.

`Data::Dump` - compact one-liner

```
use Data::Dump qw(dd);
dd \%config;                    # prints to STDERR in compact form
```

Gisle Aas's `Data::Dump`. Useful when the default `Data::Dumper` output is too verbose for a single warn line.

When to reach for which

- `Data::Dumper` when you might feed the output back into `eval` or compare two dumps.
- `Data::Printer` (DDP) when a human is reading the output.
- `Data::Dump` when you want a one-line summary.
- `Data::Dumper::Concise` on projects already using it.

Conditional debug prints

A compile-time constant is optimised out entirely when false:

```
use constant DEBUG => 0;

warn "state=$state\n" if DEBUG;
```

With `DEBUG => 0`, the `if DEBUG` test is folded at compile time and the `warn` never appears in the op tree - zero runtime cost.

Wire to the environment so the flag is flippable without editing:

```
use constant DEBUG => $ENV{APP_DEBUG} // 0;
```

Multiple channels via bitmask:

```
use constant { WEB => 1, SQL => 2, REGEX => 4 };
my $DEBUG = $ENV{APP_DEBUG} // 0;

sub dbg { my $c = shift; warn "[DBG] @_\n" if $DEBUG & $c }

dbg WEB,   "request $req_id start\n";
dbg SQL,   "query: $sql\n";
```

(continues on next page)

(continued from previous page)

```
dbg REGEX, "matched: $&\n";         # (unless you care about perf - see below)
```

Source-filter alternative: `Smart::Comments`

```
use Smart::Comments;

my @stuff = compute();
### @stuff                  # prints: @stuff: [...]
### Checking key: $key      # print, plus asserts $key is truthy
### Processing |=| ... for @stuff
```

Lines beginning `###` are rewritten into prints (and, with special syntax, progress bars and assertions). The module is a source filter - it rewrites the program text before compile - so reported line numbers can drift by one. When not imported, the `###` comments compile to nothing. The drift is the price of not cluttering the code with explicit `warn` lines.

Taint and $& - incidental costs to know about

Referencing `$&`, `$\``, or `$'` **anywhere** in the program forces perl to maintain these match-context globals for **every** regex match in the entire program, including matches inside loaded modules. Historically this imposed a noticeable slowdown; on modern perls the cost is smaller but non-zero, and the rule still applies: do not touch them if you can avoid it.

Use `@-` / `@+` (match position arrays), named captures, or the `/p` modifier with `${^MATCH}`, `${^PREMATCH}`, `${^POSTMATCH}` instead. The `/p` variants impose the cost only for matches that opt in.

Timestamped logging

Hand-rolled `warn` with `localtime` is adequate for one-off investigations:

```
use Time::HiRes qw(gettimeofday);
sub t { my ($s, $us) = gettimeofday; sprintf "%d.%06d", $s, $us }
warn sprintf "[%s] state=%s\n", t(), $state;
```

For anything more structured, use a logging framework rather than rolling your own:

- `Log::Any` - library-safe; emit logs without picking a backend. Pair with an adapter (`Log::Any::Adapter::Log4perl`, `::Stdout`, `::Syslog`) in the application.
- `Log::Log4perl` - category / level / appender / layout; config file-driven.
- `Log::Dispatch` - lower-level router; often used under `Log::Any`.

Standard levels, lowest to highest: `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR`, `FATAL`. Gate `TRACE` / `DEBUG` behind an env flag in production.

Find out more

- *exceptions* - die, $@, $SIG{__DIE__}, typed exception classes, the modern try/catch.
- *interactive-debugger* - when inserting prints costs more than launching perl -d.
- *tracing* - full-program line-by-line trace without editing the source.

Exceptions, die, and typed error objects

This chapter covers how to raise, catch, and classify runtime failures: die and eval, the native try/catch, special-variable hygiene ($@, $!, $?), $SIG{__DIE__}, and the design of exception classes.

Readers who know die and eval { } will find here the modern replacements for string-dispatching on exception messages and the hazards of $SIG{__DIE__} that the stock eval pattern silently hides.

die and eval

die throws an exception; the nearest enclosing eval { } catches it and leaves the message in $@:

```
eval {
    risky();
};
if (my $err = $@) {
    warn "caught: $err";
}
```

Without a trailing \n, die's argument gets at FILE line N. appended from the caller's frame. With \n, no location is appended - use for messages that will be shown to users, omit for messages you want to trace.

Never use the string form eval "..." for exception handling. It compiles its argument and has every hazard of user-supplied code.

The native try/catch

On Perl 5.34 and later, a dedicated try/catch is available behind a feature flag. Since 5.40 it is non-experimental:

```
use v5.40;
use feature 'try';

try {
    risky();
}
catch ($e) {
    warn "caught: $e";
}
```

`finally` remains experimental as of 5.44. Expect:

```
use feature 'try';
no warnings 'experimental::try';
```

on perls between 5.34 and 5.40.

For code that must run on 5.34 or 5.36 and also on newer perls, `Feature::Compat::Try` picks the core feature where available and falls back to `Syntax::Keyword::Try`:

```
use Feature::Compat::Try;

try { risky(); }
catch ($e) { warn "caught: $e" }
```

Why native `try` over `Try::Tiny`

`Try::Tiny` still works and remains common. Prefer the native form for new code: it avoids the sub-call frame that `Try::Tiny` introduces (which shows up in stack traces), has zero startup cost, and composes with the `isa` operator cleanly.

These are retired - do not adopt:

- `TryCatch` - abandoned.
- `Error.pm` - superseded.
- `Exception::Class::TryCatch` - superseded by `Throwable`.

`$@` clobbering - the classic `eval` hazard

`$@` is set by `eval`; it is also observable globally. Any code that runs between `eval`'s return and your check on `$@` - a `DESTROY` method, a tied-variable handler, an unrelated nested `eval` - can overwrite it. Always copy first:

```
eval { risky(); };
my $err = $@;
return unless $err;
```

Native `try`/`catch` and `Syntax::Keyword::Try` avoid this pitfall by giving you the error as the `catch` parameter; raw `eval` does not. Prefer the former for any code that takes exceptions seriously.

Special variables adjacent to exceptions

Variable	Set by	Read immediately because...
`$@`	`eval { }` / `die`	any intervening call can overwrite.
`$!`	Last failed system call	any following syscall overwrites.
`$?`	`system` / backticks child	the next child overwrites.
`$^E`	Extended OS error	on non-Unix; on Linux usually mirrors `$!`.

`$?` decoding after `system` / backticks:

```
if ($? == -1)         { warn "child failed to start: $!" }
elsif ($? & 127)      { warn "child killed by signal " . ($? & 127)␣
↪}
elsif ($? & 128)      { warn "core dumped" }        # combined with␣
↪above
else                  { my $exit = $? >> 8; ... }
```

Classifying exceptions: dispatch on type

For non-trivial programs, distinguishing "user not found" from "database is down" from "programming error" by parsing error strings is fragile. Throw typed objects and match on them:

```
use v5.40;
use feature 'try';
use Scalar::Util qw(blessed);

try {
    do_business();
}
catch ($e) {
    if    ($e isa MyApp::Err::NotFound)    { return http_404($e) }
    elsif ($e isa MyApp::Err::AuthFailure) { return http_403($e) }
    elsif ($e isa MyApp::Err)              { return http_500($e) }
    elsif (blessed($e) && $e->isa('DBIx::Class::Exception')) {
        return http_500($e);
    }
    else {
        MyApp::Err::Unknown->throw(cause => "$e");
    }
}
```

The `isa` operator (stable on 5.36+) does the right thing for overloaded objects and is zero-cost when the left operand is not a reference. Pair it with a typed wrapper at the outermost boundary so everything downstream is a `MyApp::Err`.

Designing exception classes

The 2026 default: `Throwable` role from CPAN, optionally via `Throwable::SugarFactory` for declarative hierarchies.

Minimum-viable class:

```
package MyApp::Err::NotFound;
use Moo;
with 'Throwable';

has resource => ( is => 'ro', required => 1 );
has id       => ( is => 'ro', required => 1 );
```

(continues on next page)

(continued from previous page)

```
sub message {
    my $self = shift;
    sprintf 'not found: %s/%s', $self->resource, $self->id;
}

1;
```

Usage:

```
MyApp::Err::NotFound->throw(resource => 'user', id => $uid);
```

`throw` is provided by the role; it dies with `$self` (the blessed object). `->new` without `throw` builds without raising.

Declarative form via `Throwable::SugarFactory`:

```
package MyApp::Err;
use Throwable::SugarFactory;

exception 'GenericError'      => 'something bad happened';
exception 'NotFound'           => 'resource not found'
                                  => (has => [ id => (is => 'ro')␣
↪]);
exception 'AuthFailure'        => 'auth failed'
                                  => (has => [ reason => (is => 'ro
↪') ]);
exception 'AuthFailureExpired' => 'session expired'
                                  => (extends => 'AuthFailure');
1;
```

This generates: the class, a `not_found(id => ...)` constructor-and-throw helper, a `is_not_found($e)` predicate, and `->to_hash` on every instance for JSON serialisation.

What a class must have

1. **A class name** - so `$e isa MyApp::Err::X` dispatches.
2. **Structured fields** - `id`, `resource`, `code`.
3. **A human `message`** - without it, `warn $e` prints `MyApp::Err=HASH(0x...)`.
4. **Serialisation** - `->to_hash` for logging / aggregators / JSON APIs.
5. **Origin** - file + line captured at `throw`. `Throwable` captures these automatically.
6. **A stack trace** if the error is worth investigating later: `with 'StackTrace::Auto'` adds `->stack_trace`.

When not to use a class

Plain-string `die` remains correct for short scripts, one-offs, and any case where no caller will dispatch on the error:

```
die "permission denied: $path\n";
```

The trailing `\n` suppresses file-and-line appending; the message is user-facing.

Hash-ref `die - die { type => 'not_found', id => $uid } -` is a transitional form. The moment a caller writes `if (ref $e eq 'HASH' && $e->{type} eq 'not_found')` three times, refactor to a class.

Retired exception frameworks - do not adopt:

- `Error.pm` - use `Throwable` (role-based) or `Exception::Class`.
- `Class::Throwable` - functional; `Throwable` is the modern choice.
- `Exception::Class::TryCatch` - superseded by native `try`.

`$SIG{__DIE__}` - handle with care

`$SIG{__DIE__}` fires on every `die`, **including** `die` inside `eval`. A naive install turns every caught exception into noise:

```
# WRONG: fires on every eval that catches an error
$SIG{__DIE__} = sub { Carp::confess(@_) };
```

The `$^S` special variable tells you whether you are inside an `eval`: true means "yes, someone is going to catch this"; false means "this is reaching the top". Guard on it:

```
$SIG{__DIE__} = sub {
    return if $^S;                # inside eval - not for us
    my ($err) = @_;
    eval { ship_to_sentry($err) };    # never let the handler itself
↪die
    warn "reporter failed: $@" if $@;
    die $err;                     # re-raise
};
```

For Sentry / aggregator integration, wrap the reporter call in its own `eval` - a failing error reporter must not obscure the original exception.

Before installing on top of an existing handler, chain:

```
my $previous = $SIG{__DIE__};
$SIG{__DIE__} = sub {
    return if $^S;
    my ($err) = @_;
    eval { ship_to_sentry($err) };
    warn "reporter failed: $@" if $@;
    goto &$previous if $previous;
    die $err;
};
```

`$SIG{__WARN__}` - turn warnings fatal, or route them

Symmetric to `$SIG{__DIE__}`, fires on every `warn`:

```
$SIG{__WARN__} = sub { die $_[0] };      # promote warnings to
↪exceptions
```

Useful during development as a tripwire; rarely wanted in production. More common production use: route warnings through a logger.

Find out more

- *print-and-die* - the Carp family and stack-trace-everywhere switches.
- `perlfunc/die`, `perlfunc/eval`, `perlfunc/warn`, `perlvar` - reference pages for the language primitives.

The interactive debugger

This chapter covers `perl -d`: launching a program under the debugger, the prompt, the essential motion and inspection commands, and the stack-navigation vocabulary. After this chapter you can investigate any running program interactively.

Readers start from knowing `print` / `warn` work but are not enough: you need to stop, look around, peek at variables, walk the call stack, then resume. That is what this chapter delivers.

Launching

For any script `app.pl`, prepend `-d`:

```
perl -d app.pl --verbose input.txt
```

Arguments after the script name pass to the script as normal.

Common launch forms:

Form	Purpose
`perl -d script.pl args`	Debug the given script.
`perl -d -e 'CODE'`	Debug a one-liner expression.
`perl -de 0`	Interactive REPL sandbox.
`perl -d:Module script.pl`	Load `Devel::Module` as debugger.
`perl -d:NYTProf script`	Profile via NYTProf plugin.
`perl -S -d script.pl`	Search `$PATH` for the script.

`-d` causes the compiler to emit per-statement hook points; the interpreter preloads `perl5db.pl` before the first runtime line.

Never pipe input to `perl -d` and never use `/dev/stdin` - the debugger needs a TTY to read commands from. Launch from a real terminal.

The prompt

```
Loading DB routines from perl5db.pl version 1.77
Editor support available.

Enter h or 'h h' for help, or 'man perldebug' for more help.

main::(app.pl:4):     my $key = 'welcome';
  DB<1>
```

The line `main::(app.pl:4):` shows the next line **about to execute** - nothing in the program has run yet. Compile-time code (`BEGIN`, `use`) already ran; the first prompt comes before the first runtime statement.

`DB<1>` - the number is a command counter that increments on each command. `!N` recalls command number N. `DB<<N>>` with double brackets means you are inside a nested debugger (one expression you stepped into called another breakable expression).

The debugger always shows the line about to execute, never the line just executed.

Quitting

```
  DB<1> q
```

Not `quit`, not `exit`. Typing `exit` runs Perl's `exit` inside the debugger's `eval`, which is not the same. `^D` at the prompt works on most terminals.

Help

```
  DB<1> h               # one-screen summary
  DB<1> h h             # full help (long - use the pager)
  DB<1> h b             # help on one command
  DB<1> |h h            # pipe full help through $ENV{PAGER}
```

Where am I?

`v` views a window around the current line. `l` lists a specified range:

```
  DB<1> v
1       #!/usr/bin/perl
2       use v5.40;
3
4==>    my $key = 'welcome';
5:      my %data = (
6               'this'    => qw(that),
7               'tom'     => qw(and jerry),
8               'welcome' => q(Hello World),
9               'zip'     => q(welcome),
10      );
```

- `==>` marks the next line to execute.

- `:` marks executable lines - you can set a breakpoint there.
- Lines without `:` are not breakable (comments, blanks, continuations of a multi-line statement).

Navigation:

Command	Effect
`v`	View a window around the current line.
`v LINE`	View a window around LINE.
`l`	List the next window; repeat advances.
`l LINE`	List just LINE.
`l MIN-MAX`	List a range.
`l SUB`	List the first window of SUB.
`-`	Previous window.
`.`	Snap view back to the current line.
`f FILE`	Switch display to FILE (regex against `%INC`).
`/PAT/`	Forward search in source.
`?PAT?`	Backward search.

Motion: `s`, `n`, Enter, `c`, *`r`*

The four essential motion commands:

Command	Effect
`s`	Step - execute next statement; descend into subroutines.
`n`	Next - execute next statement; step over subroutines.
`<Enter>`	Repeat the last `s` or `n`.
`c`	Continue until next breakpoint or end.
`c LINE`	Continue to a one-shot breakpoint at LINE, then stop.
`c SUB`	Continue to the first line of SUB.
`r`	Return from the current sub; stop at the caller.

```
DB<1> s                         # descend into the call
DB<2> n                         # stay at this level
DB<3>                           # <Enter> repeats the last motion
DB<4> c 42                      # run to line 42 once
DB<5> r                         # pop the current frame
```

A multi-line statement steps atomically - one `s` executes a whole multi-line hash assignment.

On a line that calls no subroutines, `s` and `n` are indistinguishable.

Printing values: p and x

```
  DB<1> p $key
welcome

  DB<2> p join ',', sort keys %data
this,tom,welcome,zip
```

`p EXPR` evaluates EXPR in the program's current package and prints the scalar-context result followed by a newline. `p` is flat - for a reference it prints `HASH(0x...)`, not contents.

For structures, use `x`:

```
  DB<3> x \%data
0  HASH(0x5651f9a82358)
   'this' => 'that'
   'tom' => 'and'
   'welcome' => 'jerry'
   'zip' => 'Hello World'
```

Always pass a **reference** (`x \%h`, `x \@a`) - `x %h` dumps the flat key/value list, which hides the structure.

See *inspecting-state* for the full inspection vocabulary (`V`, `X`, `y`, `m`, `M`, `i`, depth controls).

The call stack: T, s, r

`T` prints a backtrace - the chain of sub calls that led to the current line:

```
  DB<1> T
$ = main::handle_request(ref(HTTP::Request)) called from file 'app.
 ↪pl' line 47
$ = main::dispatch('GET', '/users/42') called from file 'app.pl'␣
 ↪line 19
@ = main::main() called from file 'app.pl' line 12
```

- `$` - called in scalar context.
- `@` - called in list context.
- `.` - void context.
- Arguments are shown with `Data::Dumper`-style truncation; the exact length cap is the `maxTraceLen` option.

Navigate the stack with `s` (into a call) and `r` (out of the current frame). There is no "frame up / frame down" command: re-step if you go too deep, or use `b` to set a breakpoint at the caller and `c` to reach it.

To inspect an outer frame's lexicals, use `y LEVEL` from *inspecting-state* - it reads via `PadWalker` and shows the enclosing scope's `my` variables.

Evaluating Perl at the prompt

Anything the debugger does not recognise as a command is `eval`'d as Perl code in package `DB` at the current lexical scope:

```
DB<1> $counter = 0                  # mutate state
DB<2> @ARGV = qw(new args)          # mutate @ARGV
DB<3> $obj->reset                   # call a method
DB<4> m/(\w+)/ && p $1              # match against $_
```

A fresh `my $x` at the prompt vanishes when the command returns - the prompt has its own lexical scope. Use a bareword, `our`, or chain on one line:

```
DB<1> our $tmp = compute();         # survives
DB<2> $tmp = compute(); p $tmp      # one-line chain
```

Force Perl interpretation when an expression looks like a debugger command:

```
DB<1> ; h EXPR                     # ; prefix forces eval
DB<2> + h EXPR                     # + prefix works too
```

Restart and rerun

`R` restarts the program via `exec()`. History, breakpoints, actions, options, and the command line are preserved:

```
DB<1> R
```

`rerun` re-executes without leaving the debugger, preserving state:

Form	Effect
`rerun`	Restart, replaying no prior commands.
`rerun -N`	Restart and replay up to N commands back.
`rerun N`	Restart from history entry N.

Redefining a sub mid-session

Paste the entire `sub NAME { ... }` at the prompt. Or edit a replacement file and load it:

```
DB<1> do 'fixed.pl'
```

Subsequent calls use the new definition. Existing stack frames continue with the old body.

Commands so far

Command	Effect
h, h CMD, h h	Help: summary / one command / full.
|CMD	Pipe the output through $ENV{PAGER}.
q	Quit.
v, v LINE	View source around current / LINE.
l, l LINE, l SUB, l MIN-MAX	List.
.	Snap to current line.
-	Previous window.
f FILE	Switch display to FILE.
/PAT/, ?PAT?	Search forward / backward.
s	Step into.
n	Step over.
<Enter>	Repeat last s / n.
r	Return from current sub.
c, c LINE, c SUB	Continue / one-shot breakpoint.
p EXPR	Print (scalar context).
x EXPR	Dump (structured, pass a reference).
T	Stack backtrace.
R	Restart via exec().
rerun [N|-N]	Restart with history replay.

Find out more

- *breakpoints* - b, conditionals, actions, watches, timing matrix.
- *inspecting-state* - V, X, y, m, M, i, depth controls.
- *tracing* - t, AutoTrace, line-info output to a file.

Breakpoints, actions, and watches

This chapter covers stopping the debugger at a specific point - a line, a subroutine, a condition, a value change - and the prompt-hook machinery that lets you run code automatically at each stop.

Readers arriving here know how to launch perl -d and step one line at a time. The next skill is to skip straight to the interesting place and stop only when the problem is about to manifest.

Breakpoints on lines and subs

Command	Effect
`b`	Breakpoint on the current line.
`b LINE`	Breakpoint at LINE in the current file.
`b LINE COND`	Conditional; break when COND is true.
`b FILE:LINE`	Cross-file; FILE matches a `%INC` entry.
`b FILE:LINE COND`	Conditional cross-file.
`b SUB`	Break at the first line of SUB (SUB must be compiled).
`b SUB COND`	Conditional sub breakpoint.
`b $coderef`	Break at the sub referenced by `$coderef` (no COND supported).
`b compile SUB`	Break after SUB finishes compiling, before its body runs.
`b postpone SUB`	Break the first time SUB is called, even if not yet loaded.
`b load FILE`	Break when FILE is `required` / do'd.
`B LINE`	Delete breakpoint at LINE.
`B *`	Delete all breakpoints.
`disable [FILE:]LINE`	Disable without removing.
`enable [FILE:]LINE`	Re-enable.
`L`	List breakpoints, actions, and watches.
`L b`	List breakpoints only.

Conditions are **raw expressions**, not `if (…)`:

```
  DB<1> b 237 $count > 10
  DB<2> b 42 /unauthorized/i
  DB<3> b validate_payload ref($obj) eq 'HASH' && keys %$obj
```

Breakpoints settle only on lines that contain executable code. A blank line, a pure comment, or a continuation of an earlier statement refuses:

```
  DB<1> b 3
Line 3 not breakable.
```

A line carries one breakpoint and one action at most; adding a second overwrites the first.

Timing matrix

Where in the program lifecycle a breakpoint fires matters - especially for code in `BEGIN` blocks or modules that are `required` lazily.

Goal	Command
Stop at the first runtime statement	Default - the debugger already stops there.
Stop inside a `BEGIN { }` block	Insert `$DB::single = 1;` inside the block.
Stop before a sub's body runs (after it compiles)	`b compile SUB`
Stop on first call of a sub not yet loaded	`b postpone SUB`
Stop when a file is `required` or do'd	`b load /full/path/from/%INC`
Stop at a specific line	`b LINE`
Stop at a line only when a condition holds	`b LINE COND`

`$DB::single = 1;` inside a `BEGIN` block is the trick for compile-time debugging: writing `b` at the prompt only works after the debugger is running, but `BEGIN` code runs before the first prompt. Drop the assignment into the block, launch under `-d`, and the debugger stops at the next breakable line inside the block.

Accessing `$DB::single` when not under `-d` autovivifies `DB::` harmlessly - the pattern is safe to commit.

Listing and managing breakpoints

```
  DB<1> L
app.pl:
 42:   $user = User->load($id);
       break if (1)
 91:   dispatch($req);
       break if ($req->method eq 'POST')

  DB<2> disable 42
  DB<3> enable 42
  DB<4> B 42
  DB<5> B *
```

`disable` / `enable` toggle a breakpoint without forgetting its condition - useful when you want to skip a hot breakpoint temporarily and re-arm it later.

Actions: run code automatically at a line

An **action** runs Perl code before the specified line executes, without stopping. Use for passive observation:

```
  DB<1> a 42 print "entering: user_id=$user_id\n"
  DB<2> a 91 warn "POST body:\n" . Dumper($body) if $req->method eq
↪'POST'
  DB<3> L a
app.pl:
 42: ...
       action: print "entering: user_id=$user_id\n"
```

(continues on next page)

(continued from previous page)

```
  DB<4> A 42              # delete action at line 42
  DB<5> A *               # delete all actions
```

Ordering at a breakable line:

1. Breakpoint condition is tested.
2. Trace output is printed (if tracing on).
3. Action runs.
4. Prompt appears (if breakpoint hit or stepping).
5. The line itself executes.

Actions and breakpoints coexist on the same line - one of each maximum.

Watches: break on value change

A watch reports the old and new value when a tracked expression changes, and drops you at the prompt:

```
  DB<1> w $global_count
  DB<2> c
Watchpoint 0:   $global_count changed:
    old value:  42
    new value:  43
main::tick(app.pl:118):    ++$global_count;
  DB<3> W $global_count   # delete this watch
  DB<4> W *               # delete all watches
```

Watches are evaluated at every breakable line. Expressions must reference globals or accessible aliases; lexical `my` variables are not watchable directly. When a lexical matters, use `Tie::Watch` in the code to install a watcher at tie time, or assign via a package-scoped alias.

Watches add per-statement cost proportional to the number of watches and the expense of each expression. Keep them narrow and remove them when done.

Prompt hooks: run code at every stop

< / > install Perl code to run automatically before / after every prompt. { / {{ do the same for debugger commands:

Command	Effect
`< CMD`	Perl code before each prompt (= after the line just executed).
`<< CMD`	Append to the pre-prompt list.
`< ?`	List pre-prompt code.
`< *`	Delete all pre-prompt code.
`> CMD` / `>> CMD`	Perl code after each prompt (= before next line).
`{ DBCMD`	Debugger command (not Perl) pre-prompt.
`{{ DBCMD`	Append to the pre-prompt debugger-command list.

Typical use: timestamp every step:

```
  DB<1> < print "stop: ", scalar localtime, "\n"
  DB<2> > print "start: ", scalar localtime, "\n"
```

Or always show the current line and a watched variable:

```
  DB<1> { v
  DB<2> { p $important
```

Multi-line: escape the newline with \:

```
  DB<1> < for my $f (@important_files) {    \
cont:     print "  $f: ", (-s $f) // 'n/a', "\n"; \
cont: }
```

Aliases

Persistent command shortcuts via =:

```
  DB<1> = len s/^len(.*)/p length($1)/
  DB<2> = quit s/^quit(\s*)/exit/
  DB<3> =                      # list all aliases
```

Aliases live in %DB::alias and apply at dispatch time. Define once in .perldb for across-session persistence.

One-shot breakpoints

c LINE sets a breakpoint at LINE, continues, and clears the breakpoint when it hits:

```
  DB<1> c 42
```

Simplest way to say "just run to line 42 and stop there". No manual b / B pairing.

Breaking on import / first-call lazily

When a sub lives in a module that may not be loaded yet, b SUB fails:

```
  DB<1> b MyApp::Worker::dispatch
Subroutine not yet compiled.
```

b postpone waits for the sub to finish compiling:

```
  DB<1> b postpone MyApp::Worker::dispatch
  DB<2> c
```

For files loaded by require / do, b load breaks on first load:

```
  DB<1> b load /usr/share/perl5/MyApp/Worker.pm
```

The filename must match the full path in %INC, not just the module name.

Find out more

- *inspecting-state* - what to look at once a breakpoint fires.
- *tracing* - when you want to watch what happens *without* stopping.

Inspecting state

This chapter covers the debugger's data-inspection vocabulary: `x`, `V`, `X`, `y`, `m`, `M`, `i`, and the options that control how deeply aggregate values are dumped. After this chapter you can ask any running program "what does this variable, this package, this object actually look like right now?"

Readers arrive here knowing `p EXPR` prints a scalar. The rest of the inspection commands earn their keep the moment the answer "what is this?" requires more than a single value.

x: dump structures

`x EXPR` pretty-prints in list context, recursing into references:

```
  DB<1> x \%config
0  HASH(0x5651f9a82358)
   'debug' => 1
   'dsn' => 'dbi:SQLite::memory:'
   'timeout' => 30
   'users' => ARRAY(0x5651f9b28a10)
      0  'alice'
      1  'bob'
      2  'carol'
```

Always pass a **reference** for aggregates. `x %h` dumps the flat key/value list, which hides the structure. `x \%h` does what you want.

```
  DB<2> x \@list
0  ARRAY(0x5651f9b28a10)
   0  'alice'
   1  'bob'
   2  'carol'

  DB<3> x $obj
0  MY::Class=HASH(0x5651fa01c2b0)
   'id' => 42
   'prefs' => HASH(0x5651fa01c330)
      'theme' => 'dark'
```

For deep or cyclic structures, cap the recursion:

```
  DB<4> x 2 \%tree      # recurse 2 levels deep
  DB<5> x 1 $obj        # one level
```

If the first token of the expression is numeric, parenthesise:

```
  DB<6> x (3 + $offset)     # without parens, 3 is mis-parsed as
↪depth
```

V and X: package variables

`V [PKG [VARS]]` dumps the globals of a package:

```
  DB<1> V main
$counter = 42
@ARGV = (
    0  '--verbose'
    1  'input.txt'
)
%ENV = (
    'PATH' => '/usr/bin:/bin'
    ...
)
```

With a second argument, filter variables:

```
  DB<2> V MyApp $config $last_user       # only these
  DB<3> V MyApp ~^config                 # regex: names matching /^
↪config/
  DB<4> V MyApp !^_                      # regex: names NOT matching
↪/^_/
```

`X` is `V` for the current package:

```
  DB<5> X                # dump current package's globals
  DB<6> X $foo @bar      # filtered
```

`V` / `X` do **not** see lexicals (`my` variables). For those, use `y`.

y: lexicals in enclosing scope

`y LEVEL [VARS]` reads lexical `my` variables via `PadWalker`:

```
  DB<1> y 0               # lexicals of the current sub
my $req = HTTP::Request=HASH(0x...)
my $user_id = 42
my @errors = ()

  DB<2> y 1               # one frame up (the caller's lexicals)
my $request_count = 118
my $dispatcher = MyApp::Dispatcher=HASH(0x...)

  DB<3> y 0 $req $user_id  # only those
```

`y` requires `PadWalker` version 0.08 or later. It works only inside a sub (the outer script body has no pad to walk).

`m`: methods of an object or class

`m EXPR` lists methods callable on an object or class:

```
  DB<1> m $user
via UNIVERSAL: isa can DOES VERSION DESTROY
via MyApp::Model: save load delete update
via MyApp::Role::Auditable: audit_log record_event

  DB<2> m 'MyApp::Worker'
via UNIVERSAL: ...
via MyApp::Worker: dispatch retry cancel
```

Groups methods by the package that provides them - useful for spotting which role or superclass a method comes from.

`M` and `i`: modules and inheritance

`M` lists loaded modules with versions:

```
  DB<1> M
  'DBI' (version 1.643)
  'JSON::PP' (version 4.16)
  'MyApp::Model' (version undef)
  ...
```

`i CLASS` or `i $obj` prints the inheritance tree:

```
  DB<2> i $user
MyApp::User 0.12
   MyApp::Model 0.42
      Moo::Object ?
```

Useful for "where is this method actually coming from?" when method resolution crosses several roles.

`S`: subroutine names

`S [[!]PAT]` lists sub names matching (or not matching) a pattern:

```
  DB<1> S ^MyApp::Worker::
MyApp::Worker::cancel
MyApp::Worker::dispatch
MyApp::Worker::retry

  DB<2> S !^(main|DB|UNIVERSAL)::      # everything except these
```

`S` alone lists everything - long.

p vs x: which to use when

Want	Use
A single scalar's value, plain	`p $x`
A joined / formatted expression	`p join ',', sort keys %h`
A hash or array	`x \%h`, `x \@a`
An object's fields	`x $obj`
Only the top level	`x 1 $deep_tree`
Package variables of `Pkg`	`V Pkg`
Lexicals of current sub	`y 0`
Method list of `$obj`	`m $obj`
`@ISA` chain of `$obj`	`i $obj`

Special case: `p` is a flat print. For a reference, `p $ref` prints `HASH(0x…)`. Use `x $ref` for the contents.

Depth and compactness options

Dump output is shaped by options set via the `o` command. The relevant settings for inspection:

Option	Effect
`dumpDepth`	Recursion depth (negative = unlimited). Default 4.
`arrayDepth`	First N array elements only; empty = all.
`hashDepth`	Same, for hashes.
`compactDump`	Short arrays on one line.
`veryCompact`	Tighter yet.
`globPrint`	Dump glob contents.
`DumpReused`	Fully expand repeated references (beware cycles).
`quote`	`"` or `'` or `auto` (default).
`undefPrint`	How `undef` renders.
`bareStringify`	Print overload-stringified form.

Set an option in a session:

```
  DB<1> o dumpDepth=6
  DB<2> o compactDump=1
  DB<3> o arrayDepth=20
  DB<4> o compactDump?              # query current value
```

Set options at startup via `PERLDB_OPTS` or in `.perldb`:

```
PERLDB_OPTS="dumpDepth=6 compactDump=1" perl -d script.pl
```

Inspecting SV internals

`Devel::Peek` exposes the scalar's internal storage - type flags, refcount, PV/NV/IV layout, magic chain - when the question is "what kind of scalar is this really?":

```
  DB<1> use Devel::Peek; Dump($sv)
```

Primarily for investigating tied variables, overloaded objects, or performance anomalies where the answer depends on whether a scalar is currently stored as an integer, a string, or both.

Note: `Devel::Peek` in pperl is a compat layer - it reports a pperl-specific `Sv` layout rather than perl5's SV layout. Fields like `REFCNT`, `FLAGS`, `PV`, `IV`, `NV` have equivalents; the exact output differs. See the *pperl-architecture guide* for layout details.

Evaluating at the prompt

Anything unrecognised as a command is evaluated as Perl. Use it to drill deeper than a single command:

```
  DB<1> p keys %{ $obj->{prefs} }
theme,locale,tz

  DB<2> x [ grep { !defined $_->{parent} } @nodes ]
  DB<3> x { map { $_->id => $_->name } @users }
```

For one-off assertions without stopping:

```
  DB<1> die "unexpected" if ref $obj ne 'MyApp::User'
```

Program I/O vs debugger I/O

The debugger's own input and output go through `$DB::IN` and `$DB::OUT`, which are opened to `/dev/tty` directly. The program can redirect STDIN / STDOUT / STDERR freely; the debugger prompt still shows up. Conversely, `p EXPR` output does not contaminate a program's captured stdout.

Find out more

- *breakpoints* - stop where you can inspect; this chapter is about what to look at once stopped.
- *tracing* - watch values change over time without stopping at every step.

Tracing: watching execution flow

This chapter covers traces that record what a program did without stopping it: the debugger's `t` command and trace options, the `Devel::Trace` module family, and the regex-engine tracer. Use tracing when the bug is "which code path actually ran?" rather than "what value does X have at line N?".

Readers coming in know how to set breakpoints and inspect state. A trace is the complement - "follow every step and tell me about it" rather than "stop at one interesting place".

t: trace inside the debugger

`t` toggles per-statement tracing. Every breakable line prints a location marker before executing:

```
  DB<1> t
Trace = on

  DB<2> c
main::(app.pl:5):           my $key = 'welcome';
main::(app.pl:6):           my %data = (
main::handle_request(app.pl:42):   my $user = load_user($req);
main::load_user(app.pl:87):        my ($req) = @_;
main::load_user(app.pl:88):        return unless $req;
...
```

Restricting the depth:

```
  DB<3> t 2           # toggle with maximum sub-depth 2
  DB<4> t 0 my_work() # trace one expression only
```

The `frame` option controls what the trace prints at sub calls and returns:

Bit	Flag	Effect
1	enter	Print on sub entry.
2	exit	Print on sub exit.
4	ctx+args	Include call context and arguments.
8	tied/ref	Mark tied-variable and reference sub calls.
16	return	Print the return value.

Set `frame` via `o`:

```
  DB<5> o frame=15       # enter + exit + ctx/args + tied/ref
```

Common combinations:

- `frame=2`: just entry/exit traces, one line per call - the call-tree view.
- `frame=6`: add caller context and argument dump - full trace.
- `frame=30`: like `frame=6` plus return values.

Trace output goes to the debugger's output stream. Redirect it to a file so it does not drown the interactive session:

```
  DB<6> o LineInfo=trace.out
```

Read the file in another terminal:

```
tail -f trace.out
```

Non-interactive tracing: NonStop + AutoTrace

For "record a full execution trace with no interaction", combine `NonStop=1` (no prompts) and `AutoTrace=1` (tracing on from start):

```
PERLDB_OPTS="NonStop=1 AutoTrace=1 frame=2 LineInfo=trace.out" \
  perl -d script.pl
```

The program runs straight through; `trace.out` captures every breakable line and every sub entry. Ideal for CI-mode failure capture, headless servers, or any environment where a TTY prompt is unavailable.

Add `inhibit_exit=0` to let the program fall off the end without the "Debugged program terminated" prompt.

Devel::Trace: every line, without the debugger

`Devel::Trace` traces every line of every file to STDERR. No command-line interaction. No breakpoints. Simple and loud:

```
perl -d:Trace script.pl 2> trace.out
```

```
>> app.pl:4: my $key = 'welcome';
>> app.pl:5: my %data = (
>> app.pl:11: print "$data{$key}\n";
```

Run it once, grep the output for the puzzle. Output size is proportional to program run length - use on a minimal reproducer, not the whole test suite.

Variants:

- `Devel::DumpTrace` - same plus per-line variable values.
- `Devel::TraceUse` - "which modules were loaded, in what order, by whom":

  ```
  perl -d:TraceUse script.pl
  ```

- `Devel::TraceINC` - narrower: just the `require` chain with timing.
- `Devel::Trace::Subs` - call-flow and stack-trace generator for "which subs actually ran".

Tracing a specific region

The `$DB::trace` variable toggles tracing from Perl code:

```
$DB::trace = 1;
interesting_region();
$DB::trace = 0;
```

Assigning to `$DB::trace` when not under `-d` autovivifies the variable harmlessly - the code is safe to commit and runs as a no-op outside the debugger.

Same pattern for single-stepping a region:

```
$DB::single = 1;        # break here when next statement runs under
↪-d
```

Regex tracing: `use re 'debug'`

The regex engine has its own tracer. Enable lexically inside a region, or on the command line:

```
{
    use re 'debug';
    $str =~ /(\w+)\s+(\w+)/;
}
```

```
perl -Mre=debug -e '"hello world" =~ /(\w+)\s+(\w+)/'
```

Compile-phase output shows what the optimiser did:

- `Compiling REx 'PATTERN'` - the source pattern.
- `size N` - internal node-table size.
- `anchored 'SUB' at K` / `floating 'SUB' at K..LIMIT` - which substring the engine will look for as a pre-filter.
- `stclass TYPE` - the starting-character class.
- `minlen N` - no match possible on input shorter than N.
- Node listing - the compiled program.

Runtime output shows each step:

```
  POS <PRE-MATCH> <POST-MATCH>    |  NODE_NUM: OPNAME
```

Indentation reflects backtracking depth; `failed...` and `failed, try continuation...` mark backtracks; `Match successful!` / `Match failed.` close the trace.

If no runtime output appears at all, the optimiser decided the match without entering the engine - either `minlen` pre-filtered or the pattern was folded to a constant test.

pperl's regex engine is a native reimplementation, not a port of perl5's. The trace is functionally equivalent but the exact node names and field formats are not byte-identical to upstream. Parsing the trace programmatically (few tools do) will need adjustment.

Partial switches narrow the output:

```
use re 'debugcolor';                    # ANSI colour
use re debug => 'EXECUTE';              # runtime only, no compile
↪output
```

`Devel::TraceUse` for module-load bugs

A program that runs slowly at startup, or a "why is this module loaded at all?", is almost always a module-graph question. `Devel::TraceUse` prints the full `use`/`require` tree:

```
perl -d:TraceUse script.pl
Modules used from script.pl:
   1. strict 1.12, script.pl line 2 [main]
   2. warnings 1.58, script.pl line 2 [main]
   3. DBI 1.643, script.pl line 5 [main]
      4. Carp 1.52, DBI line 13 [DBI]
      5. DynaLoader 1.47, DBI line 14 [DBI]
   ...
```

Indentation shows the dependency chain. Cross-reference with the line number in the parent file to see where each load happens.

Combining traces with the debugger

Trace to a file, step interactively:

```
PERLDB_OPTS="frame=6 LineInfo=trace.out" perl -d app.pl
```

The debugger prompt appears normally; `trace.out` accumulates the call-tree as you step and continue. Useful when a call chain is too deep to follow by stepping but you still want to stop at specific points to inspect.

Find out more

- *breakpoints* - stopping at a specific point rather than recording every step.
- *inspecting-state* - the commands that answer "what is this value?".

IDE and DAP integration

This chapter covers editor-driven debugging: `Perl::LanguageServer` in VS Code and its counterparts in Neovim / Emacs / IntelliJ, remote-over-SSH debugging, and container-mode debugging via `kubectl` or `docker`.

Readers who reach for an IDE rather than a terminal prompt will find here the minimum viable setup, the config keys that matter, and the known fragility points (Coro, path mapping, attach vs launch).

The landscape

Two wire protocols, both with Perl backends:

Protocol	Perl backend	Status
DAP	`Perl::LanguageServer`	Active.
DBGp	`Devel::Debug::DBGp`	Stagnant since 2018.

For new setups, use DAP via `Perl::LanguageServer`. DBGp survives only where an existing client (vim vdebug, pugdebug) is already in use.

`Perl::LanguageServer` bundles an LSP (code intelligence, diagnostics, go-to-definition) and a DAP (debugger) adapter in one process. The LSP half works on any Perl file in the editor; the DAP half runs only during a debug session.

Minimal VS Code setup

CPAN side - install into the Perl the debuggee will run under:

```
cpanm Perl::LanguageServer
```

Editor side - install the `richterger.perl` extension:

```
Ctrl-P  →  ext install richterger.perl
```

`.vscode/launch.json`:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "perl",
      "request": "launch",
      "name": "Debug current file",
      "program": "${workspaceFolder}/${relativeFile}",
      "stopOnEntry": true,
      "args": [],
      "cwd": "${workspaceFolder}",
      "env": {},
      "reloadModules": true
    }
  ]
}
```

Open a `.pl` or `.pm`, set breakpoints in the gutter, press `F5`. The Debug Console accepts any Perl expression, evaluated in the current stack frame.

`.vscode/settings.json` for `@INC` and interpreter selection:

```
{
  "perl.perlInc": [
    "${workspaceFolder}/lib",
    "${workspaceFolder}/local/lib/perl5"
  ],
  "perl.perlCmd": "/usr/bin/perl"
}
```

Capabilities

`Perl::LanguageServer` supports the DAP features that matter:

- Launch, pause, step in / over / out, return.
- Conditional and unconditional breakpoints, added at any time.
- Variable inspection across stack frames.
- **Set variable** - change a value mid-session from the editor.
- Watch expressions.
- Evaluate-in-context at the Debug Console.
- Module hot-reload via `reloadModules: true`.

Not supported:

- **Attach to a running process.** `request: "attach"` is not implemented; only `"launch"` (the LS starts the debuggee itself).

Remote debugging over SSH

Edit locally; debug on a remote host. `settings.json`:

```
{
  "perl.sshCmd": "ssh",
  "perl.sshAddr": "deploy@host.example",
  "perl.sshUser": "deploy",
  "perl.sshWorkspaceRoot": "/srv/app",
  "perl.pathMap": [
    ["/srv/app", "${workspaceFolder}"]
  ]
}
```

`Perl::LanguageServer` runs on the remote host; the editor stays local. `pathMap` translates filenames the debuggee reports (`/srv/app/lib/X.pm`) to local editor paths (`${workspaceFolder}/lib/X.pm`).

Breakpoints that silently fail to bind are almost always a `pathMap` mismatch.

Container-mode debugging

`launch.json` accepts container keys that invoke `docker`, `podman`, `docker-compose`, or `kubectl`:

Key	Values
`containerCmd`	`"docker"`, `"docker-compose"`, `"podman"`, `"kubectl"`
`containerMode`	`"run"` (fresh container) or `"exec"` (existing)
`containerName`	Image (for `run`) or container name (for `exec`)
`containerArgs`	Extra args passed to the container runtime.
`pathMap`	`[["/in/container", "/on/host"], ...]`

Attach to an already-running Docker container:

```
{
  "type": "perl",
  "request": "launch",
  "name": "Debug in container",
  "program": "/app/bin/worker.pl",
  "containerCmd": "docker",
  "containerMode": "exec",
  "containerName": "my-running-worker",
  "pathMap": [["/app", "${workspaceFolder}"]]
}
```

Kubernetes pod:

```
{
  "type": "perl",
  "request": "launch",
  "name": "Debug in pod",
  "program": "/app/bin/worker.pl",
  "containerCmd": "kubectl",
  "containerMode": "exec",
  "containerName": "my-pod-abc123",
  "containerArgs": ["-n", "staging", "--container", "app"],
  "pathMap": [["/app", "${workspaceFolder}"]]
}
```

Constraints:

- **`Perl::LanguageServer` must be installed inside the image.** Bake it into the Dockerfile or build a dev-image variant.
- `pathMap` is the single biggest failure mode. Breakpoints that never bind mean the debuggee reports a path that does not translate.

Other editors

- **Neovim** - use `nvim-dap` with `Perl::LanguageServer` as the adapter. The adapter spec is a short Lua snippet that points at `perl -MPerl::LanguageServer -e '...'`.
- **Emacs** - `dap-mode` with the same adapter.
- **IntelliJ** - `Devel::Camelcadedb` + the JetBrains Perl plugin. Not DAP; proprietary protocol. Viable but outside the `Perl::LanguageServer` ecosystem.
- **Sublime Text** - DAP clients exist via plugins; use `Perl::LanguageServer` as the adapter.

Special-mode launches

`launch.json` keys for non-default invocation:

Key	Effect
`useTaintForDebug`	Injects `-T` into the debuggee's perl invocation.
`sudoUser`	Re-execs the debuggee as that user (needs passwordless sudo).
`reloadModules`	Enables module hot-reload during the session.
`stopOnEntry`	Break at the first statement.

Gotchas

- **BEGIN-time errors** surface as DAP "output" events, not as stopped-at-breakpoint events. Use `stopOnEntry: true` to get a stop at all if the script dies in `BEGIN`.
- **Application `$SIG{__DIE__}`** can swallow exceptions before the debugger's handler runs. Wrap the app's handler in an `if ($^S)` guard (see *exceptions*).
- **Watch expressions re-evaluate on every step.** Avoid expensive or side-effect expressions.
- **No source maps.** Perl has no concept equivalent to a JavaScript source map. Source filters (`Smart::Comments`) can shift reported line numbers by one; breakpoints in filtered files may land on the wrong line.
- **`Future::AsyncAwait` stepping.** Works for simple cases; stepping through complex `await` chains is unreliable. Drop to `perl -d` for deep async investigation, or use logging.
- **Windows native** - `Perl::LanguageServer`'s stdin handling is broken on Windows. WSL works.
- **Coro.** `Perl::LanguageServer` relies on Coro internally; the most-cited fragility point is Coro interacting badly with unusual build configurations. If the server refuses to start, check Coro's install log first.

Find out more

- *interactive-debugger* - `perl -d` directly, when the IDE stack is overkill or unavailable.

Coverage: which code your tests run

A coverage report answers a question the debugger cannot: which statements, branches, conditions and subroutines a test run never reached. pperl runs `Devel::Cover`, the coverage tool perl5 uses, and reports the same numbers perl5 does.

Installing Devel::Cover

The compiled part of `Devel::Cover` is built into pperl. Its Perl modules and its `cover` command come from CPAN:

```
pperl --cpan Devel::Cover
```

They land under `~/.pperl` (or `$PPERL_HOME`, see *Installing CPAN modules*): the modules in `~/.pperl/lib/perl5`, which pperl searches without being told, and the command as `~/.pperl/bin/cover`.

If `PERL5LIB` points at a perl5 tree that already has Devel::Cover, `pperl --cpan` finds that copy, reports the module as up to date and installs nothing. Run it with `PERL5LIB` unset. Keep the perl5 tree's copy out of covered runs, too: pperl needs its own.

Measuring a program

Load `Devel::Cover` with `-M`, then run `cover` in the same directory:

```
pperl -MDevel::Cover yourprog.pl
pperl ~/.pperl/bin/cover
```

The run collects into `cover_db/` in the current directory. `cover` prints a summary per file and writes an HTML report to `cover_db/coverage.html`. For a program `prog.pl` that calls one of the two subs of its module `lib/Shape.pm`:

```
------------ ------ ------ ------ ------ ------ ------
File           stmt   bran   cond    sub   time  total
------------ ------ ------ ------ ------ ------ ------
lib/Shape.pm   50.0   50.0   33.3   50.0    0.0   45.4
prog.pl       100.0    n/a    n/a  100.0  100.0  100.0
Total          81.8   50.0   33.3   75.0  100.0   70.0
------------ ------ ------ ------ ------ ------ ------
```

Every option of `Devel::Cover` and of `cover` works as under perl5: `cover -report text` for a plain-text report, `-report json` for JSON, `cover -delete` to start over. A pod column appears when `Pod::Coverage` is installed, as it does under perl5.

Measuring a test suite

`HARNESS_PERL_SWITCHES` adds `-MDevel::Cover` to every test the harness starts. Run `prove` itself under pperl, so that the tests run under pperl too; `-S` finds `prove` on the `PATH`:

```
HARNESS_PERL_SWITCHES=-MDevel::Cover=-ignore,^t/ pperl -S prove -l t
pperl ~/.pperl/bin/cover
```

`-ignore,^t/` keeps the test files themselves out of the report. `prove --exec pperl` does not apply `HARNESS_PERL_SWITCHES`, under perl5 either, nor `-l`; with `--exec`, put both in the command itself: `prove --exec "pperl -Ilib -MDevel::Cover=-ignore,^t/" t`.

Choosing what is reported

Devel::Cover leaves out modules that belong to the installation, `~/.pperl/lib/perl5` among them. The check it uses for that does not know `PPERL5LIB` or `PPERL_HOME`, so with either set, modules from those trees are covered and reported like your own code. In that setup, choose the files with `-select` and `-ignore`, both regular expressions over file names:

```
pperl -MDevel::Cover=-select,^lib/,-ignore,^t/ yourprog.pl
```

Speed

While Devel::Cover collects, pperl does not compile loops to machine code: compiled code would run past the counters Devel::Cover reads. A covered run is therefore as fast as pperl's interpreter, not as fast as a normal pperl run. A run without `-MDevel::Cover` is unaffected.

4.2.7 The Definitive Perl One-Liner Guide

A Perl one-liner is a complete program delivered as an argument to `pperl -e` (or `-E`), run directly from the shell. It occupies the same niche as `awk`, `sed`, and ad-hoc shell pipelines, with a richer regex engine, a full standard library, and the ability to grow into a real script the moment one line stops being enough.

This guide is for a Linux shell user who already composes `grep`/`sed`/`awk`/`xargs` pipelines and has written a small Perl script or two. It teaches the command-line surface end to end: the switches, a progressive recipe catalogue, regex-driven transformations, numeric work, how to wrap recipes into reusable shell aliases, and the traps that cost time.

The binary is `pperl`. Every example in this guide is copy-pasteable into a Linux shell. pperl is a Rust reimplementation of Perl 5; its command-line interface matches Perl 5.44.

Which chapter do I want?

You want to ...	Start here
Know what each switch (`-n`, `-p`, `-a`, ...) does	*switches*
Browse recipes from trivial substitutions up to data reductions	*progression*
Match, extract, or rewrite text with regexps	*regex-recipes*
Sum, average, generate primes, convert IPs, do date arithmetic	*numeric*
Turn a one-liner into a reusable shell function or alias	*aliases*
Debug quoting, encoding, in-place edits, and record-separator surprises	*gotchas*

The three layers of this guide

- **Orientation** - this page.
- **Working programmer** - the six chapters above. Each stands on its own; skim the ones relevant to your current problem.
- **Deep reference** - the compact switch/variable table at the end of *switches*.

Two examples before you start

A sed/awk user reading this guide wants to see what a one-liner buys them. Two demonstrations.

Sum the numbers in the last column of a whitespace-separated file

```
pperl -lane '$s += $F[-1]; END { print $s }' data.txt
```

`-l` chomps input newlines and appends one on output. `-a` splits each line into `@F`. `-n` wraps the program in an implicit read loop. `$F[-1]` is the last column. `END { … }` runs once after input is exhausted. No temporary file, no `awk 'BEGIN{…}END{…}'`, no boilerplate.

Extract every unique IPv4 address from a log, newest first

```
pperl -lne 'print for /\b(?:\d{1,3}\.){3}\d{1,3}\b/g' access.log \
  | awk '!seen[$0]++'
```

The one-liner handles the hard part (finding all matches per line, not just the first), the awk keeps the first sighting of each. Written as two short tools, each doing one job - that is the working mode of this guide.

Read in this order on a first pass

1. *switches* - the vocabulary.
2. *progression* - the recipes, ordered by difficulty.
3. *regex-recipes*, *numeric* - applied pieces.
4. *aliases* - lifting what you use daily into shell functions.
5. *gotchas* - the traps, once you are running into them.

On later passes, go directly to whichever chapter names the problem you are holding.

Bibliography

This guide is not derived from any single book. It was researched the way a technical writer would research the topic - perldoc, CPAN module documentation, conference talks, blog posts, mailing-list archives, and the books listed below - and then distilled in several passes: removing references to tools and versions nobody runs anymore, collapsing the same advice repeated by different authors into one statement, and merging multiple partial treatments of one topic into a single complete treatment. The result covers more ground than any single source we consulted, because no single source covers the whole field. It is also more densely cross-linked than any of them: when the regex chapter touches something the regex guide already explains in depth, that is one click away - not a chapter you have to find by yourself in another volume.

The two books below are the closest published companions in print form. Buy them if the one-liner idiom is a daily tool for you and you want a printed reference on the shelf.

- Peteris Krumins, *Perl One-Liners: 130 Programs That Get Things Done*, No Starch Press, 2014.
- Sundeep Agarwal, *Perl One-Liners Guide* (learnbyexample), 2023.

The switches

Every one-liner is a combination of a program (passed to `-e` or `-E`) and a handful of switches that wrap implicit behaviour around it. This chapter covers the switches you will reach for. Each one earns its keep in the progression, regex, and numeric chapters that follow.

Readers coming from `awk` or `sed` already know what per-line iteration, field splitting, and in-place editing look like; the question is which pperl flag turns each on.

`-e` - run a program given on the command line

```
pperl -e 'print "hello\n"'
```

`-e` takes the Perl source as its argument. Multiple `-e` chunks concatenate; they behave as one program with semicolons between them.

```
pperl -e 'my $x = 41;' -e 'print $x + 1, "\n"'    # 42
```

Use single quotes around the program so the shell leaves `$_`, `$1`, `\n`, and backticks alone. See *gotchas* for what goes wrong with double quotes.

`-E` - `-e` plus the modern feature set

`-E` is `-e` with all optional features enabled: `say`, `state`, `fc`, the `//` defined-or operator, unicode string semantics under `use feature 'unicode_strings'`, and more.

```
pperl -E 'say "hello"'                    # instead of print "hello\
 ↪n"
pperl -E 'say "pi = " . (atan2(1,1)*4)'
```

Reach for `-E` when writing new one-liners. Reach for `-e` only when copy-paste compatibility with very old scripts matters.

`-n` - wrap an implicit read loop around the program

`-n` turns the program into the body of a loop over every line of every input file (or STDIN if none are given). The loop is exactly:

```
while (<>) {
    # your program
}
```

`$_` holds each input line, including its trailing newline. Nothing is printed automatically - you decide what reaches STDOUT.

```
# Print lines containing "error"
pperl -ne 'print if /error/' app.log
```

Equivalent long form, shown once so the mechanical expansion is clear:

```
while (<>) {
    print if /error/;
}
```

Every subsequent `-n` recipe in this guide relies on that expansion - assume it without re-stating.

`BEGIN { ... }` runs before the loop, `END { ... }` runs after. Both are ordinary Perl blocks and compose with `-n` cleanly.

```
# Count lines matching a pattern
pperl -ne '$n++ if /WARN/; END { print "$n\n" }' app.log
```

`-p` - like `-n`, but print `$_` after each iteration

```
pperl -pe 's/foo/bar/g' input.txt
```

Mechanical expansion:

```
while (<>) {
    # your program
} continue {
    print or die "-p destination: $!";
}
```

Reach for `-p` when the output is the (possibly modified) input. Reach for `-n` when the output is a summary, a filtered subset, or nothing at all.

`-l` - line-end handling

`-l` does two things at once:

1. On input, `chomp` each line as it is read under `-n` / `-p`. `$_` no longer ends in `"\n"`.
2. On output, set `$\` (the output record separator) so every `print` appends `"\n"`.

```
# Print just the first field of each tab-separated line
pperl -F'\t' -lane 'print $F[0]' table.tsv
```

Without `-l`, you would either write `print "$F[0]\n"` (the newline moves into the program) or produce glued output.

`-l` accepts an optional octal argument that sets `$\` to a different character: `-l012` keeps newline (the default), `-l0` sets `$\` to NUL, and so on. The common use is the no-argument form.

`-a` - autosplit into @F

Combined with `-n` or `-p`, `-a` splits `$_` on whitespace and puts the fields into `@F`. The default split is `split(' ', $_)` - it trims leading whitespace and treats any run of whitespace as one separator.

```
# Third column of every line
pperl -lane 'print $F[2]' data.txt

# Reverse column order
pperl -lane 'print "@{[reverse @F]}"' data.txt
```

`@F` is zero-indexed. `$F[-1]` is the last field. `scalar @F` is the field count.

`-F` - change the split pattern

`-F` sets the delimiter used by `-a`. The argument is a regex; a literal comma is `-F,`, a colon is `-F:`, a tab is `-F'\t'`.

```
# Second field of a colon-separated file
pperl -F: -lane 'print $F[1]' /etc/passwd

# Split on any run of commas or semicolons
pperl -F'[,;]+' -lane 'print scalar @F' mixed.csv
```

`-F` implies `-a`, so the split happens whether or not you spell `-a` out. The common idiom `-F: -lane` is colon-split, chomped, with `@F` available.

Quoting a tab requires a literal tab in the shell or the regex form:

```
pperl -F'\t' -lane 'print $F[0]' data.tsv       # regex form, always␣
↪safe
pperl -F'        ' -lane 'print $F[0]' data.tsv     # literal tab -␣
↪fragile
```

`-i` - edit files in place

`-i` rewrites the file `pperl` is reading from, replacing its contents with whatever `-p` prints. The whole file's output goes to a temporary file; when the loop finishes, that temporary replaces the original.

```
# Replace CRLF with LF in every .txt under cwd (no backup)
pperl -i -pe 's/\r\n/\n/g' *.txt

# Same, keeping a .bak copy beside each original
pperl -i.bak -pe 's/\r\n/\n/g' *.txt
```

Always use `-i.bak` (or another suffix) until you have verified the edit against a representative file. A bug in the program with bare `-i` destroys the input. See *gotchas*.

`-i` has no effect without `-p` (or explicit prints under `-n`): if nothing is printed, the replacement file is empty.

`-M` and `-m` - load modules before the program runs

`-Mmodule` is equivalent to `use module;` at the top of the program. `-Mmodule=sym1,sym2` is `use module qw(sym1 sym2);`.

```
pperl -MList::Util=sum -lane 'print sum @F' data.txt
pperl -MPOSIX=strftime -le 'print strftime("%F", localtime)'
pperl -MData::Dumper -e 'print Dumper({a=>1, b=>[2,3]})'
```

Lower-case `-m` is `use module ()` - the module is loaded but imports nothing; it is rarely needed on the command line.

Modules most common in one-liners: `List::Util` (sum/min/max/uniq/any), `POSIX` (strftime, mktime, fmod), `Data::Dumper` (inspect structures), `JSON::PP` (parse/emit JSON), `Text::CSV` (quoted CSV), `MIME::Base64`, `Digest::MD5`, `Digest::SHA`, `Socket`.

`-0` - set the input record separator

Perl normally reads a line at a time. `-0NNN` sets `$/` to the character with the given octal code, changing what "one record" means for `<>`, `<STDIN>`, and the `-n` / `-p` loop.

Common forms:

Form	What `$/` becomes	Effect
`-0`	`"\0"` (NUL)	Read NUL-delimited records (matches `find -print0`).
`-0777`	`undef`	Slurp the whole file as one record.
`-00`	`""`	Paragraph mode: blank lines separate records.
`-0NNN`	`chr(0NNN)`	Any single octal byte.

```
# Count files from find -print0
find . -type f -print0 | pperl -0 -ne '$n++; END { print "$n\n" }'

# Slurp, then run a cross-line regex
pperl -0777 -ne 'print "yes\n" if /BEGIN.*?END/s' text.txt

# Print every paragraph containing "TODO"
pperl -00 -ne 'print if /TODO/' notes.txt
```

Paragraph mode (`-00`) interacts with `-l`: without `-l`, the blank separator stays attached to each record; with `-l`, it is chomped.

`-C` - Unicode on stdio and/or `@ARGV`

`-C` enables Unicode handling on specified streams. The argument is either a number (bitmask) or a string of letters.

```
pperl -CSD -ne 'print' input.txt      # stdin, stdout, stderr all␣
↪UTF-8
pperl -CA  -e  'print $ARGV[0]' äöü   # @ARGV treated as UTF-8
pperl -CSA -nE 'say length'           # S + A
```

Letter codes: `I` = stdin, `O` = stdout, `E` = stderr, `S` = `I+O+E`, `A` = `@ARGV`, `D` = set default I/O layers (includes file opens).

For a fully UTF-8 session, `-CSD`. For one-off processing of named UTF-8 files, `-CSAD`. See *gotchas* for the symptoms that tell you `-C` is missing.

Reference table

Every switch, its mechanical expansion, and the chapter where it first shows up in a recipe.

Switch	Expands to	First used in
`-e CODE`	Run CODE as the program	*progression*
`-E CODE`	`-e` plus all current features enabled	*progression*
`-n`	`while (<>) { CODE }`	*progression*
`-p`	`while (<>) { CODE } continue { print }`	*progression*
`-l`	chomp input under `-n`/`-p`; set `$\ = "\n"` for output	*progression*
`-a`	Auto-split `$_` into `@F`	*progression*
`-F PAT`	Change `-a` split pattern to regex PAT	*progression*
`-i[EXT]`	Edit in place; with EXT, keep backup `original + EXT`	*progression*
`-M MOD`	`use MOD;` before program	*numeric*
`-M MOD=X`	`use MOD qw(X);`	*numeric*
`-0`	`$/ = "\0"` - NUL-delimited records	*progression*
`-00`	`$/ = ""` - paragraph mode	*progression*
`-0777`	`$/ = undef` - slurp entire file	*progression*
`-C`	Enable Unicode on stdio, `@ARGV`, and/or default layers	*gotchas*

Special variables the switches set up

Short catalogue; full definitions live in `perlvar`.

Vari-able	Role under `-n`/`-p`
`$_`	The current input line (chomped if `-l`, raw otherwise).
`$.`	Current input line number (not reset between files by default).
`$/`	Input record separator. `-0` family changes it.
`$\`	Output record separator appended by `print`. `-l` sets it to `"\n"`.
`$,`	Output field separator inserted between `print`'s arguments.
`$"`	Separator between array elements when interpolated in `"@array"`.
`@F`	Auto-split fields under `-a`.
`@ARGV`	Remaining command-line arguments; filenames are consumed as processed.
`$ARGV`	Name of the file currently being read (`"-"` means STDIN).

Find out more

- *progression* applies every switch in recipes of increasing complexity.
- `perlvar` lists every special variable pperl defines, not just the ones the switches touch.
- `perlop` covers `qr//`, the flip-flop operator, and the diamond operator used by the `-n` / `-p` loops.

Progressive recipes

This chapter works from trivial substitutions up through reductions, extraction, and structured output. Each section reuses the switches introduced in the previous one, so a reader going straight through builds up the idiom vocabulary without ever meeting a new flag mid-recipe.

Readers coming from `awk`/`sed` should recognise the first sections; the interesting material starts around *Reductions across lines* and *Context-windowed filters*.

Every recipe is copy-pasteable. When a sample input matters, it is shown inline so the expected output is unambiguous.

The baseline: `-e` and `-E`

The shortest one-liner. No I/O loop, no field splitting - just a program that runs once.

```
pperl -e 'print 2 ** 32, "\n"'                    # 4294967296
pperl -E 'say 2 ** 32'                            # same, with -E/
 ↪say
pperl -E 'say for 1 .. 5'                         # 1\n2\n3\n4\n5
```

The `,` in `print` separates arguments; the output field separator `$,` (default empty) goes between them.

Filtering: `-n` + `print if`

`-n` wraps the program in a read loop over every line of every input file (or STDIN). Nothing is printed unless the program says so.

```
# Sample input
$ printf 'gate\napple\nwhat\nkite\n' > words.txt

pperl -ne 'print if /at/' words.txt               # gate, what
pperl -ne 'print unless /e/' words.txt            # what
pperl -ne 'print if /^[aeiou]/' words.txt         # apple
```

The regex match is against `$_` by default; `/at/` is short for `$_ =~ m/at/`.

Multiple conditions

```
# Lines containing both "ERROR" and "timeout"
pperl -ne 'print if /ERROR/ && /timeout/' app.log

# Lines containing neither
pperl -ne 'print if !/ERROR/ && !/timeout/' app.log

# Events A, B, C appearing in order on one line
pperl -ne 'print if /A.*B.*C/' events.log
```

By line number

```
pperl -ne 'print if $. == 1'                    # head -n 1
pperl -ne 'print if $. <= 10'                   # head -n 10
pperl -ne 'print if 17 .. 30'                   # lines 17-30␣
↪(range op)
pperl -ne 'print if /START/ .. /END/'           # regex range,␣
↪inclusive
pperl -ne 'print unless $. % 2'                 # even lines
```

The range operator `..` in scalar context is a flip-flop: compared against `$.` when both endpoints are integers, against `$_` when they are patterns. See `perlop`.

By length

Use `-l` so the counted length excludes the trailing newline.

```
pperl -lne 'print if length >= 80' src.c        # long lines
pperl -lne 'print if length() < 5' words.txt    # short lines
```

Bare `length` with no argument applies to `$_`.

Substitution: `-p` + `s///`

`-p` is `-n` plus an implicit `print` after each iteration. Use it when the output is the (possibly modified) input.

```
# Replace every "foo" with "bar"
pperl -pe 's/foo/bar/g' input.txt

# First occurrence per line only
pperl -pe 's/foo/bar/' input.txt

# Conditional: replace only on lines that also match BAZ
pperl -pe 's/foo/bar/g if /BAZ/' input.txt
```

Case transformations

```
pperl -nle 'print uc'                                    # to uppercase
pperl -nle 'print lc'                                    # to lowercase
pperl -nle 'print ucfirst lc'                            # Sentence case
pperl -ple 's/(\w+)/\u$1/g'                              # Title Case Per␣
↪Word
```

`uc`, `lc`, and `ucfirst` with no argument operate on `$_`.

Whitespace normalisation

```
pperl -ple 's/^\s+//'                                    # trim left
pperl -ple 's/\s+$//'                                    # trim right
pperl -ple 's/^\s+|\s+$//g'                              # trim both
pperl -ple 's/\s+/ /g'                                   # collapse runs
pperl -ne  'print if /\S/'                               # drop blank␣
↪lines
```

Line-ending conversion

```
pperl -pe 's/\r\n/\n/g' win.txt                          # DOS  → UNIX
pperl -pe 's/(?<!\r)\n/\r\n/g' unix.txt                  # UNIX → DOS
```

The lookbehind `(?<!\r)` avoids producing `\r\r\n` when the input is already DOS-encoded. See *regex-recipes* for lookarounds in depth.

Field-oriented work: `-a` and `-F`

`-a` auto-splits each line into `@F`. `-F` changes the delimiter.

```
$ cat table.txt
brown bread mat hair 42
blue cake mug shirt -7
yellow banana window shoes 3.14

# Second field
pperl -lane 'print $F[1]' table.txt
# bread, cake, banana

# Last field
pperl -lane 'print $F[-1]' table.txt
# 42, -7, 3.14

# Count fields per line
pperl -lane 'print scalar @F' table.txt
# 5, 5, 5

# Lines whose last field is negative
```

(continues on next page)

(continued from previous page)

```
pperl -lane 'print if $F[-1] < 0' table.txt
# blue cake mug shirt -7
```

Custom delimiter

```
# /etc/passwd: print username (field 1) and shell (last field)
pperl -F: -lane 'print "$F[0] $F[-1]"' /etc/passwd

# Tab-separated
pperl -F'\t' -lane 'print $F[2]' metrics.tsv

# Comma or semicolon, one or more
pperl -F'[,;]+' -lane 'print scalar @F' mixed.csv
```

Proper CSV (quoted fields with embedded commas) is not a job for `-F`. Use `-MText::CSV` - see *numeric* for the pattern.

Rejoining fields

Interpolating `@F` inside double quotes uses `$"` (default: one space) as the separator.

```
# Reverse field order, space-separated
pperl -lane 'print "@{[reverse @F]}"' table.txt

# Reverse field order, colon-separated
pperl -F: -lane 'local $" = ":"; print "@{[reverse @F]}"' /etc/
 ↪passwd
```

`@{[ EXPR ]}` is the array-ref-then-deref idiom that lets arbitrary expressions be interpolated into a double-quoted string.

Reductions across lines

The move from per-line filters to cross-line computation.

Counts

```
# wc -l
pperl -lne 'END { print $. }' big.txt

# grep -c pattern
pperl -lne '$n++ if /pattern/; END { print $n + 0 }' file

# Count blank lines
pperl -lne '$n++ if /^$/; END { print $n + 0 }' file

# Total fields across the file
pperl -lane '$t += @F; END { print $t }' table.txt
```

The `$n + 0` idiom prints `0` rather than empty when the counter was never incremented.

Sums, minima, maxima

Use `List::Util` via `-M`.

```
# Per line
pperl -MList::Util=sum -lane 'print sum @F' data.txt
pperl -MList::Util=min -lane 'print min @F' data.txt
pperl -MList::Util=max -lane 'print max @F' data.txt

# Across all lines
pperl -MList::Util=sum -lane '$s += sum @F; END { print $s }' data.
↪txt
pperl -MList::Util=max -alne '$m = max($m // (), @F); END { print $m␣
↪}' data.txt
```

`$m // ()` returns an empty list when `$m` is undefined, so the first iteration does not pass `undef` to `max`. See *numeric* for the full arithmetic chapter.

Longest and shortest line

```
pperl -ne '$l = $_ if length > length $l; END { print $l }' file
pperl -ne '$s = $_ if $. == 1 || length < length $s; END { print $s␣
↪}' file
```

Unique lines (first sighting only)

```
pperl -ne 'print unless $seen{$_}++' file

# Or the List::Util one-liner (slurps):
pperl -MList::Util=uniq -e 'print uniq <>' file
```

The first form streams - memory scales with distinct lines, not total lines. The second slurps everything into memory, then emits. Prefer the first for large inputs.

Duplicates only

```
# Print each duplicate line once, when first seen for the second␣
↪time
pperl -ne 'print if ++$seen{$_} == 2' file
```

Context-windowed filters

Printing the line *around* a match, not just the match itself.

```
# Line immediately before each match (grep -B 1, with caveats)
pperl -ne '/pattern/ && $prev && print $prev; $prev = $_' file
```

(continues on next page)

(continued from previous page)

```
# Line immediately after each match (grep -A 1)
pperl -ne 'print if $p; $p = /pattern/' file

# tail -n 10
pperl -ne 'push @a, $_; shift @a if @a > 10; END { print @a }' file
```

For rich `-A` / `-B` / `-C` behaviour, `grep` is the right tool. One-liners fit when the window rule is unusual (e.g. "print the line matching X, plus every line until the next blank line").

```
# Print from /SECTION/ to the next blank line
pperl -ne 'print if /SECTION/ .. /^$/' file
```

Record mode: `-0`, `-00`, `-0777`

Switch away from line-based reading when the natural record is not a line.

Slurp with `-0777`

```
# Single-shot regex across the whole file
pperl -0777 -ne 'print "match\n" if /BEGIN.*?END/s' text.txt

# Replace only the first occurrence across the file
pperl -0777 -i.bak -pe 's/TODO/DONE/' notes.txt
```

`-0777` sets `$/ = undef`. The entire file comes in as one `$_`. The `/s` modifier makes `.` match newlines. See *regex-recipes* for the full multi-line story.

Paragraph mode with `-00`

`-00` sets `$/ = ""`. Each read returns everything up to the next blank-line delimiter.

```
# Paragraphs containing "TODO"
pperl -00 -ne 'print if /TODO/' notes.md

# Reverse paragraph order
pperl -00 -e 'print reverse <>' notes.md
```

NUL-delimited records with `-0`

```
# Count files produced by find -print0
find . -type f -print0 | pperl -0 -ne '$n++; END { print "$n\n" }'

# Convert NUL-delimited stream to newline-delimited
find . -type f -print0 | pperl -0 -pe 's/\0/\n/'
```

In-place editing with `-i`

`-i` overwrites each input file with what `-p` prints. `-i.bak` keeps a backup.

```
# Lowercase every occurrence of HOSTNAME in every .conf (keep␣
↪backup)
pperl -i.bak -pe 's/HOSTNAME/\L$&/g' *.conf

# Delete every line matching DEPRECATED (no backup - only do this
# after you've validated the pattern on one file)
pperl -i -ne 'print unless /DEPRECATED/' file.txt
```

The backup files pile up. Clean them up with `rm *.bak` once verified. See *gotchas* for the classic in-place traps.

Structured output

CSV-like (lightweight - commas are not quoted)

```
# Rebuild a TSV as CSV (naive, assumes no commas in fields)
pperl -F'\t' -lane 'print join ",", @F' data.tsv

# Add a column of line numbers
pperl -lane 'print join "\t", $., @F' data.tsv
```

JSON (stdlib)

```
# Emit one JSON object per line (no commas in values assumed)
pperl -MJSON::PP -F: -lane '
    print JSON::PP->new->encode({
        user => $F[0], uid => 0+$F[2], shell => $F[-1]
    })
' /etc/passwd
```

Multi-line one-liners like this are where an *alias* starts to pay for itself.

Numbered input

```
pperl -pe '$_ = "$. $_"'                          # 1 gate, 2 apple,␣
↪...
pperl -pe 's/^/sprintf "%5d  ", $./e'             # padded width
```

The `/e` modifier on `s///` treats the replacement as Perl code. See *regex-recipes*.

Find out more

- *regex-recipes* - the regex-dependent one-liners: captures, lookaround, named groups, `/e`, `/r`.
- *numeric* - arithmetic, statistics, randoms, date math.

- *aliases* - turn the recipes you use weekly into shell functions.
- *gotchas* - quoting, encoding, `-i` backups, record-separator surprises.
- `perlop` - the flip-flop operator, the diamond operator, `s///`, `tr///`, quoting forms.

Regex-driven one-liners

The previous chapter kept regexps at the level `sed` or `awk` users already know: literal patterns inside `m//` and `s///`. This chapter covers one-liners whose shape depends on a regex feature beyond plain matching - extractions, captures, lookarounds, code-in-replacement, and non-destructive rewrites.

Readers who already know the constructs (`(?:...)`, `(?<name>...)`, `(?=...)`, the `/e` and `/r` modifiers) can skim for the command-line framing. Readers who know what `/g` does but have never written `/e` will get the most out of reading straight through.

For the regex language itself, see the *regular expressions guide*.

Extracting all matches

`/g` in list context returns every match, not just the first.

```
# Every integer in the input
pperl -ne 'print "$_\n" for /-?\d+/g' file.txt

# Every IPv4-shaped substring
pperl -lne 'print for /\b(?:\d{1,3}\.){3}\d{1,3}\b/g' access.log

# Every HTTP header value (naive: no folded headers)
pperl -nE 'say $1 if /^[\w-]+:\s*(.+)$/' headers.txt
```

`say` (enabled by `-E`) appends a newline, saving one `"\n"` and keeping the program short.

Non-greedy quantifiers

`*?`, `+?`, `??`, `{n,m}?` - match as few characters as possible.

```
# Everything between the first < and the next > (tag-by-tag)
pperl -lne 'print for /<(.+?)>/g' markup.html

# Every quoted string (double quotes, no escape handling)
pperl -lne 'print for /"([^"]*)"/g' config.txt
```

The negated character class `[^"]*` is usually a faster and safer non-greedy than `.*?` when you know what must not appear inside.

Captures in substitution

The right-hand side of `s///` can reference left-hand captures.

```
# Swap every "word1 word2" pair
pperl -pe 's/(\w+)\s+(\w+)/$2 $1/g' file.txt

# Quote every unquoted key in key=value pairs
pperl -pe 's/(\w+)=/$1="/g; s/=([^"\n]+)/"$1"/g' config.txt

# Increase every trailing number by one
pperl -pe 's/(\d+)$/$1 + 1/e' file.txt
```

`/e` - replacement is Perl code

The `/e` modifier treats the replacement as code; its return value is what gets substituted in.

```
# Number every word in the file (global counter)
pperl -pe 's/(\w+)/++$i . ".$1"/ge' file.txt

# Number words per line (counter reset each line)
pperl -pe '$i = 0; s/(\w+)/++$i . ".$1"/ge' file.txt

# Wrap every integer in base 16
pperl -pe 's/\d+/sprintf "0x%x", $&/ge' file.txt

# Convert Celsius-labelled numbers to Fahrenheit
pperl -pe 's/(\d+)C\b/sprintf "%.1fF", $1 * 9 / 5 + 32/ge' weather.
↪txt
```

`/ge` together: evaluate the right-hand side as code, do it for every match. `$&` is the whole matched text; `$1`, `$2`, ... are captures.

`/ee` evaluates twice - the first `e` produces a string, the second evaluates that string. Useful when the replacement string is itself dynamic code read from input; needed rarely, a hazard often.

`/r` - non-destructive substitution

`/r` returns the modified string instead of rewriting `$_`. The original stays intact.

```
# Print filename and its .bak variant side by side
ls *.txt | pperl -pe '$_ = $_ . ($_ =~ s/$/.bak/r)'

# Compute a transformed version without touching $_
pperl -ne 'my $upper = $_ =~ tr/a-z/A-Z/r; print $_, $upper' file.
↪txt
```

Reach for `/r` when a single line needs to be printed in two forms, or when the transformation is conditional and you want the untouched line back in the failure branch.

Named captures

`(?<name>...)` captures into `$+{name}` and into `%+`.

```
# Parse combined-log-format access lines
pperl -nE '
    if (/^(?<ip>\S+).*?"(?<method>\S+) (?<path>\S+)/) {
        say "$+{ip} $+{method} $+{path}";
    }
' access.log
```

Named captures pay for themselves once the pattern has more than two groups, or when the same group needs to be referenced both in the pattern and in the replacement code.

Lookahead and lookbehind

Zero-width assertions: match a position where the surrounding text satisfies a condition, without consuming the condition itself.

```
# Digits preceded by $ (e.g. "$42" → 42, ignore "42")
pperl -lne 'print for /(?<=\$)\d+/g' prices.txt

# Words NOT followed by "!"
pperl -lne 'print for /\b\w+\b(?!!)/g' shouts.txt

# Insert CRLF only where LF is not already preceded by CR
pperl -pe 's/(?<!\r)\n/\r\n/g' unix.txt
```

Common pairs:

Lookaround	Meaning
`(?=X)`	Next characters match `X`
`(?!X)`	Next characters do not match `X`
`(?<=X)`	Preceding characters match `X`
`(?<!X)`	Preceding characters do not match `X`

Lookbehind in pperl supports variable-length patterns; see *regular expressions: anchors and assertions*.

Multi-line matches across slurped input

`-0777` slurps the entire file into `$_`. The regex modifiers that matter in that mode:

- `/s` - `.` matches `\n`.
- `/m` - `^` and `$` match at every line start/end, not just the start/end of the whole string.

```
# Every BEGIN...END block (non-greedy), each on one output line
pperl -0777 -lne '
    while (/BEGIN(.*?)END/sg) {
```

(continues on next page)

(continued from previous page)

```
        (my $body = $1) =~ s/\n/ /g;
        print $body;
    }
' text.txt

# Paragraphs where every line starts with ">"
pperl -00 -ne 'print if /\A(?:>.*\n?)+\z/m' mail.txt
```

`\A` and `\z` anchor the whole string start and end, unaffected by `/m`. Use them when `/m` is on but you really do mean the whole string.

Transliteration with `tr`

`tr` (also spelt `y`) is character-by-character substitution. Not a regex, but a close neighbour - and the right tool for character-set jobs.

```
pperl -pe 'tr/A-Za-z/N-ZA-Mn-za-m/'                    # ROT13
pperl -pe 'tr/A-Za-z/a-zA-Z/'                          # swap case
pperl -pe 'tr/a-z//d'                                  # delete lowercase␣
↪letters
pperl -pe 'tr/ \t/ /s'                                 # squeeze␣
↪whitespace runs to one space

# Count commas on each line (tr returns the count)
pperl -ne 'print tr/,//, "\n"' data.csv
```

The `d` flag deletes characters in the left set that have no counterpart on the right. The `s` flag squeezes runs. The `c` flag complements the left set (everything NOT listed).

Parsing delimited substrings

CSV with quoted commas

A quick-and-dirty split that honours double-quoted fields:

```
pperl -ne '
    my @F = /"([^"]*)"|([^,]+)/g;
    @F = grep defined, @F;
    print join("|", @F), "\n";
' quoted.csv
```

Correct and fast enough for a one-off. For anything that must handle embedded quotes (`"he said ""hi"""`), use `Text::CSV`; see *numeric*.

URL query strings

```
# Each key=value pair on its own line
pperl -lne 'print for /([^?&=]+)=([^&]*)/g' urls.txt
```

Practical extraction one-liners

Words by length

```
# Words of exactly 5 characters
pperl -lne 'print for /\b\w{5}\b/g' file.txt
```

Numbers with units

```
pperl -lne 'print for /\b\d+(?:\.\d+)?\s*(?:ms|s|kb|mb|gb)\b/gi'␣
↪timings.txt
```

HTTP User-Agent from a request log

```
pperl -nE 'say $1 if /^User-Agent: (.+)$/' request.log
```

Balanced braces (as many as pperl's regex engine supports)

pperl's regex engine supports recursion via `(?R)` and `(?1)`, so single-level brace balancing is a one-liner:

```
pperl -0777 -nE 'say $& while /\{(?:[^{}]|(?R))*\}/g' code.txt
```

For deeply nested structures, reach for a parser. Regex is the right tool up to the complexity where it stops being one.

Find out more

- *regular expressions guide* - the language underneath every recipe here, including modifiers, performance, and Unicode.
- `m` - the match operator.
- `s` - substitution, the full set of modifiers.
- `tr` - transliteration reference.
- *progression* - simpler `s///` recipes without captures or code.
- `perlop` - the `qr//` form for precompiling patterns referenced by name inside a one-liner.

Numeric and date one-liners

pperl as a calculator, stats tool, and date-arithmetic engine. This chapter collects the recipes that turn numeric columns, timestamps, and IP addresses into useful output without a script file.

The assumed baseline is a reader who has read *switches* and *progression*. The `-lane` / `-MList::Util=...` idioms used below were introduced there.

Where this chapter reaches for performance: large numeric reductions are one of the cases where pperl's JIT earns its cost. A `for (1 .. 100_000_000) { $s += $_ }` accumulation runs as compiled machine code after the interpreter warms up. You do not opt in - it triggers automatically on arithmetic-heavy numeric loops.

Plain-arithmetic one-liners

```
pperl -E 'say 2 ** 64'                                  # 1.
↪84467440737096e+19
pperl -E 'say 2 ** 64 - 1'                              # rounds (float)
pperl -MMath::BigInt -E 'say Math::BigInt->new(2)->bpow(64)'
                                                        #␣
↪18446744073709551616

pperl -E 'say 7 / 3'                                    # 2.33333333333333
pperl -E 'say int(7 / 3)'                               # 2
pperl -E 'say 7 % 3'                                    # 1
pperl -E 'say sqrt(2)'                                  # 1.4142135623731
pperl -E 'say atan2(1, 1) * 4'                          # 3.14159265358979␣
↪(pi)
```

`sprintf` controls output width and precision:

```
pperl -E 'printf "%.4f\n", 22/7'                        # 3.1429
pperl -E 'printf "%10.2f\n", 42.5'                      #      42.50
pperl -E 'printf "%08d\n", 42'                          # 00000042
pperl -E 'printf "%x %o %b\n", 255, 255, 255'           # ff 377 11111111
```

Sums, minima, maxima, averages

`List::Util` ships with Perl 5 - no install needed.

Per line

```
pperl -MList::Util=sum -lane 'print sum @F'                 data.txt
pperl -MList::Util=min -lane 'print min @F'                 data.txt
pperl -MList::Util=max -lane 'print max @F'                 data.txt

# Per-line mean
pperl -MList::Util=sum -lane 'print sum(@F) / @F'           data.txt
```

Across all lines

```
# Total sum
pperl -MList::Util=sum -lane '$s += sum @F; END { print $s }' data.
↪txt

# Running min / max using // (defined-or)
```

(continues on next page)

(continued from previous page)

```
pperl -MList::Util=min -alne '$m = min($m // (), @F); END { print $m
 ↪}' data.txt
pperl -MList::Util=max -alne '$m = max($m // (), @F); END { print $m
 ↪}' data.txt

# Mean of one column (third), ignoring blank lines
pperl -lane 'next unless /\S/; $s += $F[2]; $n++;
             END { printf "%.4f\n", $s / $n }' data.txt
```

`$m // ()` returns an empty list when `$m` is undefined, so the first iteration does not taint `min`/`max` with an undef. See `perlop` for `//`.

Sum a single column

```
# Sum of column 3 (zero-indexed: $F[2])
pperl -lane '$s += $F[2]; END { print $s }' data.txt

# Negative or zero entries only
pperl -lane '$s += $F[2] if $F[2] <= 0; END { print $s }' data.txt
```

Standard deviation

```
pperl -lane '
    $n++; $s += $_; $s2 += $_**2 for @F;
    END {
        my $n2 = $n * @F;    # total count (assumes rectangular
 ↪data)
        my $mean = $s / $n2;
        printf "mean=%.4f sd=%.4f\n",
            $mean, sqrt($s2 / $n2 - $mean ** 2);
    }
' data.txt
```

For anything beyond mean+sd, reach for `Statistics::Descriptive`:

```
pperl -MStatistics::Descriptive -lane '
    BEGIN { our $stat = Statistics::Descriptive::Full->new }
    $stat->add_data(@F);
    END { printf "median=%.2f mean=%.2f sd=%.2f\n",
          $stat->median, $stat->mean, $stat->standard_deviation }
' data.txt
```

Random numbers and sampling

```
# 10 random integers in [5, 15)
pperl -E 'say join ",", map { int(rand(10)) + 5 } 1 .. 10'
```

(continues on next page)

(continued from previous page)

```
# 8-character random lowercase password
pperl -E 'say join "", map { ("a".."z")[rand 26] } 1 .. 8'

# 8-character random alphanumeric
pperl -E 'say join "", map { ("a".."z", 0..9)[rand 36] } 1 .. 8'

# Random UUID (using Data::UUID)
pperl -MData::UUID -E 'say Data::UUID->new->create_str'

# Shuffle the fields of each line
pperl -MList::Util=shuffle -alne 'print join " ", shuffle @F' data.
↪txt

# Pick one random line from a file (reservoir sample, size 1)
pperl -ne 'rand($.) < 1 && ($pick = $_); END { print $pick }' file.
↪txt
```

The reservoir trick is the right answer when the file is too large to slurp or its line count is unknown - every line has an equal probability of being the one printed, with exactly one pass.

Factorials, GCD, LCM, primes

```
# Factorial
pperl -E '$f = 1; $f *= $_ for 1 .. 20; say $f'
pperl -MMath::BigInt -E 'say Math::BigInt->new(100)->bfac'

# GCD / LCM (built into Math::BigInt under bgcd / blcm)
pperl -MMath::BigInt=bgcd -anle 'print bgcd(@F)' nums.txt
pperl -MMath::BigInt=blcm -anle 'print blcm(@F)' nums.txt

# Euclid's algorithm for two numbers
pperl -le '$n=20; $m=35; ($m,$n) = ($n, $m % $n) while $n; print $m'
```

Primality:

```
# Regex-based primality test (Abigail). Input: one integer per line.
pperl -lne '(1 x $_) !~ /^1?$|^(11+?)\1+$/ && print "$_ prime"'␣
↪nums.txt

# Reliable for large numbers: use Math::Prime::Util
pperl -MMath::Prime::Util=is_prime -lne '
    print "$_ prime" if is_prime($_)
' nums.txt
```

The regex trick is a curiosity; `Math::Prime::Util` is the serious answer.

IP ↔ integer

```
# Dotted-quad → 32-bit integer
pperl -MSocket -le 'print unpack "N", inet_aton "127.0.0.1"'
# 2130706433

# Integer → dotted-quad
pperl -MSocket -le 'print inet_ntoa pack "N", 2130706433'
# 127.0.0.1
```

`unpack "N"` reads four bytes as a big-endian 32-bit unsigned integer. The matching `pack "N"` writes one.

Date and time

Today, now, epoch

```
pperl -E 'say time'                                    # epoch seconds
pperl -E 'say scalar localtime'                        # Mon Apr 22␣
↪14:31:07 2026
pperl -E 'say scalar gmtime'                           # same, UTC
pperl -MPOSIX=strftime -E 'say strftime "%F %T", localtime'
                                                       # 2026-04-22␣
↪14:31:07
```

`strftime` is the clean way to format; `localtime` in list context returns `(sec, min, hour, mday, mon, year, wday, yday, isdst)`.

Date arithmetic

`POSIX::mktime` normalises out-of-range fields (month 13 becomes month 1 of next year, day 32 becomes day 1 of next month, etc.), so date arithmetic means "subtract N from a field, mktime, strftime".

```
# Yesterday's date
pperl -MPOSIX=strftime -MPOSIX=mktime -E '
    my @t = localtime;
    $t[3] -= 1;
    say strftime "%F", localtime mktime @t
'

# 30 days ago
pperl -MPOSIX=strftime -MPOSIX=mktime -E '
    my @t = localtime;
    $t[3] -= 30;
    say strftime "%F", localtime mktime @t
'

# First Monday of next month
pperl -MPOSIX=strftime -MPOSIX=mktime -E '
```

(continues on next page)

(continued from previous page)

```
    my @t = localtime;
    $t[3] = 1; $t[4]++;             # first of next month
    my $epoch = mktime @t;
    my @nm = localtime $epoch;
    $epoch += ((8 - $nm[6]) % 7 || 7) * 86400 if $nm[6] != 1;
    # ^ if not already Monday, advance to next Monday
    say strftime "%F", localtime $epoch
'
```

For anything more elaborate (time zones, business-day math, ISO weeks), reach for `DateTime`:

```
pperl -MDateTime -E '
    my $d = DateTime->now->subtract(days => 30);
    say $d->strftime("%F %T %Z")
'
```

Timestamps in logs

```
# Sum the durations (last field, milliseconds) of lines matching
 ↪"slow"
pperl -lane '$s += $F[-1] if /slow/; END { print $s, " ms total" }'␣
 ↪app.log

# Count events per hour (assumes ISO timestamp first field)
pperl -lane '
    (my $hour = $F[0]) =~ s/:\d{2}:\d{2}.*//;
    $h{$hour}++;
    END { print "$_ $h{$_}" for sort keys %h }
' events.log
```

CSV and TSV

TSV with no embedded tabs: `-F'\t'` is fine. Real CSV needs `Text::CSV`:

```
# Sum the third column of a properly quoted CSV
pperl -MText::CSV -E '
    my $csv = Text::CSV->new({ binary => 1 });
    my $total = 0;
    while (my $row = $csv->getline(*STDIN)) {
        $total += $row->[2];
    }
    say $total
' < data.csv

# Re-emit a CSV with only selected columns
pperl -MText::CSV -E '
    my $csv = Text::CSV->new({ binary => 1, eol => "\n" });
```

(continues on next page)

(continued from previous page)

```
    while (my $row = $csv->getline(*STDIN)) {
        $csv->print(*STDOUT, [ @{$row}[0, 2, 4] ]);
    }
' < data.csv > trimmed.csv
```

The `binary => 1` option keeps bytes intact; the `eol => "\n"` option makes `print` append newlines.

JSON

`JSON::PP` ships with Perl; `Cpanel::JSON::XS` is faster if it is installed.

```
# Extract the "name" field from each line of NDJSON
pperl -MJSON::PP -lne '
    my $o = JSON::PP->new->decode($_);
    print $o->{name};
' events.ndjson

# Convert each line of TSV to JSON
pperl -MJSON::PP -F'\t' -lane '
    print JSON::PP->new->encode({ id => $F[0], name => $F[1] })
' data.tsv

# Pretty-print a single JSON document
pperl -MJSON::PP -0777 -ne '
    print JSON::PP->new->pretty->encode(JSON::PP->new->decode($_))
' small.json
```

For large JSON, streaming parsers (`JSON::Streaming::Reader`) beat slurping; single-document files under a few hundred megabytes are fine with the above.

Base64, URL encoding, hex

```
# Base64 encode/decode (of a whole file)
pperl -MMIME::Base64 -0777 -ne 'print encode_base64($_)' file
pperl -MMIME::Base64 -0777 -ne 'print decode_base64($_)' file.b64

# URL encode/decode
pperl -MURI::Escape -lne 'print uri_escape($_)'   < urls.txt
pperl -MURI::Escape -lne 'print uri_unescape($_)' < encoded.txt

# Hex ↔ integer
pperl -E 'printf "%x\n", 255'                     # ff
pperl -E 'say hex "ff"'                           # 255

# Hex dump of a file (like xxd -p)
pperl -0777 -ne 'print unpack "H*", $_' file.bin
```

(continues on next page)

(continued from previous page)

```
# Reverse: hex dump → bytes
pperl -0777 -ne 'chomp; print pack "H*", $_' file.hex > file.bin
```

Find out more

- *aliases* - wrap the recipes you use weekly into shell functions with positional arguments.
- *progression* - the reduction idioms these numeric recipes build on.
- `sprintf`, `printf` - formatted output.
- `time`, `localtime`, `gmtime` - the built-ins underneath the date recipes.

From one-liner to shell alias

A one-liner is disposable by design. The moment you type the same one twice, something has earned a place in your shell config. This chapter covers the mechanics of lifting a pperl recipe into a bash or zsh shell function, how to parametrise it, and when to stop and write a real script instead.

Assumed baseline: you have a bash or zsh shell, a `~/.bashrc` or `~/.zshrc` (or an `rc.d` / `functions` directory your shell sources), and the patience to close and reopen one terminal after editing it.

Alias versus function

Bash and zsh have two mechanisms. They are not interchangeable.

- `alias` is textual substitution. It runs at the start of a command line, replacing the alias name with its value before the shell re-parses. No arguments, no pipelining quirks beyond what substitution gives you.
- A shell function is a named piece of shell code. It takes positional arguments (`$1`, `$2`, `$@`), reads stdin like any other command, and can pipe into and out of other commands.

Rule of thumb. If the recipe takes no arguments, alias is enough. If it needs any argument (a column number, a regex, a filename), use a function.

Alias - the zero-argument recipe

```
# In ~/.bashrc
alias sum-stdin='pperl -MList::Util=sum -lane '"'"'print sum @F'"'"'
alias count-lines='pperl -lne '"'"'END { print $. }'"'"'
```

The quoting is ugly because a bash single-quoted string cannot contain a single quote; the idiom `'"'"'` closes-and-reopens. This is the main reason to prefer functions: they let you quote the Perl once, cleanly.

Function - the common case

```
# In ~/.bashrc
sum-stdin() { pperl -MList::Util=sum -lane 'print sum @F'; }
count-lines() { pperl -lne 'END { print $. }'; }
```

Usage is the same:

```
$ printf '1 2 3\n4 5 6\n' | sum-stdin
6
15
```

Parametrised functions

The shell interpolates `$1`, `$@`, `"$1"` into the function body before executing it. Inside the Perl program, the shell-level `$1` appears as a literal value - Perl's own `$1` (the regex capture) is untouched by shell interpolation only if you quote with single quotes.

The mixture is solvable with care. Two idioms cover almost every case.

Idiom 1: shell-interpolated column / value

Use double quotes around the Perl program so the shell expands `$1`, then escape Perl's own `$` with `\$`.

```
# Sum column N of stdin: sumcol N
sumcol() {
    pperl -MList::Util=sum -lane "\$s += \$F[$1]; END { print \$s }"
}

# Usage
$ cat data.tsv
10 apple 3
15 pear  7
20 plum  2
$ sumcol 2 < data.tsv
12
```

Every `\$` is a literal dollar sign for Perl; every `$1` (unescaped) is the first positional argument expanded by the shell.

Idiom 2: program in single quotes, data via environment

Single-quoted Perl stays readable, and values reach the program via `%ENV`.

```
# Grep with a pperl regex: pgrep PATTERN [FILE...]
pgrep() {
    local pat=$1
    shift
    pat="$pat" pperl -ne 'print if /$ENV{pat}/' "$@"
}
```

(continues on next page)

(continued from previous page)

```
$ pgrep '\berror\b' app.log
```

This keeps $1 (Perl capture) usable and avoids the backslash cascade.

Idiom 3: mixing both

```
# Replace column N's commas with semicolons, in place: csv-fix N␣
↪FILE...
csv-fix() {
    local col=$1
    shift
    col="$col" pperl -i.bak -F'\t' -lane '
        my $c = $ENV{col};
        $F[$c] =~ tr/,/;/;
        print join("\t", @F);
    ' "$@"
}
```

The column number arrives in `$ENV{col}` - no escaping dance. The file list propagates via `"$@"` so spaces in filenames survive.

Practical aliases worth keeping

Annotated list of recipes that earn their place in a shell config for anyone who processes text at the command line daily.

`sumcol N` - sum column N (whitespace-separated)

```
sumcol() {
    pperl -MList::Util=sum -lane "\$s += \$F[$1]; END { print \$s }"
}
```

`sumcolf N FS` - sum column N with custom field separator

```
sumcolf() {
    local col=$1 fs=$2
    col="$col" pperl -MList::Util=sum -F"$fs" -lane '
        $s += $F[$ENV{col}]; END { print $s }
    '
}

$ sumcolf 3 : < /etc/passwd    # sum of UIDs
```

`colstats N` - min / max / mean / count for column N

```
colstats() {
    col="$1" pperl -MList::Util=qw(min max sum) -lane '
        my $c = $ENV{col};
        push @v, $F[$c];
        END {
            printf "n=%d min=%g max=%g mean=%.4f\n",
                scalar @v, min(@v), max(@v), sum(@v) / @v
        }
    '
}
```

`uniq-stream` - streaming unique, keeping first sighting

```
uniq-stream() { pperl -ne 'print unless $seen{$_}++'; }
```

Unlike `sort -u`, order is preserved. Unlike `uniq`, input need not be sorted.

`extract PAT` - print every match of a regex, one per line

```
extract() { pat="$1" pperl -lne 'print for /$ENV{pat}/g'; }

$ extract '\b\d{3}-\d{4}\b' < phonebook.txt
555-1212
555-9876
```

`between A B` - print everything between regex A and regex B, inclusive

```
between() {
    a="$1" b="$2" pperl -ne 'print if /$ENV{a}/ .. /$ENV{b}/'
}

$ between 'BEGIN cert' 'END cert' < bundle.pem
```

`json-pretty` - pretty-print a single JSON document

```
json-pretty() {
    pperl -MJSON::PP -0777 -ne '
        print JSON::PP->new->pretty->canonical->encode(
            JSON::PP->new->decode($_)
        )
    '
}
```

`passwd-json` - /etc/passwd as NDJSON

```
passwd-json() {
    pperl -MJSON::PP -F: -lane '
        print JSON::PP->new->encode({
            user => $F[0], uid => 0+$F[2], gid => 0+$F[3],
            home => $F[5], shell => $F[-1]
        })
    ' < /etc/passwd
}
```

`words-by-freq` - count word frequencies, sorted

```
words-by-freq() {
    pperl -ne '
        $w{lc $_}++ for /\b[[:alpha:]]+\b/g;
        END { print "$w{$_} $_\n" for sort { $w{$b} <=> $w{$a} }␣
↪keys %w }
    '
}

$ words-by-freq < essay.txt | head -20
```

`strip-comments LANG` - strip comments for a few common languages

```
strip-comments() {
    case "$1" in
        sh|py|rb|pl) pperl -pe 's/(?<!\\)#.*//' ;;
        c|cpp|rs|java|js) pperl -0777 -pe 's{//[^\n]*|/\*.*?\*/}{}gs
↪' ;;
        sql|lua) pperl -pe 's/--[^\n]*//' ;;
        *) echo "unknown language: $1" >&2; return 1 ;;
    esac
}
```

When to stop and write a script

The one-liner → alias transition ends when:

- The program needs an `if` / `else` branch longer than a single `? :` conditional.
- There are more than two or three `-M` modules.
- You find yourself debugging via trial-and-error on the command line.
- The program is the only alias in your config that gets a comment above it explaining what it does.
- A colleague is going to run it.

Move to a script file. Drop the switches into the shebang line and let the program read naturally:

```
#!/usr/bin/env pperl
use strict;
use warnings;
use List::Util qw(sum);

my $col = shift @ARGV // die "usage: $0 COLUMN [FILE...]\n";
my $total = 0;
while (<>) {
    my @f = split;
    $total += $f[$col];
}
print "$total\n";
```

Give it a sensible name, `chmod +x`, and move on. The one-liner did its job: it was the shortest path from "I have an idea" to "I know this works"; the script is the path from "I know this works" to "my colleague can read it."

Find out more

- *progression* - the recipes these aliases wrap.
- *gotchas* - quoting inside shell functions and aliases.
- *numeric* - more candidates for the alias list, especially the date-arithmetic ones.

Traps and surprises

The ways pperl one-liners silently do something other than what the shell user meant. Each section names a trap, shows its symptom, and gives the fix. Read straight through once to prime the pattern-match; come back when a recipe is misbehaving.

Quoting hell

The single biggest source of wasted time in one-liner work.

Single quotes are your friend

Under bash and zsh, text inside single quotes is not interpolated. No `$VAR` expansion, no backtick execution, no `!` history substitution (zsh doesn't do `!`; bash does, and it bites). Use single quotes for Perl programs unless you have a reason not to.

```
# Correct
pperl -ne 'print if /$var/' file              # Perl sees literal
 ↪$var

# Trap
pperl -ne "print if /$var/" file              # shell expands $var␣
 ↪before pperl sees it
```

The ! surprise in interactive bash

```
$ pperl -E 'say "hello!"'
bash: !": event not found
```

Bash's history expansion fires on unquoted `!` (and some escaped forms). Workarounds:

- `set +H` in `~/.bashrc` disables history expansion.
- Escape the `!` when writing the command: `"hello\!"`.
- Keep the shell's single-quoted string around the program - `!` is inert inside `'...'` once history expansion is off.

A single quote inside a single-quoted string

Bash has no escape for it. The workaround is the four-character close-open dance: `'\''` reads as close-quote, literal-quote, open-quote.

```
# Want Perl to see:  print "it's here"
pperl -e 'print "it'\''s here\n"'
```

For any program with more than one embedded apostrophe, give up on `'...'` and use `q/.../` inside double quotes:

```
pperl -e "print q/it's here/, qq/\n/"
```

or move the program into a file.

The $ escape cascade under double quotes

If you quote the Perl program in double quotes (because you want `$1` to interpolate from the shell), Perl's own variables need `\$`:

```
col=3
pperl -lane "\$s += \$F[$col]; END { print \$s }"      # works
```

That is legible once. After the third such one-liner in a row, move to environment variables:

```
col=3
col=$col pperl -lane '$s += $F[$ENV{col}]; END { print $s }'
```

See *aliases#parametrised-functions*.

Locale and encoding

pperl's default I/O is bytes. If your input is UTF-8, Perl still sees a byte string until you ask for otherwise.

The symptom

```
$ echo 'héllo' | pperl -lne 'print length'
6                                                  # "h" + 2 bytes for é
↪+ 3 more = 6
$ echo 'héllo' | pperl -CSD -lne 'print length'
5                                                  # correct
```

The fix is `-C` (see *switches*):

- `-CSD` - STDIN/STDOUT/STDERR and file opens all UTF-8.
- `-CS` - stdio only.
- `-CA` - treat `@ARGV` as UTF-8.

Regex matching on UTF-8

Without `-C`, `\w` matches ASCII word characters only. With `-C`, `\w` matches Unicode letters/digits as defined by Perl's Unicode property tables.

```
# Count "word" characters in a multilingual file
pperl -CSD -lne '$c += () = /\w/g; END { print $c }' file.txt
```

Mojibake on the way out

If pperl is reading UTF-8 cleanly but the terminal shows garbled characters, the problem is further down - likely `LC_ALL` or the terminal's own encoding. `locale` and `tput` diagnose it; this is not a pperl problem.

`-i` in-place editing

Always `.bak` until verified

```
# Destructive if the pattern is wrong
pperl -i -pe 's/OLD/NEW/g' *.conf

# Safe
pperl -i.bak -pe 's/OLD/NEW/g' *.conf
# ...verify...
rm *.conf.bak
```

A wrong pattern with bare `-i` replaces the file's contents with something useless. There is no recovery short of version control. Use `-i.bak`, confirm, delete.

The empty-file trap

`-i` writes whatever `-p` (or `-n` plus explicit prints) emits. If the program prints nothing, the file becomes empty.

```
# Wrong: -i with -n and no print - empties the file
pperl -i -ne '/KEEP/ && print' file.txt      # prints only lines
↪matching KEEP

# Same thing, probably what you meant: keep the matching lines
pperl -i -ne 'print if /KEEP/' file.txt
```

Both are correct. The point: under `-n`, you own every byte of output.

In-place across many files with ownership caveats

`-i` creates a new file and renames. On Linux, this means the new file inherits the calling user's ownership and umask. Root-owned configs edited by non-root users will fail; root-owned configs edited by root will keep the mode but may lose SELinux context. Know your threat model before pointing `-i` at `/etc`.

The diamond operator <> and magic open

<> (the "diamond") is what `-n` / `-p` reads from. It reads from each argument in `@ARGV` as a filename, falling back to STDIN if `@ARGV` is empty.

The "-" is STDIN

If an argument is `"-"`, <> reads STDIN at that point. Useful for interleaving:

```
echo prefix | pperl -ne 'print' - trailer.txt
# prints: prefix, then contents of trailer.txt
```

Filename metacharacters (and why they matter less than they used to)

Historically, bare <> treated `">foo"` as "open foo for writing" - a two-argument open that read the mode from the filename. Modern Perl's <> uses three-argument open internally, so filenames starting with <, >, |, or containing pipes are treated as plain filenames. No attack surface on input filenames with modern pperl.

Filename is in $ARGV

```
pperl -lne 'print "$ARGV: $_" if /\bERROR\b/' *.log
```

`$ARGV` updates as each file opens. `"-"` is the special value for STDIN.

$. keeps counting across files

`$.` is not reset when <> moves to the next file. If you want per-file line numbers:

```
pperl -lne 'close ARGV if eof; print "$ARGV:$.: $_" if /TODO/' *.pl
```

`close ARGV if eof` resets `$.` when the current file ends.

Record-separator traps

`-l` removes then restores - once

`-l` chomps input and sets `$\` to the chomped separator for output. Fine under `-n` / `-p`. But `-l` applied twice (say, you added `-l` after `-0`) confuses people:

```
# Read NUL-delimited input, output newline-terminated
pperl -0 -lpe ''                              # correct: -l sets $\
 ↪= "\n"

# But if you write -l0 Perl parses that as -l with argument 0:
pperl -l0 -pe ''                              # $\ becomes NUL -
 ↪probably NOT what you wanted
```

Put `-l` after `-0` (no argument to `-l`) when you want the standard pair.

Paragraph mode keeps trailing blanks

`-00` (paragraph mode) sets `$/ = ""`. Perl reads up to and including the blank-line separator. Without `-l`, the separator stays attached to each paragraph; printing it back out preserves the blank line. With `-l`, the separator is chomped and `$\ = "\n"` appends one newline on output - paragraphs collapse.

```
$ printf 'a\n\nb\n\nc\n' | pperl -00 -pe ''
a

b

c
$ printf 'a\n\nb\n\nc\n' | pperl -00 -lpe ''
a
b
c
```

Know which one you want.

`-0777` and `$.`

Under `-0777`, each file is one record. `$.` increments per record, so `$.` is the number of files processed, not the number of lines.

```
pperl -0777 -e 'while (<>) { print "file $.: length=", length, "\n"
 ↪}' *.txt
# file 1: length=1024
# file 2: length=2048
```

The -F interpretation of its argument

`-F` takes a regex. Perl versions before 5.10 required you to write the delimiters; modern pperl accepts both forms:

```
pperl -F:      -lane '...'                    # bare: Perl quotes it
pperl -F/:/    -lane '...'                    # explicit delimiters
pperl -F'\t'   -lane '...'                    # tab - regex form
```

The common mistake is a literal tab in a shell where the terminal swallowed it:

```
pperl -F'        ' -lane '...'                  # fragile: tab may␣
↪be a space in your paste
```

Write `-F'\t'` unless you are pasting from known-good source.

Operator precedence around `print`

`print` has lower precedence than most operators. This trips people whose `awk` reflex is wrong for Perl:

```
# Wrong: prints into a filehandle named "..."
pperl -ne 'print "out.txt" $_'

# Right: parentheses or commas
pperl -ne 'print $_, "\n"'
pperl -ne 'print("foo\n")'

# Wrong: the binary || applies to print's argument list
pperl -ne 'print if $_ || "never"'            # works but obscure

# One particular case worth memorising:
pperl -ne 'print (1+1)*2'                     # prints 2, not 4
# ^ print(1+1) is the function call; *2 is applied to its return␣
↪value
```

When in doubt, put parentheses around `print`'s arguments or end the program with an explicit semicolon.

`tr` is not a regex

`tr` / `y` take character sets, not patterns. `tr/\d/X/` does NOT match digits - it replaces literal backslash-d characters.

```
pperl -pe 'tr/0-9/X/'                         # replace digits with␣
↪X (character range)
pperl -pe 's/\d/X/g'                          # replace digits with␣
↪X (regex)
```

Use `tr` for character-class work (case conversion, deletion, ranges) and `s///` when you need any regex machinery.

BEGIN and END scope

`BEGIN { ... }` and `END { ... }` blocks are part of the program, not the loop body. Variables declared with `my` inside `BEGIN` are not visible to the loop body. Use `our` or global variables for state that needs to persist from `BEGIN` into the loop:

```
# Wrong: $greeting is not in scope inside the loop
pperl -ne 'BEGIN { my $greeting = "Hi"; } print "$greeting $_"'

# Right: declare without my, or use our
pperl -ne 'BEGIN { $greeting = "Hi"; } print "$greeting $_"'
```

In practice, most `BEGIN` blocks do setup with side effects (`$\ = "\n"`, setting a format) rather than bindings.

die and the \n convention

`die` without a trailing `\n` appends `" at -e line N.\n"` to its message. With a trailing `\n`, it prints the message as-is. Choose deliberately:

```
# Developer diagnostic - want the location
pperl -E 'die "unreachable" if $x > 100'

# User-facing message - don't leak program structure
pperl -E 'die "bad input\n" unless /\d+/'
```

The same rule applies to `warn`.

Missing newline at end of file

If the last line of an input file has no trailing newline, `-l` still chomps what's there (nothing), and `$_` still lacks a `\n`. `-p` reprints it without a final newline, and the next line of shell output runs into it. Rarely a correctness problem; often a display problem.

```
pperl -pe '' no-final-newline.txt              # output lacks␣
↪trailing \n
```

Find out more

- *switches* - the mechanical expansions of every flag mentioned here.
- `perlvar` - the full definition of every special variable that the traps above depend on (`$/`, `$\`, `$.`, `$ARGV`, `@ARGV`).
- `perlop` - the diamond operator, `tr`, and the quoting forms referenced above.

4.2.8 Tying variables

A **tied** variable looks like an ordinary scalar, array, hash, or filehandle, but every read, write, and iteration on it is rerouted through methods you write. The variable is the public face; a class of your design is the machinery behind it. Assigning to a tied scalar calls your `STORE`; reading a tied hash key calls your `FETCH`; running `keys` over a tied

hash drives your FIRSTKEY and NEXTKEY. The caller writes plain Perl and never sees the class.

That indirection is what makes tie powerful. A hash can be a façade over an on-disk database. A scalar can recompute itself on every read. A filehandle can collect everything printed to it into a string. The code using the variable does not change; only the class behind it does.

This guide is about **writing the class** - the part perltie calls the implementation. The built-in side (tie, tied, untie) and the full method menus per variable type are already documented on the tie reference page. This guide links there for signatures and concentrates on what those pages do not teach: how to build a working tie class, what each method must return, and exactly when the runtime calls it.

Which chapter you want

Each variable type has its own constructor and its own method set. Read the chapter that matches the sigil in front of you.

- *Tied scalars* - TIESCALAR, FETCH, STORE, DESTROY. The smallest tie interface; start here to learn the shape of all the others.
- *Tied hashes* - TIEHASH plus the iteration protocol (FIRSTKEY / NEXTKEY), EXISTS, DELETE, CLEAR, and SCALAR. The richest of the data types.
- *Tied arrays* - TIEARRAY, the size protocol (FETCHSIZE / STORESIZE / EXTEND), and the mutators (PUSH, POP, SHIFT, UNSHIFT, SPLICE).
- *Tied filehandles* - TIEHANDLE and the I/O hooks (PRINT, PRINTF, WRITE, READLINE, READ, GETC, and friends).

How each chapter is built

Every chapter follows the same shape:

- **The contract by example.** Rather than restate the method menu (the tie reference page already lists every signature), each chapter builds one complete, runnable class and explains each method as it appears.
- **Call timing.** The non-obvious part of tie is *when* the runtime invokes each hook - which operator triggers which method, in which context, with which arguments. Each chapter spells this out.
- **Minimum viable vs. full.** The fewest methods you can define and still have a usable tie, then the rest of the menu and what each one buys you.
- **The easy way.** Each data type ships a Tie::* base class that implements the tedious methods in terms of a few core ones. Inherit from it and override only the interesting hooks.

What ties cost

Every operation on a tied variable is a method call. A tied hash lookup is a FETCH dispatch where a plain hash lookup is a single memory access - visibly slower in a tight loop. Tie is the right tool when the indirection earns its keep (a variable backed by a file, a computed value, a logging façade), not when you simply want a fancier hash. For bulk

work over a tied container, read it into a plain array or hash once, operate, and write back.

Differences from upstream

The tie mechanism is Perl 5.44's own, so tie classes behave exactly as they do on upstream Perl.

Tied scalars

A tied scalar is the smallest tie interface and the best place to learn the shape of all the others. You define a constructor and two accessor methods; from then on, every read of the variable calls one of them and every write calls the other.

The class needs four methods, two of them optional:

- `TIESCALAR` - the constructor, called once by `tie`.
- `FETCH` - called on every read of the variable.
- `STORE` - called on every write to the variable.
- `DESTROY` - called when the tied variable goes away (optional).

The full signature list is on the `tie` reference page. This chapter shows what each method does by building one.

A complete class

A counter that clamps writes so the value never drops below zero:

```
use v5.36;

package Counter {
    sub TIESCALAR {
        my ($class, $start) = @_;
        my $n = $start // 0;
        return bless \$n, $class;
    }
    sub FETCH { my $self = shift; return $$self }
    sub STORE {
        my ($self, $value) = @_;
        $$self = $value < 0 ? 0 : $value;   # never go below zero
    }
    sub DESTROY { }
}

tie my $hits, 'Counter', 10;
say $hits;            # 10
$hits = 25;
say $hits;            # 25
$hits = -7;
say $hits;            # 0
```

The object behind the scalar can be anything blessed - here a reference to a plain number. The `LIST` after the class name in the `tie` call (`10`) is passed straight to `TIESCALAR`.

The method contract

`TIESCALAR classname, LIST`
: The constructor. It receives the class name and whatever extra arguments were passed to `tie`. It must return a blessed reference; that reference becomes the tied object and the return value of the `tie` call. A common shortcut is a blessed scalar ref holding the value, as above, but a blessed hash ref is just as valid when the class needs more state.

`FETCH this`
: Called every time the variable is read. It takes only the object. Whatever it returns is what the reader sees. `FETCH` is where a computed scalar earns its keep: it can recompute, look something up, or log the access and return a stored value.

`STORE this, value`
: Called every time the variable is assigned. It receives the object and the single new value. Its return value is ignored - the assignment expression's value comes from a subsequent `FETCH`, not from `STORE`. This is where validation, clamping, or write-through to a backing store lives.

`DESTROY this`
: Called when the tied variable is destroyed (scope exit, or the last reference dropping). Define it only when the class holds a resource that needs releasing - an open handle, a lock. A counter has nothing to clean up, so its `DESTROY` is empty or omitted entirely.

Call timing

- Reading in any context calls `FETCH`. `say $hits`, `$x = $hits + 1`, and `"$hits"` each trigger one `FETCH`.
- Assignment calls `STORE` with the right-hand value. `$hits = 25` calls `STORE($obj, 25)`.
- Compound assignment reads then writes: `$hits += 5` calls `FETCH`, adds 5, then `STORE`. So does `$hits++`.
- The value of an assignment expression comes from `FETCH`, not `STORE`'s return. After `my $r = ($hits = -7)`, `$r` is `0` because the clamp ran in `STORE` and the expression's value was fetched back.

Minimum viable class

The smallest useful tied scalar defines exactly `TIESCALAR`, `FETCH`, and `STORE`. Omit `DESTROY` unless you allocate something. A class with only `TIESCALAR` and `FETCH` produces a read-only scalar: it ties fine, but the first assignment dies with `Can't locate object method "STORE"`.

The easy way: inherit from `Tie::StdScalar`

`Tie::StdScalar` (shipped inside `Tie::Scalar`) implements `TIESCALAR`, `FETCH`, `STORE`, and `DESTROY` over a blessed scalar ref. Inherit from it and override only the hook you want to change:

```
package Uppercase {
    use Tie::Scalar;
    our @ISA = ('Tie::StdScalar');
    sub STORE { my ($self, $v) = @_; $$self = uc $v }
}

tie my $shout, 'Uppercase';
$shout = 'hello';
say $shout;          # HELLO
```

`STORE` upper-cases on the way in; `FETCH` and the constructor come from the base class unchanged.

`Tie::Scalar` itself is the *abstract* base - it defines `FETCH` and `STORE` that croak, so subclassing it directly means supplying your own `TIESCALAR`. Reach for `Tie::StdScalar` when you want a working constructor for free; reach for `Tie::Scalar` only when you intend to write every method anyway and want the interface documented by inheritance.

See also

- `tie` - the built-in, with the full per-type method menu
- `tied` - recover the object backing the variable to call class-specific methods on it
- `untie` - dissolve the binding
- *Tied hashes* - the next step up: a constructor plus an iteration protocol
- `Tie::Scalar` - the base-class module; `Tie::StdScalar` is the concrete version you usually want

Tied hashes

A tied hash is the richest of the tie types, because a hash does more than read and write single elements - it iterates, it reports its size, it tests for key existence, and it clears. Each of those is a separate hook. Hashes were the first Perl data type to be tied (the original use case was binding `%hash` to an on-disk DBM file), and the interface still reflects that heritage.

The full method menu is on the `tie` reference page. This chapter builds one class and explains each hook, with attention to the part that trips people up: the iteration protocol.

A complete class

A hash whose keys are case-insensitive - `$h{Name}` and `$h{NAME}` address the same slot:

```
use v5.36;

package CaseInsensitive {
    sub TIEHASH  { my $class = shift; return bless {}, $class }
    sub STORE    { my ($self, $key, $val) = @_; $self->{lc $key} =
 ↪$val }
    sub FETCH    { my ($self, $key) = @_; return $self->{lc $key} }
    sub EXISTS   { my ($self, $key) = @_; return exists $self->{lc
 ↪$key} }
    sub DELETE   { my ($self, $key) = @_; return delete $self->{lc
 ↪$key} }
    sub CLEAR    { my $self = shift; %$self = () }
    sub FIRSTKEY { my $self = shift; my $reset = keys %$self; return␣
 ↪each %$self }
    sub NEXTKEY  { my $self = shift; return each %$self }
    sub SCALAR   { my $self = shift; return scalar %$self }
}

tie my %h, 'CaseInsensitive';
$h{Name}  = 'Ada';
$h{EMAIL} = 'ada@example.org';

say $h{name};                           # Ada
say $h{email};                          # ada@example.org
say exists $h{NAME} ? 'yes' : 'no';     # yes
say scalar %h ? 'nonempty' : 'empty';   # nonempty

for my $k (sort keys %h) {
    say "$k => $h{$k}";
}
# email => ada@example.org
# name => Ada
```

The object is a blessed hash ref used as the real storage; the hooks lower-case the key before touching it. Notice that iteration returns the *stored* (lower-cased) keys, which is exactly what a case-insensitive hash should do.

The method contract

`TIEHASH classname, LIST`
: The constructor. Returns a blessed reference - usually but not necessarily a hash ref - that becomes the tied object.

`FETCH this, key` / `STORE this, key, value`
: Single-element read and write, the same pair as for scalars but with a key argument. `STORE`'s return value is ignored.

`EXISTS this, key` / `DELETE this, key`
: Back the `exists` and `delete` built-ins. `DELETE`'s return value becomes `delete`'s

return value; to match a plain hash, return what FETCH would have returned for that key before removing it.

CLEAR this
Triggered when the whole hash is emptied, typically by assigning the empty list (%h = ()). Remove everything.

FIRSTKEY this / NEXTKEY this, lastkey
The iteration protocol - see below. Together they drive keys, values, and each.

SCALAR this
Called when the hash is evaluated in scalar or boolean context (scalar %h, if (%h), and since 5.28 keys %h in boolean context). Return a value that is true when the hash is non-empty and false when empty.

DESTROY this and UNTIE this
Optional cleanup hooks, identical in role to the scalar case. UNTIE is covered under the untie gotcha below.

The iteration protocol

This is the part of the hash interface that has no analogue in the scalar one, and the part that most often goes wrong.

When a keys, values, or each iteration begins, the runtime calls FIRSTKEY to get the first key, then NEXTKEY repeatedly, passing the previously returned key each time, until a method returns the empty list or undef. Both are always called in scalar context and should return just a key; the runtime calls FETCH itself to get each value.

Two rules keep iteration correct:

- **FIRSTKEY must reset any internal iterator.** When the backing store is a real hash, evaluate keys %$self in void or scalar context first - that resets Perl's per-hash each cursor - then call each. The throwaway my $reset = keys %$self in the example does exactly this. Skip it and a fresh keys %h may resume from wherever the last iteration stopped.
- **Signal the end with the empty list.** When the backing store is not a real hash and has no each of its own, return the empty list (or undef) once you have handed out the last key.

Why SCALAR is worth defining

Without a SCALAR method, the runtime guesses at the hash's truth in boolean context, and the guess can be wrong - notably, it may report a hash as non-empty right after you have emptied it by repeated DELETE. If if (%tied_hash) matters to your callers, define SCALAR and return the real answer, as the example does with scalar %$self.

Call timing

- $h{k} reads → FETCH; $h{k} = $v → STORE.
- exists $h{k} → EXISTS; delete $h{k} → DELETE.
- %h = () → CLEAR; %h = (a => 1) → CLEAR then STORE.

- `keys %h` / `values %h` / `each %h` → `FIRSTKEY` once, then `NEXTKEY` per step. `keys` and `values` in list context call `FETCH` for each key as well.
- `scalar %h`, `if (%h)` → `SCALAR` (if defined).
- A hash slice `@h{qw(a b)}` calls `FETCH` once per key; there is no slice hook.

Minimum viable class

A read/write tied hash needs at least `TIEHASH`, `FETCH`, and `STORE`. If callers will ever iterate it, add `FIRSTKEY` and `NEXTKEY` - without them, `keys` and `each` see an empty hash. Add `EXISTS` and `DELETE` if callers use those built-ins; without them, `exists`/`delete` die at the point of use. `SCALAR` and `CLEAR` round out correctness for boolean tests and bulk emptying.

The easy way: inherit from `Tie::StdHash`

`Tie::StdHash` (shipped inside `Tie::Hash`) implements the entire menu over a blessed hash ref. Inherit from it and override only the hooks that should behave differently:

```
package LoudHash {
    use Tie::Hash;
    our @ISA = ('Tie::StdHash');
    sub STORE { my ($self, $k, $v) = @_; $self->{$k} = uc $v }
}

tie my %loud, 'LoudHash';
$loud{greeting} = 'hi';
say $loud{greeting};                # HI
```

Only `STORE` changes; `FETCH`, `EXISTS`, the iteration protocol, and the constructor all come from the base class. `Tie::Hash` (the parent) is the abstract version that supplies a default `TIEHASH` and `new` but leaves the access methods for you to write; `Tie::StdHash` is the concrete one you usually want.

See also

- `tie` - the built-in and the full hash method menu
- `tied` - reach the backing object for class-specific methods
- `each` - the iterator that `FIRSTKEY` / `NEXTKEY` implement; its own page documents the end-of-iteration contract
- `exists` and `delete` - the built-ins behind `EXISTS` and `DELETE`
- *Tied scalars* - the simpler interface this one builds on
- `Tie::Hash` - the base-class module; `Tie::StdHash` is the concrete version

Tied arrays

A tied array has two layers of interface. The lower layer - `FETCH`, `STORE`, `FETCHSIZE`, `STORESIZE`, `CLEAR` - is mandatory and handles element access and resizing. The upper layer - `PUSH`, `POP`, `SHIFT`, `UNSHIFT`, `SPLICE`, `DELETE`, `EXISTS` - is optional and lets the array respond to the matching built-ins; without a given hook, that built-in either falls back to the lower layer or dies.

The full menu is on the `tie` reference page. This chapter builds a class, then explains the size protocol and the optional mutators, with their call timing verified against the running interpreter.

A complete class

A numeric array whose empty slots read back as `0` instead of `undef`, and whose `STORESIZE` pads new slots with `0`:

```
use v5.36;

package ZeroFill {
    sub TIEARRAY  { my $class = shift; return bless { data => [] },
 ↪$class }
    sub FETCH     { my ($self, $i) = @_; return $self->{data}[$i] //
 ↪0 }
    sub STORE     { my ($self, $i, $v) = @_; $self->{data}[$i] = $v
 ↪}
    sub FETCHSIZE { my $self = shift; return scalar @{ $self->{data}
 ↪} }
    sub STORESIZE {
        my ($self, $n) = @_;
        my $data = $self->{data};
        if ($n > @$data) {
            push @$data, (0) x ($n - @$data);   # pad with 0, not
 ↪undef
        } else {
            $#$data = $n - 1;
        }
    }
    sub CLEAR { my $self = shift; $self->{data} = [] }
    sub PUSH  { my $self = shift; push @{ $self->{data} }, @_;
 ↪return scalar @{ $self->{data} } }
    sub POP   { my $self = shift; return pop @{ $self->{data} } }
}

tie my @a, 'ZeroFill';
$a[0] = 11;
$a[2] = 33;                 # FETCH on the skipped slot 1 reads back
 ↪0
say scalar @a;              # 3
say $a[1];                  # 0
$#a = 4;                    # STORESIZE grows the array, padding with
```

(continues on next page)

(continued from previous page)

```
↪0
say "@a";                   # 11 0 33 0 0
push @a, 44;
say pop @a;                 # 44
```

The object is a blessed hash ref wrapping a real array in its `data` slot - a common pattern, because the class usually wants room for state beyond the array itself.

The mandatory layer

`TIEARRAY classname, LIST`
: The constructor; returns a blessed reference that becomes the tied object.

`FETCH this, index` / `STORE this, index, value`
: Single-element read and write. A negative index is translated to a positive one - the runtime calls `FETCHSIZE` first and adds it to the index - unless the class sets `$NEGATIVE_INDICES` to a true value, in which case the raw negative index reaches `FETCH` and `STORE`.

`FETCHSIZE this`
: Returns the number of elements. This backs `scalar @array` and the high-water mark for `$#array`. It is the single most-called method on a tied array, since the runtime needs the size to normalise negative indices and to bound iteration.

`STORESIZE this, count`
: Sets the array's length. Growing should fill the new slots with the class's notion of "empty" (here, `0`); shrinking should drop the excess. This backs `$#array = N` and assignment-driven resizing.

`CLEAR this`
: Empties the array - triggered by `@array = ()`.

The size protocol in practice

`FETCHSIZE` and `STORESIZE` are where tied arrays differ most from tied hashes. Three things to keep straight:

- **`$#a` reads through `FETCHSIZE` and writes through `STORESIZE`.** Setting `$#a = 4` calls `STORESIZE($obj, 5)`.
- **Skipped slots are the class's problem.** Assigning `$a[2]` when the array has length 1 calls `STORE($obj, 2, ...)` directly - the runtime does *not* call `STORESIZE` to fill the gap. If a read of slot 1 should return `0` rather than `undef`, `FETCH` has to say so (the `// 0` in the example).
- **`EXTEND` is advisory.** When the runtime expects an array to grow - for instance just before a large assignment - it may call `EXTEND this, count` as a hint to pre-allocate. It is always safe to leave `EXTEND` undefined or empty; it is purely an optimisation opportunity, and making it behave like `STORESIZE` is a bug, because the runtime calls `EXTEND` at times when it does not actually want the visible length to change.

The optional mutators

Each of `push`, `pop`, `shift`, `unshift`, `splice`, `delete`, and `exists` dispatches to the uppercase hook of the same name when the class defines it. Their call timing, confirmed against the interpreter:

- `push @a, 1, 2` → one `PUSH` call with the whole list.
- `pop @a` → `POP`; `shift @a` → `SHIFT`; `unshift @a, 0` → `UNSHIFT`.
- `splice @a, 1, 1, 'x', 'y'` → one `SPLICE` call carrying offset, length, and the replacement list; its return value becomes `splice`'s.
- `exists $a[0]` → `EXISTS`; `delete $a[0]` → `DELETE`.

When a mutator hook is **absent**, behaviour depends on the operator. `Tie::Array` (below) supplies fallbacks for the common five in terms of the mandatory layer; without that base class, calling an unimplemented mutator dies at the point of use.

Call timing

- `$a[$i]` → `FETCH`; `$a[$i] = $v` → `STORE`.
- `scalar @a`, normalising a negative index → `FETCHSIZE`.
- `$#a = $n`, `@a = (...)` resizing → `STORESIZE` (and `CLEAR` first for whole-array assignment).
- push/pop/shift/unshift/splice/delete/exists → the matching uppercase hook when defined.
- An array slice `@a[1, 3]` calls `FETCH` once per index; there is no slice hook.

Minimum viable class

The smallest tied array defines `TIEARRAY`, `FETCH`, `STORE`, `FETCHSIZE`, and `STORESIZE`. With those five, element access, `scalar @a`, `$#a`, and iteration all work. Add `CLEAR` so `@a = ()` behaves. Add the mutators only for the built-ins your callers actually use - or inherit them.

The easy way: inherit from `Tie::Array`

`Tie::Array` implements `PUSH`, `POP`, `SHIFT`, `UNSHIFT`, and `SPLICE` in terms of the mandatory five, so a subclass that supplies `TIEARRAY`, `FETCH`, `STORE`, `FETCHSIZE`, and `STORESIZE` gets all the list mutators for free:

```
package Logging {
    use Tie::Array;
    our @ISA = ('Tie::Array');
    sub TIEARRAY  { my $class = shift; return bless { data => [] },
↪$class }
    sub FETCH     { my ($self, $i) = @_; return $self->{data}[$i] }
    sub STORE     { my ($self, $i, $v) = @_; warn "store $i\n";
↪$self->{data}[$i] = $v }
    sub FETCHSIZE { my $self = shift; return scalar @{ $self->{data}␣
```

(continues on next page)

(continued from previous page)

```
 ↪} }
    sub STORESIZE { my ($self, $n) = @_; $#{ $self->{data} } = $n -␣
 ↪1 }
}

tie my @log, 'Logging';
push @log, 'x', 'y';        # PUSH inherited; calls STORE twice␣
 ↪(warns)
say "@log";                 # x y
```

`push @log, 'x', 'y'` works without a `PUSH` method because the inherited one expands into two `STORE` calls - which is why the example warns twice. `Tie::Array` does not define `TIEARRAY` itself, so the subclass must supply the constructor; its default `DELETE` and `EXISTS` simply croak, so override them if your callers need `delete` and `exists`.

See also

- `tie` - the built-in and the full array method menu
- `tied` - reach the backing object
- `splice` - the most intricate mutator the `SPLICE` hook stands in for
- `push` and `pop` - the mutators `Tie::Array` derives from the mandatory layer
- *Tied hashes* - the other rich container interface
- `Tie::Array` - the base-class module that supplies the list mutators

Tied filehandles

A tied filehandle reroutes I/O the way a tied scalar reroutes reads and writes. Print to it and your `PRINT` runs; read a line and your `READLINE` runs. The classic uses are redirecting output somewhere unusual (into a string, across a network, through a filter) and embedding Perl in a host program that needs `STDOUT` and `STDERR` handled specially.

The interface is larger than the others but mostly optional: you implement only the operations the handle will actually receive. The full menu is on the `tie` reference page; this chapter builds a class and explains the hooks and their timing.

A complete class

A write-only handle that collects everything printed to it into a string:

```
use v5.36;

package Collector {
    sub TIEHANDLE { my $class = shift; my $buf = ''; return bless \
 ↪$buf, $class }
    sub PRINT     { my $self = shift; $$self .= join($, // '', @_);␣
 ↪return 1 }
```

(continues on next page)

(continued from previous page)

```
    sub PRINTF    { my $self = shift; my $fmt = shift; $$self .=␣
↪sprintf($fmt, @_); return 1 }
    sub buffer    { my $self = shift; return $$self }
}

tie *OUT, 'Collector';
print OUT "hello ";
print OUT "world\n";
printf OUT "%d items\n", 3;
my $obj = tied *OUT;
print $obj->buffer;
# hello world
# 3 items
```

The object is a blessed scalar ref holding the accumulated text. The extra `buffer` method is reached through `tied`, not through any I/O operation - that is the standard way a tie class exposes operations with no filehandle syntax.

Tying the glob, not the scalar

The first argument to `tie` for a handle must be a glob - it begins with an asterisk:

- To tie a named handle: `tie *OUT, 'Collector'`.
- To tie a handle held in a scalar `$fh`: `tie *$fh, 'Collector'`. Writing `tie $fh, ...` ties the *scalar* `$fh`, not the handle inside it - a common and confusing mistake.

The method contract

`TIEHANDLE classname, LIST`
: The constructor; returns a blessed reference of any kind. It can hold whatever internal state the handle needs.

`PRINT this, LIST`
: Backs `print` and `say`. It receives the printed list. Honour `$,` (the output field separator) between items and `$\` (the output record terminator) at the end if you want full fidelity - `print` does not insert them for you.

`PRINTF this, format, LIST`
: Backs `printf`. It receives the format string and arguments; run them through `sprintf` yourself.

`WRITE this, scalar, length, offset`
: Backs `syswrite` - the raw, buffer-oriented write, distinct from `PRINT`.

`READLINE this`
: Backs `<$fh>` and `readline`. In scalar context return the next line or `undef` at end of input; in list context return all remaining lines or an empty list. The returned strings should include the input record separator `$/`, matching what a real handle gives you.

`READ this, scalar, length, offset` / `GETC this`
: Back `read` / `sysread` and `getc` respectively.

OPEN, CLOSE, EOF, BINMODE, FILENO, SEEK, TELL
: Back the corresponding built-ins when used on the handle. Implement the ones that make sense for your backing store and leave the rest out.

DESTROY this / UNTIE this
: Optional cleanup, identical in role to the other tie types. A handle class often closes its backing resource here.

Call timing

- `print FH @list` and `say FH @list` → `PRINT`. `say` additionally localises `$\` to `"\n"` around the call, so a `PRINT` that honours `$\` handles `say` with no special case.
- `printf FH $fmt, @args` → `PRINTF`.
- `syswrite FH, $buf, $len, $off` → `WRITE`.
- `<FH>` / `readline FH` → `READLINE`, in the surrounding context.
- `read`/`sysread` → `READ`; `getc FH` → `GETC`.
- `close FH`, `eof FH`, `binmode FH`, `seek`, `tell`, `fileno`, `open FH, ...` → the matching uppercase hook when defined.

A note on `STDERR`: when `STDERR` is tied, its `PRINT` is what issues warnings and error messages. The runtime disables the tie for the duration of that call, so you can `warn` from inside `PRINT` without triggering infinite recursion.

Minimum viable class

There is no single mandatory method beyond `TIEHANDLE` - you implement whichever operations the handle will receive. An output-only handle needs `TIEHANDLE` plus `PRINT` (and usually `PRINTF`). An input-only handle needs `TIEHANDLE` plus `READLINE` (and perhaps `GETC` / `READ`). Operations whose hook is missing die at the point of use, so define exactly the set your callers exercise.

The easy way: inherit from `Tie::StdHandle`

`Tie::StdHandle` (shipped inside `Tie::Handle`) ties a handle onto a real underlying one: its `OPEN` opens an actual file and the I/O hooks forward to it. Inherit from it and override only the operations you want to intercept:

```
package Doubler {
    use Tie::StdHandle;
    our @ISA = ('Tie::StdHandle');
    sub READLINE {
        my $self = shift;
        my $line = $self->SUPER::READLINE;
        return defined $line ? $line . $line : undef;   # echo each␣
↪line twice
    }
}

my $data = "a\nb\n";
```

(continues on next page)

(continued from previous page)

```
tie *DUP, 'Doubler';
tied(*DUP)->OPEN('<', \$data);
print while defined($_ = <DUP>);
# a
# a
# b
# b
```

`READLINE` calls up to the base class for the real next line, then doubles it; every other operation flows through `Tie::StdHandle` unchanged. `Tie::Handle` (the parent) is the abstract base that documents the interface and supplies a default `PRINT`/`PRINTF` in terms of `WRITE`; `Tie::StdHandle` is the concrete, ready-to-use one.

See also

- `tie` - the built-in and the full handle method menu
- `tied` - the only way to reach a handle class's non-I/O methods, like `buffer` above
- `print` and `printf` - the built-ins behind `PRINT` and `PRINTF`
- `readline` - its page documents the scalar/list return contract `READLINE` must honour
- *Tied scalars* - the simplest tie type, a good warm-up for the handle interface
- `Tie::Handle` - the base-class module; `Tie::StdHandle` is the concrete version

4.2.9 Coming from another language

You already know how to program. You reach for a hash, a loop, a function, a regex without thinking - in some other language. These guides teach Perl by mapping what you already do onto how Perl does it, one construct at a time. Your dictionary is a Perl hash; here is the syntax, and here is the one place the analogy leaks.

Each guide assumes you are **fluent in the source language and new to Perl**. It does not teach the source language back to you. It teaches the Perl side of each pairing: the syntax, the idiom, and - most useful of all - the gotcha that the surface similarity hides.

Every guide in this series covers the **same topics in the same order**, so the mapping is comparable across languages and the series grows cleanly as new languages are added. The order moves from what every language shares (variables, strings, numbers) through Perl's distinctive shapes (sigils, the array/hash split, and **context** - the one idea no other language has), into structure (subroutines, references, modules, objects), and out through the working surface (regex, I/O, error handling, idioms), ending with a per-language collection of the traps that actually bite.

Which guide you want

- *Python to Perl* - sigils where Python has none, the array/hash split where Python has `list` and `dict`, comprehensions rewritten as `map`/`grep`, and the context concept Python has no word for.

- *PHP to Perl* - you already have `$`, but in Perl it shifts by access, PHP's one `array` splits into `@array` and `%hash`, and `.` versus `+` (and `==` versus `eq`) stop being interchangeable.
- *Ruby to Perl* - blocks become Perl blocks and closures, method-calls-on-everything become plain functions, and Ruby's "only `nil` and `false` are falsy" inverts into Perl's truthiness.
- *Lua to Perl* - the one table type splits into `@array` and `%hash`, indexing turns 0-based, Lua patterns become real regex, and multiple-return-values are your bridge to Perl's context.
- *Sed to Perl* - the same `s///` you already know, sed's `y///` becomes `tr///`, address ranges become the `/a/ .. /b/` flip-flop, and the hold space becomes an ordinary Perl variable.
- *Awk to Perl* - Perl grew out of awk, so almost everything maps directly: fields become `@F` and `split`, `NR` becomes `$.`, associative arrays become hashes, and `BEGIN`/`END` keep their names.
- *Shell to Perl* - for when a shell script outgrew itself: `$(cmd)` becomes `qx{}`, the `[ ]`/`-f` file tests carry straight over, and the `${var#...}` parameter expansions become real regex.
- *Tcl to Perl* - a carry-over for Tcl maintainers: `set x` becomes `my $x`, `[expr {}]` disappears into native operators, lists and `array` become Perl arrays and hashes, and `regsub` becomes `s///`.

The one concept to read first

If you read only one section of whichever guide matches your background, read **Context**. Every other language in this series collapses a collection to a count, a first element, or a boolean with explicit syntax (`len(x)`, `x[0]`, `bool(x)`). Perl does it through *context*: the same expression yields a list in one place and a single scalar in another, decided by where you wrote it. It is the source of most "why did my array become the number 3?" confusion, and once it clicks the rest of Perl falls into place.

A note on the runtime

These guides target `pperl`, a faithful Rust reimplementation of Perl 5.44. Every Perl example that runs as a script was executed on `pperl` and shows real output. The text-processing guides (sed, awk, shell) lead with the classic `perl -ne`/`perl -pe` one-liner forms, the idiomatic Perl equivalent, and pair each with the explicit-loop script the switches expand to - the form you grow a one-liner into.

Python to Perl

You know Python. You write list comprehensions in your sleep, you reach for a `dict` without thinking, and `for x in xs:` is muscle memory. This guide maps each of those habits onto Perl, so you can be productive in `pperl` in an afternoon instead of relearning programming from scratch.

Two shocks land in the first five minutes, so meet them now:

- **Perl uses braces and semicolons, not indentation.** There is no significant whitespace. A block is `{ ... }`; a statement ends with `;`. Indent for your own sanity, but the parser ignores it.
- **Variables wear sigils.** A scalar is `$x`, an array is `@x`, a hash is `%x`. The sigil is part of how you read the variable, not just its name. This is the single biggest surface difference from Python, and section one is about nothing else.

Every example here runs on `pperl`, a Rust reimplementation of Perl 5.44. Each Perl snippet was executed on `pperl`; the `# ...` comments show its real output. Start your scripts with:

```
use v5.44;
```

That one line turns on `strict` (no accidental globals), `warnings`, the `say` built-in (a `print` with a trailing newline, like Python's `print`), subroutine signatures, and the modern feature set used throughout this guide.

How to read this

- *Variables and sigils* - the `$`/`@`/`%` shift.
- *Strings* - interpolation and concatenation.
- *Numbers* - one surface type, float division.
- *Arrays and lists* - your `list`, plus `map`/`grep`.
- *Hashes* - your `dict`.
- *Context* - the concept Python has no word for. Read this one.
- *Truthiness* - what counts as false, with surprises.
- *Control flow* - `if`, loops, the postfix forms.
- *Subroutines* - `def` becomes `sub`.
- *References* - nested data and closures.
- *Regular expressions* - regex as syntax, not a library.
- *File and stream I/O* - open, line reads, slurp.
- *Modules and packages* - `import` becomes `use`.
- *Objects* - the modern `class` syntax.
- *Error handling* - `try`/`except` becomes `eval` or `try`/`catch`.
- *Idioms* - the daily one-liners.
- *Gotchas coming from Python* - scan before you write.

Variables and sigils

In Python a name just names a value. In Perl the **sigil tells you the shape**: `$` for a single value, `@` for an ordered list, `%` for a key/value map. `my` declares a lexically scoped variable, the Perl equivalent of an ordinary local binding.

```
name = "Ada"
langs = ["Python", "Perl"]
ages = {"Ada": 36, "Linus": 56}
```

```
use v5.44;
my $name = "Ada";
my @langs = ("Python", "Perl");
my %ages  = (Ada => 36, Linus => 56);
say $name;            # Ada
say $langs[0];        # Python
say $ages{Ada};       # 36
```

The gotcha: when you reach *into* an aggregate for one element, the sigil follows the value you get back, not the container. A single element of `@langs` is a scalar, so it is `$langs[0]`, not `@langs[0]`; one value from `%ages` is `$ages{Ada}`. The `@`/`%` form is the whole collection; the `$` form is one piece of it.

Strings

Double quotes interpolate variables, single quotes do not. There is no f-string because the default string *is* an f-string. Concatenation is `.`, not `+` (which is reserved for numbers).

```
name = "Ada"
greeting = f"Hello, {name}"
shout = "AB" * 3
```

```
use v5.44;
my $name = "Ada";
my $greeting = "Hello, $name";   # Hello, Ada
my $literal  = 'Hello, $name';   # Hello, $name  (no interpolation)
my $joined   = "AB" . "CD";      # ABCD
my $shout    = "AB" x 3;         # ABABAB
```

The gotcha: string repetition is `x`, not `*`, and the join operator is `.`, not `+`. Mixing them up is the most common first-day typo for a Python developer. Array elements and hash values interpolate inside double quotes too: `"First: $langs[0]"` works as written.

Numbers

Perl has one numeric type at the surface; you do not declare `int` vs `float`. Division always produces a float, exactly like Python 3.

```
print(7 / 2)        # 3.5
print(7 // 2)       # 3
print(2 ** 10)      # 1024
print(10 % 3)       # 1
```

```
use v5.44;
say 7 / 2;          # 3.5
```

(continues on next page)

(continued from previous page)

```
say int(7 / 2);      # 3
say 2 ** 10;         # 1024
say 10 % 3;          # 1
```

The gotcha: comparison and equality operators come in two families, and you pick by the *operands' type*, not the variables'. Numbers use `==`, `!=`, `<`, `>`, `<=>`; strings use `eq`, `ne`, `lt`, `gt`, `cmp`. `"10" == "10.0"` is true (numeric), but `"10" eq "10.0"` is false (string). Using `==` on strings, or `eq` on numbers, is a frequent and quiet bug.

Arrays and lists

A Perl array is your Python `list`. The verbs are renamed but the operations are the same, and negative indices work as you expect.

```
nums = [1, 2, 3, 4, 5]
nums.append(6)
last = nums.pop()
print(len(nums))    # 5
print(nums[1:3])    # [2, 3]
```

```
use v5.44;
my @nums = (1, 2, 3, 4, 5);
push @nums, 6;
my $last = pop @nums;       # 6
say scalar @nums;           # 5
say "@nums[1,2]";           # 2 3   (a slice)
```

Comprehensions become `map` and `grep`. `map` transforms (Python's `[f(x) for x in xs]`); `grep` filters (Python's `[x for x in xs if p(x)]`). `$_` is the current element, Perl's implicit loop variable.

```
squares = [x * x for x in nums]
evens   = [x for x in nums if x % 2 == 0]
desc    = sorted(nums, reverse=True)
```

```
use v5.44;
my @nums    = (1, 2, 3, 4, 5);
my @squares = map  { $_ * $_ }   @nums;   # 1 4 9 16 25
my @evens   = grep { $_ % 2 == 0 } @nums; # 2 4
my @desc    = sort { $b <=> $a }  @nums;  # 5 4 3 2 1
```

The gotcha: `sort` compares as strings by default, so `sort (10, 9, 100)` gives `10 100 9`. For numbers you must pass a comparator: `sort { $a <=> $b } @nums`. `$a` and `$b` are the two elements being compared, and `<=>` is the numeric three-way operator (`cmp` is its string sibling).

Hashes

A Perl hash is your Python `dict`. Declared with `%`, accessed one value at a time with `$h{key}`.

```python
h = {"apple": 3, "pear": 5, "plum": 2}
print(h["apple"])                # 3
print(sorted(h.keys()))          # ['apple', 'pear', 'plum']
print("pear" in h)               # True
del h["plum"]
for k in sorted(h):
    print(k, "=>", h[k])
```

```perl
use v5.44;
my %h = (apple => 3, pear => 5, plum => 2);
say $h{apple};                           # 3
say join ",", sort keys %h;              # apple,pear,plum
say exists $h{pear} ? "yes" : "no";      # yes
delete $h{plum};
for my $k (sort keys %h) {
    say "$k => $h{$k}";
}
```

The gotcha: membership is `exists $h{key}`, not Python's `in`, and the fat comma `=>` is the idiomatic key/value separator. `=>` also auto-quotes the bareword on its left, so `apple => 3` needs no quotes around `apple`. A hash has no order; iterate `sort keys %h` when you want a stable one.

Context

This is the chapter to read twice. Python has no equivalent, and almost every "Perl is weird" story traces back to it.

Every expression in Perl runs in either **list context** or **scalar context**, decided by where it appears, not by what it is. The same array behaves differently in each. Assigning an array to a *scalar* gives its length; assigning it to a *list* (a parenthesised set of variables) copies elements.

```perl
use v5.44;
my @list = (10, 20, 30);

my $count = @list;          # scalar context -> 3 (the length)
say $count;                 # 3

my ($first) = @list;        # list context -> first element copied
say $first;                 # 10

my ($a, $b) = @list;        # 10 and 20
say "$a $b";                # 10 20
```

The parentheses on the left are the whole difference. `my $x = @list` asks "how many?" and gets 3. `my ($x) = @list` asks "give me the elements" and gets `10`. A Python devel-

oper reads both as "assign the list," which is exactly the trap.

A subroutine can ask which context it was called in, with `wantarray`, and return different things accordingly:

```
use v5.44;
sub items {
    return wantarray ? (1, 2, 3) : "three items";
}
my @got = items();          # list context
my $msg = items();          # scalar context
say "@got";                 # 1 2 3
say $msg;                   # three items
```

Built-ins use this too. `scalar @array` forces scalar context to get a count; a regex match with `/g` returns the list of matches in list context and a true/false in scalar context. The classic idiom for "count the matches" forces a list assignment in scalar context with the so-called *countof* form:

```
use v5.44;
my $n = () = "a1b2c3" =~ /\d/g;
say $n;                     # 3
```

The gotcha: there is no single answer to "what does this array do" - it depends on the surrounding expression. When something returns a number you expected to be a list (or vice versa), context is the first suspect. Reach for `scalar` to force a count and parentheses on the left to force element copying.

Truthiness

Perl's false values are a short, specific list: the number `0`, the empty string `""`, the string `"0"`, and `undef` (Perl's None). **Everything else is true** - and that includes some strings a Python developer assumes are false.

```
use v5.44;
for my $v (0, 1, "", "0", "00", "0.0", "false") {
    say "'$v' is ", ($v ? "true" : "false");
}
```

This prints, exactly:

```
'0' is false
'1' is true
'' is false
'0' is false
'00' is true
'0.0' is true
'false' is true
```

The gotcha: `"00"` and `"0.0"` are **true** in Perl, even though they look numerically zero, because only the exact string `"0"` is false. The string `"false"` is true (it is a non-empty, non-`"0"` string). And unlike Python, an empty array or empty hash is not itself a false *value* - in boolean context an array yields its length, so `if (@list)` is true when the list

is non-empty, which is the test you want. For a "use this if defined, else a default" choice, use defined-or `//`, which falls back only on `undef` (not on `0` or `""`):

```
use v5.44;
my %cfg = (host => "localhost");
my $port = $cfg{port} // 8080;   # key absent -> undef -> 8080
say $port;                       # 8080
```

Control flow

The keywords are familiar; the spelling differs. `elif` is `elsif`, and there is no `:` and no indentation - blocks are braced.

```
if n < 0:
    sign = "neg"
elif n == 0:
    sign = "zero"
else:
    sign = "pos"

for x in nums:
    print(x)

while running:
    step()
```

```
use v5.44;
my $sign;
if    ($n < 0)  { $sign = "neg"  }
elsif ($n == 0) { $sign = "zero" }
else            { $sign = "pos"  }

for my $x (@nums) { say $x }

while ($running) { step() }
```

Perl adds two things Python does not have. **Postfix modifiers** let a simple statement carry its own condition or loop, which reads naturally for one-liners:

```
use v5.44;
say $_ for @nums;                # loop, one statement
say "negative" if $n < 0;        # guarded statement
$sum += $_ for @nums;            # accumulate
```

And `unless`/`until` are the negated forms of `if`/`while`, for when the positive phrasing reads awkwardly: `say "ok" unless $error`.

The gotcha: the ternary chains the same as Python's, but with `?`/`:`, not `if`/`else`: `my $sign = $n < 0 ? "neg" : $n == 0 ? "zero" : "pos"`. The C-style `?:` is the idiom; there is no `x if c else y` form.

Subroutines

`def` becomes `sub`. Modern Perl has signatures that look much like Python's parameter lists, including defaults.

```python
def greet(name, greeting="Hello"):
    return f"{greeting}, {name}!"

def total(*nums):
    return sum(nums)
```

```perl
use v5.44;
sub greet ($name, $greeting = "Hello") {
    return "$greeting, $name!";
}
say greet("Ada");           # Hello, Ada!
say greet("Linus", "Hi");   # Hi, Linus!

sub total (@nums) {
    my $sum = 0;
    $sum += $_ for @nums;
    return $sum;
}
say total(1, 2, 3, 4);      # 10
```

Keyword arguments map to a hash parameter, since `key => value` pairs flatten into a list that a `%opts` signature gathers up:

```perl
use v5.44;
sub connect_to (%opts) {
    return "host=$opts{host} port=$opts{port}";
}
say connect_to(host => "localhost", port => 8080);
# host=localhost port=8080
```

The gotcha: a sub returns the value of its last expression even without `return`, and the return is **context-sensitive** (see *Context*). If you `return @list`, the caller gets the list in list context but the *count* in scalar context. Be deliberate about what you return when it might be used either way.

References

Perl's arrays and hashes do not nest directly; you nest them through **references**, which are the Perl analogue of Python's object handles (in Python every container is already a reference, so this step is invisible to you there). A reference is a scalar that points at an aggregate. Take one with `\`, follow it with `->`.

```python
data = {"name": "Ada", "langs": ["Perl", "Python"]}
print(data["langs"][1])     # Python
data["langs"].append("Ruby")
```

```
use v5.44;
my $data = { name => "Ada", langs => ["Perl", "Python"] };
say $data->{name};                  # Ada
say $data->{langs}[1];              # Python
push $data->{langs}->@*, "Ada";     # postfix deref to push
say "@{$data->{langs}}";           # Perl Python Ada
```

`{ ... }` builds an anonymous hash reference and `[ ... ]` an anonymous array reference - the literal building blocks of nested data, matching Python's `{}` and `[]` literals. The `->` arrow follows a reference one step; between two subscripts it is optional, so `$data->{langs}[1]` and `$data->{langs}->[1]` are the same.

Anonymous subs are closures, exactly like Python's `lambda` but with no single-expression limit:

```
use v5.44;
my $double = sub ($n) { $n * 2 };
say $double->(21);          # 42
```

The gotcha: the postfix deref forms (`->@*`, `->%*`, `->$*`) are the modern, readable way to dereference a whole aggregate, and `"@{ ... }"` is how you interpolate a dereferenced array into a string. The older `@$ref` and `${$ref}{key}` forms still work and appear in existing code, but reach for the arrow and postfix forms in new code.

Regular expressions

In Python regex is the `re` library. In Perl it is **syntax**: `=~` binds a pattern to a string, `m//` matches, `s///` substitutes. Capture groups land in `$1`, `$2`, and named captures in `%+`.

```
import re
m = re.search(r"(\d{4})-(\d{2})-(\d{2})", s)
if m:
    year, month, day = m.groups()
t = re.sub(",", ";", "a,b,c")
parts = "one,two,three".split(",")
```

```
use v5.44;
my $s = "2026-06-20";
if ($s =~ /(\d{4})-(\d{2})-(\d{2})/) {
    say "$1 $2 $3";                 # 2026 06 20
}
if ($s =~ /(?<year>\d{4})/) {
    say $+{year};                   # 2026
}
(my $t = "a,b,c") =~ s/,/;/g;       # a;b;c
my @parts = split /,/, "one,two,three";   # one two three
```

A `/g` match in list context returns *all* captures, which is the direct analogue of `re.findall`:

```
use v5.44;
my @nums = "x1y2z3" =~ /(\d)/g;     # 1 2 3
say "@nums";
```

The gotcha: `s///` modifies its target *in place* and returns the number of substitutions, not a new string. That is why the example copies first with `(my $t = "a,b,c")` and then substitutes on the copy - otherwise you would be editing a string literal. To get a new string without touching the original, use the `/r` flag: `my $t = $s =~ s/,/;/gr`.

File and stream I/O

The idiom is the three-argument open with a lexical filehandle, the direct counterpart of Python's `with open(...) as f:`. The angle-bracket operator `<$fh>` reads a line, the same role as iterating a Python file object.

```
with open("/tmp/sample.txt", "w") as f:
    for i in range(1, 4):
        f.write(f"line {i}\n")

with open("/tmp/sample.txt") as f:
    for line in f:
        print("got:", line.rstrip())
```

```
use v5.44;
my $path = "/tmp/sample.txt";

open my $wh, '>', $path or die "cannot write: $!";
print {$wh} "line $_\n" for 1 .. 3;
close $wh;

open my $rh, '<', $path or die "cannot read: $!";
while (my $line = <$rh>) {
    chomp $line;
    say "got: $line";       # got: line 1 ... got: line 3
}
close $rh;
```

To slurp every line at once, read the handle in list context:

```
use v5.44;
open my $fh, '<', "/tmp/sample.txt" or die $!;
my @lines = <$fh>;          # all lines, including newlines
say scalar @lines;          # 3
```

The gotcha: a line read with `<$fh>` keeps its trailing newline; call `chomp` to strip it (Python's `line.rstrip()` reflex). And open returns false on failure rather than raising, so the `or die "...: $!"` idiom is mandatory - `$!` is the system error message, Perl's `errno`.

Modules and packages

`import` becomes `use`. The namespace separator is `::`, not `.`, and a module `Foo::Bar` lives at `Foo/Bar.pm` on the include path `@INC` (Perl's `sys.path`). A module's import list is a selection of names, much like `from module import name`.

```
from math import sqrt
import json
data = json.loads(text)
```

```
use v5.44;
use List::Util qw(sum0 max first);
say sum0(1 .. 10);                          # 55
say max(3, 7, 2);                           # 7
say first { $_ % 2 == 0 } (1, 3, 4, 5);  # 4
```

The standard library is large and much of what you would reach to PyPI for ships in core or on CPAN, the Perl package archive. `List::Util`, `Scalar::Util`, `POSIX`, `File::Spec`, and `Data::Dumper` are the everyday ones.

The gotcha: `use` runs at *compile time*, not at the point in the file where it appears, so the order of `use` lines relative to your code does not matter the way `import` placement does. Importing is also opt-in per name: `use List::Util qw(sum0 max)` brings exactly `sum0` and `max` into your namespace, nothing else.

Objects

For new code, Perl 5.44 has a first-class `class` syntax that will look familiar coming from Python. Fields are declared with `field`, methods with `method`, and `$self` is available inside every method just as `self` is in Python (you do not list it as a parameter).

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
    def to_string(self):
        return f"({self.x}, {self.y})"
    def move(self, dx, dy):
        self.x += dx
        self.y += dy
        return self

p = Point(x=3, y=4)
print(p.to_string())        # (3, 4)
p.move(1, 1)
print(p.to_string())        # (4, 5)
```

```
use v5.44;
use feature 'class';

class Point {
```

(continues on next page)

(continued from previous page)

```
    field $x :param = 0;
    field $y :param = 0;
    method to_string { return "($x, $y)" }
    method move ($dx, $dy) {
        $x += $dx;
        $y += $dy;
        return $self;
    }
}

my $p = Point->new(x => 3, y => 4);
say $p->to_string;          # (3, 4)
$p->move(1, 1);
say $p->to_string;          # (4, 5)
```

The constructor is generated for you: `:param` marks a field that the auto-built `new` accepts as a named argument, and `= 0` is its default. Inheritance is `:isa`:

```
use v5.44;
use feature 'class';

class Point3D :isa(Point) {
    field $z :param = 0;
    method z { return $z }
}

my $q = Point3D->new(x => 1, y => 2, z => 3);
say $q->to_string, " z=", $q->z;   # (1, 2) z=3
```

The gotcha: the `class` feature is still marked experimental in Perl 5.44, and on `pperl` the "class is experimental" notices print to **stderr** even with `no warnings 'experimental::class'` in scope. They do not touch your program's output, so a script that prints to stdout behaves exactly as shown above; just expect the notices on the error stream. For the older `bless`-based object model you will meet in existing code, and for roles, inheritance details, and migration, see the *Object-oriented Programming guide*.

Error handling

Python's `try`/`except` has two counterparts. The classic one wraps code in `eval { }` and inspects `$@` (the caught error) afterward:

```
try:
    risky(-1)
except Exception as e:
    print("caught:", e)
```

```
use v5.44;
my $result = eval { risky(-1) };
if ($@) {
```

(continues on next page)

(continued from previous page)

```
    print "caught: $@";     # caught: negative!
}
```

The modern form is `try`/`catch`, which reads almost exactly like Python's:

```
use v5.44;
use feature 'try';

try {
    risky(-4);
}
catch ($e) {
    print "caught: $e";     # caught: negative!
}
```

where the raising side is `die`:

```
use v5.44;
sub risky ($n) {
    die "negative!\n" if $n < 0;
    return sqrt $n;
}
```

The gotcha: `die` with a string that ends in `\n` produces exactly that message; `die` with a string that does *not* end in a newline gets `" at SCRIPT line N.\n"` appended automatically. End your error strings with `\n` when you want a clean message and omit it when you want the file-and-line trace. Note also the two error variables: `$@` is the last caught exception (Python's `except ... as e`), while `$!` is the last *system* error (a failed open, the `errno` string).

Idioms

A few patterns you will see daily and want in your fingers.

Sort a list by a computed key (the Schwartzian transform - Python's `sorted(xs, key=f)` done by hand, decorate then sort then undecorate):

```
use v5.44;
my @words = qw(apple fig pear);
my @by_length =
    map  { $_->[1] }
    sort { $a->[0] <=> $b->[0] }
    map  { [ length $_, $_ ] } @words;
say "@by_length";           # fig pear apple
```

Build a hash from a list, the dict-comprehension analogue (`{w: len(w) for w in words}`):

```
use v5.44;
my @words = qw(apple pear fig);
```

(continues on next page)

(continued from previous page)

```
my %len = map { $_ => length $_ } @words;
say "$_ => $len{$_}" for sort keys %len;
# apple => 5
# fig => 3
# pear => 4
```

And `pperl` is a command-line tool in the `awk`/`sed` niche. The implicit-loop one-liner is the Perl developer's everyday reflex:

```
pperl -lane '$s += $F[-1]; END { print $s }' data.txt
```

The gotcha: read the *One-Liners guide* for the switches (`-n`, `-p`, `-a`, `-l`, `-e`) and the dozens of recipes built on them; it is one of the highest-leverage things to learn coming from a shell-and-Python background.

Gotchas coming from Python

The traps, collected for a final scan before you start writing:

- **Sigils change with usage.** One element of `@a` is `$a[0]`; one value of `%h` is `$h{k}`. The container sigil and the element sigil differ.
- **`.` joins strings, `+` adds numbers.** Never `+` two strings to concatenate; that coerces them to numbers and warns.
- **Two comparison families.** `==`/`<`/`<=>` for numbers, `eq`/`lt`/`cmp` for strings. Pick by operand type, not by variable.
- **`sort` is string-sorted by default.** Pass `{ $a <=> $b }` for numeric order.
- **Context.** `my $x = @a` is the *length*; `my ($x) = @a` is the *first element*. The parentheses are the whole difference. Re-read *Context* when a value comes back the wrong shape.
- **Truthiness surprises.** `"0"` is false, but `"00"` and `"0.0"` are **true**. Only `0`, `""`, `"0"`, and `undef` are false.
- **`s///` edits in place** and returns a count; use the `/r` flag to get a new string instead.
- **`open` does not raise.** Use `open ... or die "...: $!"`.
- **`chomp` your line reads.** `<$fh>` keeps the trailing newline.
- **Non-ASCII source needs `use utf8;`.** Under `use v5.44;` a literal non-ASCII byte in your source is a compile error without it. Keep string data ASCII unless you specifically add `use utf8;`.
- **The `class` feature prints experimental notices to stderr** on `pperl`. Correct output, noisy error stream.

PHP to Perl

You know PHP. You reach for `$arr['key']` without thinking, `foreach ($items as $item)` is muscle memory, and you join strings with `.` because of course you do. Perl and PHP are close cousins: both are sigil-using, pragmatic, get-the-job-done languages,

and PHP borrowed heavily from Perl in its early days. This guide maps your PHP habits onto Perl so you can be productive in `pperl` in an afternoon.

Two shocks land in the first five minutes, so meet them now:

- **The `$` sigil is not the whole story.** In PHP every variable is `$x`, always. In Perl the sigil changes with how you *access* the value: `@a` is the whole array but `$a[0]` is one element; `%h` is the whole hash but `$h{k}` is one value. The dollar you already know marks a single scalar; `@` and `%` are new and section one is about nothing else.
- **PHP's one `array` becomes Perl's two types.** A PHP `array` is an ordered map that does duty as both list and dictionary. Perl splits that into `@array` (integer-indexed) and `%hash` (string-keyed), and you choose up front which one you have. This split drives the Arrays and Hashes sections.

Every example here runs on `pperl`, a Rust reimplementation of Perl 5.44. Each Perl snippet was executed on `pperl`; the `# ...` comments show its real output. Start your scripts with:

```
use v5.44;
```

That one line turns on `strict` (no accidental globals), `warnings`, the `say` built-in (a `print` with a trailing newline, like PHP's `echo` plus `PHP_EOL`), subroutine signatures, and the modern feature set used throughout this guide.

How to read this

- *Variables and sigils* - the `$`/`@`/`%` shift.
- *Strings* - interpolation and `.` concatenation.
- *Numbers* - `==` vs `eq`, the classic PHP trap.
- *Arrays and lists* - half of your `array`.
- *Hashes* - the other half of your `array`.
- *Context* - the concept PHP has no word for. Read this one.
- *Truthiness* - what counts as false, with surprises.
- *Control flow* - `if`, loops, the postfix forms.
- *Subroutines* - `function` becomes `sub`.
- *References* - nested data and closures.
- *Regular expressions* - regex as syntax, not `preg_*`.
- *File and stream I/O* - `fopen`, line reads, slurp.
- *Modules and packages* - `require`/`use` becomes `use`.
- *Objects* - the modern `class` syntax.
- *Error handling* - `try`/`catch`, the same idea.
- *Idioms* - the daily one-liners.
- *Gotchas coming from PHP* - scan before you write.

Variables and sigils

You already use `$`, so the first half of Perl's sigils is free. The new idea is that **the sigil marks the access, not the variable**. The whole array is `@a`; one element is `$a[0]`. The whole hash is `%h`; one value is `$h{k}`. PHP keeps `$` on everything and varies the brackets (`$a[0]`, `$h['k']`); Perl varies the sigil instead. `my` declares a lexically scoped variable, the equivalent of an ordinary local binding.

```
$name = "Ada";
$langs = ["PHP", "Perl"];
$ages = ["Ada" => 36, "Rasmus" => 56];
```

```
use v5.44;
my $name  = "Ada";
my @langs = ("PHP", "Perl");
my %ages  = (Ada => 36, Rasmus => 56);
say $name;            # Ada
say $langs[0];        # PHP
say $ages{Ada};       # 36
say scalar @langs;    # 2
```

The gotcha: when you reach *into* an aggregate for one element, the sigil follows the value you get back, not the container. A single element of `@langs` is a scalar, so it is `$langs[0]`, not `@langs[0]`; one value from `%ages` is `$ages{Ada}`. Read the leading sigil as "what shape am I getting" - `$` for one thing, `@` for a list, `%` for the whole map - and the brackets as "from where."

Strings

Concatenation is `.`, exactly as in PHP. Double quotes interpolate variables, single quotes do not - the same split PHP makes between `"..."` and `'...'`. There is no `sprintf`-into-the-string syntax to learn because the default double-quoted string already interpolates.

```
$name = "Ada";
$greeting = "Hello, $name";
$literal  = 'Hello, $name';
$joined   = "AB" . "CD";
$shout    = str_repeat("AB", 3);
```

```
use v5.44;
my $name = "Ada";
my $greeting = "Hello, $name";   # Hello, Ada
my $literal  = 'Hello, $name';   # Hello, $name  (no interpolation)
my $joined   = "AB" . "CD";      # ABCD
my $shout    = "AB" x 3;         # ABABAB
```

The gotcha: string repetition is the `x` operator, not a function - `"AB" x 3`, not `str_repeat`. Interpolating an array or hash element needs no curly-brace escape: `"First: $langs[0]"` and `"Name: $h{name}"` both work as written, where PHP would make you write `"{$arr['k']}"`. To interpolate the result of an *expression*, wrap it in `@{[`

`... ]}`: `"sum: @{[ 2 + 3 ]}"` prints `sum: 5`.

Numbers

Perl has one numeric type at the surface; you do not declare `int` vs `float`. The thing that will actually bite you is operators, so read the gotcha twice.

```
echo 7 / 2;           // 3.5
echo intdiv(7, 2);    // 3
echo 2 ** 10;         // 1024
echo 10 % 3;          // 1
```

```
use v5.44;
say 7 / 2;          # 3.5
say int(7 / 2);     # 3
say 2 ** 10;        # 1024
say 10 % 3;         # 1
```

The gotcha: Perl has **two operator families and you pick by operand type, not by the variable**. Numbers use `==`, `!=`, `<`, `>`, `<=>`; strings use `eq`, `ne`, `lt`, `gt`, `cmp`. And `+` is *always* numeric - there is no operator overloading the way PHP's `+` does array union. Concatenate with `.`, add with `+`, and never cross them:

```
use v5.44;
say "12" . "3";     # 123   (string concat)
say "12" + "3";     # 15    (numeric add)
say "10" == "10.0" ? "same" : "diff";   # same  (numeric compare)
say "10" eq "10.0" ? "same" : "diff";   # diff  (string compare)
```

PHP retired the loose `==` mess in modern versions; Perl never had it, but it solves the same problem by making you say which comparison you mean. `"10" == "10.0"` is true (numeric), `"10" eq "10.0"` is false (string). Using `==` on strings, or `eq` on numbers, is the single most common PHP-to-Perl bug.

Arrays and lists

Here is where PHP's one `array` splits in two. When you used an `array` as a numerically indexed list - `$items[0]`, `array_push`, `foreach` over values - that is a Perl **array**, declared with `@`. The verbs are renamed but the operations are the same, and negative indices work as you expect.

```
$nums = [1, 2, 3, 4, 5];
$nums[] = 6;
$last = array_pop($nums);
echo count($nums);              // 5
print_r(array_slice($nums, 1, 2)); // [2, 3]
```

```
use v5.44;
my @nums = (1, 2, 3, 4, 5);
push @nums, 6;
```

(continues on next page)

(continued from previous page)

```
my $last = pop @nums;       # 6
say scalar @nums;           # 5
say "@nums[1,2]";           # 2 3   (a slice)
```

Your `array_map` and `array_filter` become `map` and `grep`. `map` transforms (PHP's `array_map`); `grep` filters (PHP's `array_filter`). `$_` is the current element, Perl's implicit loop variable, the way the callback's parameter is in PHP - except you do not name it.

```
$squares = array_map(fn($x) => $x * $x, $nums);
$evens   = array_filter($nums, fn($x) => $x % 2 == 0);
rsort($nums);   // descending, in place
```

```
use v5.44;
my @nums    = (1, 2, 3, 4, 5);
my @squares = map  { $_ * $_ }    @nums;  # 1 4 9 16 25
my @evens   = grep { $_ % 2 == 0 } @nums; # 2 4
my @desc    = sort { $b <=> $a }  @nums;  # 5 4 3 2 1
```

The gotcha: `sort` compares as strings by default, so `sort (10, 9, 100)` gives `10 100 9`. For numbers you must pass a comparator: `sort { $a <=> $b } @nums`. `$a` and `$b` are the two elements being compared, and `<=>` is the numeric three-way operator (`cmp` is its string sibling). Note also that `grep` keeps elements, returning a new list, where PHP's `array_filter` preserves keys - Perl arrays have no keys to preserve, so the result is simply re-indexed.

Hashes

The other half of PHP's `array` - the part you used with string keys, `$config['host']`, `array_key_exists`, `foreach ($h as $k => $v)` - is a Perl **hash**, declared with `%` and accessed one value at a time with `$h{key}`.

```
$h = ["apple" => 3, "pear" => 5, "plum" => 2];
echo $h["apple"];                  // 3
$keys = array_keys($h);
sort($keys);
echo array_key_exists("pear", $h) ? "yes" : "no";  // yes
unset($h["plum"]);
foreach ($h as $k => $v) {
    echo "$k => $v\n";
}
```

```
use v5.44;
my %h = (apple => 3, pear => 5, plum => 2);
say $h{apple};                          # 3
say join ",", sort keys %h;             # apple,pear,plum
say exists $h{pear} ? "yes" : "no";     # yes
delete $h{plum};
for my $k (sort keys %h) {
    say "$k => $h{$k}";                 # apple => 3 ... pear => 5
```

(continues on next page)

(continued from previous page)

```
}
```

The gotcha: membership is `exists $h{key}` (the key is present) or `defined $h{key}` (the key is present *and* its value is not undef) - between them they cover PHP's `array_key_exists` and `isset`. The fat comma `=>` is the idiomatic key/value separator and auto-quotes the bareword on its left, so `apple => 3` needs no quotes around `apple`. A Perl hash has **no insertion order** (unlike a PHP array, which keeps it); iterate `sort keys %h` when you want a stable order. To walk key and value together, PHP's `foreach ($h as $k => $v)` becomes either `for my $k (keys %h)` or, when you want both at once, `while (my ($k, $v) = each %h)`.

Context

This is the chapter to read twice. PHP has no equivalent, and almost every "Perl is weird" story traces back to it.

Every expression in Perl runs in either **list context** or **scalar context**, decided by where it appears, not by what it is. The same array behaves differently in each. Assigning an array to a *scalar* gives its length; assigning it to a *list* (a parenthesised set of variables) copies elements.

```
use v5.44;
my @list = (10, 20, 30);

my $count = @list;          # scalar context -> 3 (the length)
say $count;                 # 3

my ($first) = @list;        # list context -> first element copied
say $first;                 # 10

my ($x, $y) = @list;        # 10 and 20
say "$x $y";                # 10 20
```

The parentheses on the left are the whole difference. `my $x = @list` asks "how many?" and gets 3 - this is where `count($arr)` lives, just spelled by context instead of a function. `my ($x) = @list` asks "give me the elements" and gets `10`. A PHP developer reads both as "assign the array," which is exactly the trap.

A subroutine can ask which context it was called in, with `wantarray`, and return different things accordingly:

```
use v5.44;
sub items {
    return wantarray ? (1, 2, 3) : "three items";
}
my @got = items();          # list context
my $msg = items();          # scalar context
say "@got";                 # 1 2 3
say $msg;                   # three items
```

Built-ins use this too. `scalar @array` forces scalar context to get a count - that is the

explicit `count()` you were looking for. A regex match with `/g` returns the list of matches in list context and a true/false in scalar context.

The gotcha: there is no single answer to "what does this array do" - it depends on the surrounding expression. When something returns a number you expected to be a list (or vice versa), context is the first suspect. Reach for `scalar` to force a count and parentheses on the left to force element copying.

Truthiness

Perl's false values are a short, specific list: the number `0`, the empty string `""`, the string `"0"`, and `undef` (Perl's `null`). **Everything else is true.** This is close to PHP's rules but not identical, and the differences are quiet.

```
use v5.44;
for my $v (0, 1, "", "0", "00", "0.0", "false") {
    say "'$v' is ", ($v ? "true" : "false");
}
```

This prints, exactly:

```
'0' is false
'1' is true
'' is false
'0' is false
'00' is true
'0.0' is true
'false' is true
```

The gotcha: `"00"` and `"0.0"` are **true** in Perl, even though PHP treats `"0"` and only `"0"` as the falsy string too - Perl agrees on `"0"` but not on its look-alikes. The string `"false"` is true (it is a non-empty, non-`"0"` string), the same as in PHP. And unlike PHP's empty-array-is-falsy rule, an empty Perl array is not itself a false *value*: in boolean context an array yields its length, so `if (@list)` is true exactly when the list is non-empty, which is the test you want. For "use this if defined, else a default," use defined-or `//`, the equivalent of PHP's `??`, which falls back only on `undef` (not on `0` or `""`):

```
use v5.44;
my %cfg = (host => "localhost");
my $port = $cfg{port} // 8080;   # key absent -> undef -> 8080
say $port;                       # 8080
```

Control flow

The keywords are familiar; the spelling differs in two small ways. `elseif` is `elsif` (one e), and conditions need parentheses while blocks need braces - there is no `:`/`endif` alternate syntax.

```
if ($n < 0) {
    $sign = "neg";
} elseif ($n == 0) {
```

(continues on next page)

(continued from previous page)

```
    $sign = "zero";
} else {
    $sign = "pos";
}

foreach ($nums as $x) {
    echo $x;
}

while ($running) {
    step();
}
```

```
use v5.44;
my $sign;
if    ($n < 0)  { $sign = "neg"  }
elsif ($n == 0) { $sign = "zero" }
else            { $sign = "pos"  }

for my $x (@nums) { say $x }

while ($running) { step() }
```

Perl adds two things PHP does not have. **Postfix modifiers** let a simple statement carry its own condition or loop, which reads naturally for one-liners:

```
use v5.44;
say $_ for @nums;                # loop, one statement
say "negative" if $n < 0;        # guarded statement
$sum += $_ for @nums;            # accumulate
```

And `unless`/`until` are the negated forms of `if`/`while`, for when the positive phrasing reads awkwardly: `say "ok" unless $error`.

The gotcha: the ternary is `?`/`:`, the same as PHP's, and chains the same way: `my $sign = $n < 0 ? "neg" : $n == 0 ? "zero" : "pos"`. Perl has no `switch`/`match` statement in core; a chained ternary or a dispatch hash (`$handler{$key}->()`) is the idiom, and a `for`-on-a-single-value block is the common substitute for a `switch`.

Subroutines

PHP's `function` becomes Perl's `sub`. Modern Perl has signatures that look much like PHP's parameter lists, including default values.

```
function greet($name, $greeting = "Hello") {
    return "$greeting, $name!";
}

function total(...$nums) {
```

(continues on next page)

(continued from previous page)

```
    return array_sum($nums);
}
```

```
use v5.44;
sub greet ($name, $greeting = "Hello") {
    return "$greeting, $name!";
}
say greet("Ada");           # Hello, Ada!
say greet("Rasmus", "Hi");  # Hi, Rasmus!

sub total (@nums) {
    my $sum = 0;
    $sum += $_ for @nums;
    return $sum;
}
say total(1, 2, 3, 4);      # 10
```

The variadic `...$nums` becomes a trailing array parameter `@nums`, which gathers all remaining arguments. PHP's named arguments (or the common `$opts` array convention) map to a hash parameter, since `key => value` pairs flatten into a list that a `%opts` signature gathers up:

```
use v5.44;
sub connect_to (%opts) {
    return "host=$opts{host} port=$opts{port}";
}
say connect_to(host => "localhost", port => 8080);
# host=localhost port=8080
```

The gotcha: a sub returns the value of its last expression even without `return`, and the return is **context-sensitive** (see *Context*). If you `return @list`, the caller gets the list in list context but the *count* in scalar context. Be deliberate about what you return when it might be used either way.

References

PHP arrays nest directly: an `array` can hold another `array` and you just keep subscripting. Perl arrays and hashes do not nest directly; you nest them through **references**, scalars that point at an aggregate. This is the same machinery as PHP's `&$x` references and its object handles, but in Perl you reach for it whenever you build nested data. Take a reference with `\`, follow it with `->`.

```
$data = ["name" => "Ada", "langs" => ["Perl", "PHP"]];
echo $data["langs"][1];      // PHP
$data["langs"][] = "Ruby";
```

```
use v5.44;
my $data = { name => "Ada", langs => ["Perl", "PHP"] };
say $data->{name};                  # Ada
```

(continues on next page)

(continued from previous page)

```
say $data->{langs}[1];              # PHP
push $data->{langs}->@*, "Ruby";    # postfix deref to push
say "@{$data->{langs}}";           # Perl PHP Ruby
```

`{ ... }` builds an anonymous hash reference and `[ ... ]` an anonymous array reference - the literal building blocks of nested data, matching the `[...]` and `[k => v]` literals you already write in PHP. The `->` arrow follows a reference one step; between two subscripts it is optional, so `$data->{langs}[1]` and `$data->{langs}->[1]` are the same. The arrow is also exactly the method-call arrow you know from PHP OO - same symbol, both for "reach inside" and "call a method."

Anonymous subs are closures, exactly like PHP's `fn` and `function() use (...)`, but with no `use` clause needed - they capture lexicals automatically:

```
use v5.44;
my $double = sub ($n) { $n * 2 };
say $double->(21);          # 42
```

The gotcha: the postfix deref forms (`->@*`, `->%*`, `->$*`) are the modern, readable way to dereference a whole aggregate, and `"@{ ... }"` is how you interpolate a dereferenced array into a string. The older `@$ref` and `${$ref}{key}` forms still work and appear in existing code, but reach for the arrow and postfix forms in new code.

Regular expressions

In PHP regex is the `preg_*` function family with pattern strings like `'/\d+/'`. In Perl it is **syntax**: `=~` binds a pattern to a string, `m//` matches, `s///` substitutes - no delimiters-inside-a-string, no function call. Capture groups land in `$1`, `$2`, and named captures in `%+`.

```
if (preg_match('/(\d{4})-(\d{2})-(\d{2})/', $s, $m)) {
    [$_, $year, $month, $day] = $m;
}
$t = preg_replace('/,/', ';', "a,b,c");
$parts = explode(",", "one,two,three");
```

```
use v5.44;
my $s = "2026-06-20";
if ($s =~ /(\d{4})-(\d{2})-(\d{2})/) {
    say "$1 $2 $3";                 # 2026 06 20
}
if ($s =~ /(?<year>\d{4})/) {
    say $+{year};                   # 2026
}
(my $t = "a,b,c") =~ s/,/;/g;       # a;b;c
my @parts = split /,/, "one,two,three";   # one two three
```

A `/g` match in list context returns *all* captures, the direct analogue of `preg_match_all`:

```
use v5.44;
my @nums = "x1y2z3" =~ /(\d)/g;     # 1 2 3
say "@nums";
```

The gotcha: `s///` modifies its target *in place* and returns the number of substitutions, not a new string - unlike `preg_replace`, which returns a fresh string. That is why the example copies first with `(my $t = "a,b,c")` and then substitutes on the copy - otherwise you would be editing a string literal. To get a new string without touching the original (the `preg_replace` reflex), use the `/r` flag: `my $t = $s =~ s/,/;/gr`.

File and stream I/O

The idiom is the three-argument open with a lexical filehandle, the counterpart of PHP's `fopen` into a `$handle`. The angle-bracket operator `<$fh>` reads a line, the same role as `fgets`.

```
$wh = fopen("/tmp/sample.txt", "w");
for ($i = 1; $i <= 3; $i++) {
    fwrite($wh, "line $i\n");
}
fclose($wh);

$rh = fopen("/tmp/sample.txt", "r");
while (($line = fgets($rh)) !== false) {
    echo "got: " . rtrim($line) . "\n";
}
fclose($rh);
```

```
use v5.44;
my $path = "/tmp/sample.txt";

open my $wh, '>', $path or die "cannot write: $!";
print {$wh} "line $_\n" for 1 .. 3;
close $wh;

open my $rh, '<', $path or die "cannot read: $!";
while (my $line = <$rh>) {
    chomp $line;
    say "got: $line";       # got: line 1 ... got: line 3
}
close $rh;
```

To slurp every line at once, read the handle in list context - the moral equivalent of `file()`:

```
use v5.44;
open my $fh, '<', "/tmp/sample.txt" or die $!;
my @lines = <$fh>;          # all lines, including newlines
say scalar @lines;          # 3
```

The gotcha: a line read with `<$fh>` keeps its trailing newline; call `chomp` to strip it (the

`rtrim` reflex, but `chomp` removes only the record separator). And `open` returns false on failure rather than raising or warning, so the `or die "...: $!"` idiom is mandatory - `$!` is the system error message, Perl's `errno` (PHP's `error_get_last()`).

Modules and packages

PHP's `require`/`include` and Composer's autoloader become Perl's `use`. The namespace separator is `::`, not `\`, and a module `Foo::Bar` lives at `Foo/Bar.pm` on the include path `@INC` (Perl's `include_path`). A module's import list is a selection of names to bring into your namespace.

```
require 'vendor/autoload.php';
use App\Util\ListHelper;
$sum = ListHelper::sum(range(1, 10));
```

```
use v5.44;
use List::Util qw(sum0 max first);
say sum0(1 .. 10);                      # 55
say max(3, 7, 2);                       # 7
say first { $_ % 2 == 0 } (1, 3, 4, 5);  # 4
```

PHP's huge flat standard-function namespace (`count`, `in_array`, `array_map`, `implode`, `sprintf`, ...) maps in Perl partly to built-in operators and partly to a few everyday modules. The common translations:

PHP	Perl
`count($a)`	`scalar @a`
`in_array($x, $a)`	`grep { $_ eq $x } @a`
`array_map($f, $a)`	`map { ... } @a`
`array_filter($a, $f)`	`grep { ... } @a`
`implode(",", $a)`	`join ",", @a`
`explode(",", $s)`	`split /,/, $s`
`sprintf(...)`	`sprintf(...)` (same)
`isset($h[$k])`	`exists $h{$k}` / `defined $h{$k}`
`array_sum($a)`	`sum0 @a` (`List::Util`)

`List::Util`, `Scalar::Util`, `POSIX`, `File::Spec`, and `Data::Dumper` are the everyday modules; most of what you would pull from Packagist ships in core or on CPAN, the Perl package archive.

The gotcha: `use` runs at *compile time*, not at the point in the file where it appears, so the order of `use` lines relative to your code does not matter the way `require` placement can in PHP. Importing is also opt-in per name: `use List::Util qw(sum0 max)` brings exactly `sum0` and `max` into your namespace, nothing else - there is no global function table the way PHP exposes every built-in everywhere.

(Web request data - PHP's `$_GET`, `$_POST`, `$_SERVER` superglobals - has no language-level analogue in Perl; that is the job of a CGI or PSGI layer, not the core language, and is out of scope for this guide.)

Objects

For new code, Perl 5.44 has a first-class `class` syntax that will look familiar coming from PHP. Fields are declared with `field`, methods with `method`, and `$self` is available inside every method just as `$this` is in PHP (you do not list it as a parameter).

```
class Point {
    public function __construct(
        public int $x = 0,
        public int $y = 0,
    ) {}
    public function to_string(): string {
        return "({$this->x}, {$this->y})";
    }
    public function move(int $dx, int $dy): static {
        $this->x += $dx;
        $this->y += $dy;
        return $this;
    }
}

$p = new Point(x: 3, y: 4);
echo $p->to_string();        // (3, 4)
$p->move(1, 1);
echo $p->to_string();        // (4, 5)
```

```
use v5.44;
use feature 'class';

class Point {
    field $x :param = 0;
    field $y :param = 0;
    method to_string { return "($x, $y)" }
    method move ($dx, $dy) {
        $x += $dx;
        $y += $dy;
        return $self;
    }
}

my $p = Point->new(x => 3, y => 4);
say $p->to_string;          # (3, 4)
$p->move(1, 1);
say $p->to_string;          # (4, 5)
```

The constructor is generated for you, like PHP's promoted constructor properties: `:param` marks a field that the auto-built `new` accepts as a named argument, and `= 0` is its default. The method-call arrow `->` is the same one you use in PHP. Inheritance is `:isa`, the counterpart of `extends`:

```
use v5.44;
```

(continues on next page)

(continued from previous page)

```
use feature 'class';

class Point3D :isa(Point) {
    field $z :param = 0;
    method z { return $z }
}

my $q = Point3D->new(x => 1, y => 2, z => 3);
say $q->to_string, " z=", $q->z;   # (1, 2) z=3
```

The gotcha: the `class` feature is still marked experimental in Perl 5.44, and on `pperl` the "class is experimental" notices print to **stderr** even with `no warnings 'experimental::class'` in scope. They do not touch your program's output, so a script that prints to stdout behaves exactly as shown above; just expect the notices on the error stream. The older object model - a package, a hash blessed into it with `bless`, and `@ISA` for inheritance - is what you will meet in existing code and most of CPAN; for it, for roles, and for migration advice, see the *Object-oriented Programming guide*.

Error handling

PHP's `try`/`catch` has a direct counterpart in modern Perl, and it reads almost identically. The raising side is `die` (PHP's `throw`):

```
function risky($n) {
    if ($n < 0) {
        throw new Exception("negative!");
    }
    return sqrt($n);
}

try {
    risky(-4);
} catch (Exception $e) {
    echo "caught: " . $e->getMessage() . "\n";
}
```

```
use v5.44;
use feature 'try';

sub risky ($n) {
    die "negative!\n" if $n < 0;
    return sqrt $n;
}

try {
    risky(-4);
}
catch ($e) {
    print "caught: $e";     # caught: negative!
```

(continues on next page)

(continued from previous page)

```
}
```

The older, still-common form wraps code in `eval { }` and inspects `$@` (the caught error) afterward - the same pattern as PHP code written before exceptions were idiomatic:

```
use v5.44;
my $result = eval { risky(-1) };
if ($@) {
    print "caught: $@";     # caught: negative!
}
```

The gotcha: `die` with a string that ends in `\n` produces exactly that message; `die` with a string that does *not* end in a newline gets `" at SCRIPT line N.\n"` appended automatically. End your error strings with `\n` when you want a clean message and omit it when you want the file-and-line trace. Note also the two error variables: `$@` is the last caught exception (PHP's `$e`), while `$!` is the last *system* error (a failed open, the `errno` string).

Idioms

A few patterns you will see daily and want in your fingers.

Sort a list by a computed key (the Schwartzian transform - PHP's `usort` with a precomputed key, done in one expression: decorate, sort, undecorate):

```
use v5.44;
my @words = qw(apple fig pear);
my @by_length =
    map  { $_->[1] }
    sort { $a->[0] <=> $b->[0] }
    map  { [ length $_, $_ ] } @words;
say "@by_length";           # fig pear apple
```

Build a hash from a list, the `array_combine`/`array_column` analogue (here, word to its length):

```
use v5.44;
my @words = qw(apple pear fig);
my %len = map { $_ => length $_ } @words;
say "$_ => $len{$_}" for sort keys %len;
# apple => 5
# fig => 3
# pear => 4
```

And `pperl` is a command-line tool in the `awk`/`sed` niche - the same job you might reach for a quick PHP CLI script to do. The implicit-loop one-liner is the Perl developer's everyday reflex:

```
pperl -lane '$s += $F[-1]; END { print $s }' data.txt
```

The gotcha: read the *One-Liners guide* for the switches (`-n`, `-p`, `-a`, `-l`, `-e`) and the dozens

of recipes built on them; it is one of the highest-leverage things to learn coming from a PHP-and-shell background.

Gotchas coming from PHP

The traps, collected for a final scan before you start writing:

- **Sigils change with usage.** One element of `@a` is `$a[0]`; one value of `%h` is `$h{k}`. PHP keeps `$` everywhere; Perl varies the sigil by what shape you are getting.
- **One PHP `array` is two Perl types.** Decide up front: integer keys and order means `@array`; string keys means `%hash`. They are not interchangeable.
- **Two comparison families.** `==`/`<`/`<=>` for numbers, `eq`/`lt`/`cmp` for strings. Pick by operand type, not by variable. `==` on strings or `eq` on numbers is the classic bug.
- **`.` joins strings, `+` only adds numbers.** Perl's `+` never does array union or concatenation; mixing `.` and `+` corrupts your data silently or warns.
- **`sort` is string-sorted by default.** Pass `{ $a <=> $b }` for numeric order.
- **Hashes have no order.** Unlike a PHP array, iteration order is not insertion order; `sort keys %h` when you need stability.
- **Context.** `my $x = @a` is the *count*; `my ($x) = @a` is the *first element*. The parentheses are the whole difference. Re-read *Context* when a value comes back the wrong shape.
- **Truthiness surprises.** `"0"` is false, but `"00"` and `"0.0"` are **true**. Only `0`, `""`, `"0"`, and `undef` are false. An empty array is not a false value - `if (@a)` tests non-empty.
- **`s///` edits in place** and returns a count, unlike `preg_replace`; use the `/r` flag to get a new string instead.
- **`open` does not raise.** Use `open ... or die "...: $!"`.
- **`chomp` your line reads.** `<$fh>` keeps the trailing newline.
- **Web superglobals are not core.** `$_GET`/`$_POST` belong to a CGI or PSGI layer, not the language.
- **Non-ASCII source needs `use utf8;`.** Under `use v5.44;` a literal non-ASCII byte in your source is a compile error without it. Keep string data ASCII unless you specifically add `use utf8;`.
- **The `class` feature prints experimental notices to stderr** on `pperl`. Correct output, noisy error stream.

Ruby to Perl

You know Ruby. You pass blocks to `each` and `map` without thinking, you reach for a `Hash` by reflex, and `arr.select { |x| ... }` is muscle memory. This guide maps each of those habits onto Perl, so you can be productive in `pperl` in an afternoon instead of relearning programming from scratch.

Two shifts land in the first five minutes, so meet them now:

- **Sigils mark the type, not the scope.** Ruby uses `@var` for instance state, `$var` for globals, and a bare `var` for locals. Perl uses `$x` for a single value, `@x` for an ordered

list, %x for a key/value map - the sigil tells you the *shape* of the data, and it is the same whether the variable is local or global. Section one is about nothing else.

- **Not everything is an object.** In Ruby `"foo".upcase` and `[1, 2].sum` are method calls because numbers, strings, and arrays are all objects. Perl is pragmatic: a scalar is not an object, there is no universal base class, and those operations are plain functions - `uc "foo"`, `sum @a`. You call methods only on things you blessed into a class.

Every example here runs on `pperl`, a Rust reimplementation of Perl 5.44. Each Perl snippet was executed on `pperl`; the `# ...` comments show its real output. Start your scripts with:

```
use v5.44;
```

That one line turns on `strict` (no accidental globals), `warnings`, the `say` built-in (a `print` with a trailing newline, like Ruby's `puts`), subroutine signatures, and the modern feature set used throughout this guide.

How to read this

- *Variables and sigils* - the $/@/% shift.
- *Strings* - interpolation and concatenation.
- *Numbers* - one surface type, float division.
- *Arrays and lists* - your `Array`, plus block-taking `map`/`grep`.
- *Hashes* - your `Hash`, and where symbols go.
- *Context* - the concept Ruby has no word for. Read this one.
- *Truthiness* - what counts as false, and it differs sharply.
- *Control flow* - `if`, loops, the postfix forms.
- *Subroutines* - `def` becomes `sub`.
- *References* - nested data, and blocks as values.
- *Regular expressions* - regex as syntax, like Ruby's.
- *File and stream I/O* - open, line reads, slurp.
- *Modules and packages* - `require` becomes `use`.
- *Objects* - the modern `class` syntax.
- *Error handling* - `begin`/`rescue` becomes `eval` or `try`/`catch`.
- *Idioms* - the daily one-liners.
- *Gotchas coming from Ruby* - scan before you write.

Variables and sigils

In Ruby a sigil tells you *scope*: `@x` is an instance variable, `$x` a global, `x` a local. In Perl the **sigil tells you shape**: `$` for a single value, `@` for an ordered list, `%` for a key/value map. `my` declares a lexically scoped variable, the Perl equivalent of an ordinary local binding - it is how you get a local in Perl, where Ruby just uses a bare name.

```ruby
name  = "Ada"
langs = ["Ruby", "Perl"]
ages  = {"Ada" => 36, "Yukihiro" => 60}
```

```perl
use v5.44;
my $name  = "Ada";
my @langs = ("Ruby", "Perl");
my %ages  = (Ada => 36, Yukihiro => 60);
say $name;            # Ada
say $langs[0];        # Ruby
say $ages{Ada};       # 36
```

The gotcha: when you reach *into* an aggregate for one element, the sigil follows the value you get back, not the container. A single element of `@langs` is a scalar, so it is `$langs[0]`, not `@langs[0]`; one value from `%ages` is `$ages{Ada}`. The `@`/`%` form is the whole collection; the `$` form is one piece of it. (And forget the Ruby mapping: a Perl `$x` is never "a global" the way Ruby's `$x` is - the `$` here just means "one value.")

Strings

Double quotes interpolate variables, single quotes do not, exactly like Ruby's `"..."` versus `'...'`. The Ruby `#{expr}` form has a direct counterpart for plain variables and a slightly longer one for arbitrary expressions. Concatenation is `.`, not `+`.

```ruby
name     = "Ada"
greeting = "Hello, #{name}"
total    = "Total: #{6 * 7}"
shout    = "AB" * 3
```

```perl
use v5.44;
my $name = "Ada";
my $greeting = "Hello, $name";     # Hello, Ada
my $literal  = 'Hello, $name';     # Hello, $name  (no
 ↪interpolation)
my $total    = "Total: @{[ 6 * 7 ]}";  # Total: 42
my $joined   = "AB" . "CD";        # ABCD
my $shout    = "AB" x 3;           # ABABAB
```

The gotcha: a bare `"$name"` interpolates a variable just like Ruby's `#{name}`, but to interpolate an *expression* you wrap it in the `@{[ ... ]}` form (it builds a throwaway array and inlines it). String repetition is `x`, not `*`, and the join operator is `.`, not `+`; `+` on strings coerces them to numbers and warns. Array elements and hash values interpolate inside double quotes too: `"First: $langs[0]"` works as written.

Numbers

Perl has one numeric type at the surface; you do not distinguish `Integer` from `Float`. Division always produces a float, like Ruby's `Float` division when either operand is a float.

```
puts 7.0 / 2        # 3.5
puts 7 / 2          # 3   (integer division in Ruby)
puts 2 ** 10        # 1024
puts 10 % 3         # 1
```

```
use v5.44;
say 7 / 2;          # 3.5   (always float)
say int(7 / 2);     # 3
say 2 ** 10;        # 1024
say 10 % 3;         # 1
```

The gotcha: comparison and equality operators come in two families, and you pick by the *operands' type*, not the variables'. Numbers use `==`, `!=`, `<`, `>`, `<=>`; strings use `eq`, `ne`, `lt`, `gt`, `cmp`. This has no Ruby analogue - Ruby's `==` dispatches on the object's class, while Perl's `==` always means *numeric* equality. `"10" == "10.0"` is true (numeric), but `"10" eq "10.0"` is false (string). Using `==` on strings, or `eq` on numbers, is a frequent and quiet bug.

Arrays and lists

A Perl array is your Ruby `Array`. The big shift is method calls becoming functions: Ruby's `arr.push(x)`, `arr.pop`, `arr.length` become `push @arr, x`, `pop @arr`, `scalar @arr` - the array is an argument, not a receiver.

```
nums = [1, 2, 3, 4, 5]
nums.push(6)
last = nums.pop          # 6
puts nums.length         # 5
puts nums[1, 2].join(" ")  # 2 3
```

```
use v5.44;
my @nums = (1, 2, 3, 4, 5);
push @nums, 6;
my $last = pop @nums;       # 6
say scalar @nums;           # 5
say "@nums[1,2]";           # 2 3   (a slice)
```

Ruby's block-taking iterators are the richest part of this mapping. `map`, `select`, and `sort_by` become Perl `map`, `grep`, and `sort`, which take a `{ ... }` block exactly the way Ruby methods do. Inside the block, `$_` is the current element - Perl's implicit block variable, the role Ruby's `|x|` parameter plays.

```
squares = nums.map    { |x| x * x }
evens   = nums.select { |x| x.even? }
desc    = nums.sort   { |a, b| b <=> a }
```

```
use v5.44;
my @nums    = (1, 2, 3, 4, 5);
my @squares = map  { $_ * $_ }    @nums;  # 1 4 9 16 25
my @evens   = grep { $_ % 2 == 0 } @nums; # 2 4
my @desc    = sort { $b <=> $a }  @nums;  # 5 4 3 2 1
```

The gotcha: `sort` compares as strings by default, so `sort (10, 9, 100)` gives `10 100 9`. For numbers you must pass a comparator: `sort { $a <=> $b } @nums`. `$a` and `$b` are the two elements being compared (the equivalent of Ruby's `|a, b|`), and `<=>` is the numeric three-way operator - the very same spaceship you know from Ruby (`cmp` is its string sibling).

Hashes

A Perl hash is your Ruby `Hash`. Declared with `%`, accessed one value at a time with `$h{key}`. The fat comma `=>` you already use in Ruby hash literals works here too, and it auto-quotes the bareword on its left.

```
h = {apple: 3, pear: 5, plum: 2}
puts h[:apple]                          # 3
puts h.keys.sort.join(",")              # apple,pear,plum
puts h.key?(:pear)                      # true
h.delete(:plum)
h.sort.each { |k, v| puts "#{k} => #{v}" }
```

```
use v5.44;
my %h = (apple => 3, pear => 5, plum => 2);
say $h{apple};                          # 3
say join ",", sort keys %h;             # apple,pear,plum
say exists $h{pear} ? "yes" : "no";     # yes
delete $h{plum};
for my $k (sort keys %h) {
    say "$k => $h{$k}";
}
```

The gotcha: Perl has **no symbol type**. Where Ruby reaches for `:apple` as a lightweight, interned key, Perl just uses the string `"apple"` - and the `=>` fat comma auto-quotes it, so `apple => 3` and `$h{apple}` need no quotes. A Ruby symbol and a Ruby string are different objects; in Perl there is only the string, so the `:sym` versus `"sym"` distinction simply vanishes. Membership is `exists $h{key}`, not Ruby's `key?`, and a hash has no order - iterate `sort keys %h` when you want a stable one.

Context

This is the chapter to read twice. Ruby has no equivalent, and almost every "Perl is weird" story traces back to it.

Every expression in Perl runs in either **list context** or **scalar context**, decided by where it appears, not by what it is. The same array behaves differently in each. Assigning an array to a *scalar* gives its length; assigning it to a *list* (a parenthesised set of variables) copies elements. Ruby never overloads assignment this way - `a = arr` always

just aliases the array - so this is genuinely new.

```
use v5.44;
my @list = (10, 20, 30);

my $count = @list;          # scalar context -> 3 (the length)
say $count;                 # 3

my ($first) = @list;        # list context -> first element copied
say $first;                 # 10

my ($a, $b) = @list;        # 10 and 20
say "$a $b";                # 10 20
```

The parentheses on the left are the whole difference. `my $x = @list` asks "how many?" and gets 3. `my ($x) = @list` asks "give me the elements" and gets `10`. The Ruby reflex reads both as "assign the array," which is exactly the trap.

A subroutine can ask which context it was called in, with `wantarray`, and return different things accordingly:

```
use v5.44;
sub items {
    return wantarray ? (1, 2, 3) : "three items";
}
my @got = items();          # list context
my $msg = items();          # scalar context
say "@got";                 # 1 2 3
say $msg;                   # three items
```

Built-ins use this too. `scalar @array` forces scalar context to get a count; a regex match with `/g` returns the list of matches in list context and a true/false in scalar context. The classic idiom for "count the matches" forces a list assignment in scalar context with the so-called *countof* form:

```
use v5.44;
my $n = () = "a1b2c3" =~ /\d/g;
say $n;                     # 3
```

The gotcha: there is no single answer to "what does this array do" - it depends on the surrounding expression. When something returns a number you expected to be a list (or vice versa), context is the first suspect. Reach for `scalar` to force a count and parentheses on the left to force element copying.

Truthiness

This is the sharpest break from Ruby in the whole guide. In Ruby only `nil` and `false` are falsy - `0` and `""` are both **true**. In Perl the false values are a short, specific list: the number `0`, the empty string `""`, the string `"0"`, and `undef` (Perl's `nil`). Everything else is true.

```
use v5.44;
for my $v (0, 1, "", "0", "00", "0.0", "false") {
    say "'$v' is ", ($v ? "true" : "false");
}
```

This prints, exactly:

```
'0' is false
'1' is true
'' is false
'0' is false
'00' is true
'0.0' is true
'false' is true
```

The gotcha: `0` and `""` are **false** in Perl but **true** in Ruby - the single most disruptive habit to unlearn. A Ruby guard like `if count` fires for `count == 0`; the Perl `if ($count)` does not. On the other side, `"00"` and `"0.0"` are **true** in Perl even though they look numerically zero, because only the exact string `"0"` is false. And unlike Ruby, where you might lean on `nil` checks, Perl distinguishes "false" from "undefined" with defined-or `//`, which falls back only on `undef` (not on `0` or `""`):

```
use v5.44;
my %cfg = (host => "localhost");
my $port = $cfg{port} // 8080;   # key absent -> undef -> 8080
say $port;                       # 8080
```

Control flow

The keywords are familiar; the spelling differs. `elsif` is Perl's `elsif` (Ruby's `elsif` too - one of the few exact matches), and blocks are braced, with no `then` and no terminating `end`.

```
if    n < 0  then sign = "neg"
elsif n == 0 then sign = "zero"
else              sign = "pos"
end

nums.each { |x| puts x }

while running
  step
end
```

```
use v5.44;
my $sign;
if    ($n < 0)  { $sign = "neg"  }
elsif ($n == 0) { $sign = "zero" }
else            { $sign = "pos"  }
```

(continues on next page)

(continued from previous page)

```
for my $x (@nums) { say $x }

while ($running) { step() }
```

Perl's **postfix modifiers** are the direct counterpart of Ruby's trailing `puts x if cond` and `do_it while cond` - a simple statement carrying its own condition or loop:

```
use v5.44;
say $_ for @nums;                # loop, one statement
say "negative" if $n < 0;        # guarded statement
$sum += $_ for @nums;            # accumulate
```

And `unless`/`until` are the negated forms of `if`/`while`, just like Ruby's `unless`/`until`: `say "ok" unless $error`.

The gotcha: the ternary uses `?`/`:`, not Ruby's `cond ? a : b` - which is actually identical, so this one is free. There is no `x if c else y` expression form; chain ternaries instead: `my $sign = $n < 0 ? "neg" : $n == 0 ? "zero" : "pos"`.

Subroutines

`def` becomes `sub`. Modern Perl has signatures that look much like Ruby's parameter lists, including defaults. And the good news a Ruby developer wants to hear: **the last evaluated expression is the return value**, exactly as in Ruby - `return` is optional.

```
def greet(name, greeting = "Hello")
  "#{greeting}, #{name}!"
end

def total(*nums)
  nums.sum
end
```

```
use v5.44;
sub greet ($name, $greeting = "Hello") {
    return "$greeting, $name!";
}
say greet("Ada");           # Hello, Ada!
say greet("Yukihiro", "Hi"); # Hi, Yukihiro!

sub area ($w, $h) { $w * $h }   # no return needed; last expr wins
say area(3, 4);             # 12

sub total (@nums) {
    my $sum = 0;
    $sum += $_ for @nums;
    return $sum;
}
say total(1, 2, 3, 4);      # 10
```

Ruby keyword arguments map to a hash parameter, since `key => value` pairs flatten into a list that a `%opts` signature gathers up:

```
use v5.44;
sub connect_to (%opts) {
    return "host=$opts{host} port=$opts{port}";
}
say connect_to(host => "localhost", port => 8080);
# host=localhost port=8080
```

The gotcha: a sub returns the value of its last expression even without `return` (the Ruby reflex carries over), but in Perl that return is also **context-sensitive** (see *Context*). If you `return @list`, the caller gets the list in list context but the *count* in scalar context. Be deliberate about what you return when it might be used either way.

References

Perl's arrays and hashes do not nest directly; you nest them through **references**, a scalar that points at an aggregate. In Ruby every container is already a reference, so nesting is invisible to you there; in Perl you take a reference with `\` (or build one with `[ ]`/`{ }`) and follow it with `->`.

```
data = {name: "Ada", langs: ["Perl", "Ruby"]}
puts data[:langs][1]      # Ruby
data[:langs].push("Python")
```

```
use v5.44;
my $data = { name => "Ada", langs => ["Perl", "Ruby"] };
say $data->{name};                  # Ada
say $data->{langs}[1];              # Ruby
push $data->{langs}->@*, "Python";  # postfix deref to push
say "@{$data->{langs}}";           # Perl Ruby Python
```

`{ ... }` builds an anonymous hash reference and `[ ... ]` an anonymous array reference - the literal building blocks of nested data, matching Ruby's `{}` and `[]` literals. The `->` arrow follows a reference one step; between two subscripts it is optional, so `$data->{langs}[1]` and `$data->{langs}->[1]` are the same.

The payoff for a Ruby developer is blocks-as-values. A Ruby block, `proc`, or `lambda` becomes a Perl anonymous sub: a closure stored in a scalar, called with `->`. It closes over its lexicals exactly like a Ruby block:

```
use v5.44;
my $doubler = sub { map { $_ * 2 } @_ };
say join ",", $doubler->(1, 2, 3);  # 2,4,6

my $factor = 10;
my $scaler = sub ($x) { $x * $factor };  # closes over $factor
say $scaler->(5);                   # 50
```

The gotcha: a Perl anonymous sub is a value you must *store and call explicitly* with `->( )` - there is no implicit `yield`, no block passed invisibly alongside the arguments. When

a builtin wants a block (`map`/`grep`/`sort`, or `first`/`reduce` from `List::Util`), you write the `{ ... }` inline; when you want to pass a callable around as data, you make a `sub { ... }` and call it through its scalar. The postfix deref forms (`->@*`, `->%*`, `->$*`) are the modern, readable way to dereference a whole aggregate.

Regular expressions

Good news: Perl regex will feel like home. Ruby borrowed its regex literal `/.../`, its `=~` operator, and `$1`/`$2` capture variables straight from Perl. The match operator `m//`, the substitution `s///`, and named captures in `%+` round it out.

```ruby
s = "2026-06-20"
if s =~ /(\d{4})-(\d{2})-(\d{2})/
  puts "#{$1} #{$2} #{$3}"
end
t = "a,b,c".gsub(",", ";")
parts = "one,two,three".split(",")
```

```perl
use v5.44;
my $s = "2026-06-20";
if ($s =~ /(\d{4})-(\d{2})-(\d{2})/) {
    say "$1 $2 $3";                 # 2026 06 20
}
if ($s =~ /(?<year>\d{4})/) {
    say $+{year};                   # 2026
}
(my $t = "a,b,c") =~ s/,/;/g;       # a;b;c
my @parts = split /,/, "one,two,three";   # one two three
```

A `/g` match in list context returns *all* captures, the direct analogue of Ruby's `String#scan`:

```perl
use v5.44;
my @nums = "x1y2z3" =~ /(\d)/g;     # 1 2 3
say "@nums";
```

The gotcha: where Ruby's `gsub` returns a new string and leaves the original alone, Perl's `s///` modifies its target *in place* and returns the number of substitutions. That is why the example copies first with `(my $t = "a,b,c")` and then substitutes on the copy. To get a new string without touching the original - the `gsub` behaviour you expect - add the `/r` flag: `my $t = $s =~ s/,/;/gr`.

File and stream I/O

The idiom is the three-argument open with a lexical filehandle, the counterpart of Ruby's `File.open(path, mode) do |f| ... end`. The angle-bracket operator `<$fh>` reads a line, the same role as Ruby's `f.each_line` or `f.gets`.

```ruby
File.open("/tmp/sample.txt", "w") do |f|
  (1..3).each { |i| f.write("line #{i}\n") }
end
```

(continues on next page)

(continued from previous page)

```ruby
File.open("/tmp/sample.txt") do |f|
  f.each_line { |line| puts "got: #{line.chomp}" }
end
```

```perl
use v5.44;
my $path = "/tmp/sample.txt";

open my $wh, '>', $path or die "cannot write: $!";
print {$wh} "line $_\n" for 1 .. 3;
close $wh;

open my $rh, '<', $path or die "cannot read: $!";
while (my $line = <$rh>) {
    chomp $line;
    say "got: $line";       # got: line 1 ... got: line 3
}
close $rh;
```

To slurp every line at once, read the handle in list context:

```perl
use v5.44;
open my $fh, '<', "/tmp/sample.txt" or die $!;
my @lines = <$fh>;          # all lines, including newlines
say scalar @lines;          # 3
```

The gotcha: a line read with `<$fh>` keeps its trailing newline; call `chomp` to strip it - Ruby's `line.chomp` is the same verb, and Perl borrowed the name. And `open` returns false on failure rather than raising the way Ruby's `File.open` does, so the `or die "...: $!"` idiom is mandatory; `$!` is the system error message, Perl's equivalent of the message in Ruby's `Errno` exception.

Modules and packages

`require`/`include` becomes `use`. The namespace separator is `::`, which you already know from Ruby constants like `Set::Foo`; Perl uses it everywhere, for both packages and the names inside them. A module `Foo::Bar` lives at `Foo/Bar.pm` on the include path `@INC` (Perl's `$LOAD_PATH`). A module's import list is a selection of names, much like pulling specific methods into scope.

```ruby
require "set"
nums = [1, 2, 3, 4].sum
biggest = [3, 7, 2].max
```

```perl
use v5.44;
use List::Util qw(sum0 max first);
say sum0(1 .. 10);                      # 55
say max(3, 7, 2);                       # 7
say first { $_ % 2 == 0 } (1, 3, 4, 5);  # 4
```

The standard library is large, and much of what you would reach to RubyGems for ships in core or on CPAN, the Perl package archive. `List::Util`, `Scalar::Util`, `POSIX`, `File::Spec`, and `Data::Dumper` are the everyday ones.

The gotcha: `use` runs at *compile time*, not at the point in the file where it appears, so the order of `use` lines relative to your code does not matter the way `require` placement can. Importing is opt-in per name: `use List::Util qw(sum0 max)` brings exactly `sum0` and `max` into your namespace, nothing else - there is no monkey-patching of built-in types the way Ruby lets you reopen `Array`.

Objects

Ruby is everything-is-an-object; Perl is pragmatic, and the difference showed up already in *Arrays* (functions, not methods). But when you *do* want objects, Perl 5.44 has a first-class `class` syntax that will look familiar. Fields are declared with `field`, methods with `method`, and `$self` is available inside every method just as `self` is in Ruby (you do not list it as a parameter).

```ruby
class Point
  attr_accessor :x, :y
  def initialize(x: 0, y: 0)
    @x = x
    @y = y
  end
  def to_string
    "(#{@x}, #{@y})"
  end
  def move(dx, dy)
    @x += dx
    @y += dy
    self
  end
end

p = Point.new(x: 3, y: 4)
puts p.to_string           # (3, 4)
p.move(1, 1)
puts p.to_string           # (4, 5)
```

```perl
use v5.44;
use feature 'class';

class Point {
    field $x :param = 0;
    field $y :param = 0;
    method to_string { return "($x, $y)" }
    method move ($dx, $dy) {
        $x += $dx;
        $y += $dy;
        return $self;
    }
```

(continues on next page)

(continued from previous page)

```
}

my $p = Point->new(x => 3, y => 4);
say $p->to_string;          # (3, 4)
$p->move(1, 1);
say $p->to_string;          # (4, 5)
```

The constructor is generated for you, the equivalent of Ruby's `attr_accessor` plus `initialize` collapsed into the field declarations: `:param` marks a field that the auto-built `new` accepts as a named argument, and `= 0` is its default. Inheritance is `:isa`, the counterpart of Ruby's `class Point3D < Point`:

```
use v5.44;
use feature 'class';

class Point3D :isa(Point) {
    field $z :param = 0;
    method z { return $z }
}

my $q = Point3D->new(x => 1, y => 2, z => 3);
say $q->to_string, " z=", $q->z;   # (1, 2) z=3
```

The gotcha: the `class` feature is still marked experimental in Perl 5.44, and on `pperl` the "class is experimental" notices print to **stderr** even with `no warnings 'experimental::class'` in scope. They do not touch your program's output, so a script that prints to stdout behaves exactly as shown above; just expect the notices on the error stream. Note also that `field` gives you a `:reader` for read access but no Ruby-style `attr_accessor` setter by default - you write a `method` to mutate, as `move` does above. For the older `bless`-based object model you will meet in existing code, and for roles, inheritance details, and migration, see the *Object-oriented Programming guide*.

Error handling

Ruby's `begin`/`rescue`/`ensure` has two counterparts. The classic one wraps code in `eval { }` and inspects `$@` (the caught error) afterward - note that `eval` here is exception *trapping*, not Ruby's `eval` of a string:

```
begin
  risky(-1)
rescue => e
  puts "caught: #{e.message}"
end
```

```
use v5.44;
my $result = eval { risky(-1) };
if ($@) {
    print "caught: $@";     # caught: negative!
}
```

The modern form is `try/catch/finally`, which reads almost exactly like `begin/rescue/ensure`:

```
use v5.44;
use feature 'try';

try {
    risky(-4);
}
catch ($e) {
    print "caught: $e";     # caught: negative!
}
finally {
    say "done";             # done   (runs either way, like ensure)
}
```

where the raising side is `die` - Perl's `raise`:

```
use v5.44;
sub risky ($n) {
    die "negative!\n" if $n < 0;
    return sqrt $n;
}
```

The gotcha: `die` with a string that ends in `\n` produces exactly that message; `die` with a string that does *not* end in a newline gets `" at SCRIPT line N.\n"` appended automatically. End your error strings with `\n` when you want a clean message and omit it when you want the file-and-line trace. The `finally` block, like Ruby's `ensure`, runs whether or not an exception was thrown. Note also the two error variables: `$@` is the last caught exception (Ruby's `rescue => e`), while `$!` is the last *system* error (a failed open). Under `pperl`, the `try/catch/finally` feature prints an "experimental" notice on stderr; stdout is unaffected.

Idioms

A few patterns you will see daily and want in your fingers.

Sort a list by a computed key. Ruby has `sort_by` built in; the Perl hand-rolled form is the Schwartzian transform - decorate, sort, undecorate - and is worth recognising on sight:

```
use v5.44;
my @words = qw(apple fig pear);
my @by_length =
    map  { $_->[1] }
    sort { $a->[0] <=> $b->[0] }
    map  { [ length $_, $_ ] } @words;
say "@by_length";           # fig pear apple
```

Build a hash from a list, the Ruby `words.to_h { |w| [w, w.length] }` analogue:

```
use v5.44;
my @words = qw(apple pear fig);
```

(continues on next page)

(continued from previous page)

```
my %len = map { $_ => length $_ } @words;
say "$_ => $len{$_}" for sort keys %len;
# apple => 5
# fig => 3
# pear => 4
```

And `pperl` is a command-line tool in the `awk`/`sed` niche - the role `ruby -e` and `ruby -ne` fill for shell glue. The implicit-loop one-liner is the Perl developer's everyday reflex:

```
pperl -lane '$s += $F[-1]; END { print $s }' data.txt
```

The gotcha: read the *One-Liners guide* for the switches (`-n`, `-p`, `-a`, `-l`, `-e`) and the dozens of recipes built on them; it is one of the highest-leverage things to learn coming from a shell-and-Ruby background.

Gotchas coming from Ruby

The traps, collected for a final scan before you start writing:

- **Sigils mark shape, not scope.** `$x`/`@x`/`%x` say single/list/map; they are not Ruby's instance/global/local markers. One element of `@a` is `$a[0]`; one value of `%h` is `$h{k}`.
- **Not everything is an object.** `uc "foo"` and `sum @a` are functions, not methods on a string or array. You call methods only on blessed references.
- **Truthiness is the big one.** `0` and `""` are **false** in Perl but **true** in Ruby. `if ($count)` is false when `$count` is `0`. Only `0`, `""`, `"0"`, and `undef` are false; `"00"` and `"0.0"` are **true**.
- **No symbols.** `:name` has no Perl equivalent; hash keys are plain strings, and `=>` auto-quotes the bareword on its left.
- **`.` joins strings, `+` adds numbers.** Never `+` two strings to concatenate; that coerces them to numbers and warns. String repeat is `x`, not `*`.
- **Two comparison families.** `==`/`<`/`<=>` for numbers, `eq`/`lt`/`cmp` for strings. Pick by operand type, not by variable - unlike Ruby's class-dispatched `==`.
- **`sort` is string-sorted by default.** Pass `{ $a <=> $b }` for numeric order.
- **Context.** `my $x = @a` is the *length*; `my ($x) = @a` is the *first element*. The parentheses are the whole difference. Re-read *Context* when a value comes back the wrong shape.
- **`s///` edits in place** and returns a count, unlike Ruby's `gsub`; use the `/r` flag to get a new string instead.
- **`open` does not raise.** Use `open ... or die "...: $!"`.
- **Blocks are not implicit.** A stored `sub { ... }` is called explicitly with `->( )`; there is no `yield` and no invisible block argument.
- **Non-ASCII source needs `use utf8;`.** Under `use v5.44;` a literal non-ASCII byte in your source is a compile error without it. Keep string data ASCII unless you specifically add `use utf8;`.

- **The `class` feature prints experimental notices to stderr** on `pperl`. Correct output, noisy error stream.

Lua to Perl

You know Lua. You build a table without thinking, you reach for `setmetatable` to get objects and operators, and `for k, v in pairs(t)` is muscle memory. This guide maps each of those habits onto Perl, so you can be productive in `pperl` in an afternoon instead of relearning programming from scratch.

Two shocks land in the first five minutes, so meet them now:

- **Lua's one table becomes two Perl types.** Lua has a single aggregate, the table, that you use as both an array (`{1, 2, 3}`) and a map (`{k = "v"}`). Perl splits that into two distinct things: an ordered `@array` keyed by integers and a `%hash` keyed by strings. Deciding "is this a list or a lookup?" up front is the central new habit, and the *Arrays* and *Hashes* sections are about nothing else.
- **Variables wear sigils.** Lua names are bare: `x`, `t`, `f`. In Perl a scalar is `$x`, an array is `@x`, a hash is `%x`. The sigil is part of how you read the variable, not just decoration. This is the single biggest surface difference from Lua, and section one is about nothing else.

Every example here runs on `pperl`, a Rust reimplementation of Perl 5.44. Each Perl snippet was executed on `pperl`; the `# ...` comments show its real output. Start your scripts with:

```
use v5.44;
```

That one line turns on `strict` (every variable must be declared with `my`, the cure for Lua's accident-prone implicit globals), `warnings`, the `say` built-in (a `print` with a trailing newline, like Lua's `print`), subroutine signatures, and the modern feature set used throughout this guide.

How to read this

- *Variables and sigils* - the `$`/`@`/`%` shift, and `local x` becoming `my $x`.
- *Strings* - interpolation, and `..` concatenation becoming `.`.
- *Numbers* - one surface type, float division.
- *Arrays and lists* - half of your table, 0-based not 1-based.
- *Hashes* - the other half of your table.
- *Context* - the concept Lua has no word for. Read this one.
- *Truthiness* - what counts as false, and it differs sharply from Lua.
- *Control flow* - `if`, loops, the postfix forms.
- *Subroutines* - `function` becomes `sub`; real closures.
- *References* - nested data and closures.
- *Regular expressions* - real regex, not Lua patterns.
- *File and stream I/O* - open, line reads, slurp.

- *Modules and packages* - `require` becomes `use`.
- *Objects* - metatables become the modern `class` syntax.
- *Error handling* - `pcall` becomes `eval` or `try`/`catch`.
- *Idioms* - the daily one-liners.
- *Gotchas coming from Lua* - scan before you write.

Variables and sigils

In Lua a name just names a value, whatever its shape. In Perl the **sigil tells you the shape**: `$` for a single value, `@` for an ordered list, `%` for a key/value map. `my` declares a lexically scoped variable, the exact role of Lua's `local`. Where Lua lets a bare assignment silently create a global, `use v5.44` turns that into a compile error: every variable is `my` unless you say otherwise.

```lua
local name = "Ada"
local langs = {"Lua", "Perl"}
local ages = {Ada = 36, Linus = 56}
```

```perl
use v5.44;
my $name = "Ada";
my @langs = ("Lua", "Perl");
my %ages  = (Ada => 36, Linus => 56);
say $name;            # Ada
say $langs[0];        # Lua
say $ages{Ada};       # 36
```

The gotcha: when you reach *into* an aggregate for one element, the sigil follows the value you get back, not the container. A single element of `@langs` is a scalar, so it is `$langs[0]`, not `@langs[0]`; one value from `%ages` is `$ages{Ada}`. The `@`/`%` form is the whole collection; the `$` form is one piece of it. In Lua the table name `t` never changes whether you write `t`, `t[1]`, or `t.key`; in Perl the leading mark changes with what you are accessing.

Strings

Double quotes interpolate variables, single quotes do not. There is no `string.format` reflex needed for the common case, because the default string already interpolates. The big swap: concatenation is `.`, where Lua uses `..`

```lua
local name = "Ada"
local greeting = "Hello, " .. name
local shout = string.rep("AB", 3)
```

```perl
use v5.44;
my $name = "Ada";
my $greeting = "Hello, $name";   # Hello, Ada
my $literal  = 'Hello, $name';   # Hello, $name  (no interpolation)
my $joined   = "AB" . "CD";      # ABCD
my $shout    = "AB" x 3;         # ABABAB
```

The gotcha: Lua's concatenation operator `..` does not exist in Perl; the single dot `.` joins strings and the `x` operator repeats them (Lua's `string.rep`). Array elements and hash values interpolate inside double quotes too, so `"First: $langs[0]"` works as written, which removes most of the `..` chaining a Lua developer reaches for.

Numbers

Perl has one numeric type at the surface; you do not pick between integer and float, much as Lua 5.3+ presents a single `number` to most code. Division always produces a float.

```lua
print(7 / 2)        -- 3.5
print(math.floor(7 / 2))  -- 3
print(2 ^ 10)       -- 1024.0
print(10 % 3)       -- 1
```

```perl
use v5.44;
say 7 / 2;          # 3.5
say int(7 / 2);     # 3
say 2 ** 10;        # 1024
say 10 % 3;         # 1
```

The gotcha: comparison and equality operators come in two families, and you pick by the *operands' type*, not the variables'. Numbers use `==`, `!=`, `<`, `>`, `<=>`; strings use `eq`, `ne`, `lt`, `gt`, `cmp`. `"10" == "10.0"` is true (numeric), but `"10" eq "10.0"` is false (string). Lua has only `==` and coerces nothing across the string/number line; Perl instead asks *you* to choose the comparison, and using `==` on strings (or `eq` on numbers) is a frequent quiet bug.

Arrays and lists

Here is half of the great table split. A Lua sequence (`{10, 20, 30}`, the table used as an array) becomes a Perl `@array`. The first and loudest difference: **Perl arrays are 0-based**, where Lua arrays conventionally start at index 1.

```lua
local t = {"x", "y", "z"}
print(t[1])    -- x   (Lua: first element is index 1)
print(t[3])    -- z
print(#t)      -- 3   (length)
```

```perl
use v5.44;
my @t = ("x", "y", "z");
say $t[0];          # x   (Perl: first element is index 0)
say $t[2];          # z
say $t[-1];         # z   (negative indices count from the end)
say scalar @t;      # 3   (length; @t in scalar context is its size)
```

The mutators are renamed but familiar. `table.insert`/`table.remove` become `push`/`pop`/`shift`/`unshift`:

```
use v5.44;
my @nums = (10, 20, 30);
push @nums, 40;              # add to the end
my $last = pop @nums;        # 40, removed from the end
say scalar @nums;            # 3
say "@nums[0,1]";            # 10 20   (a slice)
```

Lua reaches for explicit `for i = 1, #t do ... end` loops to transform or filter; Perl has `map` (transform) and `grep` (filter), with `$_` as the current element:

```
use v5.44;
my @nums    = (1, 2, 3, 4, 5);
my @squares = map  { $_ * $_ }   @nums;   # 1 4 9 16 25
my @evens   = grep { $_ % 2 == 0 } @nums; # 2 4
my @desc    = sort { $b <=> $a }  @nums;  # 5 4 3 2 1
```

The gotcha: beyond 0-based indexing, `#t` has no single Perl spelling - you use `scalar @a` for an array's length and `keys %h` for a hash's. And `sort` compares as strings by default, so `sort (10, 9, 100)` gives `10 100 9`. For numbers you must pass a comparator: `sort { $a <=> $b } @nums`. `$a` and `$b` are the two elements being compared, and `<=>` is the numeric three-way operator (`cmp` is its string sibling).

Hashes

Here is the other half of the table split. A Lua table used as a map (`{name = "Ada", age = 36}`, or `t["key"]`) becomes a Perl `%hash`. Declared with `%`, accessed one value at a time with `$h{key}`.

```
local h = {apple = 3, pear = 5, plum = 2}
print(h.apple)              -- 3
print(h["pear"])            -- 5
h.plum = nil                -- delete a key
for k, v in pairs(h) do
    print(k, v)
end
```

```
use v5.44;
my %h = (apple => 3, pear => 5, plum => 2);
say $h{apple};                          # 3
say join ",", sort keys %h;             # apple,pear,plum
say exists $h{pear} ? "yes" : "no";     # yes
delete $h{plum};
for my $k (sort keys %h) {
    say "$k => $h{$k}";                 # apple => 3 ... pear => 5
}
```

The gotcha: Lua merges the array and the map into one table and keeps both an array part and a hash part inside it; Perl keeps them strictly apart, so you decide up front whether a thing is an `@array` (integer positions, ordered) or a `%hash` (string keys, unordered). Membership is `exists $h{key}`, the counterpart of Lua's `h[k] ~= nil` test, and the fat comma `=>` is the idiomatic key/value separator that auto-quotes the bare-

word on its left (so `apple => 3` needs no quotes). A hash has no order, so iterate `sort keys %h` when you want a stable one; Lua's `pairs` is unordered for the same reason.

Context

This is the chapter to read twice. Lua has no general equivalent, and almost every "Perl is weird" story traces back to it. But Lua *does* give you the one bridge that makes it click: **multiple return values**.

In Lua, `local a, b = f()` spreads a function's several return values across several variables, and `#t` or `select("#", ...)` collapses a list to a count. Perl generalises that idea into a property of *every* expression. Every expression runs in either **list context** or **scalar context**, decided by where it appears. The same array behaves differently in each: assigned to a *scalar* it gives its length, assigned to a *list* (parenthesised variables) it copies elements.

```
use v5.44;
my @list = (10, 20, 30);

my $count = @list;          # scalar context -> 3 (the length)
say $count;                 # 3

my ($first) = @list;        # list context -> first element copied
say $first;                 # 10

my ($a, $b) = @list;        # 10 and 20
say "$a $b";                # 10 20
```

The parentheses on the left are the whole difference. `my $x = @list` asks "how many?" and gets 3. `my ($x) = @list` asks "give me the elements" and gets `10`. The list form `my ($a, $b) = @list` is exactly Lua's `local a, b = ...` - the bridge you already know.

A subroutine returning several values is the same as a Lua function with multiple `return` values; the caller's context decides what comes back:

```
use v5.44;
sub minmax (@nums) {
    my @s = sort { $a <=> $b } @nums;
    return ($s[0], $s[-1]);
}
my ($lo, $hi) = minmax(4, 1, 9, 2);
say "$lo $hi";              # 1 9
```

A sub can even ask which context it was called in, with `wantarray`, and answer differently - something Lua functions cannot do:

```
use v5.44;
sub items {
    return wantarray ? (1, 2, 3) : "three items";
}
my @got = items();          # list context
my $msg = items();          # scalar context
```

(continues on next page)

(continued from previous page)

```
say "@got";                # 1 2 3
say $msg;                  # three items
```

The gotcha: there is no single answer to "what does this array do" - it depends on the surrounding expression, where Lua resolves multiple-value spreading at the specific call site only. When something returns a number you expected to be a list (or vice versa), context is the first suspect. Reach for `scalar` to force a count and parentheses on the left to force element copying.

Truthiness

This is a sharp break from Lua, so slow down. In Lua **only `nil` and `false` are falsy** - the number `0` and the empty string `""` are both *true*. Perl draws the line in a completely different place. Its false values are: the number `0`, the empty string `""`, the string `"0"`, and undef (Perl's `nil`). **Everything else is true.**

```
use v5.44;
for my $v (0, 1, "", "0", "00", "0.0", "false") {
    say "'$v' is ", ($v ? "true" : "false");
}
```

This prints, exactly:

```
'0' is false
'1' is true
'' is false
'0' is false
'00' is true
'0.0' is true
'false' is true
```

The gotcha: a Lua developer's instinct that `0` is true will silently flip the meaning of every `if ($count)`. In Perl `0` and `""` *are* false, so `if ($n)` is false when `$n` is zero - usually what you want, but the opposite of Lua. Going the other way, `"00"` and `"0.0"` are **true** even though they look numerically zero, because only the exact string `"0"` is false. For a "use this if defined, else a default" choice - Lua's `x = y or default` idiom, which trips on `0` and `false` - use Perl's defined-or `//`, which falls back only on undef (Perl's `nil`):

```
use v5.44;
my %cfg = (host => "localhost");
my $port = $cfg{port} // 8080;   # key absent -> undef -> 8080
say $port;                       # 8080
```

Control flow

The keywords are familiar; the spelling differs. Lua's `elseif` is Perl's `elsif`, there is no then/do/end, and blocks are braced rather than keyword-delimited.

```
if n < 0 then
    sign = "neg"
elseif n == 0 then
    sign = "zero"
else
    sign = "pos"
end

for _, x in ipairs(nums) do
    print(x)
end

while running do
    step()
end
```

```
use v5.44;
my $sign;
if    ($n < 0)  { $sign = "neg"  }
elsif ($n == 0) { $sign = "zero" }
else            { $sign = "pos"  }

for my $x (@nums) { say $x }

while ($running) { step() }
```

Perl adds two things Lua does not have. **Postfix modifiers** let a simple statement carry its own condition or loop, which reads naturally for one-liners:

```
use v5.44;
say $_ for @nums;                # loop, one statement
say "negative" if $n < 0;        # guarded statement
$sum += $_ for @nums;            # accumulate
```

And `unless`/`until` are the negated forms of `if`/`while`, for when the positive phrasing reads awkwardly: `say "ok" unless $error`.

The gotcha: Perl has no `x if c else y` statement; the conditional expression is the C-style ternary with `?`/`:`, which Lua usually fakes with `c and x or y`. The Perl form is direct and chains: `my $sign = $n < 0 ? "neg" : $n == 0 ? "zero" : "pos"`. Loop control is `last`/`next` where Lua has `break` and (in 5.4) `goto continue`.

Subroutines

`function` becomes `sub`. Modern Perl has signatures that name their parameters, including defaults, rather than Lua's positional `...` and manual `arg` handling.

```
local function greet(name, greeting)
    greeting = greeting or "Hello"
    return greeting .. ", " .. name .. "!"
```

(continues on next page)

(continued from previous page)

```lua
end

local function total(...)
    local sum = 0
    for _, n in ipairs({...}) do sum = sum + n end
    return sum
end
```

```perl
use v5.44;
sub greet ($name, $greeting = "Hello") {
    return "$greeting, $name!";
}
say greet("Ada");           # Hello, Ada!
say greet("Linus", "Hi");   # Hi, Linus!

sub total (@nums) {
    my $sum = 0;
    $sum += $_ for @nums;
    return $sum;
}
say total(1, 2, 3, 4);      # 10
```

Lua's named-argument idiom (passing a single table, `f{host=..., port=...}`) maps to a hash parameter, since `key => value` pairs flatten into a list that a `%opts` signature gathers up:

```perl
use v5.44;
sub connect_to (%opts) {
    return "host=$opts{host} port=$opts{port}";
}
say connect_to(host => "localhost", port => 8080);
# host=localhost port=8080
```

The gotcha: a sub returns the value of its last expression even without `return` (like Lua's explicit-only return, except Perl also has the implicit last-value form), and the return is **context-sensitive** (see *Context*). If you `return @list`, the caller gets the list in list context but the *count* in scalar context - a degree of freedom Lua's multiple-return does not have.

References

Perl's arrays and hashes do not nest directly; you nest them through **references**, a scalar that points at an aggregate. In Lua every table is already a reference (assigning a table shares it, never copies), so this concept is invisible to you there; in Perl you take a reference explicitly with `\`, and follow it with `->`.

```lua
local data = {name = "Ada", langs = {"Perl", "Lua"}}
print(data.langs[2])        -- Lua
table.insert(data.langs, "Ruby")
```

```
use v5.44;
my $data = { name => "Ada", langs => ["Perl", "Lua"] };
say $data->{name};                  # Ada
say $data->{langs}[1];              # Lua
push $data->{langs}->@*, "Ruby";    # postfix deref to push
say "@{$data->{langs}}";           # Perl Lua Ruby
```

`{ ... }` builds an anonymous hash reference and `[ ... ]` an anonymous array reference - the literal building blocks of nested data. The `->` arrow follows a reference one step; between two subscripts it is optional, so `$data->{langs}[1]` and `$data->{langs}->[1]` are the same.

Anonymous subs are real closures, exactly like Lua's closures over upvalues - one of the cleanest carry-overs in the whole guide:

```
use v5.44;
sub counter ($start) {
    my $n = $start;
    return sub { return $n++ };
}
my $c = counter(10);
say $c->();                 # 10
say $c->();                 # 11
say $c->();                 # 12
```

The gotcha: the postfix deref forms (`->@*`, `->%*`, `->$*`) are the modern, readable way to dereference a whole aggregate, and `"@{ ... }"` is how you interpolate a dereferenced array into a string. The older `@$ref` and `${$ref}{key}` forms still work and appear in existing code, but reach for the arrow and postfix forms in new code.

Regular expressions

This is an upgrade, not a translation. Lua has **patterns** - `string.match`, `string.gsub`, character classes like `%d` - which are deliberately *not* regular expressions and lack alternation, quant, and backreferences. Perl has full regex as **syntax**: `=~` binds a pattern to a string, `m//` matches, `s///` substitutes. Capture groups land in `$1`, `$2`, and named captures in `%+`.

```
local s = "2026-06-20"
local y, m, d = string.match(s, "(%d+)-(%d+)-(%d+)")
local t = string.gsub("a,b,c", ",", ";")   -- a;b;c
```

```
use v5.44;
my $s = "2026-06-20";
if ($s =~ /(\d{4})-(\d{2})-(\d{2})/) {
    say "$1 $2 $3";                 # 2026 06 20
}
if ($s =~ /(?<year>\d{4})/) {
    say $+{year};                   # 2026
}
```

(continues on next page)

(continued from previous page)

```
(my $t = "a,b,c") =~ s/,/;/g;       # a;b;c
my @parts = split /,/, "one,two,three";   # one two three
```

A `/g` match in list context returns *all* the matches, the direct analogue of looping `string.gmatch`:

```
use v5.44;
my @nums = "x1y2z3" =~ /(\d)/g;     # 1 2 3
say "@nums";
```

The gotcha: the metacharacters differ from Lua patterns - Perl uses `\d` where Lua uses %d, */+/? quantifiers, `|` alternation, and `(?<name>...)` named groups, none of which Lua patterns support. And `s///` modifies its target *in place* and returns the number of substitutions, not a new string, where Lua's `gsub` returns a new string. That is why the example copies first with `(my $t = "a,b,c")` and then substitutes on the copy. To get a new string without touching the original, use the `/r` flag: `my $t = $s =~ s/,/;/gr`.

File and stream I/O

The idiom is the three-argument open with a lexical filehandle, the counterpart of Lua's `io.open`. The angle-bracket operator `<$fh>` reads a line, the role of Lua's `fh:read("l")` or the `fh:lines()` iterator.

```
local f = io.open("/tmp/sample.txt", "w")
for i = 1, 3 do
    f:write("line " .. i .. "\n")
end
f:close()

f = io.open("/tmp/sample.txt", "r")
for line in f:lines() do
    print("got:", line)
end
f:close()
```

```
use v5.44;
my $path = "/tmp/sample.txt";

open my $wh, '>', $path or die "cannot write: $!";
print {$wh} "line $_\n" for 1 .. 3;
close $wh;

open my $rh, '<', $path or die "cannot read: $!";
while (my $line = <$rh>) {
    chomp $line;
    say "got: $line";       # got: line 1 ... got: line 3
}
close $rh;
```

To slurp every line at once, read the handle in list context:

```
use v5.44;
open my $fh, '<', "/tmp/sample.txt" or die $!;
my @lines = <$fh>;          # all lines, including newlines
say scalar @lines;          # 3
```

The gotcha: a line read with `<$fh>` keeps its trailing newline (Lua's `fh:lines()` strips it, `fh:read("L")` keeps it); call `chomp` to strip it. And `open` returns false on failure rather than raising, much like Lua's `io.open` returning `nil, err`, so the `or die "...: $!"` idiom is mandatory - `$!` is the system error message, Perl's `errno`.

Modules and packages

Lua's `require` becomes Perl's `use`. The namespace separator is `::`, not `.`, and a module `Foo::Bar` lives at `Foo/Bar.pm` on the include path `@INC` (Lua's `package.path`). Unlike Lua's `require`, which returns a table you assign yourself, `use` imports selected names directly into your namespace.

```
local sqrt = math.sqrt
local sum = 0
for i = 1, 10 do sum = sum + i end
```

```
use v5.44;
use List::Util qw(sum0 max first);
say sum0(1 .. 10);                      # 55
say max(3, 7, 2);                       # 7
say first { $_ % 2 == 0 } (1, 3, 4, 5);  # 4
```

Here is a pleasant surprise coming from Lua's deliberately tiny standard library: Perl ships a large built-in set, and much of what you would hand-roll or pull from LuaRocks is in core or one `use` away on CPAN, the Perl package archive. `List::Util`, `Scalar::Util`, `POSIX`, `File::Spec`, and `Data::Dumper` are the everyday ones.

The gotcha: `use` runs at *compile time*, not at the line where it appears, so its placement relative to your code does not matter the way a `require` call's does. Importing is opt-in per name: `use List::Util qw(sum0 max)` brings exactly `sum0` and `max` into your namespace, nothing else.

Objects

Lua has no built-in object system; you build one from tables and metatables - `setmetatable(t, {__index = Class})` for method lookup and inheritance, `__add`/`__eq` for operators. Perl 5.44 gives you a first-class `class` syntax instead, so the common case needs no metatable wiring. Fields are declared with `field`, methods with `method`, and `$self` is available inside every method (you do not pass it as the first parameter the way Lua's `self` / colon-call does).

```
local Point = {}
Point.__index = Point

function Point.new(x, y)
```

(continues on next page)

(continued from previous page)

```
    return setmetatable({x = x or 0, y = y or 0}, Point)
end
function Point:to_string()
    return "(" .. self.x .. ", " .. self.y .. ")"
end
function Point:move(dx, dy)
    self.x = self.x + dx
    self.y = self.y + dy
    return self
end

local p = Point.new(3, 4)
print(p:to_string())        -- (3, 4)
p:move(1, 1)
print(p:to_string())        -- (4, 5)
```

```
use v5.44;
use feature 'class';

class Point {
    field $x :param = 0;
    field $y :param = 0;
    method to_string { return "($x, $y)" }
    method move ($dx, $dy) {
        $x += $dx;
        $y += $dy;
        return $self;
    }
}

my $p = Point->new(x => 3, y => 4);
say $p->to_string;          # (3, 4)
$p->move(1, 1);
say $p->to_string;          # (4, 5)
```

The constructor is generated for you, replacing Lua's hand-written `Point.new`: `:param` marks a field that the auto-built new accepts as a named argument, and `= 0` is its default. Inheritance, which Lua builds from a chain of `__index` metatables, is the `:isa` attribute:

```
use v5.44;
use feature 'class';

class Point3D :isa(Point) {
    field $z :param = 0;
    method z { return $z }
}

my $q = Point3D->new(x => 1, y => 2, z => 3);
say $q->to_string, " z=", $q->z;   # (1, 2) z=3
```

The gotcha: the `class` feature is still marked experimental in Perl 5.44, and on `pperl` the "class is experimental" notices print to **stderr** even when you try to silence them. They do not touch your program's output, so a script that prints to stdout behaves exactly as shown above; just expect the notices on the error stream. Lua's operator metamethods (__add, __eq, and friends) map to Perl's `overload` pragma rather than to `class`; for that, the older `bless`-based object model, roles, and inheritance details, see the *Object-oriented Programming guide*.

Error handling

Lua's protected call `pcall(f)` - which returns `false, err` instead of unwinding - has two Perl counterparts. The classic one wraps code in `eval { }` and inspects `$@` (the caught error) afterward:

```lua
local ok, err = pcall(function() risky(-1) end)
if not ok then
    print("caught: " .. err)
end
```

```perl
use v5.44;
my $result = eval { risky(-1) };
if ($@) {
    print "caught: $@";     # caught: negative!
}
```

The modern form is `try`/`catch`, which reads more like a conventional exception block than `pcall`'s boolean-return dance:

```perl
use v5.44;
use feature 'try';

try {
    risky(-4);
}
catch ($e) {
    print "caught: $e";     # caught: negative!
}
```

where the raising side is `die`, Perl's `error`:

```perl
use v5.44;
sub risky ($n) {
    die "negative!\n" if $n < 0;
    return sqrt $n;
}
```

The gotcha: `die` with a string that ends in `\n` produces exactly that message; `die` with a string that does *not* end in a newline gets `" at SCRIPT line N.\n"` appended automatically (Lua's `error` does a similar position-prefixing unless you pass level `0`). End your error strings with `\n` when you want a clean message and omit it when you want the file-and-line trace. Note also the two error variables: `$@` is the last caught exception, while `$!` is the last *system* error (a failed open, the `errno` string).

Idioms

A few patterns you will see daily and want in your fingers.

Sort a list by a computed key (the Schwartzian transform - decorate, sort, undecorate in one pipeline, the Perl reflex where Lua passes a comparator function to `table.sort`):

```
use v5.44;
my @words = qw(apple fig pear);
my @by_length =
    map  { $_->[1] }
    sort { $a->[0] <=> $b->[0] }
    map  { [ length $_, $_ ] } @words;
say "@by_length";           # fig pear apple
```

Build a hash from a list, turning a sequence into a lookup table in one step:

```
use v5.44;
my @words = qw(apple pear fig);
my %len = map { $_ => length $_ } @words;
say "$_ => $len{$_}" for sort keys %len;
# apple => 5
# fig => 3
# pear => 4
```

And `pperl` is a command-line tool in the `awk`/`sed` niche, a role Lua does not target. The implicit-loop one-liner is the Perl developer's everyday reflex:

```
pperl -lane '$s += $F[-1]; END { print $s }' data.txt
```

The gotcha: read the *One-Liners guide* for the switches (`-n`, `-p`, `-a`, `-l`, `-e`) and the dozens of recipes built on them; it is one of the highest-leverage things to learn coming from a scripting background where this niche was someone else's job.

Gotchas coming from Lua

The traps, collected for a final scan before you start writing:

- **One table becomes two types.** A Lua table is an `@array` (integer-keyed, ordered) *or* a `%hash` (string-keyed, unordered) in Perl, never both at once. Decide which you have before you reach for it.
- **Arrays are 0-based.** Lua's `t[1]` is Perl's `$t[0]`; the last element is `$t[-1]`. This is the most common porting off-by-one.
- **Sigils change with usage.** One element of `@a` is `$a[0]`; one value of `%h` is `$h{k}`. The container sigil and the element sigil differ - the bare Lua name `t` has no such shift.
- **Concatenation is `.`, not `..`.** Lua's `..` does not exist; `.` joins strings and `x` repeats them.
- **Truthiness is inverted at zero.** In Lua `0` and `""` are *true*; in Perl they are **false**. Only `0`, `""`, `"0"`, and `undef` are false, so `"00"` and `"0.0"` are surprisingly **true**.

- **Two comparison families.** ==/</<=> for numbers, eq/lt/cmp for strings. Pick by operand type; Lua's single == has no equivalent split.
- **`sort` is string-sorted by default.** Pass `{ $a <=> $b }` for numeric order.
- **Context.** `my $x = @a` is the *length*; `my ($x) = @a` is the *first element*. The list form is your friend `local a, b = f()`; the scalar form is new. Re-read *Context* when a value comes back the wrong shape.
- **Regex is not Lua patterns.** Perl has full regex (\d, |, *, named groups); Lua patterns (%d, no alternation) are a different, smaller language. `s///` edits in place and returns a count; use `/r` for a new string.
- **`open` does not raise.** Use `open ... or die "...: $!"`, the way Lua's `io.open` returns `nil, err`.
- **`chomp` your line reads.** `<$fh>` keeps the trailing newline that Lua's `lines()` would have stripped.
- **Non-ASCII source needs `use utf8;`.** Under `use v5.44;` a literal non-ASCII byte in your source is a compile error without it. Keep string data ASCII unless you specifically add `use utf8;`.
- **The `class` feature prints experimental notices to stderr** on `pperl`. Correct output, noisy error stream. Operator metamethods map to the `overload` pragma, not to `class`.

Sed to Perl

You know `sed`. You think in `s/old/new/`, you address lines with `/pattern/` and `$`, you reach for `-n` and `p` to print selectively, and `-i` to edit files in place. This guide maps each of those habits onto Perl, so you can keep every reflex you have and pick up the things sed makes hard.

The one idea to carry in from the first line: **Perl is a near-superset of sed for text work.** Everything sed does, Perl does - usually with the *same* `s///` syntax - plus a real regex engine, real variables, and arbitrary code in the replacement. The awkward corners of sed (the hold space, multi-line tricks, anything resembling arithmetic) become ordinary Perl that you would write without thinking. This guide leads with the one-to-one mappings and then shows where Perl simply removes a sed limitation.

Every runnable example here was executed on `pperl`, a Rust reimplementation of Perl 5.44; the `# ...` comments show its real output. Start scripts with:

```
use v5.44;
```

That one line turns on `strict`, `warnings`, and the `say` built-in (a `print` with a trailing newline), the modern feature set used throughout.

How to read this

- *The line cycle* - sed's implicit loop becomes `while (<$fh>)` and `$_`.
- *Substitution: `s///`* - the same operator, with real regex.
- *Transliteration: `y///` becomes `tr///`* - the y command, renamed.

- *Addresses and ranges* - line numbers, patterns, and `/a/ .. /b/`.
- *Deleting, printing, quitting:* `d`, `p`, `q` - `next`, `print`, `last`.
- *The hold space becomes a variable* - where sed is hard and Perl is trivial.
- *In-place editing and I/O* - `-i`, reading files, pipes.
- *One-liners and switches* - `-n`, `-p`, `-e`, `BEGIN`/`END`.
- *Gotchas coming from sed* - scan before you write.

The line cycle

sed runs your script once per input line, with the current line in its pattern space, and prints the pattern space at the end of the cycle (unless `-n`). Perl's equivalent is an explicit loop with the line in `$_`, the implicit topic variable. The `-p` switch reproduces "auto-print every line"; `-n` reproduces "print nothing unless I say so."

```
sed 's/bar/BAR/' input.txt
```

```
perl -pe 's/bar/BAR/' input.txt
```

The same loop written out in full - what `-p` expands to - opens the file and iterates it explicitly. `$_` is the current line, exactly like sed's pattern space:

```
use v5.44;
open my $fh, '<', '/tmp/data.txt' or die "open: $!";
while (<$fh>) {           # each line lands in $_
    s/a/A/g;              # operate on $_ by default
    print;               # print $_ (note: keeps the newline)
}
# Apple
# bAnAnA
# cherry
```

(The input file held `apple`, `banana`, `cherry`.) The gotcha: a line read with `<$fh>` keeps its trailing newline, just as sed's pattern space does. `print` with no argument prints `$_` including that newline, so it round-trips cleanly; `chomp` only when you intend to strip it. The line number is in `$.`, the sed analogue of the line counter you reach for with `$=` in GNU sed scripts.

Substitution: `s///`

This is the headline: **sed's `s///` and Perl's `s///` are the same operator.** The flags line up too - `g` is global, `i` is case-insensitive - and Perl adds flags sed has no equivalent for.

```
sed 's/colou\?r/COLOR/g' input.txt
```

```
perl -pe 's/colou?r/COLOR/g' input.txt
```

By default `s///` edits its target in place and returns the count of substitutions. Captured groups are `$1`, `$2`, ... and you can run **arbitrary code** in the replacement with the `/e` flag - something sed cannot do at all:

```
use v5.44;
my $s = "items: 3 and 4";
(my $t = $s) =~ s/(\d+)/$1 * 10/ge;   # /e: replacement is Perl code
say $t;                               # items: 30 and 40
```

To rearrange captures (sed's `\1`, `\2` backreferences in the replacement), use `$1`, `$2` and, when you want a *new* string instead of editing in place, the `/r` flag, which returns the result and leaves the original untouched:

```
use v5.44;
my $date = "2026-06-21";
my $us = $date =~ s{(\d+)-(\d+)-(\d+)}{$2/$3/$1}r;   # /r: return a␣
↪copy
say $us;                                              # 06/21/2026
say $date;                                            # 2026-06-21
```

The gotcha: bare `s///` modifies its target and returns a *count*, not the new string. That is why the examples above either copy first (`(my $t = $s) =~ s/.../.../`) or use `/r`. Writing `my $new = ($s =~ s/a/b/)` puts the substitution count in `$new`, not the edited string - a classic first-day surprise. Note also that Perl delimiters are free: `s{...}{...}` and `s#...#...#` avoid backslashing slashes in a path pattern.

Transliteration: `y///` becomes `tr///`

sed's `y/abc/xyz/` is Perl's `tr/abc/xyz/`. Perl keeps `y///` as an exact alias, so `y/a-z/A-Z/` works verbatim, but `tr` is the spelled name you will see in Perl code.

```
sed 'y/abc/xyz/' input.txt
```

```
use v5.44;
my $s = "hello";
(my $t = $s) =~ tr/a-z/A-Z/;   # uppercase, like y/a-z/A-Z/
say $t;                        # HELLO
```

`tr` also does things `y` cannot. With an empty replacement it *counts* matches, and the `s` modifier squeezes runs of a character - the common `tr -s` idiom:

```
use v5.44;
my $s = "hello";
my $vowels = ($s =~ tr/aeiou//);   # count, replacement empty
say $vowels;                       # 2

my $dots = "a....b...c";
(my $squeezed = $dots) =~ tr/././s; # squeeze repeats
say $squeezed;                      # a.b.c
```

The gotcha: `tr///` is **not** a regex - the left side is a literal character set, the same as sed's `y`. `tr/a-z//` means the range `a` through `z`, but `tr/.*//` means the two literal characters `.` and `*`, not "any string." For pattern work, reach for `s///`. And like `s///`, `tr///` edits in place and returns a count, so copy first or use the `/r` flag for a non-destructive transliteration.

Addresses and ranges

A sed command can be prefixed by an address: a line number, `$` for the last line, a `/pattern/`, or a `addr1,addr2` range. In Perl an address is just a condition on the statement - a postfix `if`. Line number is `$.`; the pattern test is `/pat/` against `$_`; the last line is `eof`.

```
sed -n '2p' input.txt              # print line 2
sed -n '/banana/p' input.txt       # print matching lines
```

```
use v5.44;
open my $fh, '<', '/tmp/data.txt' or die "open: $!";
while (<$fh>) {
    print if $. == 2;              # sed -n '2p'
}
# banana
```

A sed range `/start/,/end/` becomes Perl's **flip-flop** range operator `/start/ .. /end/`, which is true from the line matching the left pattern through the line matching the right, inclusive:

```
use v5.44;
open my $fh, '<', '/tmp/data.txt' or die "open: $!";
while (<$fh>) {
    print if /banana/ .. /cherry/;  # sed '/banana/,/cherry/p' with␣
 ↪-n
}
# banana
# cherry
```

The gotcha: in a boolean test `..` is the flip-flop (range) operator, **not** the list-range operator (which is `1 .. 10` building a list). Perl tells them apart by context - in the scalar/boolean position of an `if`, `..` is the line-range selector that mirrors sed's `addr1,addr2`. It is stateful: it remembers whether it has seen the start pattern, so it does exactly what a sed range does across the loop.

Deleting, printing, quitting: d, p, q

sed's single-letter cycle commands map to Perl loop keywords. `d` (delete pattern space, skip to next cycle) is `next`. `q` (quit) is `last`. `p` (print) is `print`. Because Perl's loop is explicit, these read as exactly what they do.

```
sed '/banana/d' input.txt          # drop matching lines
sed '/banana/q' input.txt          # stop after the match
```

```
use v5.44;
open my $fh, '<', '/tmp/data.txt' or die "open: $!";
while (<$fh>) {
    next if /banana/;              # sed '/banana/d'
    print;
}
```

(continues on next page)

(continued from previous page)

```
# apple
# cherry
```

```
use v5.44;
open my $fh, '<', '/tmp/data.txt' or die "open: $!";
while (<$fh>) {
    print;
    last if /banana/;              # sed '/banana/q'
}
# apple
# banana
```

The gotcha: under sed's `-p`-style auto-print, `d` must skip the auto-print, which it does by aborting the cycle. In an explicit Perl loop you control printing yourself, so "delete" is simply *not printing* the line - `next if /pat/` before the `print`, or guard the `print` with `unless`. There is no hidden auto-print to fight; what you `print` is what comes out.

The hold space becomes a variable

This is the section where sed stops being convenient and Perl stops being hard. sed has exactly one auxiliary buffer - the hold space - and shuffles data into it with `h`, `H`, `g`, `G`, `x`. Anything needing two pieces of remembered state turns into hold-space contortions. In Perl you just declare a variable. Or an array. Or a hash.

The canonical hold-space exercise is reversing a file (`tac`), which in sed is an opaque `1!G;h;$!d` incantation. In Perl it is one obvious line:

```
sed -n '1!G;h;$!d' input.txt       # reverse the file (tac)
```

```
use v5.44;
open my $fh, '<', '/tmp/data.txt' or die "open: $!";
my @hold;                          # the "hold space" is just an␣
↪array
while (<$fh>) {
    unshift @hold, $_;             # newest line to the front
}
print @hold;
# cherry
# banana
# apple
```

The gotcha: there is no gotcha - that is the point. Once you have real variables, the entire category of sed problems that exist only because there is a single hold space disappears. Need to remember the previous line? `my $prev`. Need a running total? `my $sum`. Need to dedup? A `my %seen` hash. When a sed script reaches for `h`/`H`/`x`, the Perl translation is almost always "introduce a named variable and write the obvious thing."

In-place editing and I/O

sed's `-i` rewrites a file in place. The idiomatic Perl is the same switch on a `-p` one-liner. Reading several files is sed's default; Perl spells it `<>`, the diamond operator, which reads each file named in `@ARGV` in turn.

```
sed -i 's/red/crimson/' file.txt   # edit in place
```

```
perl -i -pe 's/red/crimson/' file.txt
```

Written out in full it is an explicit read-modify-write: slurp the lines, transform them, write them back. This is exactly what `-i` does under the hood:

```
use v5.44;
my $path = '/tmp/inplace_data.txt';   # holds: red green blue

open my $in, '<', $path or die "read: $!";
my @lines = <$in>;                    # slurp all lines (list
 ↪context)
close $in;

s/e/E/g for @lines;                   # transform each line's $_

open my $out, '>', $path or die "write: $!";
print {$out} @lines;                  # write them back
close $out;
# file now holds:
# rEd
# grEEn
# bluE
```

For pipes - sed in the middle of a shell pipeline - Perl opens a pipe filehandle directly: `open my $fh, '-|', 'ls', '-l'` reads a command's output, and `open my $fh, '|-', 'sort'` writes to a command's input. The gotcha: `open` returns false on failure rather than raising, so the `or die "...: $!"` idiom is mandatory - `$!` is the system error string, sed's silent failure made explicit. And reading multiple files with `<>` resets `$.` per file only if you check `eof` explicitly; the plain `$.` keeps counting across the concatenated stream.

One-liners and switches

The one-liner is sed's home turf and Perl's too. The switch family that makes it work: `-e` supplies the program, `-n` wraps it in a no-auto-print loop (sed's `-n`), `-p` wraps it in an auto-print loop (sed's default), `-i` edits in place, `-l` handles line endings, and `-a`/`-F` autosplit each line into `@F` (awk territory, useful here for field edits). `BEGIN { }` and `END { }` blocks run before and after the loop, the direct analogue of sed's `1i`/`$a` setup-and-teardown tricks.

```
sed -n '/error/p' log.txt                  # print error lines
sed 's/[0-9]\+/N/g' input.txt              # mask numbers
```

```
perl -ne 'print if /error/' log.txt        # print error lines
perl -pe 's/\d+/N/g' input.txt             # mask numbers
perl -nE 'END { say $n } $n++ if /error/' log.txt   # count, BEGIN/
↪END
```

Each of those expands to an explicit loop: `perl -ne 'CODE'` is `while (<$fh>) { CODE }`, and `perl -pe 'CODE'` is the same with a trailing `print`. Written out, the "count error lines" form:

```
use v5.44;
open my $fh, '<', '/tmp/data.txt' or die "open: $!";
my $n = 0;
while (<$fh>) {
    $n++ if /banana/;          # the body of -ne
}
say $n;                        # 1
```

The gotcha: when a one-liner grows past a line or two, write out the explicit loop it expands to. Consult the *One-Liners guide* for the full switch catalogue and dozens of recipes - it is the highest-leverage thing to read coming from a sed background. Perl also has full subroutines, references, modules, and OO if your script outgrows a one-liner; see the *Object-oriented Programming guide* for that side of the language.

Gotchas coming from sed

The traps, collected for a final scan before you write:

- **`s///` returns a count, not the new string.** `my $x = $s =~ s/a/b/` puts the substitution *count* in `$x`. Copy first (`(my $t = $s) =~ s/.../.../`) or use the `/r` flag to get a new string.
- **`tr///` (sed's y) is a literal character set, not a regex.** `tr/.*//` matches the two characters `.` and `*`. For patterns, use `s///`.
- **`..` in a boolean is the flip-flop range, not a list range.** In an `if`, `/a/ .. /b/` is the sed `addr1,addr2` selector; `1 .. 10` in list context builds a list. Same operator, decided by context.
- **`<$fh>` keeps the trailing newline,** like sed's pattern space. `print` round-trips it; `chomp` only when you mean to strip it.
- **There is no hidden auto-print.** sed's `-p` default is something you spell yourself with `print`. "Delete a line" is just not printing it (`next if /pat/`), not a special command.
- **`open` does not raise on failure.** Use `open my $fh, '<', $path or die "...: $!"`. `$!` is the system error string.
- **The hold space is gone - use a variable.** Any sed script reaching for h/H/x translates to a named Perl variable, array, or hash. This is a simplification, not a loss.

Awk to Perl

You know awk. You think in `pattern { action }`, you split every line into `$1`, `$2`, `$3` without asking, you count things with `a[key]++`, and you frame the whole job with `BEGIN { }` and `END { }`. Perl was written partly as "awk, but bigger" - almost everything you do in awk has a direct Perl form, and then Perl keeps going: real data structures, full regex, modules, and a language that does not run out of room when the script grows.

The one idea to carry in from the first line: **Perl is a near-superset of awk.** The record loop, the fields, the associative arrays, the `BEGIN`/`END` keywords, `print` and `printf`, `length`/`substr`/`split` - they all map across, often by the *same name*. What changes is that fields become a real array `@F` you can `push` and `pop`, the associative array becomes a Perl hash you can nest, and the regex engine is Perl's own - the one PCRE copies - rather than POSIX ERE. This guide leads with the one-to-one mappings and points out the few places the analogy leaks.

Every runnable example here was executed on `pperl`, a Rust reimplementation of Perl 5.44; the `# ...` comments show its real output. Start scripts with:

```
use v5.44;
```

That one line turns on `strict`, `warnings`, and the `say` built-in (a `print` with a trailing newline), the modern feature set used throughout.

How to read this

- *The line/record loop* - awk's implicit `{ ... }` becomes `while (<$fh>)` and `$_`.
- *Fields and splitting* - `$1`/`$NF`/`NF`/`FS` become `@F`, `$F[0]`, `split`, `-F`.
- *Regex and substitution* - `gsub`/`sub`/`match` become `s///`/`=~`, a strict superset.
- *Addresses and ranges* - `/re/`, `cond { }`, and `/a/,/b/` become postfix `if` and the flip-flop.
- *I/O, pipes, in-place edit* - `getline`, redirection, and `-i`.
- *One-liners and switches* - `-n`, `-a`, `-F`, `BEGIN`/`END`.
- *Control flow* - `next`, `for`, the postfix forms.
- *Variables and data* - associative arrays become hashes, the happy mapping.
- *Gotchas coming from awk* - scan before you write.

The line/record loop

awk runs your program once per input record, with the record in `$0` and the action in `{ ... }`. Perl's equivalent is an explicit loop with the record in `$_`, the implicit topic variable. The `-n` switch reproduces awk's "run this block per line"; awk's `NR` (record number) is Perl's `$.`.

```
awk '{ print }' input.txt
```

```
perl -ne 'print' input.txt
# alice 85 90
# bob 70 60
# carol 95 100
# alice 50 55
```

The same loop written out in full - what `-n` expands to - opens the file and iterates it explicitly. `$_` is the current record, exactly like awk's `$0`:

```
use v5.44;
open my $fh, '<', '/tmp/scores.txt' or die "open: $!";
while (<$fh>) {           # each line lands in $_, like awk's $0
    chomp;
    say "line $. : $_";   # $. is awk's NR
}
# line 1 : alice 85 90
# line 2 : bob 70 60
# line 3 : carol 95 100
# line 4 : alice 50 55
```

(The input file held `alice 85 90`, `bob 70 60`, `carol 95 100`, `alice 50 55`.) The gotcha: a line read with `<$fh>` keeps its trailing newline, unlike awk's `$0`, which has the record separator stripped. Call `chomp` to strip it (chomp is the closest thing to "awk already removed the newline for you"). The record number is `$.` = awk's `NR`; when you read several files with `<>`, `$.` keeps counting across them, so the awk `FNR` (per-file counter) reset is something you arrange yourself by checking `eof`.

Fields and splitting

This is awk's heart, and Perl gives you the same picture with one twist: the fields live in a real array `@F`, so awk's `$1` is Perl's `$F[0]` (zero-based), `$0` is `$_`, and `NF` is the count `scalar @F`. Under the `-a` switch Perl autosplits each line into `@F` exactly as awk autosplits into `$1`..`$NF`; written out, that is one explicit `split`.

```
awk '{ print $1, NF }' input.txt          # first field, field count
awk -F, '{ print $1 }' input.txt          # comma separator (FS)
```

```
perl -ane 'print "$F[0] @{[scalar @F]}\n"' input.txt   # -a fills @F
perl -F, -ane 'print "$F[0]\n"' input.txt              # -F sets the␣
↪split pattern
```

The full-script form does the split itself. Bare `split` with no arguments splits `$_` on whitespace and discards leading blanks - exactly awk's default `FS`:

```
use v5.44;
open my $fh, '<', '/tmp/scores.txt' or die "open: $!";
while (<$fh>) {
    my @F = split;                          # awk's automatic field␣
↪split
    say "$F[0] has ", scalar(@F) - 1, " scores"; # $F[0]=$1, @F=NF
```

(continues on next page)

(continued from previous page)

```
}
# alice has 2 scores
# bob has 2 scores
# carol has 2 scores
# alice has 2 scores
```

Arithmetic across fields is the everyday awk action `print $1, $2+$3`, and it reads almost identically:

```
use v5.44;
open my $fh, '<', '/tmp/scores.txt' or die "open: $!";
while (<$fh>) {
    my @F = split;
    say "$F[0]: ", $F[1] + $F[2];   # awk: print $1, $2+$3
}
# alice: 175
# bob: 130
# carol: 195
# alice: 105
```

awk's `$NF` (the last field) is Perl's negative index `$F[-1]`, and a non-default `FS` is the pattern argument to `split`:

```
use v5.44;
open my $fh, '<', '/tmp/data.csv' or die "open: $!";  # alice,85,90
↪...
while (<$fh>) {
    chomp;
    my @F = split /,/;          # awk -F,
    say $F[-1];                 # awk's $NF: last field
}
# 90
# 60
# 100
```

To join fields back with a separator (awk's `OFS`), set `$,` (the output field separator that `print`/`say` inserts between its list arguments):

```
use v5.44;
open my $fh, '<', '/tmp/data.csv' or die "open: $!";
while (<$fh>) {
    chomp;
    my @F = split /,/;
    local $, = "|";             # OFS
    say @F;                     # join the list with OFS
}
# alice|85|90
# bob|70|60
# carol|95|100
```

The gotcha: fields are **zero-based**. awk's `$1` is `$F[0]`, `$2` is `$F[1]`, and `$0` (the whole

record) is `$_`, not an element of `@F`. `NF` is `scalar @F` (or `$#F + 1`); the last field is `$F[-1]`, not `$F[NF]`. And `split` with **one space as a string** (`split ' '`) is the special whitespace-and-strip mode; `split / /` (a regex with one space) is literal and does *not* strip leading blanks. Plain `split` with no argument is the awk-default mode you want most of the time.

Regex and substitution

awk's `sub`, `gsub`, `match`, and `~` all map onto Perl's single regex syntax: `=~` binds a pattern to a string, `s///` substitutes, `s///g` is global (awk's `gsub` vs `sub`). Perl's engine is PCRE, a strict superset of awk's POSIX ERE, so every awk pattern works and Perl adds backreferences, lookaround, named captures, and the `/e`, `/r`, `/x` flags awk has no equivalent for.

```
awk '/^alice/ { sub(/alice/, "ALICE"); print }' input.txt
```

```
perl -ne 'if (/^alice/) { s/alice/ALICE/; print }' input.txt
# ALICE 85 90
# ALICE 50 55
```

The full-script form:

```
use v5.44;
open my $fh, '<', '/tmp/scores.txt' or die "open: $!";
while (<$fh>) {
    next unless /^alice/;   # awk: /^alice/ { ... }
    s/alice/ALICE/;         # awk: sub(/alice/, "ALICE") (first
 ↪match)
    print;
}
# ALICE 85 90
# ALICE 50 55
```

Use `s///g` for awk's `gsub` (every match), and capture groups land in `$1`, `$2` just as awk's `match` populates `RSTART`/`RLENGTH` plus the `gawk` array. The `/e` flag runs **arbitrary Perl** in the replacement, which awk cannot do:

```
use v5.44;
my $s = "items: 3 and 4";
(my $t = $s) =~ s/(\d+)/$1 * 10/ge;   # /e: replacement is Perl code
say $t;                               # items: 30 and 40
```

`length`, `substr`, `index`, and the case functions map by name - watch the indexing:

```
use v5.44;
my $s = "alice";
say length $s;          # 5         (awk length($s))
say substr $s, 0, 3;    # ali       (awk substr($s,1,3) - awk is 1-
 ↪based!)
say uc $s;              # ALICE     (awk toupper($s))
```

The gotcha: `substr` is **zero-based** where awk's `substr` is one-based, so awk

`substr($s,1,3)` is Perl `substr($s,0,3)`. And `s///` edits its target in place and returns the **count** of substitutions, not the new string - so `my $new = ($s =~ s/a/b/)` puts a number in `$new`. Copy first (`(my $t = $s) =~ s/.../.../`) or use the `/r` flag to get a fresh string. POSIX character classes (`[[:digit:]]`) work, but the Perl shorthands (`\d`, `\w`, `\s`) are shorter and what you will see in Perl code.

Addresses and ranges

An awk rule is `pattern { action }`: the action runs only on records matching the pattern. In Perl that pattern is just a condition on the statement - a postfix `if` testing `$_`. A bare `/re/` matches against `$_`, a numeric test uses `$.` (awk's NR), and awk's range `/start/,/end/` is Perl's **flip-flop** operator `/start/ .. /end/`.

```
awk 'NR==2'              input.txt       # print record 2
awk '/banana/'           input.txt       # print matching records
awk '/bob/,/carol/'      input.txt       # range, inclusive
```

```
perl -ne 'print if $. == 2'            input.txt
perl -ne 'print if /banana/'           input.txt
perl -ne 'print if /bob/ .. /carol/'   input.txt
```

The full-script flip-flop, true from the line matching the left pattern through the line matching the right, inclusive - exactly awk's two-pattern range:

```
use v5.44;
open my $fh, '<', '/tmp/scores.txt' or die "open: $!";
while (<$fh>) {
    print if /bob/ .. /carol/;   # awk: /bob/,/carol/
}
# bob 70 60
# carol 95 100
```

The gotcha: a bare `pattern` with no action in awk means "print the record"; in Perl the loop is explicit, so you write the `print` (or `say`) yourself. And in a boolean test `..` is the flip-flop (range) operator, **not** the list-range operator (`1 .. 10` builds a list). Perl tells them apart by context: in the scalar position of an `if`, `..` is the stateful line-range selector that mirrors awk's `addr1,addr2`.

I/O, pipes, in-place edit

awk reads every file named on its command line automatically and has `getline` for explicit reads. Perl's `<>` (the diamond operator) reads each file in `@ARGV` in turn - the same default - and `<$fh>` is the explicit `getline`. Writing to a command or reading from one (awk's `print | "cmd"` and `"cmd" | getline`) is a pipe filehandle.

```
awk '{ print > "out.txt" }' input.txt      # redirect output
awk '{ print | "sort" }'     input.txt     # pipe to a command
```

```
use v5.44;
open my $out, '>', '/tmp/out.txt' or die "write: $!";
```

(continues on next page)

(continued from previous page)

```
open my $in,  '<', '/tmp/scores.txt' or die "read: $!";
while (<$in>) {
    print {$out} $_;        # awk: print > "out.txt"
}
close $out;
```

For pipes, `open my $fh, '|-', 'sort'` writes to a command's input and `open my $fh, '-|', 'ls', '-l'` reads a command's output - awk's `| "cmd"` in both directions.

awk has no built-in in-place edit; you reach for a tmp-file shuffle or GNU `gawk -i inplace`. Perl spells it `-i` on a one-liner:

```
perl -i -pe 's/red/crimson/' file.txt      # edit in place
```

Written out in full it is an explicit read-modify-write - slurp the lines, transform, write back - which is exactly what `-i` does under the hood:

```
use v5.44;
my $path = '/tmp/inplace.txt';          # holds: red green blue

open my $in, '<', $path or die "read: $!";
my @lines = <$in>;                      # slurp all lines (list
 ↪context)
close $in;

s/e/E/g for @lines;                     # transform each line's $_

open my $out, '>', $path or die "write: $!";
print {$out} @lines;                    # write them back
close $out;
# file now holds: rEd / grEEn / bluE
```

The gotcha: open returns false on failure rather than aborting, so the `or die "...: $!"` idiom is mandatory - `$!` is the system error string, awk's silent failure made explicit. And `print {$out} ...` needs the braces around the filehandle when it is a variable; the bare `print FH ...` form (no comma) is for named handles, which is a common first-day stumble.

One-liners and switches

The one-liner is awk's home turf and Perl's too. The switch family that reproduces awk: `-e` supplies the program, `-n` wraps it in a per-record loop (awk's implicit `{ }`), `-p` does the same with an auto-print at the end of each cycle, `-a` autosplits into `@F` (awk's automatic field split), `-F` sets the split pattern (awk's `FS`), and `-l` handles line endings (awk's `ORS`/`RS` convenience). `BEGIN { }` and `END { }` are the **same keywords** as awk, running before and after the loop.

```
awk '/error/ { print }'          log.txt        # print error
 ↪records
```

(continues on next page)

(continued from previous page)

```
awk '{ s += $NF } END { print s }' input.txt    # sum the last field
awk -F, '{ print $1 }'             input.txt    # first CSV field
```

```
perl -ne 'print if /error/'                     log.txt
perl -ane 'END { print "$s\n" } $s += $F[-1]'    input.txt
perl -F, -ane 'print "$F[0]\n"'                  input.txt
```

Each of those expands to an explicit loop: `perl -ne 'CODE'` is `while (<$fh>) { CODE }`, `-ane` adds `my @F = split` at the top, and `BEGIN`/`END` blocks work in a full script exactly as in awk. Written out, the "sum a column with BEGIN/END" form:

```
use v5.44;
my $n = 0;
my $sum = 0;
BEGIN { say "start" }              # runs before the loop, like awk␣
↪BEGIN
END   { say "lines=$n sum=$sum" }  # runs after, like awk END
open my $fh, '<', '/tmp/scores.txt' or die "open: $!";
while (<$fh>) {
    my @F = split;
    $n++;
    $sum += $F[1];                 # awk: sum += $2
}
# start
# lines=4 sum=300
```

The gotcha: when a one-liner grows past a line or two, write out the explicit loop it expands to. Consult the *One-Liners guide* for the full switch catalogue and dozens of recipes - it is the highest-leverage thing to read coming from an awk background. Perl also has full subroutines, references, modules, and OO if your script outgrows a one-liner; see the *Object-oriented Programming guide* and the other coming-from guides.

Control flow

awk's control keywords map onto Perl's almost name-for-name. awk's `next` (skip to the next record) is Perl's `next`; `break`/`continue` in awk loops are Perl's `last`/`next`; awk's `for (k in arr)` becomes a Perl `for` over `keys`. The braces and `;` are the same; Perl adds the postfix forms that read like awk patterns.

```
awk '{ if ($2 < 70) next; print }' input.txt
```

```
use v5.44;
open my $fh, '<', '/tmp/scores.txt' or die "open: $!";
while (<$fh>) {
    my @F = split;
    next if $F[1] < 70;        # awk: if ($2 < 70) next
    print;
}
# alice 85 90
```

(continues on next page)

(continued from previous page)

```
# carol 95 100
```

The gotcha: Perl's comparison operators come in two families and you pick by the *operands' type*, where awk has only </== for both. Numbers use ==, <, >, <=>; strings use `eq`, `lt`, `gt`, `cmp`. awk decides numeric-vs-string comparison by the values at run-time; Perl makes you choose the operator. Using == on strings (or `eq` on numbers) is a frequent, quiet bug - `"abc" == "xyz"` is true in Perl (both numify to 0), which is almost never what you meant.

Variables and data

This is the happiest mapping in the guide: awk's associative array is a Perl **hash**, and the daily idioms are near-identical. awk's `count[$1]++` is Perl's `$count{$F[0]}++`; `for (k in count)` is `for my $k (keys %count)`; `if (k in count)` is `exists $count{$k}`. The one shift is the sigil - the whole hash is `%count`, but one value is `$count{$key}`.

```
awk '{ total[$1] += $2 + $3 }
     END { for (k in total) print k, total[k] }' input.txt
```

```
use v5.44;
open my $fh, '<', '/tmp/scores.txt' or die "open: $!";
my %total;
while (<$fh>) {
    my @F = split;
    $total{$F[0]} += $F[1] + $F[2];   # awk: total[$1] += $2 + $3
}
for my $name (sort keys %total) {     # awk END loop over the array
    say "$name $total{$name}";
}
# alice 280
# bob 130
# carol 195
```

The two-file `NR==FNR` idiom - load keys from the first file, then test membership in the second - is one explicit loop per file in Perl, which is clearer than the awk one-liner it replaces:

```
use v5.44;
my %seen;
open my $f1, '<', '/tmp/data.csv' or die "open: $!";
while (<$f1>) {
    my ($k) = split /,/;          # first field only
    $seen{$k} = 1;
}
close $f1;
open my $f2, '<', '/tmp/scores.txt' or die "open: $!";
while (<$f2>) {
    my ($k) = split;
```

(continues on next page)

(continued from previous page)

```
    print if $seen{$k};           # awk: ($1 in seen)
}
# alice 85 90
# bob 70 60
# carol 95 100
# alice 50 55
```

The gotcha: a hash has no inherent order (awk's `for (k in arr)` is also unordered, so this will not surprise you), so iterate `sort keys %h` when you want a stable order. Membership is `exists $h{$k}`, not awk's `k in arr` - and merely *reading* `$h{$k}` to test it will autovivify the key into existence in some contexts, where `exists` never does. Numbers and strings auto-coerce in Perl as in awk, but remember the `==` vs `eq` split from *Control flow*.

Gotchas coming from awk

The traps, collected for a final scan before you write:

- **Fields are zero-based.** awk's `$1` is `$F[0]`, `$2` is `$F[1]`, and the whole record `$0` is `$_`. `NF` is `scalar @F`; the last field is `$F[-1]`, not `$F[NF]`.
- **`<$fh>` keeps the trailing newline,** unlike awk's `$0`. `chomp` to strip it.
- **`substr` is zero-based,** where awk's is one-based: awk `substr($s,1,3)` is Perl `substr($s,0,3)`.
- **`s///` returns a count, not the new string.** `my $x = $s =~ s/a/b/` puts the substitution *count* in `$x`. Copy first, or use the `/r` flag to get a new string.
- **Two comparison families.** Numbers use `==`/`<`/`<=>`; strings use `eq`/`lt`/`cmp`. awk's single `==`/`<` is split in two, chosen by operator, not by value. `"abc" == "xyz"` is true (both numify to 0).
- **Bare `split` is the awk-default mode.** `split` (no args) splits `$_` on whitespace and strips leading blanks. `split / /` (a regex with one space) is literal and does *not* strip - use plain `split` for awk's default `FS`.
- **Membership is `exists $h{$k}`, not `k in arr`.** Testing a key by reading `$h{$k}` can autovivify it; `exists` never does.
- **open does not abort on failure.** Use `open my $fh, '<', $path or die "... : $!"`. `$!` is the system error string.

Shell to Perl

You know the shell. You glue programs together with pipes, you reach for `sed`, `awk`, `grep`, and `cut` in a pipeline without thinking, you capture command output with `$(...)`, and you script control flow with `for f in *`, `while read line`, and `case`. This guide maps each of those habits onto Perl, so you can keep the pipeline reflexes and pick up the things the shell makes painful.

When a shell script grows past about 50 lines of quoting and `sed`/`awk`/`grep` pipelines, Perl is the natural next step: it does all of it in one language, with real data structures instead of whitespace-delimited strings. The framing to carry in: **Perl is the shell pipeline, but**

the filter language is a real language. You still open pipes, you still glob, you still run commands and read their output. What changes is that between the pipes you now have `@arrays`, `%hashes`, real regex, and arithmetic that needs no `$(( ))`. The shell's one data type is the string; Perl gives you actual containers.

Every runnable example here was executed on `pperl`, a Rust reimplementation of Perl 5.44; the `# ...` comments show its real output. Start scripts with:

```
use v5.44;
```

That one line turns on `strict`, `warnings`, and the `say` built-in (a `print` with a trailing newline), the modern feature set used throughout.

How to read this

- *The record loop* - `while read line` becomes `while (<$fh>)` and `$_`.
- *Fields and splitting* - `cut`/`IFS`/`awk` fields become `split` and a real array.
- *Regex and substitution* - the shell's `${var#...}`/`${var/a/b}` parameter expansions become real regex.
- *Addresses and ranges* - `grep -n`, `awk NR==2`, and line ranges.
- *I/O, pipes, and in-place edit* - the heart: `open`, `-|`/`|-`, redirection, here-docs, `$(...)`.
- *One-liners and switches* - `-n`, `-p`, `-e`, `BEGIN`/`END`.
- *Control flow* - `for`/`while`/`case`/`if`, file tests, exit status.
- *Variables and data* - `$var` becomes `my $var`, plus the `@arrays` and `%hashes` the shell never had.
- *Gotchas coming from the shell* - scan before you write.

The record loop

The shell's line-at-a-time idiom is `while read line; do ... done`, which reads one line per iteration into `$line` (and strips it according to `IFS`). Perl's equivalent is `while (<$fh>)`, which reads one line per iteration into `$_`, the implicit topic variable. The crucial difference: Perl keeps the trailing newline, so you `chomp` it yourself, exactly where the shell's `read` would have eaten it.

```
while read line; do
    echo "got: $line"
done < data.txt
```

```
use v5.44;
open my $fh, '<', '/tmp/data.txt' or die "open: $!";
while (my $line = <$fh>) {      # each line, with its newline
    chomp $line;                # strip it, like read does
    say "got: $line";
}
close $fh;
# got: apple
```

(continues on next page)

(continued from previous page)

```
# got: banana
# got: cherry
```

(The input file held `apple`, `banana`, `cherry`.) Read with the bare `<$fh>` and the line lands in `$_`, the default that most line built-ins operate on. The line number is in `$.`. This is the form to reach for when you are translating a filter:

```
use v5.44;
open my $fh, '<', '/tmp/data.txt' or die "open: $!";
while (<$fh>) {                 # line lands in $_
    chomp;                      # chomp $_ by default
    say "$. : $_";              # $. is the line number
}
close $fh;
# 1 : apple
# 2 : banana
# 3 : cherry
```

The gotcha: `read` splits on `IFS` and trims; `<$fh>` does neither. A Perl line read is the whole raw line including the `\n`, so `chomp` is your reflex, and field-splitting is a separate, explicit step (the next section). What you lose in automatic trimming you gain in not having `IFS` quietly mangle your data.

Fields and splitting

In the shell you reach fields with `cut -d: -f1`, with `awk '{print $1}'`, or by setting `IFS` and letting `read` split for you. Perl's one answer is `split`: give it a separator and a string, get a real array back. Positional fields that the shell numbers `$1`, `$2` become array elements `$f[0]`, `$f[1]` - **0-based**, unlike `awk`'s 1-based `$1`.

```
line='root:x:0:0:root:/root:/bin/bash'
user=$(echo "$line" | cut -d: -f1)
shell=$(echo "$line" | cut -d: -f7)
echo "$user $shell"
```

```
use v5.44;
my $line = "root:x:0:0:root:/root:/bin/bash";
my @f = split /:/, $line;
say "user: $f[0]";             # user: root
say "shell: $f[-1]";           # shell: /bin/bash   (negative index)
say "fields: ", scalar @f;     # fields: 7
```

To split on runs of whitespace - the shell's default word-splitting, and `awk`'s default - use the special pattern `split ' '` (a literal single-space string, which also skips leading blanks). To join fields back, `join`:

```
use v5.44;
my @words = split ' ', "  the   quick brown  ";
say scalar @words;             # 3   (leading/inner blanks collapsed)
say join "|", @words;          # the|quick|brown
```

The gotcha: `split /:/` with a literal pattern keeps empty trailing fields only up to a point - by default Perl drops trailing empty fields, so `split /:/, "a::"` yields `("a")`, not `("a", "", "")`. Pass a negative limit (`split /:/, $s, -1`) to keep them, the equivalent of caring about trailing empty `cut` fields. And `split ' '` (the magic-whitespace form) is **not** the same as `split / /` (a single literal space): the first collapses runs and trims, the second does not.

Regex and substitution

The shell's `${}` parameter expansions become real regex - and this is a large upgrade. `${var#prefix}`, `${var%suffix}`, `${var/a/b}`, and `${var//a/b}` are each a special-purpose, limited string edit. Perl replaces the whole family with one operator, `s///`, that takes an actual regular expression and the full set of flags.

```
f='report.txt.bak'
base=${f%.bak}              # strip suffix
path='/usr/local/bin'
tail=${path##*/}            # basename
dir=${path%/*}              # dirname
s='a-b-c'
one=${s/-/_}                # replace first
all=${s//-/_}               # replace all
```

```
use v5.44;
my $f = "report.txt.bak";
(my $base = $f) =~ s/\.bak$//;       # ${f%.bak}
say $base;                           # report.txt

my $path = "/usr/local/bin";
(my $tail = $path) =~ s{.*/}{};      # ${path##*/}  basename
say $tail;                           # bin
(my $dir = $path) =~ s{/[^/]*$}{};   # ${path%/*}   dirname
say $dir;                            # /usr/local

my $s = "a-b-c";
(my $one = $s) =~ s/-/_/;            # ${s/-/_}   first only
say $one;                            # a_b-c
(my $all = $s) =~ s/-/_/g;           # ${s//-/_}  /g = global
say $all;                            # a_b_c
```

Because it is a real regex, the things the shell cannot do at all are trivial: capture groups land in `$1`, `$2`; the `/e` flag runs Perl code in the replacement; the `/r` flag returns a copy instead of editing in place; and `/i`, `/x` modify matching.

```
use v5.44;
my $s = "items: 3 and 4";
(my $t = $s) =~ s/(\d+)/$1 * 10/ge;  # /e: replacement is Perl code
say $t;                              # items: 30 and 40

my $date = "2026-06-21";
my $us = $date =~ s{(\d+)-(\d+)-(\d+)}{$2/$3/$1}r;  # /r: return a
```

(continues on next page)

(continued from previous page)

```
↪copy
say $us;                                # 06/21/2026
say $date;                              # 2026-06-21   (unchanged)
```

The gotcha: bare `s///` edits its target and returns the *count* of substitutions, not the new string - so the examples copy first with `(my $t = $s) =~ ...` or use the `/r` flag. Writing `my $new = ($s =~ s/a/b/)` puts a number in `$new`. This is the single most common first-day surprise. Note also that the delimiters are free: `s{...}{...}` and `s#...#...#` save you backslashing the slashes in a path pattern, the way the shell's `${path##*/}` already avoids them.

Addresses and ranges

Selecting lines by number or pattern - `grep`, `grep -n`, `awk 'NR==2'`, `sed -n '/a/,/b/p'` - becomes a postfix `if` on a condition inside the loop. The line number is `$.`; the pattern test is `/pat/` against `$_`.

```
grep -n an data.txt                 # number + matching lines
awk 'NR==2' data.txt                # the second line
```

```
use v5.44;
open my $fh, '<', '/tmp/data.txt' or die "open: $!";
while (<$fh>) {
    chomp;
    say "$. : $_" if /an/;          # grep -n 'an'
}
close $fh;
# 2 : banana
```

A line range - the sed/awk `/start/,/end/` selector - becomes Perl's **flip-flop** range operator `/start/ .. /end/`, true from the line matching the left pattern through the line matching the right, inclusive:

```
use v5.44;
open my $fh, '<', '/tmp/data.txt' or die "open: $!";
while (<$fh>) {
    print if /banana/ .. /cherry/;   # the /start/,/end/ range
}
close $fh;
# banana
# cherry
```

The gotcha: in a boolean test `..` is the flip-flop (range) operator, **not** the list-range operator (which is `1 .. 10` building a list). Perl tells them apart by context - in the boolean position of an `if`, `..` is the stateful line-range selector; in list context it builds a sequence. Same two dots, decided by where you write them.

I/O, pipes, and in-place edit

This is the heart of the guide, because it is the heart of shell scripting. Everything you do with `|`, `>`, `>>`, `2>&1`, `<<EOF`, and `$(...)` has a direct Perl spelling, and once a script outgrows a single pipeline this is the section that pays off.

Command substitution. The shell's `$(cmd)` and backticks become Perl's backticks or `qx{...}`. The capture is context-sensitive: in scalar context you get the whole output as one string; in list context you get one element per line.

```
out=$(echo hello)
mapfile -t lines < <(seq 1 3)
```

```
use v5.44;
my $out = `echo hello`;           # scalar: whole output, one string
chomp $out;
say "got: $out";                  # got: hello

my @lines = qx{seq 1 3};          # list: one element per line
chomp @lines;
say "lines: @lines";              # lines: 1 2 3
```

When you do not need the output, `system()` runs a command and gives you its exit status (see *Control flow* for `$?`). Pass `system` a *list* and no shell is involved - the safe form that avoids quoting hell:

```
use v5.44;
system('mkdir', '-p', '/tmp/sh_demo');   # list form: no shell, no␣
↪quoting
say "made it" if -d '/tmp/sh_demo';      # made it
```

Pipes. A pipeline stage - reading a command's output or feeding a command's input - is an open with a pipe mode. `open my $fh, '-|', LIST` reads a command's stdout; `open my $fh, '|-', LIST` writes to a command's stdin. The list form (command plus args as separate strings) runs no shell, so there is nothing to quote.

```
seq 1 3 | while read n; do echo "n=$n"; done    # read from a␣
↪command
printf '%s\n' banana apple cherry | sort         # write to a␣
↪command
```

```
use v5.44;
open my $ph, '-|', 'seq', '1', '3' or die "pipe: $!";  # read␣
↪command output
my @nums = <$ph>;
close $ph;
chomp @nums;
say "got: @nums";                 # got: 1 2 3
```

```
use v5.44;
open my $sort, '|-', 'sort' or die "pipe: $!";  # write to command␣
```

(continues on next page)

(continued from previous page)

```
↪stdin
print {$sort} "$_\n" for qw(banana apple cherry);
close $sort;
# apple
# banana
# cherry
```

Redirection. > and >> are the open modes `'>'` (truncate) and `'>>'` (append); a three-arg open with a lexical handle is the idiom. `print {$fh} ...` writes to a handle. To merge a child's stderr into the captured stream (`2>&1`), keep the redirection in the command you hand to backticks:

```
use v5.44;
open my $fh, '>', '/tmp/sh_out.txt' or die "write: $!";
print {$fh} "line $_\n" for 1 .. 3;
close $fh;

my $both = `sh -c 'echo to-out; echo to-err 1>&2' 2>&1`;  # merge␣
↪stderr
print $both;
# to-out
# to-err
```

Here-docs. The shell's `<<EOF` (interpolating) and `<<'EOF'` (literal) map one-to-one to Perl's `<<"END"` and `<<'END'`. Double quotes on the terminator interpolate variables; single quotes do not.

```
use v5.44;
my $name = "world";
print <<"END";                # interpolating, like <<EOF
hello $name
END
print <<'END';                # literal, like <<'EOF'
no $interpolation here
END
# hello world
# no $interpolation here
```

In-place edit. `sed -i` rewrites a file in place; the idiomatic Perl is the same switch on a `-p` one-liner. Written out in full it is the explicit read-modify-write that `-i` performs under the hood - slurp, transform, write back:

```
use v5.44;
my $path = '/tmp/inplace_data.txt';   # holds: red green blue

open my $in, '<', $path or die "read: $!";
my @lines = <$in>;                    # slurp all lines
close $in;

s/e/E/g for @lines;                   # transform each line's $_
```

(continues on next page)

(continued from previous page)

```
open my $out, '>', $path or die "write: $!";
print {$out} @lines;                    # write them back
close $out;
# file now holds:
# rEd
# grEEn
# bluE
```

The gotcha: open returns false on failure rather than raising, so the `or die "...: $!"` idiom is mandatory - `$!` is the system error string, the shell's silent `2>/dev/null` failure made loud and explicit. And when you build a command from variables, prefer the **list form** of `system`/open (`'ls', '-l', $dir`) over a single interpolated string: the list form bypasses the shell entirely, so there is no word splitting, no glob expansion, and no quoting to get wrong - the cure for the shell's quoting hell.

One-liners and switches

The one-liner is the shell developer's home turf and Perl's too: the filter you drop into a pipeline. The switch family that makes it work: `-e` supplies the program, `-n` wraps it in a no-auto-print loop, `-p` wraps it in an auto-print loop, `-i` edits files in place, `-l` handles line endings, and `-a`/`-F` autosplit each line into `@F` (the direct `awk` analogue, fields in `@F` instead of `$1`..`$NF`). `BEGIN { }` and `END { }` run before and after the loop, like `awk`'s `BEGIN`/`END`.

```
grep error log.txt                              # filter lines
awk -F: '{print $1}' /etc/passwd                # first field
awk '{s += $NF} END {print s}' data.txt         # sum last field
```

```
perl -ne 'print if /error/' log.txt             # filter lines
perl -F: -ane 'print "$F[0]\n"' /etc/passwd     # first field, -F sets␣
↪split
perl -lanE '$s += $F[-1]; END { say $s }' data.txt  # sum last field
```

Each of those expands to an explicit loop: `perl -ne 'CODE'` is `while (<$fh>) { CODE }`, and `perl -pe 'CODE'` is the same with a trailing `print`. Written out, the "first field of each line" form:

```
use v5.44;
open my $fh, '<', '/tmp/passwd.txt' or die "open: $!";
while (<$fh>) {                 # the implicit loop of -ane
    chomp;
    my @F = split /:/;          # what -F: gives you in @F
    say $F[0];                  # print the first field
}
close $fh;
# root
# daemon
# bin
```

The gotcha: when a one-liner grows past a line or two, write out the explicit loop it expands to. Consult the *One-Liners guide* for the full switch catalogue and dozens of recipes - it is the highest-leverage thing to read coming from a shell background, where the one-liner is the everyday reflex.

Control flow

The shell's `for`, `while`, `case`, and `if`/`test` map onto Perl loops and conditionals, and several of the mappings are pleasantly direct.

Loops. `for f in *; do` becomes `for my $f (glob "*");` the file glob `*.txt` is `glob "*.txt"` or the readline form `<*.txt>`.

```
for f in /tmp/sh_*.txt; do
    echo "file: $f"
done
```

```
use v5.44;
for my $f (sort glob "/tmp/sh_*.txt") {
    say "file: $f";
}
# file: /tmp/sh_data.txt
# file: /tmp/sh_passwd.txt
```

Conditionals and `test`. This is a happy mapping: the shell's `[ -e ]`, `[ -d ]`, `[ -f ]` file tests are the **same `-X` operators** in Perl. String and numeric comparison split into two families, as they do between `[ "$a" = "$b" ]` and `[ "$a" -eq "$b" ]`: Perl uses `eq`/`ne` for strings and `==`/`!=` for numbers. The shell's `-z`/`-n` (empty / non-empty string) become `!length` / `length`.

```
if [ -f "$f" ] && [ -n "$name" ]; then
    echo ok
fi
[ "$a" = "$b" ]        # string equality
[ "$x" -eq "$y" ]      # numeric equality
```

```
use v5.44;
my $f = '/tmp/sh_data.txt';
my $name = 'set';
say "ok" if -f $f && length $name;   # same -f, same idea
# ok

say "abc" eq "abc" ? "eq" : "ne";    # string compare
say 10 == 10.0 ? "num-eq" : "num-ne"; # numeric compare
# eq
# num-eq
```

`case`. The shell's `case` has no single keyword twin; the idiomatic Perl is a `for`/`if` chain or a ternary cascade (or, for many branches, a dispatch hash mapping keys to code refs).

```
case "$cmd" in
    start)   echo starting ;;
    stop)    echo stopping ;;
    *)       echo "unknown: $cmd" ;;
esac
```

```
use v5.44;
for my $cmd (qw(start stop bogus)) {
    my $r =
        $cmd eq 'start' ? "starting"
      : $cmd eq 'stop'  ? "stopping"
      :                   "unknown: $cmd";
    say "$cmd -> $r";
}
# start -> starting
# stop -> stopping
# bogus -> unknown: bogus
```

Exit status. After a `system()`, the child's status is in `$?`, exactly the shell variable. But `$?` holds the raw `wait` status, not the bare exit code: shift right by 8 to get the code the shell would show as `$?`. The loop keywords `last` and `next` are the shell's `break` and `continue`.

```
use v5.44;
system('sh', '-c', 'exit 3');
say "raw \$? = $?";           # raw $? = 768
say "exit code = ", $? >> 8;  # exit code = 3
```

The gotcha: `$? >> 8` is the exit code; the low 8 bits hold the signal that killed the child (if any). The shell hides this behind its own `$?`; in Perl you do the shift yourself. Forgetting it is why `if ($? == 1)` after a command that exited 1 is almost always wrong - the raw value is 256, not 1.

Variables and data

A shell variable is always a string - `var=value`, used as `$var` or `${var}`, and quoted as `"$var"` to survive word splitting. Perl's scalar is `my $var`, lexically scoped, and **no quoting dance is needed**: a Perl scalar holds one value and does not get re-split on use. Where the shell forces everything through strings, Perl adds the two containers the shell never had: ordered `@arrays` and key/value `%hashes`.

```
name="Ada"
greeting="Hello, ${name}"
langs="Perl Python"          # a "list" is really a string + IFS
```

```
use v5.44;
my $name = "Ada";
my $greeting = "Hello, $name";   # interpolation, like "${name}"
my @langs = ("Perl", "Python");  # a real list, not a split string
say $greeting;                   # Hello, Ada
```

(continues on next page)

(continued from previous page)

```
say "first: $langs[0]";          # first: Perl
say "count: ", scalar @langs;    # count: 2
```

Script arguments. The shell's positional parameters map to `@ARGV`. The quoting minefield of `"$@"` versus `$*` simply does not exist: `@ARGV` is a real array, each argument a clean element no matter what spaces or globs it contained. `$1`..`$9` become `$ARGV[0]`..`$ARGV[8]`, `$#` (the count) becomes `scalar @ARGV`, and `$0` (the script name) becomes `$0`.

```
echo "count: $#"
echo "first: $1"
echo "all: $@"
```

```
use v5.44;
say "count: ", scalar @ARGV;   # like $#
say "first: $ARGV[0]";         # like $1
say "all: @ARGV";              # like "$@"
# invoked as: pperl script.pl one two three
# count: 3
# first: one
# all: one two three
```

Associative arrays. A Perl hash is the shell's `declare -A` done right - declared with `%`, accessed one value at a time with `$h{key}`, and far more ergonomic than the shell's bolted-on associative arrays.

```
use v5.44;
my %count;
$count{$_}++ for qw(apple pear apple fig apple pear);
for my $k (sort keys %count) {
    say "$k: $count{$k}";
}
# apple: 3
# fig: 1
# pear: 2
```

The gotcha: a shell variable is global by default and a Perl `my` variable is lexically scoped to its block - the opposite default. The shell's `local` (dynamic scoping inside a function) is closest to Perl's `local`, but for ordinary variables you want `my`, which is true lexical scope. And interpolation only happens in double quotes: `"$name"` interpolates, `'$name'` is the literal four characters, just like the shell.

> Perl also has full subroutines, references, modules, and OO if your script outgrows a one-liner - see the *Object-oriented Programming guide* and the other coming-from guides.

Gotchas coming from the shell

The traps, collected for a final scan before you write:

- **`<$fh>` keeps the trailing newline.** Shell `read` trims; Perl does not. `chomp` is your reflex on every line read.
- **`split` is a separate step, and it is 0-based.** Fields are `$f[0]`, `$f[1]`, not awk's 1-based `$1`. `split ' '` (magic whitespace) trims and collapses; `split / /` (a literal space) does not.
- **`s///` returns a count, not the new string.** `my $x = $s =~ s/a/b/` puts the substitution *count* in `$x`. Copy first (`(my $t = $s) =~ s/.../.../`) or use the `/r` flag.
- **`$? >> 8` is the exit code.** The raw `$?` after `system` is the full wait status; the exit code is the high 8 bits. `if ($? == 1)` is almost always a bug; you want `if (($? >> 8) == 1)`.
- **open does not raise on failure.** Use `open my $fh, '<', $path or die "...: $!"`. `$!` is the system error string - the shell's silent failure made explicit.
- **Prefer the list form of `system`/`open` over an interpolated string.** `system('ls', '-l', $dir)` runs no shell: no word splitting, no glob surprise, no quoting hell. The string form (`system "ls -l $dir"`) reintroduces every shell-quoting trap you came here to escape.
- **`my` is lexical; shell variables are global.** Declare with `my` and scope to the block; reach for `local` only for the shell's dynamic-scoping behaviour, which is rarely what you want.
- **String vs numeric comparison is two families.** `eq`/`ne` for strings, `==`/`!=` for numbers - the `[ = ]` vs `[ -eq ]` split, but the operators are reversed from the shell, so read them carefully.

Tcl to Perl

You know Tcl. You write `set x 5`, you substitute with `$x`, you reach for `[expr {...}]` to do arithmetic, you build lists with `lappend` and walk them with `foreach`, and you bend strings with `regexp` and `regsub`. This guide maps each of those habits onto Perl, so you can keep every reflex you have and pick up the things Tcl makes hard.

Tcl and Perl share the same roots: Unix scripting, a string-first worldview, and `$`-substitution as the way you reach a variable. Most of what you know maps almost directly - and Perl gives you real data structures (`@arrays` and `%hashes` as first-class types, not lists pressed into service) and a vast module ecosystem to land in. Two specific pieces of friction get the weight in this guide: Perl's distinct array/hash types where Tcl has lists and assoc arrays, and Perl's regex (a superset of Tcl's). The line-at-a-time and one-liner material is lighter here, because Tcl is not a one-liner language; you will spend your time in variables, data, control flow, and regex.

Every runnable example here was executed on `pperl`, a Rust reimplementation of Perl 5.44; the `# ...` comments show its real output. Start scripts with:

```
use v5.44;
```

That one line turns on `strict`, `warnings`, and the `say` built-in (a `print` with a trailing newline, the rough twin of Tcl's `puts`), the modern feature set used throughout.

How to read this

- *Variables and substitution* - `set` becomes `my`, `[expr]` disappears, `$x` substitution stays.
- *Lists and arrays* - `lindex`/`lappend`/`lsort` become Perl array operations.
- *Assoc arrays and hashes* - `set a(k) v` becomes `$a{k}`; `dict` becomes a hash.
- *Procs and references* - `proc` becomes `sub`; `upvar`/`uplevel` become references.
- *Regex and substitution* - `regexp`/`regsub`/`string map` become `m//`/`s///`/`tr///`.
- *Control flow* - `if`/`while`/`foreach`/`switch` become Perl's, plus `{}` vs `""` quoting.
- *The line/record loop* - reading a file line by line with `$_` and `$.`.
- *Fields and splitting* - `split`/`join`, the Tcl `split` analogue.
- *I/O, pipes, and in-place edit* - open/gets/puts/exec become open/<$fh>/print/qx.
- *One-liners and switches* - `-n`, `-p`, `-e`, kept brief.
- *Gotchas coming from Tcl* - scan before you write.

Variables and substitution

In Tcl, `set x 5` creates a variable and `$x` substitutes its value. In Perl, `my $x = 5;` declares a lexically scoped variable and `$x` substitutes it - the substitution syntax you already know carries over verbatim. The relief is that Perl has native operators: there is no `[expr {...}]`. Arithmetic is just arithmetic.

```
set x 5
set name "world"
set sum [expr {$x + 3}]
puts "x is $x"
puts "sum is $sum"
```

```
use v5.44;
my $x = 5;                  # set x 5
my $name = "world";
my $sum = $x + 3;           # native operators, no [expr] needed
say "x is $x";              # x is 5
say "sum is $sum";          # sum is 8
```

Quoting maps cleanly, and this is a happy mapping: Tcl's `"..."` (substitution happens) is Perl's double-quoted string, and Tcl's `{...}` (literal, no substitution) is Perl's single-quoted string.

```
use v5.44;
my $name = "world";
say "hello, $name";         # hello, world   (interpolates)
say 'no $interp here';      # no $interp here (literal)
```

The gotcha: Tcl is "everything is a string" and Perl is close to it - a scalar auto-coerces between number and string by context - but Perl splits *comparison* into two operator families, and you pick by the operation, not the value. Numbers use ==, !=, <, >; strings use eq, ne, lt, gt.

```
use v5.44;
my $x = "5";
say $x + 3;                 # 8    (numeric context coerces "5")
say $x . "3";               # 53   (string context)
say "10" == "10.0" ? "num-equal" : "num-differ";   # num-equal
say "10" eq "10.0" ? "str-equal" : "str-differ";   # str-differ
```

== compares "10" and "10.0" as numbers (equal); eq compares them as strings (different). Using == where you meant eq, or the reverse, is the one quiet bug the string-first worldview sets you up for.

Lists and arrays

A Tcl list is a Perl array, and the verbs line up one for one. The big shift: in Tcl *everything is a list*, including strings you split on the fly; in Perl an array (@a) is a distinct, first-class type with its own sigil. When you reach for one element, the sigil follows the *element*, so it is $a[0] (a scalar), not @a[0].

```
set items [list apple pear fig]
puts [lindex $items 0]
puts [llength $items]
lappend items plum
set sorted [lsort $items]
foreach it $items { puts "item: $it" }
```

```
use v5.44;
my @items = ("apple", "pear", "fig");
say $items[0];                  # apple        (lindex $items 0)
say scalar @items;              # 3            (llength $items)
push @items, "plum";            # lappend items plum
my @sorted = sort @items;       # lsort
say "@sorted";                  # apple fig pear plum
for my $it (@items) {           # foreach it $items
    say "item: $it";
}
```

lsearch - "find the index of a matching element" - has no single built-in; the idiomatic Perl is grep over the index range, with $#items as the last index:

```
use v5.44;
my @items = ("apple", "pear", "fig", "plum");
my ($idx) = grep { $items[$_] eq "fig" } 0 .. $#items;   # lsearch
say "fig at $idx";              # fig at 2
```

The gotcha: scalar @items is how you get the length (Tcl's llength); writing my $n = @items works because assigning an array to a scalar yields its count, but spell it scalar @items when you want the length inside a larger expression. And sort compares as

strings by default, so `sort (10, 9, 100)` gives `10 100 9`; pass a numeric comparator `sort { $a <=> $b } @nums` for numeric order, where `$a` and `$b` are the two elements and `<=>` is the numeric three-way operator.

Assoc arrays and hashes

A Tcl associative array - `set a(red) 1` - is a Perl hash. The element access maps directly: Tcl's `$a(red)` is Perl's `$a{red}`, and the whole collection is `%a` (Tcl has no syntax for the array as a single value; Perl does). Modern Tcl's `dict` maps to the same Perl hash, or to a hash reference when you need to nest or pass it around.

```
set a(red) 1
set a(green) 2
set a(blue) 3
puts $a(red)
puts [array exists a]
foreach k [lsort [array names a]] { puts "$k => $a($k)" }
unset a(blue)
```

```
use v5.44;
my %a = (red => 1, green => 2, blue => 3);
say $a{red};                              # 1     ($a(red))
say exists $a{green} ? "yes" : "no";      # yes   (info exists
 ↪a(green))
for my $k (sort keys %a) {                # array names a
    say "$k => $a{$k}";
}
delete $a{blue};                          # unset a(blue)
```

That prints:

```
1
yes
red => 1
green => 2
blue => 3
```

The gotcha: membership is `exists $a{key}`, not a separate command, and the fat comma `=>` is the idiomatic key/value separator in a hash literal. `=>` also auto-quotes the bareword on its left, so `red => 1` needs no quotes around `red`. A hash has no inherent order (neither does a Tcl array); iterate `sort keys %a` when you want a stable one, exactly as you reach for `lsort [array names a]`.

Procs and references

`proc name {args} {body}` becomes Perl's `sub name { ... }`. The classic Tcl way to read arguments - positional, by name - has a direct Perl twin: `my ($a, $b) = @_;` unpacks the argument array `@_`. Modern Perl also has signatures, which read like Tcl's `{args}` list and add defaults.

```
proc add {a b} {
    return [expr {$a + $b}]
}
puts [add 3 4]

proc greet {who {greeting Hello}} {
    return "$greeting, $who!"
}
puts [greet Tcl]
puts [greet Perl Hi]
```

```
use v5.44;
sub add {
    my ($a, $b) = @_;       # unpack the argument array
    return $a + $b;
}
say add(3, 4);              # 7

sub greet ($who, $greeting = "Hello") {   # signature with default
    return "$greeting, $who!";
}
say greet("Tcl");           # Hello, Tcl!
say greet("Perl", "Hi");    # Hi, Perl!
```

Tcl's pass-by-name - `upvar`, where a proc reaches up and mutates a caller's variable - becomes a Perl **reference**: take one with `\`, follow it with `$$ref`. You pass the reference explicitly instead of naming the variable, which makes the data flow visible at the call site.

```
use v5.44;
sub increment {
    my ($ref) = @_;
    $$ref++;                # mutate through the reference (Tcl: upvar)
}
my $count = 10;
increment(\$count);         # pass a reference to $count
say $count;                 # 11
```

The gotcha: `@_` is the argument array, and `my ($a, $b) = @_;` *copies* the arguments into named locals - the line you almost always want at the top of a sub. Reaching for `upvar`-style aliasing is rarely needed in Perl; when you do need to mutate a caller's data, pass a reference (`\$x`, `\@arr`, `\%h`) deliberately rather than aliasing by name. References are also how Perl nests data (an array of hashes, a hash of arrays); the same mechanism that replaces `upvar` is the one that replaces Tcl's string-encoded nested lists.

Regex and substitution

This is a payload section. Tcl's `regexp` and `regsub` use Advanced Regular Expressions; Perl's regex is a **superset**, so the patterns you know carry over and gain features. Perl makes regex *syntax*, not a command: `=~` binds a pattern to a string, `m//` matches, `s///` substitutes, and `tr///` transliterates.

```tcl
set s "order 42 shipped"
if {[regexp {(\d+)} $s -> num]} {
    puts "number: $num"
}
set t [regsub -all {o} "foo boo zoo" 0]
puts $t
set u [string map {a A b B} "abcabc"]
puts $u
```

```perl
use v5.44;
my $s = "order 42 shipped";
if ($s =~ /(\d+)/) {            # regexp; capture lands in $1
    say "number: $1";          # number: 42
}
(my $t = "foo boo zoo") =~ s/o/0/g;    # regsub -all  ->  s///g
say $t;                                # f00 b00 z00
(my $u = "abcabc") =~ tr/ab/AB/;       # string map (char-for-char)
say $u;                                # ABcABc
```

The capture maps too: Tcl's `-> num` match variable becomes Perl's `$1`, `$2`, and so on, with named captures landing in `%+`:

```perl
use v5.44;
if ("2026-06-21" =~ /(?<y>\d{4})-(?<m>\d{2})/) {
    say "$+{y} / $+{m}";        # 2026 / 06
}
```

Tcl's `string match` is glob matching, not regex (`string match *.txt $name`). In Perl, anchor a regex instead - `/\.txt$/` is the equivalent - because Perl's one pattern language covers both jobs.

```perl
use v5.44;
my $name = "report.txt";
say "matches" if $name =~ /\.txt$/;     # string match *.txt $name
# matches
```

The gotcha: `s///` modifies its target *in place* and returns the *count* of substitutions, not the new string - which is why the examples above copy first with `(my $t = "...")` before substituting. To get a new string and leave the original untouched, use the `/r` flag: `my $new = $s =~ s/o/0/gr`. And `tr///` (the `string map` analogue for single characters) is **not** a regex: its left side is a literal character set, so `tr/ab/AB/` maps a and b only. For pattern work, reach for `s///`.

Control flow

The control structures map almost one to one; the spelling shifts and the braces stay. Tcl's `if {cond} {body}` becomes Perl's `if (cond) { body }`, `elseif` becomes `elsif`, and `foreach` becomes `for` (the two spellings are aliases in Perl; `foreach` works too).

```
if {$n < 0} {
    set sign neg
} elseif {$n == 0} {
    set sign zero
} else {
    set sign pos
}

set i 0
while {$i < 3} { puts "i=$i"; incr i }

foreach w {a b c} { puts "w=$w" }
```

```
use v5.44;
my $n = 0;
my $sign;
if    ($n < 0)  { $sign = "neg"  }
elsif ($n == 0) { $sign = "zero" }   # elseif -> elsif
else            { $sign = "pos"  }
say $sign;                   # zero

my $i = 0;
while ($i < 3) { say "i=$i"; $i++ }  # incr i -> $i++

for my $w (qw(a b c)) { say "w=$w" } # foreach w {a b c}
```

Tcl's `switch` has no single keyword in Perl; the idiomatic replacement is a **dispatch hash** mapping each case to a code reference, which is faster and more flexible than a chain of `if`:

```
use v5.44;
my $cmd = "start";
my %dispatch = (
    start => sub { "starting" },
    stop  => sub { "stopping" },
);
say $dispatch{$cmd}->();     # starting
```

The gotcha: the `{}` vs `""` distinction you rely on in Tcl for deferring versus substituting maps to Perl's `''` vs `""` for strings (covered above), but Perl's *block* braces are not quoting - a Perl `{ ... }` after `if`/`while`/`for` is a code block, always evaluated, never a deferred string. There is no "the body is a string until I run it" step; Perl parses the block at compile time. Postfix modifiers are a bonus Tcl has no form for: `say $_ for @items;` and `say "neg" if $n < 0;` carry the loop or condition after the statement.

The line/record loop

Tcl reads a file with `set fh [open f r]; while {[gets $fh line] >= 0} {... }`. Perl's equivalent is a `while (<$fh>)` loop, with each line landing in `$_` (the implicit topic variable) or in a named scalar. The line number is in `$.`.

```
set fh [open /tmp/tcl_data.txt r]
while {[gets $fh line] >= 0} {
    puts "got: $line"
}
close $fh
```

```
use v5.44;
open my $fh, '<', '/tmp/tcl_data.txt' or die "open: $!";
while (my $line = <$fh>) {        # gets $fh line
    chomp $line;                  # <$fh> keeps the newline; strip it
    say "got: $line";             # puts
}
close $fh;
# got: apple
# got: banana
# got: cherry
```

(The input file held `apple`, `banana`, `cherry`.) Using `$_` and the line counter `$.` reads even closer to a record loop:

```
use v5.44;
open my $fh, '<', '/tmp/tcl_data.txt' or die "open: $!";
while (<$fh>) {                   # each line lands in $_
    chomp;                        # chomp defaults to $_
    say "$. : $_";               # $. is the current line number
}
close $fh;
# 1 : apple
# 2 : banana
# 3 : cherry
```

The gotcha: Tcl's `gets` strips the trailing newline for you; Perl's `<$fh>` keeps it, so call `chomp` when you do not want it. With no argument, `chomp` and `say`/`print` both default to `$_`, which is why the second loop never names the line variable.

Fields and splitting

Tcl's `split $line $sep` returns a list; Perl's `split /sep/, $line` returns an array, and `join` puts it back together. The difference: a Perl split separator is a *regex*, so you get the full pattern language for free.

```
set line "a:b:c"
set parts [split $line ":"]
puts [llength $parts]
puts [lindex $parts 1]
```

(continues on next page)

(continued from previous page)

```
puts [join $parts ","]
```

```
use v5.44;
my $line = "a:b:c";
my @parts = split /:/, $line;   # split $line ":"  (separator is a␣
↪regex)
say scalar @parts;              # 3        (length $parts)
say $parts[1];                  # b        (index $parts 1)
say join ",", @parts;           # a,b,c    (join $parts ",")
```

The gotcha: `split ' '` with a literal single-space string is a special case that splits on *any run of whitespace* and discards leading empty fields - the awk-style "split into words" behaviour, and usually what you want for free-form input. `split / /` (a regex with one space) is the literal "split on each single space" and does not collapse runs. Reach for `split ' '` to tokenise words:

```
use v5.44;
my @words = split ' ', "  the  quick brown  ";
say "@words";                   # the quick brown
```

I/O, pipes, and in-place edit

Tcl's open/gets/puts/read/close map to Perl's open/<$fh>/print/slurp/close. The idiom is the three-argument open with a lexical filehandle. Tcl's `exec` - run a command and capture its output - becomes Perl's qx{...} (backticks) or `system` when you do not need the output.

```
set out [exec echo hello from shell]
puts "exec said: $out"
```

```
use v5.44;
my $out = qx{echo hello from shell};    # exec  ->  qx// (backticks)
chomp $out;
say "exec said: $out";                  # exec said: hello from␣
↪shell
```

Tcl has no in-place file edit built in; you read, transform, and write back. Perl's `-i` switch is the one-liner form of exactly that, and the full-script equivalent is the same read-modify-write written out:

```
use v5.44;
my $path = '/tmp/tcl_inplace.txt';      # holds: red green blue

open my $in, '<', $path or die "read: $!";
my @lines = <$in>;                      # slurp all lines (list␣
↪context)
close $in;

s/e/E/g for @lines;                     # transform each line's $_
```

(continues on next page)

(continued from previous page)

```
open my $out, '>', $path or die "write: $!";
print {$out} @lines;                    # write them back
close $out;
# file now holds:
# rEd
# grEEn
# bluE
```

The gotcha: `open` returns false on failure rather than raising (Tcl throws an error you catch with `catch`), so the `or die "...: $!"` idiom is mandatory - `$!` is the system error string. For pipes, Perl opens a pipe filehandle directly: `open my $fh, '-|', 'ls', '-l'` reads a command's output and `open my $fh, '|-', 'sort'` writes to a command's input, the structured form of Tcl's `open "|command"`.

One-liners and switches

Tcl is not a one-liner language, so this section is brief - but Perl is, and once you leave Tcl behind it is worth knowing the reflex exists. The switch family: `-e` supplies the program, `-n` wraps it in a no-auto-print loop, `-p` wraps it in an auto-print loop, `-i` edits in place, and `-a`/`-F` autosplit each line into `@F`. `BEGIN { }` and `END { }` blocks run before and after the loop.

```
perl -ne 'print if /error/' log.txt        # print matching lines
perl -pe 's/\d+/N/g' input.txt             # mask numbers
perl -nE 'END { say $n } $n++ if /error/' log.txt   # count, END␣
↪block
```

Each of those expands to an explicit loop: `perl -ne 'CODE'` is `while (<$fh>) { CODE }`, and `perl -pe 'CODE'` is the same with a trailing `print`. Written out, the "count matching lines" form:

```
use v5.44;
open my $fh, '<', '/tmp/tcl_data.txt' or die "open: $!";
my $n = 0;
while (<$fh>) {
    $n++ if /banana/;          # the body of -ne
}
say $n;                        # 1
```

The gotcha: when a one-liner grows past a line or two, write out the explicit loop it expands to. Consult the *One-Liners guide* for the full switch catalogue and recipes. Perl also has full subroutines, references, modules, and OO if your script outgrows a glue task: Tcl's `namespace eval` and `[incr Tcl]` classes map to Perl packages and the modern `class` syntax; see the *Object-oriented Programming guide* for that side of the language.

Gotchas coming from Tcl

The traps, collected for a final scan before you write:

- **Two comparison families.** ==/</<=> compare numerically; eq/lt/cmp compare as strings. `"10" == "10.0"` is true, `"10" eq "10.0"` is false. Pick by the operation, not the value - the one quiet bug the string-first worldview invites.
- **The sigil follows the element.** One element of `@items` is `$items[0]` (a scalar), not `@items[0]`; one value of `%a` is `$a{red}`. The container sigil and the element sigil differ.
- **`sort` is string-sorted by default,** like comparing with eq. Pass `{ $a <=> $b }` for numeric order.
- **`s///` returns a count, not the new string,** and edits in place. Copy first (`(my $t = $s) =~ s/.../.../`) or use the `/r` flag for a fresh string.
- **`tr///` (the `string map` analogue) is a literal character set, not a regex.** `tr/ab/AB/` maps single characters. For patterns, use `s///`.
- **`split ' '` and `split / /` differ.** The literal-space string splits on whitespace runs and drops leading empties (the word tokeniser); the one-space regex does not. Use `split ' '` for free-form input.
- **`<$fh>` keeps the trailing newline,** unlike Tcl's `gets`. Call `chomp` to strip it.
- **`open` does not raise on failure.** Use `open my $fh, '<', $path or die "...: $!"`. `$!` is the system error string; there is no `catch` to wrap it.
- **References replace `upvar`.** To mutate a caller's variable, pass a reference (`\$x`, `\@arr`, `\%h`) deliberately rather than aliasing by name. References are also how Perl nests data.
- **A `{ ... }` block is code, not a deferred string.** Perl parses blocks at compile time; there is no "the body is a string until I run it" step. Only `''`/`""` carry Tcl's `{}`/`""` substitution distinction.

4.2.10 The Definitive Guide to 2D Graphics with Perl

Perl is not the language people reach for when they think "graphics", and that reputation is mostly an accident of history rather than a statement about the tools. The vector engine everyone else uses under the hood (`Cairo`) ships natively here, `SDL2` gives a window and an accelerated renderer, and the raster libraries are mature. What Perl has lacked is not libraries but a single place that puts them in order, from the pixel up to a running program.

This guide is that place, for **two dimensions**. It starts at the theory every graphics programmer eventually needs (how a line becomes pixels, what a colour space actually is, how transparency composites), moves through the working CPAN toolkits, and then goes one step further than the usual tour: it uses pperl's native FFI to reach C graphics libraries directly, the same way pperl already reaches libgmp to give you `Math::GMP`. It ends with two things you can run: a drawing application and a 2D game.

Three dimensions and the GPU are the subject of the companion *3D guide*, which builds on this one. The final chapter draws the line between the two.

Which chapter do I want?

I want to ...	Start here
understand how pixels, paths, and colour actually work	*foundations*
implement a line, circle, fill, or blur myself	*raster-algorithms*
draw on, load, or save a bitmap image	*raster-imager*
draw on, load, or save a bitmap with GD	*raster-gd*
use an ImageMagick codebase, or its specific features	*raster-gd-magick*
produce resolution-independent vector paths, or SVG output	*vector-cairo-svg*
read image metadata, or draw a chart	*metadata-and-charts*
draw in a live window on Linux	*gui-toolkits*
reach a C graphics library that has no CPAN binding	*ffi-graphics-primer*
see how pperl's shipped native `Cairo` binding works	*ffi-cairo-showcase*
build a real drawing application	*capstone-drawing-app*
build a real 2D game, or drive SDL2 directly	*capstone-2d-game*
go on to 3D and the GPU	*whats-next*

The three arcs of this guide

- **Foundations** - *foundations* and *raster-algorithms*. The mental model and the classic algorithms, written in plain Perl so you own them before any library hides them. Read these once; every later chapter assumes the vocabulary.
- **Toolkits** - the six chapters from *raster-imager* through *gui-toolkits*. What each library is good at and where one wins over another, ending with the window and frame loop. Each chapter is self-contained; reach for the one that names your problem.
- **Reaching further** - *ffi-graphics-primer* onward. pperl's native FFI, the two capstone projects, and an honest map of where the road currently ends.

A note on the runtime

These chapters target `pperl`, a faithful Rust reimplementation of Perl 5.44. The CPAN modules in the toolkit chapters are used exactly as they are on any Perl 5, and the FFI chapters use pperl's own `Peta::FFI`, which has no perl5 equivalent.

Known gaps

Everything three-dimensional, every GPU-programming topic, computer vision and image analysis, and connectors to hosted image-generation services are out of scope for this version. They are named, with the reason and the plan, in *what's next*. The FFI chapters also mark, at the point where it matters, which C interfaces `Peta::FFI` reaches and what a hand-written binding costs you.

See also

- *Auto-FFI (Peta::FFI)* - the native FFI layer the "reaching further" arc is built on: `dlopen`, `call`, the type-signature codes, and the native bindings built on it.

- *PPerl Architecture guide* - the JIT and how pperl provides C-backed modules natively.

Foundations: pixels, paths, and colour

Before any library, four ideas carry the whole subject: the difference between a raster and a vector, the coordinate system a drawing lives in, what a colour actually is once you have to store it, and how two colours combine when one is partly transparent. Every chapter after this one leans on these without re-explaining them. None of it is Perl-specific, but all of the examples are Perl, because the point is to make the ideas concrete in code you can run.

Raster versus vector

A **raster** image is a grid of coloured cells. It has a fixed width and height in pixels, and every pixel holds a colour. Scaling it up past its resolution invents nothing: the cells just get bigger. Photographs, screenshots, and anything with continuous tone are rasters. `Imager`, `GD`, and ImageMagick are raster tools.

A **vector** image is a set of shapes described mathematically: "a line from here to there", "a circle of this radius", "fill this path". There is no resolution until you draw it, so it scales to any size cleanly. Fonts, logos, and diagrams are vectors. `Cairo` and `SVG` are vector tools.

The two meet at **rasterisation**: turning a vector description into pixels. A vector library still has to decide which pixels a diagonal line touches, and that decision is *the first algorithm in the next chapter*. Everything you draw ends up as pixels on a screen; the question is only whether you commit to a resolution early (raster) or late (vector).

A raster in Perl is, at its plainest, an array of rows:

```
my $w = 4;
my $h = 3;
# each pixel is [r, g, b], each channel 0..255
my @img = map { [ map { [0, 0, 0] } 1 .. $w ] } 1 .. $h;

$img[1][2] = [255, 0, 0];   # row 1, column 2, red
```

That nested-array picture is all a raster is. Real libraries store the same grid in a packed C buffer for speed, but the mental model does not change: a pixel is addressed by an integer row and column, and it holds a colour.

The coordinate system

Graphics coordinates are not math-class coordinates. The origin `(0, 0)` is the **top-left** corner, x increases to the right, and y increases **downward**. This catches everyone once: a larger y is lower on the screen, not higher.

```
# (0,0) top-left            x ->
#   +-----------------
#   | (0,0)      (w-1, 0)
# y |
```

(continues on next page)

(continued from previous page)

```
# | |
# v | (0, h-1)   (w-1, h-1)
```

A pixel is a little square, and there is a lasting ambiguity about whether the coordinate `(3, 5)` names the square's corner or its centre. Raster libraries usually treat integer coordinates as pixel centres; vector libraries treat them as the grid lines between pixels, which is why a one-unit-wide vertical line at `x = 3` can come out straddling two columns of pixels at half intensity. When a thin line or a rectangle edge looks blurry or one pixel off, this is almost always why, and the fix is a half-pixel offset. The chapters that hit it say so.

Sizes and positions are integers in raster space and real numbers in vector space. That is the practical difference you feel: in `Imager` you set pixel `(120, 47)`; in `Cairo` you can move to `(120.5, 47.25)` and the rasteriser works out the coverage.

Colour: what a pixel holds

A colour on a screen is three numbers: how much **red**, **green**, and **blue** light to emit. Add all three at full strength and you get white; leave them all off and you get black. This is the **RGB** model, and it is additive because you are adding light, not mixing paint.

Each channel is almost always stored in 8 bits, so `0 .. 255` per channel, and a colour is three bytes. That is the `#RRGGBB` hex notation from the web: `#ff0000` is `(255, 0, 0)`, pure red.

```
sub rgb_to_hex {
    my ($r, $g, $b) = @_;
    return sprintf "#%02x%02x%02x", $r, $g, $b;
}

print rgb_to_hex(255, 128, 0), "\n";   # #ff8000, an orange
```

RGB is how pixels are stored, but it is a terrible model for a human choosing a colour. "Make this a bit more orange but keep it the same brightness" is not an obvious move in RGB. **HSV** (hue, saturation, value) is the model for that: **hue** is the position on the colour wheel in degrees `0 .. 360` (0 red, 120 green, 240 blue), **saturation** is how vivid versus grey `0 .. 1`, and **value** is how bright `0 .. 1`. Rotating the hue while holding saturation and value walks the rainbow at constant vividness and brightness.

Every colour-picker and gradient tool works in HSV or a close relative, then converts to RGB to store. The conversion is worth having in plain sight, because you will reimplement it more than once:

```
# h in 0..360, s and v in 0..1  ->  (r, g, b) in 0..255
sub hsv_to_rgb {
    my ($h, $s, $v) = @_;
    my $c = $v * $s;                       # chroma
    my $hp = $h / 60;
    my $x = $c * (1 - abs($hp % 2 - 1 + ($hp - int $hp)));
    my ($r, $g, $b) =
        $hp < 1 ? ($c, $x, 0) :
```

(continues on next page)

(continued from previous page)

```
        $hp < 2 ? ($x, $c, 0) :
        $hp < 3 ? (0, $c, $x) :
        $hp < 4 ? (0, $x, $c) :
        $hp < 5 ? ($x, 0, $c) :
                  ($c, 0, $x);
    my $m = $v - $c;
    return map { int(($_ + $m) * 255 + 0.5) } ($r, $g, $b);
}

my @rainbow = map { [ hsv_to_rgb($_, 1, 1) ] } 0, 60, 120, 180, 240,␣
↪300;
# red, yellow, green, cyan, blue, magenta
```

There are other spaces you will meet: greyscale (one channel, a brightness), CMYK (subtractive, for print), and perceptually uniform spaces like CIELAB where equal numeric steps look like equal perceptual steps. For screen drawing in two dimensions, RGB for storage and HSV for choosing cover almost everything.

Alpha and compositing

A fourth channel, **alpha**, records opacity: `0` fully transparent, `255` (or `1.0`) fully opaque. A pixel with alpha is **RGBA**. Alpha is what lets you draw one image over another and have the top one's transparent parts show through.

Combining a foreground pixel over a background pixel is **compositing**, and the standard rule is **source-over**. With alpha as a fraction `0 .. 1`:

```
# composite one RGBA pixel over another (all channels 0..255,
# alpha treated as 0..1 for the blend)
sub over {
    my ($fg, $bg) = @_;
    my $a = $fg->[3] / 255;                    # source alpha
    my @out = map {
        int($fg->[$_] * $a + $bg->[$_] * (1 - $a) + 0.5)
    } 0 .. 2;
    return \@out;
}

my $red_half = [255, 0, 0, 128];    # red at 50% opacity
my $white    = [255, 255, 255, 255];
my $result   = over($red_half, $white);
print "@$result\n";                 # 255 128 128, a pink
```

Red at half opacity over white gives pink, exactly as you would expect paint-thinned-with-water to look. That one formula, applied per pixel, is how every layer, brush stroke, and sprite with soft edges lands on the canvas beneath it. When you stack many semi-transparent layers, you apply it repeatedly, bottom to top.

The subtlety that bites later is **premultiplied alpha**: storing each colour channel already multiplied by its alpha. It makes repeated compositing cheaper and filtering correct at edges, and it is why a library's raw pixel bytes sometimes are not the RGB you expect.

`Cairo` uses premultiplied alpha internally; the *Cairo chapter* notes where it shows.

The one 3D idea, named and deferred

You will hear about the **Z-buffer** the moment 3D comes up, and it is worth one paragraph here so the term is not a mystery. In 2D, "what is on top" is decided by draw order: the last thing drawn wins, which is exactly the source-over compositing above. In 3D, objects have a depth (a z coordinate), and the thing nearest the viewer should win regardless of draw order. A **Z-buffer** is a second grid, the same size as the image, holding the depth of the closest surface seen so far at each pixel; before drawing a pixel you check whether the new surface is nearer than what is recorded, and skip it if not.

That is the whole concept, and it is as far as this 2D guide goes with it. Depth buffering, projection, and the rest of the third dimension belong to the *3D guide*. In two dimensions, draw order is your depth buffer.

Where this goes next

You now have the vocabulary the rest of the guide speaks: raster and vector, top-left coordinates, RGB and HSV, alpha and source-over compositing. The *next chapter* spends that vocabulary on the classic drawing algorithms, implemented from scratch, so that when a library draws a line or fills a polygon you know exactly what it is doing on your behalf.

Drawing primitives from scratch

Every graphics library hands you `line`, `circle`, `fill`, and `blur`. This chapter implements them in plain Perl so that when a library draws them for you, nothing is magic. These are the classic algorithms, and they are classic because they are the right answers: fast, integer where they can be, and correct at the awkward cases. You will not ship these implementations (a C library will always be faster), but knowing them tells you why a library behaves the way it does, and gives you the tools for the one case the library did not anticipate.

Throughout, the canvas is the nested-array raster from the *foundations chapter*: a grid of `[r, g, b]` pixels addressed by integer `(x, y)`, origin top-left.

```
sub new_canvas {
    my ($w, $h) = @_;
    return { w => $w, h => $h,
             px => [ map { [ map { [255, 255, 255] } 1 .. $w ] } 1 .
↪. $h ] };
}

sub set_px {
    my ($c, $x, $y, $color) = @_;
    return if $x < 0 || $y < 0 || $x >= $c->{w} || $y >= $c->{h};
    $c->{px}[$y][$x] = $color;
}
```

Lines: Bresenham

A line from `(x0, y0)` to `(x1, y1)` has to choose, for each column it crosses, which row of pixels to light. The naive approach computes `y = m*x + b` with floating point per step; **Bresenham's algorithm** does it with integer addition only, by tracking an error term that says how far the ideal line has drifted from the pixels chosen so far.

The insight is that at each step you either stay on the same cross-axis row or move one, and a running integer error decides which. The version below handles every slope and direction in one loop by working with absolute deltas and step signs:

```
sub draw_line {
    my ($c, $x0, $y0, $x1, $y1, $color) = @_;
    my $dx = abs($x1 - $x0);
    my $dy = -abs($y1 - $y0);
    my $sx = $x0 < $x1 ? 1 : -1;
    my $sy = $y0 < $y1 ? 1 : -1;
    my $err = $dx + $dy;                    # error accumulator
    while (1) {
        set_px($c, $x0, $y0, $color);
        last if $x0 == $x1 && $y0 == $y1;
        my $e2 = 2 * $err;
        if ($e2 >= $dy) { $err += $dy; $x0 += $sx; }   # step in x
        if ($e2 <= $dx) { $err += $dx; $y0 += $sy; }   # step in y
    }
}
```

No multiplication, no division, no floating point in the loop. That is why Bresenham survived from 1962: it is the line-drawing you would build in hardware. Every library's thin, aliased line is this or a direct descendant.

Circles: the midpoint method

A circle has eight-fold symmetry: compute one octant and mirror it into the other seven. The **midpoint circle algorithm** walks one octant with the same integer-error idea as Bresenham, deciding at each step whether to move straight or diagonally by testing a decision variable against zero.

```
sub draw_circle {
    my ($c, $cx, $cy, $r, $color) = @_;
    my ($x, $y) = ($r, 0);
    my $err = 1 - $r;                       # decision variable
    while ($x >= $y) {
        # eight mirrored points, one per octant
        for my $p ([$x,$y],[$y,$x],[-$x,$y],[-$y,$x],
                   [$x,-$y],[$y,-$x],[-$x,-$y],[-$y,-$x]) {
            set_px($c, $cx + $p->[0], $cy + $p->[1], $color);
        }
        $y++;
        if ($err < 0) { $err += 2 * $y + 1; }
        else          { $x--; $err += 2 * ($y - $x) + 1; }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Compute one eighth, draw eight. The same symmetry trick, with the error test adjusted, gives ellipses and arcs.

Filling polygons: the scanline method

Drawing a shape's outline is one problem; filling its interior is another. The **scanline fill** sweeps a horizontal line down the image, and for each scanline finds where it crosses the polygon's edges. Those crossings come in pairs, and the interior is between each pair: the **even-odd rule**. Sort the x crossings, then fill between the first and second, the third and fourth, and so on.

```
# poly is an arrayref of [x, y] vertices, in order, implicitly
 ↪closed
sub fill_polygon {
    my ($c, $poly, $color) = @_;
    my ($ymin, $ymax) = ($poly->[0][1], $poly->[0][1]);
    for my $v (@$poly) {
        $ymin = $v->[1] if $v->[1] < $ymin;
        $ymax = $v->[1] if $v->[1] > $ymax;
    }
    for my $y ($ymin .. $ymax) {
        my @xs;
        for my $i (0 .. $#$poly) {
            my $a = $poly->[$i];
            my $b = $poly->[($i + 1) % @$poly];
            my ($ay, $by) = ($a->[1], $b->[1]);
            # does this edge straddle the scanline?
            next if ($ay <= $y) == ($by <= $y);
            # x where the edge crosses y (linear interpolation)
            my $t = ($y - $ay) / ($by - $ay);
            push @xs, $a->[0] + $t * ($b->[0] - $a->[0]);
        }
        @xs = sort { $a <=> $b } @xs;
        # fill between crossing pairs
        for (my $i = 0; $i + 1 < @xs; $i += 2) {
            for my $x (int($xs[$i] + 0.5) .. int($xs[$i + 1] - 0.5))
 ↪{
                set_px($c, $x, $y, $color);
            }
        }
    }
}
```

The `($ay <= $y) == ($by <= $y)` test is the crossing check written to count each vertex exactly once, so a scanline passing through a shared vertex does not double-count and leak the fill. That half-open convention is the detail that separates a correct fill from one with occasional stray lines, and it is why libraries and this code agree on where an

edge “belongs”.

Anti-aliasing: coverage instead of on-or-off

The line and circle above are **aliased**: each pixel is fully on or fully off, and a diagonal edge looks like a staircase. **Anti-aliasing** softens that by setting a pixel’s intensity to how much of it the shape actually covers. A pixel the edge crosses at 40 percent coverage gets 40 percent of the shape colour blended with 60 percent of the background, which is exactly the *source-over compositing* from the last chapter with alpha equal to coverage.

The cheapest way to approximate coverage is **supersampling**: draw at several times the resolution with the aliased algorithm, then average each block of high-resolution pixels down to one. Averaging a 2x2 block where three sub-pixels are on gives 75 percent coverage for free:

```
# downsample a 2x-oversampled canvas by averaging 2x2 blocks
sub downsample_2x {
    my ($big) = @_;
    my ($w, $h) = ($big->{w} / 2, $big->{h} / 2);
    my $out = new_canvas($w, $h);
    for my $y (0 .. $h - 1) {
        for my $x (0 .. $w - 1) {
            my @sum = (0, 0, 0);
            for my $dy (0, 1) {
                for my $dx (0, 1) {
                    my $p = $big->{px}[2 * $y + $dy][2 * $x + $dx];
                    $sum[$_] += $p->[$_] for 0 .. 2;
                }
            }
            $out->{px}[$y][$x] = [ map { int($_ / 4 + 0.5) } @sum ];
        }
    }
    return $out;
}
```

Real libraries compute coverage analytically rather than by brute-force supersampling, but the effect is identical: the staircase becomes a smooth ramp of intensities. When `Cairo` gives you a crisp anti-aliased curve and `Imager` offers an `aa` flag on its primitives, this is what they are doing.

Tessellation: breaking shapes into triangles

Hardware and many algorithms only really know how to fill **triangles**; everything more complex is first broken into triangles, a process called **tessellation** or **triangulation**. A convex polygon triangulates trivially by fanning from one vertex:

```
# fan triangulation of a convex polygon -> list of [v0, v1, v2]␣
↪triangles
sub triangulate_convex {
    my ($poly) = @_;
    my @tris;
```

(continues on next page)

(continued from previous page)

```
    for my $i (1 .. $#$poly - 1) {
        push @tris, [ $poly->[0], $poly->[$i], $poly->[$i + 1] ];
    }
    return @tris;
}
```

Concave polygons need a real algorithm (ear clipping is the standard one: repeatedly find a triangle that pokes out with no other vertex inside it, snip it off, repeat). The convex fan is enough to see the idea, and it is the idea that carries straight into 3D, where every surface is ultimately a mesh of triangles. In two dimensions you meet tessellation whenever you fill a complex vector path; in the *3D guide* it is the foundation of everything.

Convolution: blur, sharpen, and edges

A whole family of image effects comes from one operation: **convolution**, sliding a small grid of weights (a **kernel**) over the image and replacing each pixel with the weighted sum of its neighbours. A kernel of all equal weights averages, which blurs. A kernel that subtracts the neighbours and adds a large centre sharpens.

```
# apply a 3x3 kernel to a canvas; divisor normalises the weights
sub convolve_3x3 {
    my ($c, $kernel, $divisor) = @_;
    $divisor ||= 1;
    my $out = new_canvas($c->{w}, $c->{h});
    for my $y (1 .. $c->{h} - 2) {
        for my $x (1 .. $c->{w} - 2) {
            my @sum = (0, 0, 0);
            for my $ky (0 .. 2) {
                for my $kx (0 .. 2) {
                    my $wgt = $kernel->[$ky][$kx];
                    my $p = $c->{px}[$y + $ky - 1][$x + $kx - 1];
                    $sum[$_] += $p->[$_] * $wgt for 0 .. 2;
                }
            }
            $out->{px}[$y][$x] =
                [ map { my $v = int($_ / $divisor + 0.5);
                        $v < 0 ? 0 : $v > 255 ? 255 : $v } @sum ];
        }
    }
    return $out;
}

my $box_blur = [[1,1,1],[1,1,1],[1,1,1]];   # divisor 9
my $sharpen  = [[0,-1,0],[-1,5,-1],[0,-1,0]]; # divisor 1
```

The box blur divides by 9 to keep brightness constant; the sharpen kernel sums to 1 so it needs no divisor. Swap in a kernel that responds to intensity change and you get edge detection, which is where image processing shades into computer vision. That boundary is deliberate: this guide uses convolution as a **drawing** tool (blur a shadow, sharpen a

downscaled sprite) and stops there. Analysis, feature detection, and recognition are a different subject, out of scope here.

What the libraries do for you

Everything above is what a graphics library implements in optimised C, usually with SIMD and careful edge handling you would not want to write by hand. Knowing the algorithms means you can read a library's documentation and know what "anti-aliased", "even-odd fill rule", "convolution matrix", and "triangulated path" mean, and you can drop to your own pixel loop for the rare effect no library ships. The *next chapter* picks up `Imager`, where these primitives are one method call away and running at C speed.

Raster drawing with Imager

`Imager` is the raster workhorse: create an image, draw on it, filter it, save it. It is actively maintained, it links its own copies of the format libraries so PNG, JPEG, TIFF, GIF, and WEBP all work out of the box, and its drawing primitives are the ones from the *algorithms chapter* running at C speed. If you are starting a raster project on Perl today and have no reason to prefer something else, start here.

Every drawing method returns false on failure and sets `errstr`, so the idiom is to check and report:

```
use Imager;

my $img = Imager->new(xsize => 400, ysize => 300, channels => 4)
    or die Imager->errstr;
```

`channels => 4` asks for RGBA (red, green, blue, alpha); use 3 for opaque RGB. The alpha channel is what lets the *compositing* from the foundations chapter work when you draw one image onto another.

Colours

Anywhere a method takes `color =>`, you can pass a name, a hex string, an `Imager::Color` object, or a plain array reference:

```
my $red   = Imager::Color->new('red');
my $red2  = Imager::Color->new('#FF0000');
my $red3  = Imager::Color->new(255, 0, 0);
my $trans = Imager::Color->new(255, 0, 0, 128);   # 50% alpha

$img->box(color => [0, 128, 255], filled => 1);   # array-ref, no␣
 ↪object
```

The array-reference form is the least ceremony for a one-off colour and is what most of the examples below use.

The primitives

Note the parameter names carefully, because they are not all the same shape. Boxes use corner coordinates `xmin/ymin/xmax/ymax`; circles and arcs use a center `x/y` with a radius `r`; arcs add start and end degrees `d1/d2`.

```
# fill the whole canvas white first
$img->box(color => 'white', filled => 1);

# a line; aa => 1 requests anti-aliasing
$img->line(x1 => 0, y1 => 0, x2 => 399, y2 => 299,
           color => [255, 0, 0], aa => 1);

# an outlined box (no fill) and a filled one
$img->box(xmin => 20, ymin => 20, xmax => 120, ymax => 90,
          color => 'black');
$img->box(xmin => 140, ymin => 20, xmax => 240, ymax => 90,
          color => [200, 220, 255], filled => 1);

# a filled circle: center (200,150), radius 60
$img->circle(x => 200, y => 150, r => 60, color => [0, 128, 255]);

# an arc is a pie slice from d1 to d2 degrees
$img->arc(x => 200, y => 150, r => 80, d1 => 0, d2 => 120,
          color => 'yellow');

# a polygon from a list of [x, y] points
$img->polygon(points => [[300, 40], [380, 40], [340, 110]],
              color => [120, 200, 120]);

# a polyline is an open path; it takes no fill
$img->polyline(points => [[300, 130], [340, 200], [380, 130]],
               color => 'black', aa => 1);

# flood fill from a seed point out to a border
$img->flood_fill(x => 5, y => 5, color => [245, 245, 245]);
```

Two traps worth stating once. `box` is filled only when you pass `filled => 1` (or a `fill =>` pattern); without it you get an outline. `polyline` draws an open path and does not accept `fill`, unlike `polygon` which closes and can fill. When a shape comes out as an outline you expected filled, or a fill silently does nothing, one of these two is why.

Loading, scaling, and saving

Reading and writing infer the format from the filename extension, or you can force it with `type =>`:

```
my $photo = Imager->new;
$photo->read(file => 'input.jpg') or die $photo->errstr;

# scale by a factor, or to fit a box
```

(continues on next page)

(continued from previous page)

```
my $thumb = $photo->scale(scalefactor => 0.25);
my $fit   = $photo->scale(xpixels => 200, ypixels => 200, type =>
 ↪'min');

$thumb->write(file => 'thumb.png') or die $thumb->errstr;
```

`scale` returns a new image; it does not modify the original. The `type => 'min'` form scales so the image fits inside the given box preserving aspect ratio; `'max'` fills the box (cropping the overflow when you then crop); `'nonprop'` ignores aspect ratio and stretches to exactly the pixels you named.

Filters

`filter` applies an effect in place, named by `type =>`. The *convolution* from the algorithms chapter is here as built-in Gaussian blur and unsharp masking, plus a general `conv` for your own kernel:

```
# Gaussian blur; larger stddev is more blur
$img->filter(type => 'gaussian', stddev => 2.5);

# unsharp mask to sharpen; both params optional
$img->filter(type => 'unsharpmask', stddev => 2.0, scale => 1.0);

# your own 1-D convolution coefficients (a horizontal blur here)
$img->filter(type => 'conv', coef => [0.25, 0.5, 0.25]);

# contrast, mosaic (pixelate), and inversion are all filters too
$img->filter(type => 'contrast', intensity => 1.4);
$img->filter(type => 'mosaic', size => 8);
$img->filter(type => 'hardinvert');
```

The full set includes `gaussian`, `gaussian2` (separate `x` and `y` deviations), `conv`, `unsharpmask`, `contrast`, `mosaic`, `noise`, `hardinvert`, `autolevels`, `postlevels`, and the procedural texture generators `fountain`, `gradgen`, `radnoise`, and `turbnoise`. Reach for the named filter before writing your own pixel loop; only drop to a `conv` coefficient list or manual access when nothing built in fits.

A complete program

Putting it together: a small poster with a background, some shapes, a blur pass, and a saved thumbnail.

```
use strict;
use warnings;
use Imager;

my $img = Imager->new(xsize => 400, ysize => 300, channels => 4)
    or die Imager->errstr;
```

(continues on next page)

(continued from previous page)

```
$img->box(color => [30, 30, 60], filled => 1);              # dark
 ↪bg
$img->circle(x => 120, y => 150, r => 70, color => [255, 200, 60]);
$img->circle(x => 260, y => 150, r => 70, color => [60, 200, 255]);
$img->filter(type => 'gaussian', stddev => 1.2);           # soften
$img->line(x1 => 0, y1 => 250, x2 => 399, y2 => 250,
           color => 'white', aa => 1);

$img->write(file => 'poster.png') or die $img->errstr;
$img->scale(scalefactor => 0.5)->write(file => 'poster-thumb.png')
    or die $img->errstr;
```

When Imager is the wrong tool

`Imager` is raster, so it has the raster limitation: it has a fixed resolution, and scaling past it degrades. For resolution-independent output (a logo that must print crisply at any size, a diagram destined for the web as markup) you want a vector tool, covered in *Cairo and SVG*. For an existing codebase built on GD or ImageMagick, or for their specific strengths, see the *next chapter*. And when the drawing needs to appear in a live window rather than a file, that is the *GUI toolkits chapter*. For everything that is "make or modify a bitmap on disk", `Imager` is the answer.

Raster drawing with GD

GD is Perl's binding to libgd, the small C library behind a great deal of the web's generated images: thumbnails, charts, captchas, badges. It creates an image, draws on it, filters it, and saves it, and its drawing primitives are the ones from the *algorithms chapter* running at C speed. On pperl it is a native module: the C half is bound directly to the system's libgd, and GD's own Perl half (`GD::Image`, `GD::Polygon`, `GD::Simple`) runs as it is, so code written for GD on any Perl 5 runs here unchanged.

The API is positional and terse. Methods take coordinates and a colour in a fixed order, and most of them return nothing useful. The ones that can fail either return false or `undef`, and the idiom is to check those, or die with the reason, as the file loaders do.

```
use GD;

my $im = GD::Image->new(400, 300, 1);   # width, height, true-colour
```

Colours

GD's defining quirk is how it names a colour. You never pass red, green and blue to a drawing method. You ask the image for a colour first, with `colorAllocate`, and then draw with the integer it returns. What that integer means depends on the kind of image.

A **palette** image (created without the third argument) holds a table of at most 256 colours, and every pixel stores an index into it. This is the model of GIF and of 256-colour displays. `colorAllocate` adds an entry and returns its index, and the first colour you allocate becomes the background.

A **true-colour** image (third argument 1) stores the colour itself in every pixel. `colorAllocate` has no table to add to, so it simply packs the channels into one integer, and that integer is the colour. Prefer true-colour for all new work: it has no 256-colour ceiling, and filters, resampling and alpha blending all assume it.

```
use strict;
use warnings;
use GD;

# a palette image: colorAllocate hands out table indices
my $pal   = GD::Image->new(100, 100);
my $white = $pal->colorAllocate(255, 255, 255);   # first one is the
 ↪background
my $red   = $pal->colorAllocate(255, 0, 0);
print "palette: white=$white red=$red, ",
      $pal->colorsTotal, " colours in the table\n";

# a truecolor image: the "index" is the colour itself, packed
my $tc   = GD::Image->new(100, 100, 1);
my $tw   = $tc->colorAllocate(255, 255, 255);
my $tr   = $tc->colorAllocate(255, 0, 0);
my $half = $tc->colorAllocateAlpha(255, 0, 0, 64);  # about 50%
 ↪transparent
printf "truecolor: white=0x%08x red=0x%08x half=0x%08x\n", $tw, $tr,
 ↪$half;
```

```
palette: white=0 red=1, 2 colours in the table
truecolor: white=0x00ffffff red=0x00ff0000 half=0x40ff0000
```

On a palette image the table does fill up. The 257th `colorAllocate` returns `-1`, and `colorResolve` is the safe alternative: it returns an exact match if there is one, allocates if there is room, and otherwise falls back to the closest colour already in the table.

```
use strict;
use warnings;
use GD;

my $pal = GD::Image->new(10, 10);
$pal->colorAllocate($_, 0, 0) for 0 .. 255;       # fill all 256
 ↪slots
print "table: ", $pal->colorsTotal, " colours\n";
print "allocate one more: ", $pal->colorAllocate(1, 2, 3), "\n";
print "resolve instead:   ", $pal->colorResolve(1, 2, 3), "\n";
```

```
table: 256 colours
allocate one more: -1
resolve instead:   1
```

`colorExact` and `colorClosest` are the two halves of `colorResolve` on their own; `rgb($colour)` turns an index or packed colour back into its three channels.

Alpha, the GD way

`colorAllocateAlpha` takes a fourth value, and it runs the opposite way to the *foundations chapter*: in GD, alpha `0` is fully **opaque** and `127` is fully **transparent**, a 7-bit range with transparency at the top. So `64` is roughly half and half.

What happens when you draw with a translucent colour depends on `alphaBlending`. With blending on (the default for true-colour images) the colour is composited source-over onto what is already there. With it off, the pixel is simply replaced, alpha and all.

```
use strict;
use warnings;
use GD;

my $im    = GD::Image->new(100, 100, 1);
my $white = $im->colorAllocate(255, 255, 255);
my $half  = $im->colorAllocateAlpha(255, 0, 0, 64);

my @before = $im->rgb($im->getPixel(50, 50));
print "fresh canvas: @before\n";                 # truecolor starts
 ↪black

$im->filledRectangle(0, 0, 99, 99, $white);
$im->filledRectangle(0, 0, 49, 99, $half);       # blending on:
 ↪composite
$im->alphaBlending(0);
$im->filledRectangle(50, 0, 99, 99, $half);      # blending off:
 ↪replace

for my $x (25, 75) {
    my $c = $im->getPixel($x, 50);
    printf "x=%d: rgb=%s alpha=%d\n", $x, join(' ', $im->rgb($c)),
 ↪$im->alpha($c);
}
```

```
fresh canvas: 0 0 0
x=25: rgb=255 128 128 alpha=0
x=75: rgb=255 0 0 alpha=64
```

The blended half is the pink `255 128 128` the foundations chapter computed by hand for red at half opacity over white. The other half keeps pure red with alpha `64` stored in the pixel, which is what you want when you are building an image with transparent regions to save (see *transparency in files* below).

The first line of output is a trap worth stating once: a new true-colour image starts **black**, not white. Always paint the background with a `filledRectangle` over the whole canvas before drawing anything else.

The primitives

Every primitive takes its geometry first and the colour last, all positional. Rectangles use two corners, both inclusive, so a full 400 x 300 canvas is `(0, 0, 399, 299)`. Ellipses and arcs use a centre plus the **full width and height** (not a radius), and arc angles are in **degrees**, starting at three o'clock and increasing clockwise (clockwise because y grows downward).

```
use strict;
use warnings;
use GD;

my $im = GD::Image->new(400, 300, 1);
my $white  = $im->colorAllocate(255, 255, 255);
my $black  = $im->colorAllocate(0, 0, 0);
my $blue   = $im->colorAllocate(0, 90, 200);
my $red    = $im->colorAllocate(220, 40, 40);
my $green  = $im->colorAllocate(60, 170, 80);
my $yellow = $im->colorAllocate(250, 200, 40);

$im->filledRectangle(0, 0, 399, 299, $white);      # background

# lines: plain, dashed, and thick
$im->line(10, 10, 130, 60, $black);
$im->dashedLine(10, 30, 130, 80, $black);
$im->setThickness(5);
$im->line(10, 60, 130, 110, $blue);
$im->setThickness(1);

# rectangles: corner to corner, both corners inclusive
$im->rectangle(150, 10, 250, 80, $black);
$im->filledRectangle(270, 10, 390, 80, $yellow);

# ellipses and arcs: centre, then full width and height
$im->ellipse(70, 170, 110, 60, $black);
$im->filledEllipse(70, 170, 50, 50, $blue);
$im->arc(200, 170, 100, 100, 0, 270, $black);          # three␣
↪quarters
$im->filledArc(330, 170, 100, 100, 30, 330, $yellow);  # a pie slice
$im->filledArc(330, 170, 100, 100, 30, 330, $black, gdEdged |␣
↪gdNoFill);

# polygons come from a GD::Polygon
my $tri = GD::Polygon->new;
$tri->addPt(20, 290);
$tri->addPt(120, 290);
$tri->addPt(70, 220);
$im->filledPolygon($tri, $green);
$im->polygon($tri, $black);

my $zig = GD::Polygon->new;
```

(continues on next page)

(continued from previous page)

```
$zig->addPt($_->[0], $_->[1]) for [150, 280], [180, 230], [210,␣
↪280], [240, 230];
$im->unclosedPolygon($zig, $red);                  # open path, no␣
↪closing edge

# flood fills
$im->rectangle(270, 230, 390, 290, $black);
$im->line(330, 230, 330, 290, $black);
$im->fill(300, 260, $red);                         # fills the left␣
↪cell only
$im->fillToBorder(360, 260, $black, $blue);        # stops at black␣
↪pixels

open my $fh, '>', 'primitives.png' or die "primitives.png: $!";
binmode $fh;
print $fh $im->png;
close $fh;
```

The result is a 3 x 3 sheet: three lines, an outlined and a filled rectangle; an ellipse around a disc, a three-quarter arc open at the top right, a pie with its mouth to the right; a filled triangle, a red zigzag, and a box whose two halves were flood-filled red and blue.

A few details the sheet shows:

- `filledArc` takes an optional style. The default fills a pie slice; `gdChord` replaces the curved edge with a straight line between the two end points, `gdNoFill` draws only the outline, and `gdEdged` adds the two radii. They combine with `|`, which is how the pie above gets its black outline on a second call.
- `polygon` (also spelled `openPolygon`) draws a closed outline; `unclosedPolygon` leaves off the edge from the last point back to the first, which is the polyline. `filledPolygon` fills using the *scanline method*.
- `fill` floods the connected region of the colour under the seed point, so it stopped at the black divider. `fillToBorder` floods until it meets one named border colour, whatever it crosses on the way.
- `setThickness` changes the pen for every later line, outline and arc, until you set it back.

Single pixels are `setPixel($x, $y, $colour)` and `getPixel($x, $y)`; the latter returns the index or packed colour, which `rgb` and `alpha` take apart.

Anti-aliasing

Everything above is aliased: each pixel is fully on or fully off, the staircase of the *algorithms chapter*. GD anti-aliases through a special colour. You name the real colour with `setAntiAliased`, then draw with the constant `gdAntiAliased` in place of a colour:

```
use strict;
use warnings;
use GD;
```

(continues on next page)

(continued from previous page)

```
my $im    = GD::Image->new(220, 60, 1);
my $white = $im->colorAllocate(255, 255, 255);
my $black = $im->colorAllocate(0, 0, 0);
$im->filledRectangle(0, 0, 219, 59, $white);

$im->line(10, 10, 100, 37, $black);                 # aliased

$im->setAntiAliased($black);
$im->line(120, 10, 210, 37, gdAntiAliased);         # anti-aliased

# the same column of each line, three pixels high
for my $x (55, 165) {
    my @grey = map { ($im->rgb($im->getPixel($x, $_)))[0] } 22 ..␣
↪24;
    print "x=$x: @grey\n";
}

open my $fh, '>', 'aa.png' or die "aa.png: $!";
binmode $fh;
print $fh $im->png;
close $fh;
```

```
x=55: 255 255 0
x=165: 255 127 128
```

At that column the ideal line passes exactly between two pixel rows. The aliased line picks one and paints it solid black; the anti-aliased line gives each row half coverage, two mid greys. That is the coverage idea from the algorithms chapter, computed for you.

`gdAntiAliased` works for lines and for polygons, outlined or filled. `filledEllipse` ignores it and keeps a hard edge. For curves that must be smooth, see *when GD is the wrong tool*.

GD has three more special colours in the same style: `gdStyled` draws with a repeating pattern of colours set by `setStyle` (a custom dash), `gdBrushed` stamps a small image set by `setBrush` along the line, and `gdTiled` fills with an image set by `setTile`.

Text

GD has two text systems. The built-in **bitmap fonts** need nothing installed: `gdTinyFont`, `gdSmallFont`, `gdMediumBoldFont`, `gdLargeFont` and `gdGiantFont`, fixed-size and ASCII-oriented, drawn with `string` (and `stringUp` for text rotated 90 degrees). The **TrueType** route, `stringFT`, renders any font file through FreeType at any size and angle, anti-aliased.

```
use strict;
use warnings;
use GD;
```

(continues on next page)

(continued from previous page)

```
my $im    = GD::Image->new(320, 150, 1);
my $white = $im->colorAllocate(255, 255, 255);
my $black = $im->colorAllocate(0, 0, 0);
my $blue  = $im->colorAllocate(0, 90, 200);
$im->filledRectangle(0, 0, 319, 149, $white);

# the built-in bitmap fonts: x, y is the top-left of the text
$im->string(gdSmallFont, 10, 10, 'gdSmallFont', $black);
$im->string(gdGiantFont, 10, 28, 'gdGiantFont', $black);
$im->stringUp(gdMediumBoldFont, 300, 140, 'stringUp', $blue);
printf "gdGiantFont cell: %dx%d\n", gdGiantFont->width, gdGiantFont-
 ↪>height;

# TrueType through FreeType: x, y is the baseline start
my $font = '/usr/share/fonts/TTF/DejaVuSans.ttf';
my @box = $im->stringFT($blue, $font, 20, 0, 10, 90, 'TrueType text
 ↪')
    or die "stringFT: $@";
print "bounding box: @box\n";

# angle is in radians, counter-clockwise
$im->stringFT($black, $font, 11, 0.3, 180, 135, 'at an angle')
    or die "stringFT: $@";

# measure without drawing: call it on the class, not an image
my @size = GD::Image->stringFT($black, $font, 20, 0, 0, 0, 'TrueType␣
 ↪text');
printf "width %d, height %d\n", $size[2] - $size[0], $size[1] -
 ↪$size[5];

# with fontconfig on, a font name works instead of a path
$im->useFontConfig(1);
$im->stringFT($black, 'DejaVu Sans:bold', 11, 0, 10, 135, 'by name')
    or die "stringFT: $@";

open my $fh, '>', 'text.png' or die "text.png: $!";
binmode $fh;
print $fh $im->png;
close $fh;
```

```
gdGiantFont cell: 9x15
bounding box: 9 95 188 95 188 70 9 70
width 178, height 24
```

The font path is a system detail; on your machine `fc-list` shows the installed fonts and their files, or turn on `useFontConfig` and name the font as the last call does.

Three things differ between the two systems, and all three catch people. The **position**: `string` places the top-left corner of the text, `stringFT` places the start of the baseline,

so the same y puts the two at different heights. The **argument order**: `string` takes the font first and the colour last, `stringFT` takes the colour first. And the **angle**: `stringFT` wants radians, counter-clockwise, while `arc` wants degrees, clockwise.

`stringFT` returns the four corners of the text's bounding box as eight numbers (lower-left, lower-right, upper-right, upper-left, each as x and y), and an empty list on failure with the reason in `$@`, so with a wrong font path the example above dies with `stringFT: Could not find/open font`. Called on the class instead of an image it measures without drawing, which is how you centre or right-align a caption.

Saving and loading

Output is one method per format, each returning the encoded bytes; you write them to a file opened in binary mode.

```
use strict;
use warnings;
use GD;

sub save {
    my ($file, $bytes) = @_;
    defined $bytes or die "$file: this libgd cannot write that␣
 ↪format\n";
    open my $fh, '>', $file or die "$file: $!";
    binmode $fh;
    print $fh $bytes;
    close $fh;
}

my $im    = GD::Image->new(200, 100, 1);
my $white = $im->colorAllocate(255, 255, 255);
my $red   = $im->colorAllocate(200, 40, 40);
$im->filledRectangle(0, 0, 199, 99, $white);
$im->filledEllipse(100, 50, 160, 80, $red);

save('shape.png',  $im->png);          # optional 0..9 compression␣
 ↪level
save('shape.jpg',  $im->jpeg(85));     # quality 0..100
save('shape.webp', $im->webp(80));     # quality 0..100
save('shape.gif',  $im->gif);          # palette format: colours are␣
 ↪reduced
save('shape.bmp',  $im->bmp);

my @loaded = (
    [ 'shape.png',  GD::Image->newFromPng('shape.png', 1) ],   # 1:␣
 ↪true-colour
    [ 'shape.jpg',  GD::Image->newFromJpeg('shape.jpg', 1) ],
    [ 'shape.webp', GD::Image->newFromWebp('shape.webp') ],
    [ 'shape.gif',  GD::Image->newFromGif('shape.gif') ],
    [ 'shape.bmp',  GD::Image->newFromBmp('shape.bmp') ],
);
```

(continues on next page)

(continued from previous page)

```
for (@loaded) {
    my ($file, $back) = @$_;
    printf "%-10s %5d bytes  %dx%d  centre=%s\n", $file, -s $file,
        $back->getBounds, join(' ', $back->rgb($back->getPixel(100,␣
↪50)));
}
```

```
shape.png    765 bytes  200x100  centre=200 40 40
shape.jpg   3216 bytes  200x100  centre=200 40 40
shape.webp  1274 bytes  200x100  centre=199 40 39
shape.gif    413 bytes  200x100  centre=204 42 44
shape.bmp  60054 bytes  200x100  centre=200 40 40
```

The centre pixel tells the story of each format. PNG and BMP are lossless and give back exactly `200 40 40`. JPEG and WEBP are lossy; on a flat area like this they come back exact or within a step. GIF is a palette format, so the true-colour image was reduced to at most 256 colours on the way out, and the red moved to the nearest colour in that palette.

Loading is one constructor per format, each taking a file name or an open filehandle: `newFromPng`, `newFromJpeg`, `newFromGif`, `newFromWebp`, `newFromBmp`. Each has a `Data` variant for bytes you already hold in a scalar, such as `newFromPngData($bytes)`. A loader either returns an image or dies: a missing file with `... not found`, a file it cannot decode with the name of the libgd call that refused it. That is why the loop above checks nothing.

The second argument of `newFromPng` and `newFromJpeg` (and their `Data` variants) decides whether a true-colour file stays true-colour, and leaving it out has a cost:

```
use strict;
use warnings;
use GD;

my $pal = GD::Image->newFromPng('shape.png');       # no flag:␣
↪palette
my $tc  = GD::Image->newFromPng('shape.png', 1);    # 1: true-colour
for ([ 'no flag' => $pal ], [ 'flag 1' => $tc ]) {
    my ($how, $im) = @$_;
    printf "%-8s isTrueColor=%d  centre=%s\n", $how, $im->
↪isTrueColor,
        join(' ', $im->rgb($im->getPixel(100, 50)));
}

my $was = GD::Image->trueColor(1);                  # the program-
↪wide default
my $any = GD::Image->new('shape.png');
printf "trueColor(1), was %d: new(file) isTrueColor=%d\n", $was,
↪$any->isTrueColor;

eval { GD::Image->newFromPng('shape.jpg') };
```

(continues on next page)

(continued from previous page)

```
my ($why) = $@ =~ /^(.*?) at /;
print "wrong format: $why\n";
```

```
no flag  isTrueColor=0  centre=204 42 44
flag 1   isTrueColor=1  centre=200 40 40
trueColor(1), was 0: new(file) isTrueColor=1
wrong format: gdImageCreateFromPng error
```

Without the flag, GD converts the loaded file to a palette image, the same reduction the GIF went through, and the red moves. Pass 1 for anything you will filter, resample or blend, which is what the rest of this chapter assumes. `GD::Image->trueColor(1)` changes the default for the whole program and returns the previous one; it also makes `GD::Image->new($width, $height)` true-colour, and it is the only way to steer `GD::Image->new($file)`, which recognises PNG, JPEG, GIF and WEBP from the file's first bytes but takes no flag of its own. A palette file such as a GIF loads as a palette image either way, and `paletteToTrueColor` converts it in place.

Which formats work is decided by how the system's libgd was built, not by GD. An encoder libgd was built without returns `undef`, which is why `save` above checks. `GD::supportsFileType` answers from libgd's format table, which lists a format as present even when its codec is a stub, so the reliable test is the one `save` does: ask for the bytes and see whether you got any.

Transparency in files

Two separate mechanisms make parts of a saved image transparent, one per image kind.

```
use strict;
use warnings;
use GD;

# a PNG with a real alpha channel
my $im = GD::Image->new(100, 100, 1);
$im->alphaBlending(0);                              # write alpha,␣
 ↪do not blend
$im->saveAlpha(1);                                  # keep it in the␣
 ↪PNG
my $clear = $im->colorAllocateAlpha(0, 0, 0, 127);  # fully␣
 ↪transparent
my $glass = $im->colorAllocateAlpha(0, 90, 200, 64);
$im->filledRectangle(0, 0, 99, 99, $clear);
$im->filledEllipse(50, 50, 80, 80, $glass);

open my $fh, '>', 'glass.png' or die "glass.png: $!";
binmode $fh;
print $fh $im->png;
close $fh;

# a palette GIF with one colour marked transparent
my $gif = GD::Image->new(100, 100);
```

(continues on next page)

(continued from previous page)

```
my $bg  = $gif->colorAllocate(255, 255, 255);
my $ink = $gif->colorAllocate(0, 128, 0);
$gif->transparent($bg);
$gif->filledRectangle(25, 25, 74, 74, $ink);

open $fh, '>', 'stamp.gif' or die "stamp.gif: $!";
binmode $fh;
print $fh $gif->gif;
close $fh;

my $back = GD::Image->newFromPng('glass.png', 1);
printf "glass.png: corner alpha %d, centre alpha %d\n",
    $back->alpha($back->getPixel(0, 0)), $back->alpha($back->
↪getPixel(50, 50));
my $g = GD::Image->newFromGif('stamp.gif');
printf "stamp.gif: transparent index %d\n", $g->transparent;
```

```
glass.png: corner alpha 127, centre alpha 64
stamp.gif: transparent index 0
```

For a true-colour PNG, per-pixel alpha survives the round trip only when both switches are set: `alphaBlending(0)` so the alpha is written into the pixels rather than blended away, and `saveAlpha(1)` so the PNG encoder keeps the channel. Forget the second and the file comes out opaque. For a palette image, `transparent($index)` marks one colour of the table as "see-through", all or nothing, which is the only kind of transparency GIF has.

Scaling and copying

GD scales by copying from one image into another of a different size. You create the destination first, then copy a rectangle of the source into a rectangle of the destination. There are two copies, and the difference is the one between the aliased and anti-aliased lines. `copyResized` picks the nearest source pixel for each destination pixel. `copyResampled` averages the source pixels that land on each destination pixel, which is the supersampling downscale from the *algorithms chapter*. The example shrinks the `primitives.png` sheet saved earlier both ways:

```
use strict;
use warnings;
use GD;

my $src = GD::Image->newFromPng('primitives.png', 1);
my ($sw, $sh) = $src->getBounds;

# fit inside 160x160, keeping the aspect ratio
my $scale = 160 / ($sw > $sh ? $sw : $sh);
my ($dw, $dh) = (int($sw * $scale + 0.5), int($sh * $scale + 0.5));

my $fast   = GD::Image->new($dw, $dh, 1);
```

(continues on next page)

(continued from previous page)

```
my $smooth = GD::Image->new($dw, $dh, 1);
#                   src   dstX dstY srcX srcY dstW dstH srcW srcH
$fast->copyResized($src,     0,   0,   0,   0, $dw, $dh, $sw, $sh);
$smooth->copyResampled($src, 0,   0,   0,   0, $dw, $dh, $sw, $sh);

for ([nearest => $fast], [resampled => $smooth]) {
    my ($name, $im) = @$_;
    my %colours;
    for my $y (0 .. $dh - 1) {
        $colours{ $im->getPixel($_, $y) }++ for 0 .. $dw - 1;
    }
    printf "%-9s %dx%d, %d distinct colours\n", $name, $dw, $dh,␣
↪scalar keys %colours;
    open my $fh, '>', "thumb-$name.png" or die "thumb-$name.png: $!
↪";
    binmode $fh;
    print $fh $im->png;
    close $fh;
}
```

```
nearest   160x120, 6 distinct colours
resampled 160x120, 133 distinct colours
```

The nearest-neighbour thumbnail keeps only the six colours it was drawn with, and pays for it: one-pixel lines fall between the sampled pixels and vanish, so the rectangle outline and the arc from the primitives sheet are simply gone. The resampled thumbnail blends them into faint but present lines, which is where its 133 in-between colours come from. Use `copyResampled` for anything a person will look at, and `copyResized` for pixel art you want to stay blocky when enlarged.

The same copy-a-rectangle shape does compositing:

```
use strict;
use warnings;
use GD;

my $canvas = GD::Image->new(100, 100, 1);
my $blue   = $canvas->colorAllocate(0, 0, 255);
$canvas->filledRectangle(0, 0, 99, 99, $blue);

my $patch = GD::Image->new(40, 40, 1);
my $red   = $patch->colorAllocate(200, 40, 40);
$patch->filledRectangle(0, 0, 39, 39, $red);

#                  src     dstX dstY srcX srcY  w   h
$canvas->copy($patch,        10,  10,   0,   0, 40, 40);       #␣
↪stamp it
$canvas->copyMerge($patch,   50,  50,   0,   0, 40, 40, 50);   # 50%␣
↪blend
```

(continues on next page)

(continued from previous page)

```
my $backup = $canvas->clone;                        # an independent␣
↪copy

printf "copy:      %s\n", join ' ', $canvas->rgb($canvas->
↪getPixel(30, 30));
printf "copyMerge: %s\n", join ' ', $canvas->rgb($canvas->
↪getPixel(70, 70));
printf "clone:     %s\n", join ' ', $backup->rgb($backup->
↪getPixel(70, 70));
```

```
copy:      200 40 40
copyMerge: 100 20 147
clone:     100 20 147
```

`copy` stamps the source unchanged. `copyMerge` takes a last argument from 0 to 100 and mixes that percentage of the source into the destination: half of `200 40 40` plus half of `0 0 255` is `100 20 147`. `clone` returns a new image with the same pixels, for when you want to keep the original before filtering it. Storable's `dclone` is no substitute: GD's Storable hooks hand it an empty object with no image behind it, and the first method called on that object dies.

Rotating, flipping and cropping

The quarter turns and mirror images are exact pixel moves, with no resampling and nothing lost. The `copy` forms return a new image and leave the source alone; `flipHorizontal` and `flipVertical` also come in place. `crop` returns a new image holding one rectangle of the source, given as an array reference of x, y, width and height. The example marks two corners of a card so the output can say where they went:

```
use strict;
use warnings;
use GD;

# a card that shows its orientation: red block top-left, blue block␣
↪top-right
my $card  = GD::Image->new(60, 40, 1);
my $white = $card->colorAllocate(255, 255, 255);
my $red   = $card->colorAllocate(200, 40, 40);
my $blue  = $card->colorAllocate(40, 90, 220);
$card->filledRectangle(0, 0, 59, 39, $white);
$card->filledRectangle(0, 0, 14, 14, $red);
$card->filledRectangle(50, 0, 59, 9, $blue);

# which corner pixel holds which block
sub corners {
    my ($im) = @_;
    my ($w, $h) = $im->getBounds;
    my %at;
    for ([ 'top-left', 0, 0 ], [ 'top-right', $w - 1, 0 ],
```

(continues on next page)

(continued from previous page)

```
        [ 'bottom-left', 0, $h - 1 ], [ 'bottom-right', $w - 1, $h
↪- 1 ]) {
        my ($name, $x, $y) = @$_;
        my $c = $im->getPixel($x, $y);
        $at{red}  = $name if $c == $red;
        $at{blue} = $name if $c == $blue;
    }
    return sprintf "%dx%d  red %-12s blue %s", $w, $h,
        $at{red} // '-', $at{blue} // '-';
}

print "original            ", corners($card), "\n";
for my $method (qw(copyRotate90 copyRotate180 copyRotate270
                   copyFlipHorizontal copyFlipVertical)) {
    my $turned = $card->$method;                    # a new image
    printf "%-19s %s\n", $method, corners($turned);
}

my $mirror = $card->clone;
$mirror->flipHorizontal;                            # in place
print "flipHorizontal      ", corners($mirror), "\n";

my $piece = $card->crop([0, 0, 30, 20]);            # [x, y, width,
↪height]
print "crop                ", corners($piece), "\n";

# trim a border: every edge row and column within 0.5% of white goes
my $trimmed = $card->cropThreshold($white, 0.5);
print "cropThreshold       ", corners($trimmed), "\n";

# a contact sheet of the card and its three rotations
my $sheet = GD::Image->new(250, 70, 1);
$sheet->filledRectangle(0, 0, 249, 69, $sheet->colorAllocate(200,
↪200, 200));
my $x = 5;
for my $im ($card, $card->copyRotate90, $card->copyRotate180, $card-
↪>copyRotate270) {
    my ($w, $h) = $im->getBounds;
    $sheet->copy($im, $x, 5, 0, 0, $w, $h);
    $x += $w + 5;
}
open my $fh, '>', 'rotations.png' or die "rotations.png: $!";
binmode $fh;
print $fh $sheet->png;
close $fh;
```

```
original            60x40  red top-left     blue top-right
copyRotate90        40x60  red top-right    blue bottom-right
copyRotate180       60x40  red bottom-right blue bottom-left
```

(continues on next page)

(continued from previous page)

```
copyRotate270       40x60  red bottom-left  blue top-left
copyFlipHorizontal  60x40  red top-right    blue top-left
copyFlipVertical    60x40  red bottom-left  blue bottom-right
flipHorizontal      60x40  red top-right    blue top-left
crop                30x20  red top-left     blue -
cropThreshold       60x15  red bottom-left  blue top-right
```

`copyRotate90` turns the image a quarter **clockwise** as it appears on screen, the red corner moving from top-left to top-right, and swaps width and height; `copyRotate270` is the quarter turn the other way. The contact sheet `rotations.png` shows the card and its three turns side by side. A flip is a mirror, not a turn: after `copyFlipHorizontal` the blue block is on the left, but both blocks are still at the top, which no rotation achieves.

`cropThreshold` strips a uniform border instead of taking a rectangle you name. It removes rows and columns from each edge for as long as every pixel in them lies within the threshold of the colour you pass, the threshold being a percentage of the largest possible colour distance, measured squared. On the card only the bottom goes: the top row, the left column and the right column each touch a coloured block, so the white below the red block's 15 rows is all it can take. A threshold of `0` matches nothing, so a small positive value is the usual choice even for an exact colour.

Filters

libgd ships a set of whole-image filters. Each works in place and returns true on success. Most of them are fixed 3 x 3 *convolutions*, the operation the algorithms chapter wrote by hand, which the example below makes visible by reading one pixel on the edge of a red square and one inside it:

```
use strict;
use warnings;
use GD;

# a test card: a red square on white, so edges show what a filter
↪does
sub card {
    my $im = GD::Image->new(40, 40, 1);
    my $white = $im->colorAllocate(255, 255, 255);
    my $red   = $im->colorAllocate(200, 40, 40);
    $im->filledRectangle(0, 0, 39, 39, $white);
    $im->filledRectangle(10, 10, 29, 29, $red);
    return $im;
}

my @filters = (
    [ gaussianBlur    => [] ],
    [ meanRemoval     => [] ],     # sharpen
    [ emboss          => [] ],
    [ edgeDetectQuick => [] ],
    [ negate          => [] ],
    [ grayscale       => [] ],
```

(continues on next page)

(continued from previous page)

```
    [ brightness      => [40] ],   # -255 .. 255
    [ contrast        => [50] ],   # positive lowers contrast,␣
↪negative raises it
    [ smooth          => [1] ],    # weight of the centre pixel
    [ pixelate        => [8, 0] ], # block size, mode
);

for my $f (@filters) {
    my ($method, $args) = @$f;
    my $im = card();
    $im->$method(@$args) or die "$method failed\n";
    printf "%-16s edge=%-12s inside=%s\n", $method,
        join('/', $im->rgb($im->getPixel(10, 20))),
        join('/', $im->rgb($im->getPixel(20, 20)));
}
```

```
gaussianBlur     edge=213/93/93    inside=200/40/40
meanRemoval      edge=35/0/0       inside=200/40/40
emboss           edge=209/255/255  inside=127/127/127
edgeDetectQuick  edge=17/0/0       inside=127/127/127
negate           edge=55/215/215   inside=55/215/215
grayscale        edge=87/87/87     inside=87/87/87
brightness       edge=240/80/80    inside=240/80/80
contrast         edge=145/105/105  inside=145/105/105
smooth           edge=218/111/111  inside=200/40/40
pixelate         edge=255/255/255  inside=200/40/40
```

Reading the table:

- **`gaussianBlur`** changes only the edge. Its kernel is the 3 x 3 `1 2 1 / 2 4 2 / 1 2 1`, divided by 16; at the edge pixel the left column (weight 4 of 16) is white, so red becomes `(4 x 255 + 12 x 200) / 16 = 213`. One pass is a small blur; call it several times for more.
- **`meanRemoval`** is the sharpen kernel: nine times the centre minus the eight neighbours. Flat areas are untouched and the edge is pushed away from its neighbours, down to `35/0/0`.
- **`emboss`** and **`edgeDetectQuick`** respond only to change, and add a mid-grey offset of 127 so that "no change" is grey rather than black. That is why the flat inside reads `127/127/127`.
- **`negate`**, **`grayscale`** and **`brightness`** are per-pixel, not convolutions, so edge and inside change alike. `grayscale` uses the luma weights (`0.299 R + 0.587 G + 0.114 B`), giving 87 for this red.
- **`contrast`** is per-pixel too, and runs the way its name does not suggest: a positive value **lowers** the contrast. Every channel's distance from mid-grey (127.5) is scaled by `((100 - c) / 100)` squared, so `50` keeps a quarter of it and red's `200` becomes `127.5 + 72.5 / 4 = 145`. A negative value pushes channels apart instead, and `-50` already drives this red to pure `255/0/0`.
- **`smooth`** is a convolution with a centre weight you choose and weight 1 on the eight

neighbours. With `1` all nine are equal, the plain box blur: three white pixels of nine give `(3 x 255 + 6 x 200) / 9 = 218`. A larger weight keeps more of the original pixel, so the blur gets weaker. Both `smooth` and `contrast` take fractional values, such as `smooth(0.5)` or `contrast(12.5)`.

- **`pixelate`** replaces each block with one colour; mode `0` takes the block's top-left pixel, mode `1` its average. The edge pixel sits in a block whose top-left corner is white.

Also available is `selectiveBlur`, which smooths regions while leaving strong edges alone (on the test card it leaves the edge exactly as it was).

GD::Simple

`GD::Simple` ships with GD and wraps an image in a friendlier API: named colours (190 of them), a pen with a current position, and separate fill (`bgcolor`) and outline (`fgcolor`) colours, so one call draws a filled shape with a border.

```
use strict;
use warnings;
use GD::Simple;

my $img = GD::Simple->new(220, 120);     # cleared to white

$img->bgcolor('yellow');                 # fill colour
$img->fgcolor('blue');                   # outline colour
$img->rectangle(10, 10, 90, 60);

$img->bgcolor('orange');
$img->fgcolor('red');
$img->moveTo(150, 40);                   # shapes draw at the pen
 ↪position
$img->ellipse(80, 50);

$img->fgcolor('black');
$img->penSize(3, 3);
$img->moveTo(10, 100);
$img->angle(0);                          # heading: 0 points right
$img->line(100);                         # turtle-style: 100 px
 ↪ahead

$img->moveTo(130, 105);
$img->font(gdMediumBoldFont);
$img->string('GD::Simple');

open my $fh, '>', 'simple.png' or die "simple.png: $!";
binmode $fh;
print $fh $img->png;
close $fh;

printf "%d colour names\n", scalar(my @names = $img->color_names);
```

```
190 colour names
```

The result is a yellow box with a blue border, an orange ellipse with a red border, a thick black line and a caption. Any method `GD::Simple` does not define is passed through to the `GD::Image` inside it, so the filters, `copyResampled` and the rest of this chapter all work on a `GD::Simple` object, and `$img->gd` returns the image itself. It is a good fit for charts, diagrams and quick sketches; for anything that needs exact control of colours and alpha, the plain `GD::Image` API is more direct.

A complete program

Putting it together: a poster with a gradient background, two translucent discs whose overlap blends, a blur pass, an anti-aliased line and a TrueType caption drawn crisp on top, saved as a PNG plus a resampled JPEG thumbnail.

```
use strict;
use warnings;
use GD;

my ($W, $H) = (400, 300);
my $img = GD::Image->new($W, $H, 1);

# background: a vertical ramp from dark blue to near black
for my $y (0 .. $H - 1) {
    my $t = $y / ($H - 1);
    my $c = $img->colorAllocate(int(30 * (1 - $t)), int(40 * (1 -
↪$t)), int(90 - 70 * $t));
    $img->line(0, $y, $W - 1, $y, $c);
}

# two translucent discs; alpha blending composites the overlap
my $sun = $img->colorAllocateAlpha(255, 200, 60, 40);
my $sea = $img->colorAllocateAlpha(60, 200, 255, 40);
$img->filledEllipse(150, 140, 160, 160, $sun);
$img->filledEllipse(250, 140, 160, 160, $sea);

# soften: each gaussianBlur pass is one 3x3 kernel, so repeat for␣
↪more
$img->gaussianBlur for 1 .. 3;

# a crisp anti-aliased horizon and a caption, drawn after the blur
$img->setAntiAliased($img->colorAllocate(255, 255, 255));
$img->line(20, 250, 380, 244, gdAntiAliased);

my $font = '/usr/share/fonts/TTF/DejaVuSans.ttf';
my $ink  = $img->colorAllocate(240, 240, 240);
$img->stringFT($ink, $font, 16, 0, 20, 285, 'Perl, drawn with GD')
    or die "stringFT: $@";

sub save {
```

(continues on next page)

(continued from previous page)

```
    my ($file, $bytes) = @_;
    open my $fh, '>', $file or die "$file: $!";
    binmode $fh;
    print $fh $bytes;
    close $fh;
}
save('poster.png', $img->png);

# a half-size thumbnail
my $thumb = GD::Image->new($W / 2, $H / 2, 1);
$thumb->copyResampled($img, 0, 0, 0, 0, $W / 2, $H / 2, $W, $H);
save('poster-thumb.jpg', $thumb->jpeg(85));
```

Note that the background allocates 300 colours in a loop. On a true-colour image that costs nothing, because `colorAllocate` only packs the channels; on a palette image the table would be full after 256 lines. Order matters too: the blur runs before the line and the caption, so the discs are soft and the text is sharp.

When GD is the wrong tool

GD is raster, so it has the raster limitation: it works at a fixed resolution, and scaling past it degrades. Its anti-aliasing covers lines and polygons only, and its coordinates are integers. For smooth curves, sub-pixel positioning, gradients and output that must stay crisp at any size (a logo, a diagram destined for print or the web), use a vector engine: *Cairo and SVG* covers both routes, and pperl's native `Cairo` binding is shown in *the Cairo showcase*.

When the job is a per-pixel algorithm no filter covers (a custom kernel, a colour transform, a generated texture), write it over the pixels yourself. For small images the plain-Perl loops of the *algorithms chapter* are enough, and `setPixel` and `getPixel` move the result in and out of GD; for large images, PDL does the same arithmetic on whole arrays at once.

When the drawing needs to appear in a live window rather than a file, that is the *GUI toolkits chapter*. For everything that is "make or modify a bitmap on disk", GD is a small, fast and complete answer.

GD and ImageMagick when you need them

`Imager` is the default, but two other raster libraries are worth knowing. GD is older, smaller, and everywhere: a great deal of existing Perl draws with it, and its colour-index model is worth understanding even if you never choose it new. ImageMagick, through `Image::Magick`, is the heavyweight: the widest format support and the deepest bag of transformations, at the cost of a large dependency. This chapter shows ImageMagick, points to the chapter on GD, and says when each beats `Imager`.

GD

GD has a chapter of its own: *raster drawing with GD* covers its colour model, the primitives, text, file formats, scaling and filters, and ends with a complete program.

When GD wins: you are maintaining code that already uses it (a large amount of older Perl does), you want the smallest dependency that still draws, or you specifically need libgd behaviour to match another tool in a pipeline that also uses libgd. For new drawing work with no such constraint, Imager's named-colour and hash-argument API is friendlier and its filter set is richer.

ImageMagick (Image::Magick)

Image::Magick, also called PerlMagick, ships with ImageMagick itself and exposes its full engine. Its strengths are **format breadth** (it reads and writes formats nothing else on CPAN touches) and **transformations** (hundreds of operations from Resize and Blur through Distort, Morphology, and format-specific tricks). Its weakness is the dependency: you need a matching ImageMagick library installed, and it is large.

The API is verb-oriented: you create an object, Read or size it, then call named operations, each returning an error string that is empty on success.

```
use Image::Magick;

my $im = Image::Magick->new;
$im->Set(size => '200x200');
$im->ReadImage('canvas:white');            # a blank white canvas

# drawing is one Draw call per primitive, SVG-path-like
$im->Draw(primitive => 'line',    points => '0,0 199,199',
          stroke => 'blue');
$im->Draw(primitive => 'circle',  points => '100,100 100,40',
          stroke => 'red', fill => 'none');
$im->Draw(primitive => 'polygon', points => '50,10 150,10 100,90',
          fill => 'skyblue');

# transformations are named methods
$im->Blur(radius => 0, sigma => 1.5);
$im->Resize(geometry => '100x100');

my $err = $im->Write('out.png');
warn "$err" if $err;
```

The circle primitive takes the center followed by any point on the perimeter (so 100,100 100,40 is center (100,100) with radius 60), which is ImageMagick's convention rather than a radius. Every geometry argument is ImageMagick's string mini-language ('200x200', '100x100+10+10'), so it is worth a glance at ImageMagick's own geometry documentation for the details.

When ImageMagick wins: you need a format nothing else reads, you are doing heavy batch transformation (thumbnailing thousands of files, format conversion pipelines), or you want one of its many operations that would be a project to reimplement. For interactive drawing or a tight dependency footprint, it is more machine than you need,

and `Imager` or `GD` fit better.

Choosing

You want ...	Reach for
New raster drawing, friendly API, good filters	`Imager`
Smallest dependency, or an existing GD codebase	`GD`
Widest formats, heavy batch transforms	`Image::Magick`

All three are raster and share the raster ceiling: fixed resolution, degradation on upscaling. When you need output that scales cleanly, the *next chapter* moves to vector graphics with `Cairo` and `SVG`.

Vector graphics: Cairo and SVG

Raster tools commit to a resolution; vector tools describe shapes and decide the pixels later, or never. `Cairo` is a full anti-aliased 2D engine that rasterises paths to a pixel surface (or to PDF and SVG): you draw with real-number coordinates and it works out the coverage. `SVG` generates SVG markup, a text description of shapes that a browser or renderer draws later at whatever size it likes. Use `Cairo` when you need pixels now with the quality of a proper rasteriser; use `SVG` when you want resolution-independent output as a file that other tools consume.

`Cairo` also returns in *chapter 9*: pperl ships a native reimplementation of it, so the same code runs with no CPAN Cairo installed, and that chapter shows how the native binding works. Here it is the ordinary CPAN module.

Cairo: paths and the drawing model

`Cairo`'s model is a **path** built up from segments, then either **stroked** (the outline drawn) or **filled** (the interior painted). You set a source colour, build a path with `move_to` and `line_to` and curves, and commit it. Colours are floats in `0 .. 1`, not `0 .. 255`.

```
use Cairo;

# an ARGB surface is a pixel buffer with alpha
my $surface = Cairo::ImageSurface->create('argb32', 200, 200);
my $cr = Cairo::Context->create($surface);

# paint a white background: set source, then paint fills everything
$cr->set_source_rgb(1, 1, 1);
$cr->paint;

# a stroked path
$cr->set_source_rgb(0, 0, 0);
$cr->set_line_width(3);
$cr->move_to(20, 20);
$cr->line_to(180, 20);
```

(continues on next page)

(continued from previous page)

```
$cr->curve_to(180, 100, 20, 100, 20, 180);   # a bezier curve
$cr->stroke;

# a filled, semi-transparent circle (arc angles are in radians)
$cr->arc(100, 100, 40, 0, 6.28318);          # full circle = 2*pi
$cr->set_source_rgba(1, 0, 0, 0.5);           # red at 50% alpha
$cr->fill;

$surface->write_to_png('output.png');
```

Three things to internalise. Coordinates are real numbers, so `move_to(100.5, 47.25)` is meaningful and the *half-pixel offset* from the foundations chapter is how you get crisp thin lines. Arc angles are in **radians**, not degrees, so a full circle is `2 * pi`. And `set_source_rgba`'s alpha gives you the *compositing* for free: fill or stroke with alpha below 1 and Cairo blends over what is already there.

`stroke` and `fill` consume the current path. If you want to both fill and stroke the same shape, use `fill_preserve` (or `stroke_preserve`), which keeps the path for a second operation:

```
$cr->arc(100, 100, 40, 0, 6.28318);
$cr->set_source_rgb(0.4, 0.8, 1);
$cr->fill_preserve;                  # fill, but keep the path
$cr->set_source_rgb(0, 0, 0);
$cr->set_line_width(2);
$cr->stroke;                         # now outline the same circle
```

Premultiplied alpha

The `argb32` surface stores pixels with *premultiplied alpha*: each colour channel is already multiplied by its alpha. This is invisible when you draw through Cairo's API, but it matters the moment you read raw pixel bytes off the surface or feed Cairo's buffer to another library, because the bytes are not the straight RGBA you might assume. When you cross that boundary, un-multiply first. Within Cairo's own drawing it never comes up.

SVG: markup, not pixels

SVG is pure Perl and does something different: it builds an XML document describing shapes, and produces text. Nothing is rasterised; the consumer (a browser, Inkscape, a PDF converter) draws it. That makes it perfect for output that must stay sharp at any zoom, or that other tools will process further.

```
use SVG;

my $svg = SVG->new(width => 200, height => 200);

# a group carries shared style; children inherit it
my $g = $svg->group(id => 'shapes',
```

(continues on next page)

(continued from previous page)

```
                    style => { stroke => 'black', fill => 'none' });

$g->line(x1 => 0, y1 => 0, x2 => 200, y2 => 200);
$g->circle(cx => 100, cy => 100, r => 50, style => { fill =>
 ↪'skyblue' });
$g->rect(x => 10, y => 10, width => 40, height => 40);
$g->path(d => 'M10 190 L100 100 L190 190 Z');   # a triangle via␣
 ↪path data

$svg->text(x => 60, y => 110)->cdata('Hello');

print $svg->xmlify;
```

The style hash maps to SVG's presentation attributes: `stroke`, `fill`, `stroke-width`, `opacity`, and the rest. A `group` (aliased `g`) with a style is how you avoid repeating attributes on every element; children inherit and can override. Text is attached with a chained `cdata` because the character content is a child node, not an attribute.

For a polygon from computed points, `get_path` assembles the points string for you:

```
my @x = (50, 150, 100);
my @y = (10, 10, 90);
my $points = $svg->get_path(x => \@x, y => \@y, -type => 'polygon');
$g->polygon(%$points, style => { fill => 'coral' });
```

`xmlify` (also spelled `to_xml` or `render`) returns the finished document as a string. Write it to a `.svg` file and it opens in any browser; hand it to a converter for PDF or PNG.

Cairo or SVG?

You want ...	Reach for
Pixels now, anti-aliased, from real-number coordinates	`Cairo`
Vector output as a file for browsers or other tools	`SVG`
To render the same drawing to PNG, PDF, and SVG from one API	`Cairo`
No compiled dependency at all (pure Perl)	`SVG`

The two also compose: generate structure as SVG markup, then render it to pixels with a rasteriser when you need a bitmap. For applied vector output in the specific shape of charts and graphs, and for reading the metadata already inside image files, the *next chapter* covers the pragmatic bookends of raster and vector work.

Image metadata and applied 2D charting

Two pragmatic corners of 2D work round out the toolkit arc. First, reading the data already inside image files: dimensions, format, and the EXIF block a camera writes. Second, the most common reason people draw at all, which is turning numbers into a chart. Neither is glamorous, and both come up constantly.

Reading metadata: Image::Info

Before you draw on or resize an image, you often just need to know about it: how big is it, what format, what did the camera record. `Image::Info` is pure Perl, reads the headers without decoding the whole image, and returns a plain hash reference.

```
use Image::Info qw(image_info dim);

my $info = image_info('photo.jpg');
die $info->{error} if $info->{error};

my ($w, $h) = dim($info);
print "format: $info->{file_media_type}\n";     # image/jpeg
print "size:   ${w}x${h}\n";
print "created: $info->{DateTime}\n" if $info->{DateTime};
```

`dim` is a convenience that pulls width and height out of the hash. The rest of the keys depend on the format and on what the file carries: resolution, colour type, and the common EXIF fields for photographs. For a quick "what is this file" it is all you need, and it costs almost nothing because it never decodes the pixels.

Reading and writing metadata: Image::ExifTool

When you need the full metadata surface, and especially when you need to **write** it, `Image::ExifTool` is the definitive tool. It handles the metadata of essentially every image and media format, reads thousands of tags, and can edit them in place without touching the image data.

```
use Image::ExifTool qw(:Public);

my $et = Image::ExifTool->new;
$et->ExtractInfo('photo.jpg');

my $model = $et->GetValue('Model');             # camera model
my $when  = $et->GetValue('DateTimeOriginal');
print "shot on $model at $when\n" if $model;

# rewrite a tag and save
$et->SetNewValue('Artist' => 'Jane Doe');
$et->WriteInfo('photo.jpg');                     # edits metadata
 ↪only
```

`Image::Info` answers "how big, what format" cheaply; `Image::ExifTool` is the one to reach for when the metadata itself is the job (stripping location tags for privacy, batch-tagging a photo library, reading maker-note fields). Both leave the pixels alone.

Charting: the honest state of the art

Charting is applied 2D drawing: axes, bars, lines, and labels composed from the primitives you already know. The Perl charting landscape is thinner than it once was. Several once-popular modules are now unmaintained, so this section leads with what still works and names the rest plainly.

GD::Graph for quick raster charts

`GD::Graph` builds on `GD` and is still maintained. It is the shortest path to a bar, line, or pie chart as a PNG. You give it data as a set of parallel arrays (the first is the x labels, the rest are data series) and it renders.

```
use GD::Graph::bars;

my @data = (
    ['Jan', 'Feb', 'Mar', 'Apr'],       # x-axis labels
    [  12,    19,    8,     23    ],     # one data series
);

my $chart = GD::Graph::bars->new(400, 300);
$chart->set(
    title       => 'Widgets shipped',
    y_max_value => 25,
    dclrs       => ['blue'],
) or die $chart->error;

my $gd = $chart->plot(\@data) or die $chart->error;

open my $fh, '>', 'chart.png' or die $!;
binmode $fh;
print $fh $gd->png;                     # plot() returns a␣
↪GD::Image
close $fh;
```

`GD::Graph::bars`, `::lines`, `::pie`, and `::points` are the workhorses. `plot` returns a `GD::Image`, so everything from the *GD chapter* about output applies. This is the pragmatic default for a chart on disk.

SVG for vector charts

When the chart must stay sharp at any size, or live in a web page as markup, build it as *SVG* directly. There is no chart abstraction to learn: you compute the bar rectangles and axis lines yourself and emit them. That is more code than `GD::Graph`, but it is resolution-independent and needs no compiled dependency.

```
use SVG;

my @values = (12, 19, 8, 23);
my $svg = SVG->new(width => 400, height => 300);
my $base = 260;                         # baseline y
my $bw   = 60;                          # bar width

for my $i (0 .. $#values) {
    my $h = $values[$i] * 8;            # scale to pixels
    my $x = 40 + $i * ($bw + 20);
    $svg->rect(x => $x, y => $base - $h, width => $bw, height => $h,
               style => { fill => 'steelblue' });
```

(continues on next page)

(continued from previous page)

```
}
$svg->line(x1 => 30, y1 => $base, x2 => 380, y2 => $base,
           style => { stroke => 'black' });

print $svg->xmlify;
```

For anything beyond a simple bar or line chart the hand-rolled SVG grows quickly, but for exactly the common cases it is clean and dependency-free.

The legacy options, named

Two charting modules you will find referenced online are effectively unmaintained and are noted here only so their state is not a surprise. **`Chart::Clicker`** produces attractive Cairo-rendered charts but has not seen a release in years and carries a large dependency tree (including Moose); if you already run it, it works, but it is not a choice for new code. **`Chart::Gnuplot`** drives an external `gnuplot` binary and is likewise dormant; if you have `gnuplot` installed and want its plotting power, calling it directly (or through a thin wrapper of your own) is more maintainable than depending on the abandoned module. For scientific plotting specifically, the PDL ecosystem's own graphics modules are the better-supported path and belong with PDL rather than here.

Where this leaves the toolkit arc

You can now create, load, draw on, filter, save, vectorise, inspect, and chart 2D images entirely on disk. The one thing missing is a **screen**: getting a drawing into a live, interactive window. That is the *next chapter*, and it is also the bridge to the two capstone projects, which both need a window to run in.

2D on a screen: windows and the frame loop

Everything so far drew to a file. This chapter draws to a **window**: a surface that repaints, responds to the mouse and keyboard, and can animate.

On pperl that means the native `SDL2` binding, which supplies a window, a hardware-accelerated 2D renderer, an event queue and timing. The CPAN GUI toolkits are covered at the end, briefly, because they cannot run here and the reason is worth stating once.

A window and a renderer

```
use SDL2 qw(:all);

SDL_Init(SDL_INIT_VIDEO) == 0 or die "SDL_Init: ", SDL_GetError(),
 ↪"\n";

my $win = SDL2::Window->new(title => 'pperl', w => 640, h => 480);
my $ren = $win->renderer(vsync => 1) or die "renderer: ", SDL_
 ↪GetError(), "\n";
```

The explicit `SDL_Init` is optional: `SDL2::Window->new` initialises the video subsystem for you if it is not already up. The window closes when the last reference to `$win` goes away. A **renderer** draws into that window, using the GPU where one is available and falling back to software where it is not.

`vsync => 1` ties frame presentation to the display refresh, which paces a loop without a sleep and avoids tearing. `software => 1` forces the software path, which is what a headless test wants.

The frame loop

Every interactive program has the same shape: drain the event queue, update state, draw, present.

```
my $running = 1;
while ($running) {
    while (my $ev = SDL_PollEvent()) {
        $running = 0 if $ev->{type} == SDL_QUIT;
    }
    $ren->set_draw_color(30, 30, 40);
    $ren->clear;
    $ren->set_draw_color(240, 180, 60);
    $ren->fill_rect([ 200, 150, 240, 180 ]);
    $ren->present;
}
```

This is **immediate mode**: the screen is a function of your state, recomputed every frame. Nothing is retained between frames, so there is no object to update when something moves; you just draw it in its new place.

Drain the queue with a `while`, not an `if`. Handling one event per frame falls progressively behind during fast mouse movement, and the symptom is input lagging seconds behind the user.

Drawing

The renderer's primitives are deliberately few:

- `$ren->clear` fills the target with the draw colour.
- `$ren->set_draw_color($r, $g, $b, $a)`, alpha defaulting to 255.
- `$ren->draw_point`, `$ren->draw_line`.
- `$ren->draw_rect($rect)` outlines, `$ren->fill_rect($rect)` fills.
- `$ren->copy($texture, $src, $dst)` blits a texture.
- `$ren->read_pixels($rect)` returns raw RGBA bytes.

A rectangle is `[$x, $y, $w, $h]`, or `undef` meaning the whole target.

There is **no circle, no polygon, no thick line and no text**. That is not an omission to work around casually: SDL's renderer is a blitting engine, and anything richer is either a texture you prepared earlier or pixels you computed yourself with the *algorithms chapter*.

Textures, and drawing offscreen

A texture is an image in the renderer's memory. Created with `SDL_TEXTUREACCESS_TARGET` it can also be drawn **into**, which is how you keep an offscreen image that survives redraws:

```
my $canvas = SDL2::Texture->new($ren, w => 320, h => 240,
                                access => SDL_TEXTUREACCESS_TARGET);

$ren->set_target($canvas);                # draw into the texture
$ren->set_draw_color(255, 255, 255);
$ren->clear;
$ren->set_target(undef);                 # undef: back to the window
$ren->copy($canvas, undef, [0, 0, 320, 240]);
```

`set_target` points the renderer at the texture, and `undef` points it back at the window. The direct function layer's `SDL_SetRenderTarget` does the same and accepts the same objects. The *drawing app* is built on exactly this.

Input

Events arrive as plain hash references with a `type` and whatever keys that type carries. A keyboard event has `scancode`, `sym`, `mod` and `key_name`; a mouse button event has `x`, `y`, `button` and `clicks`.

For continuous movement, ask the keyboard state rather than the queue:

```
SDL_PumpEvents();
$x -= 4 if SDL2::key_pressed(SDL_SCANCODE_LEFT);
```

Events report transitions, so moving per `SDL_KEYDOWN` inherits the operating system's key-repeat delay and stutters. `key_pressed` answers "is it down right now", which is what smooth movement needs. The *game capstone* covers the scancode and keysym distinction that goes with it.

What SDL2 is not

It is not a widget toolkit. There are no buttons, no text fields, no layout, no focus handling, no accessibility. A control is a rectangle you draw and a coordinate test you write, which is fine for four colour swatches and is not fine for a settings dialogue.

If an application needs real widgets, SDL2 is the wrong layer and there is currently no right one on pperl. That is a genuine gap, stated rather than papered over.

The CPAN toolkits

`Prima`, `Perl/Tk`, `Gtk3` and `Wx` are the toolkits a search will find, and they are the right answer on perl5. **None of them runs on pperl**, for one structural reason: each is an XS module, and pperl cannot load a compiled extension, because XS binds to perl5's C-level ABI and pperl has no such ABI to bind to. `Prima.pm` calls `bootstrap Prima` unconditionally at load time, so `use Prima` fails before any of your code runs.

The natural next question is why they are not reached through `Peta::FFI` the way `Cairo` and `SDL2` are. The answer differs per toolkit, and for Prima it is decisive: **there is no library to reach.** Cairo binds `libcairo`, and SDL2 binds `libSDL2`. Prima is not a binding to anything; it *is* the implementation, with its own graphics engine and its own X11 widget layer. Prima 1.77 is roughly 117,000 lines of C and header alongside 95,000 lines of Perl module code. There is no `.so` on the far side of an FFI call, so a native Prima would mean reimplementing a windowing toolkit, not wrapping one.

`Gtk3` and `Wx` do wrap real libraries and are therefore reachable in principle, at the cost of a large binding each. Both are also in poor health upstream: `Wx` and its `Alien::wxWidgets` dependency have not had a release in years, and `Gtk3` is in slow maintenance.

For this guide they are mention-only. If you are writing for perl5 and want widgets, they remain good answers; on pperl the working answer for drawing to a screen is SDL2.

Choosing

You want ...	Reach for
A window, drawing and input on pperl	`SDL2`
Per-frame animation, a game loop	`SDL2`
An offscreen surface that survives redraws	`SDL2` target texture
Vector output to a file rather than a screen	`Cairo`
Real widgets on pperl	nothing today

With a window and a frame loop, the toolkit arc is complete. The *next arc* goes under it: pperl's native FFI, how the bindings this chapter used are built, and two projects that put the whole guide together.

Reaching further: Peta::FFI for graphics

The toolkit chapters covered what CPAN already wraps. This chapter, and the rest of the guide, goes further: reaching a C graphics library that has no CPAN binding at all, using pperl's native FFI. This is a pperl-exclusive capability. Binding a C library the traditional way means writing a C shim, compiling it, and shipping a build toolchain with your code; pperl calls the C library directly, the same mechanism that already gives you `Math::GMP` with no compiled binding to build at all.

`Peta::FFI` is built into the runtime, and so are the native bindings built on its binding layer: they are Rust modules, not Perl-side declarations. (pperl's built-in `FFI::Platypus` is a layer over Peta::FFI's engine, for code written for CPAN's FFI::Platypus.) This matters for graphics because a real graphics binding needs to manage C-allocated objects (a surface, a context) across many calls, and pperl's model handles that the way the C library itself does.

The authoritative reference for the FFI layer is the *Auto-FFI page*; this chapter applies it to graphics specifically.

The two levels

`Peta::FFI` is best understood as two levels: raw calls you write yourself, and native bindings someone has written for you.

Raw calls. You name a library, a function, and a type-signature string, and pperl dispatches the call through `libffi`. Nothing is generated ahead of time; any signature you can spell works.

```
use Peta::FFI qw(dlopen call dlclose);

my $lib = dlopen("libm.so.6");
my $r = call($lib, "sqrt", "(d)d", 2.0);     # sqrt(2) = 1.4142...
dlclose($lib);
```

The signature `"(args)ret"` uses one letter per type: `v` void, `i` int, `l` long, `L` unsigned long or size_t, `d` double, `f` float, `p` a C string in, `P` a mutable output buffer, and `o` an opaque pointer carried as a plain integer. Around those codes sit two more sets of primitives, and they are what make the rest of a graphics API reachable, beyond the functions that take plain scalars: `alloc`, `free`, `peek`, `peek_cstr`, `poke`, `pack_ptr` and `unpack_ptr` for memory you own, and `closure` and `closure_free` for handing a Perl coderef to C as a function pointer.

Native bindings. The other level is a hand-written native binding, built on Peta::FFI's binding layer: a module that opens the library once, resolves every symbol it needs, and presents a clean Perl API, managing the C-allocated objects behind Perl values. It is the faster and more comfortable form of the same reach. `Math::GMP` is exactly this: `use Math::GMP` and you get arbitrary-precision integers with operator overloading, and nowhere did you touch `dlopen` or a signature string. The *next chapter* takes one of these that pperl already ships for a graphics library, `Cairo`, and shows how it works.

What raw calls can and cannot reach

Most of a C graphics API is reachable right now, because most of it is scalars, handles, and memory the caller owns:

- Functions that take and return **numbers** (set a line width, a radius, a colour component) map straight onto `i`, `d`, `f`, and friends.
- Functions that take a **C string** (load a font by name, open a file) use `p`.
- Functions that fill a **caller-provided buffer** (read pixels into memory you own) use `P`, or `alloc` plus `o` when you want to keep the buffer across several calls.
- An **opaque handle** returned by the library (a surface pointer, a context pointer) is an address, and `o` is its type code. Declared with `handle_type`, it becomes an object of a class of your own that frees itself, as a `<Class>` type in the signature. The native pattern in the next chapter stores these object handles the same way: as an integer inside a blessed Perl reference.
- A **struct**, passed or returned by value (a `Color { r, g, b, a }`, a `Rectangle`) or through a pointer (a rectangle out-parameter), is reachable. `record` declares it member by member, Peta::FFI lays it out as the C compiler does, and the signature names it as `{Color}` by value or `*{Color}` by pointer.

- A **union** (an event record) is reachable too, by pointer: `union` declares it as `record` declares a struct, every member at offset 0, and the signature passes it as `*{Event}` for the library to fill in. A union cannot be passed by value.
- **Arrays and out-parameters** take a Perl reference: `\@points` for a `T[]`, `\my $count` for a `*T`, `\$buf` for a `P` buffer C fills.
- A **callback** is reachable. `closure` wraps a coderef as a genuine C function pointer, so a library that calls into your code (a comparator, a progress handler, an event or draw callback) gets one.

Where raw calls stop: **anything derived from a header.** Nothing reads the library's `.h` files for you. You write the signature from the C prototype yourself and declare each record's and union's members yourself. Getting those right is your responsibility. A signature that disagrees with the real prototype is undefined behaviour rather than an error you get told about.

The practical shape of that: the *mechanism* for a graphics library is in place, and what a hand-written binding costs you now is the careful work of transcribing prototypes and declarations correctly. This guide's value is telling you which side of the line a given task falls on, so you neither avoid what works nor promise what does not.

Why a native binding, not raw calls, for real work

You could drive a graphics library entirely through raw `call`s, and for a quick experiment that is fine. For anything real, a native binding is the right unit, for three reasons.

First, **object lifetime**. A graphics context is allocated by the library and must be freed exactly once. A native binding ties that lifetime to a Perl value's `DESTROY`, so the C object is released when the Perl object goes away, with no manual bookkeeping. With raw calls you declare that yourself, with `handle_type` for each C object type, or keep track of every `o` pointer by hand.

Second, **ergonomics**. `$surface->fill` reads better and is harder to get wrong than a `call` with a hand-written signature at every use site.

Third, **the signature is written once**. In a native binding each C function's exact type is declared once, in one Rust module, and the compiler checks every argument against it; every caller just uses the method. Scattering raw signatures through application code is how you get a mismatch that corrupts the stack.

The *Cairo showcase* makes all three concrete with the `Cairo` binding pperl ships, a real vector library whose API is almost entirely scalars and one opaque handle, which is precisely the shape that reaches cleanly today.

The shape of what follows

The next chapter opens up the native Cairo binding pperl ships, built the way `Math::GMP` is, so you can see a graphics library brought up on pperl end to end. Then the two capstones build a drawing application and a 2D game you can run, both on the native `SDL2` binding, with the raw layer shown where it clarifies what the binding removes. The *final chapter* draws the map of what is left.

A native Cairo binding, shipped

The *vector chapter* drew with `Cairo` as an ordinary CPAN module. This chapter shows the other side of that same code: pperl ships a **native reimplementation of `Cairo`**, so `use Cairo` works on pperl with no CPAN Cairo distribution installed and no C compiler involved. The Perl you write is identical; what backs it is pperl's own FFI reaching `libcairo` directly. Cairo makes the ideal worked example because its C API is almost entirely scalars and one opaque handle, which is precisely the shape that reaches cleanly today.

This is a real, shipped binding, not a sketch. Reading it tells you what a native graphics binding is, how the pieces fit, and how to recognise (or ask for) the same treatment for another library. Building such a binding is a runtime task; what you write as a user is the Perl on the far side of it, and that Perl is just the `Cairo` *API* you already saw.

The Perl API: exactly the CPAN one

The native binding exposes the **real `Cairo` package names and methods**: `Cairo::ImageSurface`, `Cairo::Context`, `Cairo::PdfSurface`, and the rest, with the upstream method signatures. Code written for CPAN `Cairo` runs unchanged; ported Cairo test suites pass against it. The program from the vector chapter is the program here:

```
use Cairo;

my $surface = Cairo::ImageSurface->create('argb32', 200, 200);
my $cr      = Cairo::Context->create($surface);

$cr->set_source_rgb(1, 1, 1);
$cr->paint;                                  # white background
$cr->set_source_rgb(0, 0, 0);
$cr->set_line_width(3);
$cr->move_to(20, 20);
$cr->line_to(180, 20);
$cr->stroke;
$cr->arc(100, 100, 40, 0, 6.28318);
$cr->set_source_rgba(1, 0, 0, 0.5);
$cr->fill;

$surface->write_to_png('out.png');
# the surface and context free their C objects when they go out of␣
 ↪scope
```

Nothing in that program hints at how it is implemented. There is no `dlopen`, no signature string, no binding object: just `Cairo`. The rest of this chapter is what makes `use Cairo` do that on pperl, and the pattern is the same one behind `Math::GMP`.

Step 1: open the library once, resolve the symbols

The binding opens `libcairo.so.2` a single time, when the module loads, and resolves each C function it needs into a table of function pointers held for the life of the process. Opening the library, the symbol table and the class each C pointer type is wrapped in come from Peta::FFI's binding layer, which every native library binding in pperl is built

on. This is the same shape as the `Math::GMP` binding, which opens `libgmp` once and resolves its `mpz_*` symbols. For Cairo the resolved symbols are the ones behind the methods above:

```
open libcairo.so.2 once when the module loads
resolve and store, as function pointers:
    cairo_image_surface_create(format, width, height) -> surface*
    cairo_create(surface*)                             -> context*
    cairo_move_to(context*, x, y)
    cairo_line_to(context*, x, y)
    cairo_arc(context*, xc, yc, radius, a1, a2)
    cairo_set_source_rgb(context*, r, g, b)
    cairo_set_source_rgba(context*, r, g, b, a)
    cairo_set_line_width(context*, w)
    cairo_stroke(context*)  cairo_fill(context*)  cairo_
↪paint(context*)
    cairo_surface_write_to_png(surface*, path) -> status
    cairo_destroy(context*)  cairo_surface_destroy(surface*)
```

If `libcairo` is not installed on the system, `use Cairo` still succeeds, and the first call that needs the library dies with a message saying how to install it. If the installed `libcairo` is too old to have one of these functions, only the method that calls it dies, naming the C function; you can catch that with `eval` like any other error. Every one of those C functions has a signature the FFI's type codes already express: the arguments are the opaque handle (an address, so an integer), doubles, and a string path. There is no by-value struct in the core drawing API, which is why Cairo is the clean example.

Cairo is a large library, so the real binding is organised by package: `Cairo::Surface`, `Cairo::Context`, `Cairo::Path`, `Cairo::Pattern`, `Cairo::Matrix`, `Cairo::Font`, and `Cairo::Region` each resolve the symbols they need. That is an implementation convenience; the user sees one `Cairo`. Each of those packages has its own reference page under `Cairo`, with the methods it carries and what each returns. When you want to look a method up rather than understand the pattern, that reference is the place to go, not this chapter.

Step 2: the opaque handle as a blessed integer

Cairo hands back a `cairo_surface_t*` and a `cairo_t*`: pointers to C structs whose insides you never touch. The binding does not need to understand their layout, only to hold the pointer and pass it back on every call. pperl represents such a handle the same way `Math::GMP` holds its `mpz_t` pointer: as an **integer inside a blessed reference**.

```
Cairo::ImageSurface object  =  bless(\ (surface_ptr as integer),
↪'Cairo::ImageSurface')
Cairo::Context object       =  bless(\ (context_ptr as integer),
↪'Cairo::Context')
```

When a method is called, the binding reads the integer back out of the blessed reference, casts it to the C pointer, and calls the function. The blessing gives you method dispatch (`$cr->stroke`), inheritance (every concrete surface class is a `Cairo::Surface` subclass), and, crucially, a `DESTROY` hook. The only difference from `Math::GMP` is that the opaque thing is a drawing surface instead of a big integer.

Step 3: lifetime tied to DESTROY

A C graphics object must be freed exactly once. Leak it and every drawing costs you memory you never get back; free it twice and you corrupt the memory the C allocator is managing, which crashes the program later, somewhere unrelated and hard to trace. The binding ties the C object's life to the Perl object's: when the blessed reference goes out of scope, Perl calls `DESTROY`, and the binding calls the matching C destructor, respecting cairo's reference counting.

```
Cairo::Context DESTROY  ->  cairo_destroy(context_ptr)
Cairo::Surface DESTROY  ->  cairo_surface_destroy(surface_ptr)
```

`Math::GMP` does precisely this: its destructor calls `mpz_clear` and frees the `mpz_t`. You, the user, never call a free function; the drawing objects clean up when the Perl variables holding them go away. The binding enforces the ordering cairo requires, releasing a context before the surface it draws to.

Step 4: the methods

Each Perl method is a small native function that reads the handle, reads the scalar arguments, and makes one C call. `Cairo::Context::move_to` reads the context pointer and the two coordinates and calls `cairo_move_to`; `Cairo::Context::set_source_rgba` reads the pointer and the four colour components and calls `cairo_set_source_rgba`; `Cairo::Surface::write_to_png` reads the pointer and a path string. The mapping is one method to one C call:

```
Cairo::Context::move_to($cr, $x, $y)
    -> read context ptr from $cr; call cairo_move_to(ptr, x, y)

Cairo::Context::set_source_rgba($cr, $r, $g, $b, $a)
    -> call cairo_set_source_rgba(ptr, r, g, b, a)

Cairo::Surface::write_to_png($surface, $path)
    -> call cairo_surface_write_to_png(ptr, path)
```

The C-level signatures live in the binding and are settled there once. A user calling `$cr->set_source_rgba(1, 0, 0, 0.5)` never writes a signature and cannot get one wrong. That single point of truth for the C calls is the third of the *three reasons* a native binding beats scattered raw calls: object lifetime, ergonomics, and the signature written down once.

What this reaches, and what it does not

With those four pieces (open once, handle as a blessed integer, `DESTROY` frees, methods over the resolved function table) pperl delivers a real, memory-safe `Cairo` with no CPAN distribution installed, no C compiler, and no build step. The full *vector chapter* drawing model is available, and the objects manage themselves.

Cairo's core drawing API is scalars plus one opaque handle throughout, which is why Cairo was the first graphics library brought up this way: the four pieces above are the whole job. A library with a busier C interface needs more of the toolbox but not a different pattern. Structs are declared records, passed by value or by pointer; union-typed

event records are declared unions the library fills in through a pointer; a library that **calls back** into your code gets a real function pointer from `closure`. What stays out of reach is the *primer's* hard line: any prototype or declaration you would want derived from a header rather than written out by hand. The *2D game capstone* works through a library on the busier end of that range.

The takeaway

A graphics library whose C API is scalars plus opaque handles can be brought up on pperl as a native binding, and `Cairo` is the first one shipped: open the library once, hold each C object as a blessed integer, free it in `DESTROY`, and back each method with its C call settled in one place. The result is the upstream `Cairo` API, running natively. The *next chapter* stops describing and starts building: a drawing application on the native `SDL2` binding, assembling the whole guide into something that runs.

Capstone: a drawing application

A small paint program you can run: drag to paint, pick a colour, and save a PNG. It is built entirely on the native `SDL2` binding pperl ships, which supplies the window, the input, an accelerated 2D renderer and image output.

The design is the one every paint program uses. An **offscreen canvas** holds the painting; the window merely displays it. The mouse draws onto the canvas, and the frame loop copies the canvas to the window. Keeping the two separate means the painting survives resizes and occlusion for free, and saving is reading back something that already exists.

The canvas is a target texture

SDL renders into a window by default. Give it a texture created with `SDL_TEXTUREACCESS_TARGET` and it will render into that instead, which is this program's offscreen image.

```
use SDL2 qw(:all);

my ($W, $H) = (320, 240);
my $STRIP   = 40;                       # button strip along the bottom

SDL_Init(SDL_INIT_VIDEO) == 0 or die "SDL_Init: ", SDL_GetError(), "\n";

my $win = SDL2::Window->new(title => 'pperl paint', w => $W, h => $H + $STRIP);
my $ren = $win->renderer or die "renderer: ", SDL_GetError(), "\n";

my $canvas = SDL2::Texture->new($ren, w => $W, h => $H,
                                access => SDL_TEXTUREACCESS_TARGET);

sub with_canvas {
    my ($code) = @_;
    $ren->set_target($canvas);
```

(continues on next page)

(continued from previous page)

```
    $code->();
    $ren->set_target(undef);            # undef means the window
↪again
}

with_canvas(sub {                       # start from a white sheet
    $ren->set_draw_color(255, 255, 255);
    $ren->clear;
});
```

A new target texture holds transparent black, so the canvas is cleared to white once, before anything is painted on it. Everything drawn inside `with_canvas` lands on the painting; everything outside it lands on the window and is replaced next frame. Getting that backwards is the one structural mistake available here, and its symptom is a painting that vanishes the moment the window redraws.

The brush

A stroke is a line from the previous mouse position to the current one, so a fast drag stays connected rather than dotting.

```
my @palette = ([0,0,0], [220,40,40], [40,160,60], [40,90,220]);
my $brush   = $palette[0];

sub stroke {
    my ($from, $to) = @_;
    with_canvas(sub {
        $ren->set_draw_color(@$brush);
        # A 3px nib: draw the segment three times, offset. Cheaper
↪than
        # a real thick-line routine and adequate for a brush.
        for my $d (-1, 0, 1) {
            $ren->draw_line($from->[0] + $d, $from->[1],
                            $to->[0]   + $d, $to->[1]);
            $ren->draw_line($from->[0], $from->[1] + $d,
                            $to->[0],   $to->[1]   + $d);
        }
    });
}
```

SDL's `draw_line` is one pixel wide with no thickness parameter, hence the offset draw passes. A proper brush would rasterise a disc along the segment, which is the *algorithms chapter* circle routine applied per step.

Controls, without widgets

SDL2 has no widget toolkit. There is no button object, no layout, no event routing by widget. A control is a rectangle you draw and a coordinate test you write. For four swatches that is less code than configuring a widget would be, and for a real application it is the point at which you would want a toolkit instead.

```
sub button_rect { my ($i) = @_; return [ 8 + $i * 44, $H + 6, 36, 28
↪] }

sub draw_frame {
    $ren->set_draw_color(60, 60, 60);
    $ren->clear;
    $ren->copy($canvas, undef, [0, 0, $W, $H]);
    for my $i (0 .. $#palette) {
        $ren->set_draw_color(@{ $palette[$i] });
        $ren->fill_rect(button_rect($i));
    }
    $ren->set_draw_color(200, 200, 200);
    $ren->draw_rect(button_rect($_)) for 0 .. $#palette;
    $ren->present;
}
```

copy blits the canvas texture into the top of the window, the swatches are filled rectangles below it, and `present` shows the frame.

Input

Events arrive as plain hash references. A press below the canvas picks a colour; a press above it starts a stroke.

```
my $last;
sub handle {
    my ($ev) = @_;
    if ($ev->{type} == SDL_MOUSEBUTTONDOWN) {
        if ($ev->{y} >= $H) {
            for my $i (0 .. $#palette) {
                my $r = button_rect($i);
                $brush = $palette[$i]
                    if $ev->{x} >= $r->[0] && $ev->{x} < $r->[0] +
↪$r->[2]
                    && $ev->{y} >= $r->[1] && $ev->{y} < $r->[1] +
↪$r->[3];
            }
        } else { $last = [ $ev->{x}, $ev->{y} ] }
    }
    elsif ($ev->{type} == SDL_MOUSEMOTION && $last) {
        my $p = [ $ev->{x}, $ev->{y} ];
        stroke($last, $p);
        $last = $p;
    }
    elsif ($ev->{type} == SDL_MOUSEBUTTONUP) { $last = undef }
}

my $running = 1;
while ($running) {
    while (my $ev = SDL_PollEvent()) {
```

(continues on next page)

(continued from previous page)

```
        $running = 0 if $ev->{type} == SDL_QUIT;
        handle($ev);
    }
    draw_frame();
    SDL_Delay(16);
}
```

`SDL_MOUSEMOTION` fires only while the pointer moves, and `$last` is what makes it a drag rather than a paint-on-hover: it is set on press and cleared on release.

Saving

The renderer writes its current target straight out, so saving is pointing it at the canvas and asking. The file takes the target's size: pointed at the canvas it is the 320 x 240 painting, without the button strip, whatever size the window has.

```
sub save_png {
    my ($path) = @_;
    with_canvas(sub { $ren->save_png($path) });
}
```

save_png croaks on failure and returns the renderer on success. Errors therefore arrive as exceptions, so call it plainly and wrap it in `eval` only if you want to catch the failure. The habitual `or die "save failed"` does no harm, but it can never fire.

`save_jpg` and `save_bmp` sit beside it with the same contract.

Running the drawing app

Assemble the pieces into one file and run it with `pperl`. Drag to paint, click a swatch to change colour, and call `save_png` to write the result.

The whole program has been checked with a click and a drag pushed onto SDL's event queue, both in a window and headless (`SDL_VIDEODRIVER=dummy`): the canvas starts white, a swatch click selects red, the drag paints red at the midpoint of the stroke and leaves a pixel 50 rows below untouched. In both cases `save_png` writes a 320 x 240 RGBA PNG holding exactly the painting: a white sheet crossed by the red stroke, with no button strip.

Where the FFI thread would enter

This capstone needs no raw FFI: one shipped binding covers all of it. The *FFI thread* enters when you outgrow it: a pressure-sensitive tablet, or a C brush engine with no Perl binding. That is reached exactly as the *Cairo showcase* described, and the *next capstone* shows the raw layer in use where it earns its place.

Capstone: a 2D game

A complete small game: a paddle, a ball, a score, keyboard control and a loop that runs at a steady frame rate. It is built on the native SDL2 binding, which gives a window, an accelerated renderer, an event queue and timing in one module.

A game is a loop: read input, update the world, draw the world, wait for the next frame, repeat. Everything else is detail.

The world is plain data

```
use SDL2 qw(:all);

my ($W, $H) = (480, 360);

my %game = (
    ball   => { x => 240, y => 180, dx => 3, dy => 3, r => 8 },
    paddle => { x => 200, w => 80, h => 12, y => $H - 30 },
    score  => 0,
    over   => 0,
);
```

Deliberately a plain hash. The update step is ordinary arithmetic on ordinary data and the draw step reads the same data; nothing about either is graphics-specific until the last moment.

Update

Advance the ball, bounce it off three walls and the paddle, end the game if it passes below.

```
sub update {
    return if $game{over};
    my $b = $game{ball};
    $b->{x} += $b->{dx};
    $b->{y} += $b->{dy};

    # Clamp AND reverse. Reversing alone lets a fast ball that
    # overshoots the wall reverse again next frame and stick in it.
    if ($b->{x} - $b->{r} < 0)  { $b->{x} = $b->{r};      $b->{dx} =
↪-$b->{dx} }
    if ($b->{x} + $b->{r} > $W) { $b->{x} = $W - $b->{r}; $b->{dx} =
↪-$b->{dx} }
    if ($b->{y} - $b->{r} < 0)  { $b->{y} = $b->{r};      $b->{dy} =
↪-$b->{dy} }

    my $p = $game{paddle};
    if ($b->{y} + $b->{r} >= $p->{y}
        && $b->{x} >= $p->{x} && $b->{x} <= $p->{x} + $p->{w}
        && $b->{dy} > 0) {                  # only when moving
↪downward
```

(continues on next page)

(continued from previous page)

```
        $b->{y}  = $p->{y} - $b->{r};
        $b->{dy} = -$b->{dy};
        $game{score}++;
    }
    $game{over} = 1 if $b->{y} - $b->{r} > $H;
}
```

Two conditions in there are load-bearing and both guard against the same class of bug. The `$b->{dy} > 0` test stops the ball scoring repeatedly while it is still overlapping the paddle on its way out. The clamps stop a ball that overshoots a wall from reversing again on the next frame and vibrating inside it. **Symptom: the ball sticks to a surface and shudders.** Cause: reversing velocity without correcting position.

Draw

The frame is a function of the world state, recomputed from scratch each time.

```
sub draw {
    my ($ren) = @_;
    $ren->set_draw_color(20, 20, 30);
    $ren->clear;

    my $p = $game{paddle};
    $ren->set_draw_color(230, 230, 230);
    $ren->fill_rect([ $p->{x}, $p->{y}, $p->{w}, $p->{h} ]);

    my $b = $game{ball};
    $ren->set_draw_color(250, 210, 60);
    $ren->fill_rect([ $b->{x} - $b->{r}, $b->{y} - $b->{r},
                      $b->{r} * 2, $b->{r} * 2 ]);
    $ren->present;
}
```

Clearing and redrawing everything is the immediate-mode style: simple to reason about, and fast enough that a game this size never needs anything cleverer. The ball is a square here because SDL's renderer has no circle primitive; a real one would use a texture, or the midpoint circle from the *algorithms chapter*.

Input: events tell you about changes, not state

This is the distinction that decides whether movement feels right.

The event queue reports **transitions**: this key went down, that one came up. Polling it and moving the paddle per `SDL_KEYDOWN` gives movement governed by the operating system's key-repeat delay, which stutters: one step, a pause, then a stream.

To know whether a key is held right now, ask the keyboard state instead.

```
sub input {
    my $p = $game{paddle};
```

(continues on next page)

(continued from previous page)

```
    SDL_PumpEvents();
    $p->{x} -= 6 if SDL2::key_pressed(SDL_SCANCODE_LEFT);
    $p->{x} += 6 if SDL2::key_pressed(SDL_SCANCODE_RIGHT);
    $p->{x} = 0            if $p->{x} < 0;
    $p->{x} = $W - $p->{w} if $p->{x} > $W - $p->{w};
}
```

`key_pressed` reads SDL's live key array, so it is a state query rather than an event, and it moves the paddle a fixed amount every frame for as long as the key is down. `SDL_PumpEvents` is what refreshes that array, and omitting it leaves the state frozen.

Note `SDL_SCANCODE_LEFT` rather than `SDLK_LEFT`. A **scancode** is a physical key position and a **keysym** is what that key means under the current layout. Arrow keys are the same either way, but WASD is not: on an AZERTY keyboard the keysyms are ZQSD while the scancodes stay where the fingers are. Movement wants scancodes; text and shortcuts want keysyms.

The loop

```
SDL_Init(SDL_INIT_VIDEO) == 0 or die "SDL_Init: ", SDL_GetError(),
 ↪"\n";
my $win = SDL2::Window->new(title => 'pperl bounce', w => $W, h =>
 ↪$H);
my $ren = $win->renderer(vsync => 1) or die "renderer: ", SDL_
 ↪GetError(), "\n";

my $running = 1;
while ($running) {
    while (my $ev = SDL_PollEvent()) {
        $running = 0 if $ev->{type} == SDL_QUIT;
    }
    input();
    update();
    draw($ren);
}
```

`vsync => 1` ties `present` to the display refresh, which paces the loop without a sleep and avoids tearing. Without it, add `SDL_Delay(16)` or the loop spins as fast as the CPU allows.

Draining the event queue **completely** each frame matters: a `while` rather than an `if`. A single-event-per-frame loop falls progressively further behind whenever events arrive faster than frames are drawn, and the symptom is input that lags seconds behind the user.

Running the game

Assemble the pieces into one file and run it with `pperl`. Arrow keys move the paddle, each hit with the paddle scores, and missing ends the game.

The whole program has been checked over a run of frames, both in a window and headless (`SDL_VIDEODRIVER=dummy`): the ball advances by its velocity, bounces off the left wall and is pushed back inside to `x=8`, scores and reverses on paddle contact, and ends the game when it passes the bottom. The drawn frame really does contain the ball's colour `(250, 210, 60)` at the ball's centre, the paddle's colour on the paddle, and the background colour everywhere else. With `vsync => 1` the loop runs at the display refresh: at 60 Hz, sixty frames take one second.

Under the hood: what the binding is doing

`use SDL2` is a native binding built on `Peta::FFI`, the same pattern as the *Cairo showcase*: the library is opened once, each handle is an integer inside a blessed reference, and `DESTROY` frees it.

The raw layer is still reachable, and worth seeing once because it shows what the binding removes. Polling an event without it means declaring `SDL_Event`, a C union, and the structs inside it, then passing one by pointer for `SDL_PollEvent` to fill in:

```
use Peta::FFI;
my $sdl = Peta::FFI::dlopen("libSDL2-2.0.so.0");

Peta::FFI::record("Keysym", scancode => "i32", sym => "i32", mod =>
 ↪"u16", unused => "u32");
Peta::FFI::record("KeyEvent", type => "u32", timestamp => "u32",␣
 ↪windowID => "u32",
                  state => "u8", repeat => "u8", pad => "u8[2]",␣
 ↪keysym => "{Keysym}");
Peta::FFI::union("Event", type => "u32", key => "{KeyEvent}",␣
 ↪padding => "u8[56]");

my $ev = Event->new;                              # one SDL_Event
while (Peta::FFI::call($sdl, "SDL_PollEvent", "(*{Event})i", $ev)) {
    next unless $ev->type == 0x300;               # SDL_KEYDOWN
    my $sym = $ev->key->keysym->sym;
}
```

`union` works as `record` does, except that every member starts at offset 0, so the union is as large as its largest member. A union is always passed by pointer, as `*{Event}`; a signature that passes one by value croaks, because a C call cannot pass it that way reliably.

That works, and every member in it was transcribed by hand from SDL's headers: the types and their order, the `0x300`, the 56 bytes of padding that make the union as large as SDL's. The signature `"(*{Event})i"` was written by hand too, from the C prototype. Peta::FFI computes the offsets, but a declaration that disagrees with the header is undefined behaviour rather than an error. The binding pays that cost once, in one place, checked once, which is the argument for a native binding over scattered raw calls.

Where the guide ends

You have built two programs that run: a drawing application and a game, both on bindings pperl ships, with the raw FFI layer shown where it clarifies rather than where it is needed. The *final chapter* draws the boundary of this guide; this game's sequel is the *3D game* at the end of the 3D guide.

Where this guide ends, and what's next

This guide covers two dimensions and stops there on purpose. This closing chapter draws the boundary explicitly: where the third dimension continues, what belongs in separate guides, and where the FFI line runs. Naming these keeps you from hunting for coverage that is not here.

The third dimension

Everything three-dimensional lives in the companion *3D guide*. The *foundations chapter* named the one concept that carries across (the Z-buffer, which decides what is nearest the viewer), and the *algorithms chapter* noted that tessellation into triangles is the bridge into 3D. The 3D guide picks both up and goes the whole way:

- **The mathematics and the pipeline**: vectors, matrices, projection, and a software rasteriser with a real depth buffer, written in plain Perl so every stage is visible.
- **Geometry**: meshes, the OBJ, STL, PLY and glTF file formats, modelling operations, and scene graphs with a camera and picking.
- **The GPU**, through pperl's native `Peta::WebGPU` binding: a window and a frame loop, shaders and buffers, and a lit 3D scene with depth, culling and instancing, plus GPU compute.
- **Three capstones**: a world generator, an interactive modeller, and a 3D game, the sequel to this guide's *2D game*.

Deliberately out of scope

Two areas are excluded on purpose and would each be a separate guide, not a chapter here or in the 3D guide:

- **Computer vision and image analysis.** This guide used *convolution* as a drawing tool (blur, sharpen) and stopped precisely at the line where the same maths becomes detection and recognition. Feature detection, segmentation, and recognition are analysis, not drawing, and belong elsewhere.
- **Hosted image-generation services.** Connectors to external image-generation APIs are network and service-integration work, not graphics programming, and are out of scope regardless of dimension.

Where the FFI line runs

The *reaching-further arc* was candid at each step about what pperl's native FFI reaches. Collected here, the line runs like this.

Raw `Peta::FFI` calls handle scalars, strings, buffers and opaque handles, which is all of a library like *Cairo*. Beyond those, a **struct** is a declared record, passed by value or by pointer, a **union** is declared the same way and passed by pointer, and arrays and out-parameters take a Perl reference. A **callback** is a real function pointer from `closure`. The *game capstone* shows SDL2's event records, a union, declared that way, so a struct-heavy realtime library is reachable rather than blocked.

All of it is reachable, at a price: **nothing is derived from a header.** You write each signature from the C prototype and each record's and union's members by hand, and a mistake is undefined behaviour rather than a reported error. That is the real cost of a hand-rolled binding, and the reason a native binding is the right unit for anything you intend to keep: it pays the transcription once, in one place, checked once. `SDL2` is the worked example. It was written on exactly that pattern and pperl ships it, which is why the capstones call `SDL2::Window->new` instead of counting struct offsets.

What you can do

Set against that line, the ground covered is broad. With this guide you can, on pperl:

- Draw raster images with primitives and filters, and read, scale, and save them (*Imager*, *GD*, *ImageMagick*).
- Produce resolution-independent vector output, as pixels through Cairo or as markup through SVG (*Cairo and SVG*).
- Read and write image metadata, and draw charts (*metadata and charts*).
- Open a window, respond to the mouse and keyboard, and animate (*windows and the frame loop*).
- Reach a C graphics library through a native binding, with no C compiler and no build step (*Cairo showcase*).
- Call a C library through raw FFI, reading its structs back through pointers and handing it callbacks (*FFI primer*, and the raw layer of the *2D game*).
- Build and run real programs: a *drawing application* and a *2D game*.

That is a complete 2D graphics capability in Perl, from the pixel to a running program. The road out is the *3D guide*.

4.2.11 The Definitive Guide to 3D Graphics with Perl

The *2D guide* ended by naming the third dimension as the road out, and this is that road. It covers the same ground for 3D that the first book covered for 2D: the theory you eventually need, the geometry and the file formats, the way a triangle actually reaches the screen, and how Perl talks to the hardware and to the engines everyone else uses. It ends with three programs you can run.

3D has a reputation for being gatekept by C++ and by engines, and the reputation is mostly about tooling rather than about the subject. The mathematics is a handful of ideas that fit on one page. The rendering pipeline is a sequence you can implement yourself before you ever hand it to a graphics card. Perl can drive that graphics card, and this guide shows exactly how.

What this guide runs on

Worth stating plainly at the top, because it shapes every chapter. pperl cannot load a compiled extension: XS binds to perl5's C-level ABI and pperl has no such ABI to bind to. That rules out the historical Perl 3D bindings, which are all XS. What it leaves is better than it sounds, and it is what this guide is built on:

- **Plain Perl**, for the mathematics, the meshes, the file formats and a software renderer you write yourself. Every example in the first two arcs runs with nothing installed.
- **The native SDL2 binding**, which pperl ships, for a window, the event loop and timing.
- **`PDL`**, for arithmetic over a whole grid at once without a Perl loop.
- **`Peta::WebGPU`**, the GPU binding, for drawing and for compute work. The *GPU arc* puts the earlier chapters' geometry on the screen through the graphics card, and the *world generator* writes its terrain stages in plain Perl, PDL and GPU compute and measures what each costs.

Which chapter do I want?

I want to ...	Start here
understand the maths and the transform pipeline	*foundations*
fill a triangle, or know what a depth buffer is for	*software-rasteriser*
light a surface, or paint a texture onto it	*shading*
store geometry, or diagnose a broken model	*meshes*
read or write OBJ, STL, PLY or glTF	*model-formats*
extrude, revolve, subdivide, or combine solids	*modelling-operations*
arrange a scene, steer a camera, or click on an object	*scene-graphs*
open a window, reach the GPU, or run a frame loop	*gpu-device-and-window*
draw a first triangle with shaders and buffers	*gpu-pipeline*
render a lit 3D scene with depth on the GPU	*gpu-3d-scene*
generate a landscape, or compare Perl, PDL and GPU compute	*capstone-world-generator*
build an interactive modeller	*capstone-modeller*
build a small 3D game	*capstone-3d-game*

A note on the second dimension

This guide assumes the *2D guide's foundations*: the difference between raster and vector, the top-left screen coordinate system, RGB and alpha, and source-over compositing. It does not repeat them. Where a 2D idea returns in a new form, the chapter says so and links back rather than re-explaining.

The one place the earlier book deliberately stopped short was the Z-buffer. It named the idea, said that in two dimensions draw order is your depth buffer, and left it there. This guide picks it up properly.

Foundations: vectors, matrices, and the transform pipeline

The screen is flat, so something must decide where a point in space lands on it. That is the **transform pipeline**: vectors, one 4x4 matrix, and the order operations compose in. Built here in plain Perl, because every later chapter uses the vocabulary.

Handedness

Point your right thumb along x and index finger along y; your middle finger points along z. That is a **right-handed** system, used by OpenGL and by most mathematics: with x right and y up, **z comes out of the screen**, so things in front of the camera have *negative* z. DirectX and Unity are left-handed, with z going into the screen.

Both appear in files you load. Getting it wrong renders a scene inside out or mirrors a model; it does not produce an error.

The cross product encodes the handedness, so it is where to check:

```
my $x = [1, 0, 0];
my $y = [0, 1, 0];
print join ", ", @{ v_cross($x, $y) };    # 0, 0, 1
```

x cross y gives +z. `0, 0, -1` means a left-handed convention has crept in somewhere.

Vectors

Three numbers, as an array reference. Either a **position** or a **direction** depending on context; the two behave differently under translation, so it is worth tracking which you have.

```
sub v_add   { my ($a, $b) = @_; return [ map { $a->[$_] + $b->[$_] }␣
↪0 .. 2 ] }
sub v_sub   { my ($a, $b) = @_; return [ map { $a->[$_] - $b->[$_] }␣
↪0 .. 2 ] }
sub v_scale { my ($a, $s) = @_; return [ map { $_ * $s } @$a ] }

sub v_dot {
    my ($a, $b) = @_;
    return $a->[0]*$b->[0] + $a->[1]*$b->[1] + $a->[2]*$b->[2];
}

sub v_cross {
    my ($a, $b) = @_;
    return [ $a->[1]*$b->[2] - $a->[2]*$b->[1],
             $a->[2]*$b->[0] - $a->[0]*$b->[2],
             $a->[0]*$b->[1] - $a->[1]*$b->[0] ];
}

sub v_len  { my ($a) = @_; return sqrt(v_dot($a, $a)) }

sub v_norm {
    my ($a) = @_;
```

(continues on next page)

(continued from previous page)

```
    my $l = v_len($a);
    return $l == 0 ? [0, 0, 0] : v_scale($a, 1 / $l);
}
```

Two of these carry meaning; the rest is bookkeeping.

Dot product: alignment. Zero when perpendicular, positive when aligned, negative when opposed. Divided by both lengths it is the cosine of the angle. Every lighting calculation in this guide is one dot product.

```
print v_dot([1,0,0], [0,1,0]);            # 0, so perpendicular
print v_len([3, 4, 0]);                   # 5
print join ", ", @{ v_norm([3, 4, 0]) };  # 0.6, 0.8, 0
```

Cross product: a vector perpendicular to both inputs. Gives a triangle's *normal* from two of its edges, and a camera's axes from a view direction. It does not commute; swapping the arguments flips the sign.

Normalising scales to length 1. Formulas that take directions assume it, and an unnormalised one scales the result silently rather than failing.

The 4x4 matrix

Rotation and scale fit a 3x3 matrix. Translation does not, because moving a point is an addition. Add a fourth component `w`, set it to 1 for positions, and use a 4x4: translation occupies the last column and one multiply does all three. Coordinates carrying that component are **homogeneous**. `w` is used again at the perspective divide below.

Storage is **column-major**, the layout OpenGL expects: element `m[col * 4 + row]`, putting translation in elements 12, 13, 14.

```
sub m_identity { return [1,0,0,0, 0,1,0,0, 0,0,1,0, 0,0,0,1] }

sub m_mul {                                   # returns a * b
    my ($a, $b) = @_;
    my @out = (0) x 16;
    for my $c (0 .. 3) {
        for my $r (0 .. 3) {
            my $s = 0;
            $s += $a->[$_ * 4 + $r] * $b->[$c * 4 + $_] for 0 .. 3;
            $out[$c * 4 + $r] = $s;
        }
    }
    return \@out;
}

sub m_translate {
    my ($tx, $ty, $tz) = @_;
    my $m = m_identity();
    @$m[12, 13, 14] = ($tx, $ty, $tz);
    return $m;
```

(continues on next page)

(continued from previous page)

```
}

sub m_scale {
    my ($sx, $sy, $sz) = @_;
    my $m = m_identity();
    @$m[0, 5, 10] = ($sx, $sy, $sz);
    return $m;
}

sub m_rotate_y {
    my ($rad) = @_;
    my ($c, $s) = (cos $rad, sin $rad);
    return [ $c,0,-$s,0,  0,1,0,0,  $s,0,$c,0,  0,0,0,1 ];
}
```

Applying one to a point, with `w` carried along:

```
sub xform {
    my ($m, $p) = @_;                          # $p is [x, y, z, w]
    my @o;
    for my $r (0 .. 3) {
        $o[$r] = $m->[$r] * $p->[0] + $m->[4 + $r] * $p->[1]
               + $m->[8 + $r] * $p->[2] + $m->[12 + $r] * $p->[3];
    }
    return \@o;
}

my $t = m_translate(5, 0, 0);
print join ", ", @{ xform($t, [1, 2, 3, 1]) };     # 6, 2, 3, 1
```

Order

Matrix multiplication does not commute. A translate and a scale give two different transforms depending on which side each sits:

```
my $ts = m_mul(m_translate(5,0,0), m_scale(2,2,2));
print join ", ", @{ xform($ts, [1, 2, 3, 1]) };    # 7, 4, 6, 1

my $st = m_mul(m_scale(2,2,2), m_translate(5,0,0));
print join ", ", @{ xform($st, [1, 2, 3, 1]) };    # 12, 4, 6, 1
```

Read a chain **right to left**. In `m_mul($translate, $scale)` the scale runs first: `(1,2,3)` scales to `(2,4,6)`, then moves 5 along x, giving `(7,4,6)`. The other order scales the translation too, putting x at 12. An object that orbits the origin instead of spinning in place has this backwards.

The pipeline: five spaces

A vertex starts in a model file and ends as a pixel position, through a fixed sequence.

1. **Model space.** Coordinates as authored, relative to the object's own origin.
2. **World space**, after the *model matrix*: where the object sits in the scene.
3. **View space**, after the *view matrix*: the scene re-expressed with the camera at the origin looking down -z. The camera never moves; the world moves the opposite way around it.
4. **Clip space**, after the *projection matrix*. Perspective applied but not divided out; `w` holds the depth the divide will use.
5. **Normalised device coordinates**, after dividing by `w`. Everything visible lies in a cube from -1 to +1. A scale maps that to pixels.

The camera

`m_look_at` builds the view matrix from an eye position, a target, and an up hint, by constructing the camera's axes with cross products.

```
sub m_look_at {
    # $up must NOT be parallel to ($target - $eye): the cross
↪product
    # below then has no perpendicular to find and the matrix
↪degenerates.
    my ($eye, $target, $up) = @_;
    my $f = v_norm(v_sub($target, $eye));     # forward
    my $s = v_norm(v_cross($f, $up));         # right
    my $u = v_cross($s, $f);                  # true up, re-squared
    return [ $s->[0], $u->[0], -$f->[0], 0,
             $s->[1], $u->[1], -$f->[1], 0,
             $s->[2], $u->[2], -$f->[2], 0,
             -v_dot($s, $eye), -v_dot($u, $eye), v_dot($f, $eye), 1
↪];
}

my $view = m_look_at([0, 0, 5], [0, 0, 0], [0, 1, 0]);
print join ", ", @{ xform($view, [0, 0, 0, 1]) };   # 0, 0, -5, 1
```

The camera sits 5 along z; the world origin lands at `z = -5`, five units in front of a camera looking down -z. `up` is only a hint, which is why `$u` is recomputed from the other two axes. An `up` parallel to the view direction breaks it: the cross product has no perpendicular to find.

Projection

The projection matrix puts view-space depth into `w`, so the divide that follows shrinks distant things.

```
sub m_perspective {
    my ($fovy_deg, $aspect, $near, $far) = @_;
    my $pi = 4 * atan2(1, 1);
    my $half = $fovy_deg * $pi / 360;
    my $f = cos($half) / sin($half);          # cot(fovy / 2)
    return [ $f / $aspect, 0, 0, 0,
             0, $f, 0, 0,
             0, 0, ($far + $near) / ($near - $far), -1,
             0, 0, 2 * $far * $near / ($near - $far), 0 ];
}
```

The `-1` in the third column is the mechanism: it copies view-space z into the output `w`. The rest scales to the field of view and squeezes depth into the cube.

One point, all the way through:

```
my $proj = m_perspective(60, 4/3, 0.1, 100);
my $clip = xform($proj, xform($view, [0, 0, 0, 1]));
# clip = 0.000, 0.000, 4.810, 5.000      <- w is 5, the distance

my @ndc = map { $_ / $clip->[3] } @{$clip}[0 .. 2];
# ndc = 0.000, 0.000, 0.962

my ($W, $H) = (800, 600);
my $sx = ($ndc[0] * 0.5 + 0.5) * $W;
my $sy = (1 - ($ndc[1] * 0.5 + 0.5)) * $H;
# screen = (400.0, 300.0)
```

Dead centre of an 800x600 window. The `1 -` in the y line is the flip between NDC, where y goes up, and the screen's *top-left origin*, where it goes down. That one line is where the two conventions meet.

Foreshortening

Nothing above special-cases distance, yet distant things shrink, because `w` grows with depth and everything is divided by it:

```
for my $z (1, 2, 4, 8) {
    my $c = xform($proj, xform($view, [1, 0, -$z, 1]));
    printf "z=%-2d  ndc.x = %.3f   w = %.2f\n", -$z, $c->[0] / $c->
↪[3], $c->[3];
}
# z=-1   ndc.x = 0.217   w = 6.00
# z=-2   ndc.x = 0.186   w = 7.00
# z=-4   ndc.x = 0.144   w = 9.00
# z=-8   ndc.x = 0.100   w = 13.00
```

The same point, one unit off the axis, drifts toward the centre as it recedes. Perspective is a division, not an effect applied on top.

Rotations

Euler angles: three numbers, one per axis. Readable, so this is what a user interface should show. Two failures: **gimbal lock**, where at certain orientations two axes align and a degree of freedom is lost, and poor interpolation, taking curved paths between orientations.

Quaternions: four numbers encoding an axis and angle together. No gimbal lock, compose by multiplication, interpolate smoothly. Not readable by inspection.

Store quaternions, expose Euler angles, convert at the edges.

Next

The *rasteriser* turns these screen positions into pixels: which triangles face away, and what to do when two land on the same pixel.

Rasterisation: triangles, depth, and draw order

Foundations produced screen positions. This chapter fills the space between them.

Three things differ from the 2D *scanline polygon fill*: everything is a triangle, each pixel carries a depth as well as a colour, and values varying across the surface cannot be interpolated linearly once perspective is involved.

Canvases here are small enough to print, so every figure is what the code produced.

Why triangles

Three points always lie in one plane, and a triangle is always convex. Neither holds for a general polygon: four points can be non-planar, and a concave outline needs the crossing-pair logic the 2D scanline fill carried. So geometry is tessellated into triangles first, and a triangle is the only primitive from here on.

The render target

A colour buffer and a depth buffer of the same size, the latter holding the distance to the nearest surface written so far, starting further than anything.

```
sub new_target {
    my ($w, $h) = @_;
    return { w => $w, h => $h,
             px    => [ map { [ ('.') x $w ] } 1 .. $h ],
             depth => [ map { [ (9e9) x $w ] } 1 .. $h ] };
}
```

Characters stand in for colours so the results print directly.

The inside test

One function carries the rasteriser. Given three points it returns twice the signed area of the triangle they form:

```
sub edge {
    my ($a, $b, $c) = @_;
    return ($b->[0] - $a->[0]) * ($c->[1] - $a->[1])
         - ($b->[1] - $a->[1]) * ($c->[0] - $a->[0]);
}
```

With a and b fixed as a line, the sign says which side c is on: positive one side, negative the other, zero exactly on it.

A point is inside the triangle when it is on the same side of all three edges, so run edge once per edge and check the signs agree.

```
my $w0 = edge($v1, $v2, $p);
my $w1 = edge($v2, $v0, $p);
my $w2 = edge($v0, $v1, $p);
next unless ($w0 >= 0 && $w1 >= 0 && $w2 >= 0)
         || ($w0 <= 0 && $w1 <= 0 && $w2 <= 0);
```

Accepting both sign conventions takes a triangle wound either way. Sampling at the pixel centre `($x + 0.5, $y + 0.5)` is the 2D guide's *half-pixel offset*: an integer coordinate names a corner, so coverage is decided at the middle.

Loop over the triangle's bounding box. A 20 by 10 target:

```
....................
.########...........
..##############....
...############......
...#########........
....#######.........
.....####...........
......#.............
....................
....................
```

54 pixels. No scanlines, edge lists or sorting: the same three comparisons at every candidate pixel, independent of each other, which is why hardware uses this formulation.

Winding and culling

edge on the triangle's own vertices gives its signed area, and the sign says which way it winds on screen:

```
print edge([1,1], [17,2], [6,8]);    # 107
print edge([1,1], [6,8], [17,2]);    # -107
```

Same points, opposite order, opposite sign. That sign distinguishes the front of a surface from the back.

A closed object is built so every triangle winds the same way seen from outside; rotate it and the ones now facing away flip sign. Those are invisible, so they can be discarded before any pixel work. That is **back-face culling**:

```
my $area = edge($v0, $v1, $v2);
return 0 if $opt{cull} && $area <= 0;
```

Both windings with culling on: 54 pixels front-facing, 0 reversed. On a closed mesh this discards roughly half the triangles for one sign test. It is also why a model with inconsistent winding renders with holes; each hole is a triangle culled that should not have been.

Barycentric interpolation

Divided by the total area, the three edge results are the **barycentric coordinates** of the point: weights summing to 1 giving each vertex's contribution.

```
my ($b0, $b1, $b2) = ($w0 / $area, $w1 / $area, $w2 / $area);
my $z = $b0 * $v0->[2] + $b1 * $v1->[2] + $b2 * $v2->[2];
```

At a vertex its own weight is 1 and the others 0; at the centre they are equal. Any per-vertex value interpolates this way: depth here, equally a colour, a normal or a texture coordinate.

The depth buffer

Two triangles overlap on screen. Which does the viewer see?

The 2D answer was draw order, last one wins: the **painter's algorithm**. It fails as soon as draw order does not match depth order. A is near (`z = 2`), B is far (`z = 8`).

Drawing B then A gives the right image, but only because that order happened to match the depth order:

```
..................
..AAAAAAAAAA.......
..AAAAAAAABBBBBBB..
..AAAAAAAABBBBBBB..
..AAAAAA..BBBBBBB..
..AAAA......BBBBB..
..AAA.........BBB..
..A............BB..
................B..
..................
```

Reversed, A then B, the same code produces a **wrong** image. The figure below is the bug, not the goal:

```
..................
..AAAAAAAAAA.......
..AAAAABBBBBBBBBB..
..AAAAAABBBBBBBB..
```

(continues on next page)

(continued from previous page)

```
..AAAAAA..BBBBBBBB..
..AAAA......BBBBBB..
..AAA.........BBBB..
..A............BBB..
................B..
....................
```

The far triangle has painted over the near one. The geometry is fine; the algorithm has no idea which is nearer.

Sorting back to front does not fix it: two long triangles can each be partly in front of the other, and three can form a cycle with no valid order. The answer must work per pixel:

```
next if $z >= $t->{depth}[$y][$x];    # something nearer is already
↪here
$t->{depth}[$y][$x] = $z;
$t->{px}[$y][$x] = $ch;
```

Keep the nearest depth per pixel, skip anything further. Both draw orders now produce the same image, confirmed pixel by pixel in the harness. Triangles can arrive in any sequence.

That is the **Z-buffer**, named and deferred in the *2D guide's foundations*. It costs one screen-sized buffer and one compare per pixel.

Perspective-correct interpolation

Linear interpolation in screen space is wrong for everything except depth. A receding surface covers less screen per unit of surface, so equal steps across the screen are not equal steps across the surface. A naively interpolated texture slides and warps, most visibly on a floor.

Interpolate the value divided by `w`, interpolate `1/w` alongside it, and divide at the end. An edge from `z = 1` to `z = 5` carrying a coordinate from 0 to 1:

```
t      screen-linear  perspective
0.00   0.000          0.000
0.25   0.250          0.062
0.50   0.500          0.167
0.75   0.750          0.375
1.00   1.000          1.000
```

Halfway across the screen is 17 percent along the surface, not 50. The two agree at the endpoints and diverge between, so the bug looks like a texture correct at the corners and wrong in the middle.

Depth needs no correction: `1/w` is what the depth buffer should store, and that interpolates linearly in screen space already.

Near-plane clipping

A vertex behind the camera has `w <= 0`, and dividing by it produces garbage. Such vertices must go before the divide. The test is `w > near`, and a segment crossing it is cut:

```
sub clip_near {
    my ($a, $b) = @_;                  # each [x, y, z, w]
    my ($da, $db) = ($a->[3] - $near, $b->[3] - $near);
    return ($a, $b) if $da >= 0 && $db >= 0;      # wholly in front
    return ()       if $da <  0 && $db <  0;      # wholly behind
    my $t = $da / ($da - $db);                    # where it crosses
    my $m = [ map { $a->[$_] + ($b->[$_] - $a->[$_]) * $t } 0 .. 3
↪];
    return $da >= 0 ? ($a, $m) : ($m, $b);
}
```

A segment from `w = 4` to `w = -2` cuts at `[5.20, 0.00, 0.00, 0.10]`, exactly on the near plane; both ends behind returns nothing. Clipping a triangle runs this against each edge, which can leave four vertices, so the result is re-triangulated before rasterising.

The far plane works the same way. The other four sides need no clipping here, because the bounding box is already clamped to the screen.

In hardware

All of the above is fixed-function silicon on a graphics card: edge functions, barycentric weights, depth compare, perspective divide, run per triangle in parallel. What you supply is the vertex positions going in and the colour decision coming out.

That is why the *GPU scene chapter* reads as familiar vocabulary: a depth test you enable, a cull-face mode you pick a winding for, and interpolated values that arrive perspective-correct.

The *next chapter* decides what colour a covered pixel is.

Shading: normals, light, and texture

The *rasteriser* decided which pixels a triangle covers; **shading** decides their colour. It rests entirely on the *dot product*: a surface is bright in proportion to how much it faces the light, and that is one dot product. The rest is refinement.

Normals

A **normal** is a unit vector pointing out of a surface. For a triangle it is the cross product of two edges:

```
my $n = v_norm(v_cross(v_sub($v1, $v0), v_sub($v2, $v0)));
# [0, 0, 1] for a triangle lying in the xy plane
```

Direction depends on winding, as with back-face culling: reversing the vertex order gives `[0, 0, -1]`. A mesh with inconsistent winding has some normals pointing inward, and those triangles render black.

One normal per triangle lights a flat surface. For curved objects, store a normal per **vertex** instead, averaging the normals of the faces meeting there. The geometry stays faceted, the lighting does not, which is why a sphere of 200 triangles can look round.

Lambert

Brightness is the cosine of the angle between the normal and the direction to the light, which for unit vectors is their dot product.

```
# Both must be unit vectors, and $light_dir points FROM the surface
# TOWARD the light, not the direction the light travels.
my $i = v_dot($normal, $light_dir);
$i = 0 if $i < 0;
```

The clamp matters: a negative value means the light is behind the surface, and without it light shines through solid objects.

```
light direction              intensity
straight on  [0,0,1]         1.000
30 deg off   [0.5,0,0.87]    0.866
60 deg off   [0.87,0,0.5]    0.500
edge on      [1,0,0]         0.000
from behind  [0,0,-1]        0.000
```

This models a **matte** surface: chalk, paper, unfinished wood. It looks identical from every viewing angle, because the formula never mentions the viewer.

Alone it leaves anything facing away from the light pure black, which no real scene is, because light bounces. The cheap fix is an **ambient** term:

```
my $i = 0.1 + 0.9 * $lambert;      # never fully black
```

Not physically justified: a stand-in for the bounced light the model does not simulate, which keeps shadowed areas readable.

Specular

Shiny surfaces have a highlight that moves with the viewer, which Lambert cannot produce. **Blinn-Phong** adds one using the **halfway vector**, the direction midway between light and viewer:

```
# $view_dir points FROM the surface TOWARD the viewer, matching
# $light_dir's convention. Clamp the dot product before the power:
# a negative base with a fractional exponent is a domain error.
my $H = v_norm(v_add($light_dir, $view_dir));
my $nh = v_dot($normal, $H);
$nh = 0 if $nh < 0;
my $spec = $nh ** $shininess;
```

When the normal aligns with the halfway vector the surface bounces light straight at the viewer and the highlight peaks. The exponent controls how tight it is:

```
shininess    specular
1            0.9659
8            0.7578
32           0.3298
128          0.0118
```

Same geometry throughout, only the exponent changes. Low values give a broad sheen, high values a small bright spot: roughly 8 for polished wood, 32 for plastic, 128 and up for glass or wet metal.

The three terms combined:

```
my $colour = $ambient + $diffuse * $lambert + $specular * $spec;
```

How often you evaluate it

The formulas are the same wherever they run. How often they run changes the result enormously.

Flat shading: one intensity per triangle, applied to every pixel. Cheap, and the faceting shows:

```
 +++++++
  +++++++++++++++++++
   ++++++++++++++++++++++++
   ++++++++++++++++++++
    ++++++++++++++++++
     +++++++++++++++
      ++++++++++++
      ++++++++
       +++++
        +
```

Gouraud shading: an intensity per vertex, interpolated with the same barycentric weights the rasteriser already computes for depth:

```
my $i = $b0 * $inten[0] + $b1 * $inten[1] + $b2 * $inten[2];
```

```
@%%%%##
 %%%####****++++===-
  %%###****++++====---::::
  %####***++++====---::
   ###***++++====----
    #****++++====--
     ***++++===-
     **++++==
      ++++=
       +
```

Three lighting calculations rather than one per pixel, and the flat region becomes a gradient. The approximation is interpolating the **result**, which shows on specular highlights:

a highlight landing mid triangle with all three vertices dim vanishes, because there is nothing at the vertices to interpolate.

Per-pixel shading, confusingly called Phong shading, interpolates the **normal** instead and runs the lighting at every pixel. More expensive, and correct. This is what a fragment shader does.

The rule that carries forward: interpolate the inputs, not the answer.

Texture mapping

Same interpolation again. Each vertex carries a **UV coordinate**, a position in an image, both axes 0 to 1 regardless of pixel size. Interpolate UVs across the triangle and look up the image at the result:

```
# $b0..$b2 must be PERSPECTIVE-CORRECT barycentric weights. Screen-
 ↪linear
# weights slide the texture across the surface; see the rasteriser␣
 ↪chapter.
my $u = $b0 * $uv[0][0] + $b1 * $uv[1][0] + $b2 * $uv[2][0];
my $v = $b0 * $uv[0][1] + $b1 * $uv[1][1] + $b2 * $uv[2][1];
my $texel = sample($image, $u, $v);
```

A checker pattern sampled that way, over the same triangle:

```
 ####...
  ......####...###...
   ...###...####......###..
   ####...###......####.
    ...####......###..
     ###......####.
       ......###..
       ...####.
        ###..
          .
```

Assigning UVs is **unwrapping**, a modelling task rather than a rendering one: deciding how the surface is cut and flattened, like the net of a paper model.

Two practical details. UVs are the value that slides visibly without **perspective-correct** interpolation. And a UV outside 0 to 1 needs a rule, repeat or clamp, which is a per-texture setting in every graphics API.

The same lookup carries more than colour: per-pixel normals (a **normal map**, bumps without geometry), shininess, or how much light a point blocks. Only the meaning of the number changes.

Physically based rendering

The model above uses numbers chosen by eye. **PBR** replaces them with quantities that conserve energy, so a surface never reflects more light than it receives, and describes materials by measurable properties: roughness, and whether the surface is metal.

The practical consequence: PBR materials hold up under any lighting, where Blinn-Phong parameters are tuned per scene and break in the next one. The pipeline shape is unchanged, with a dot product still at the centre.

Blinn-Phong is still worth implementing first; every PBR treatment assumes it.

Next

You can light a surface and texture it, but have nothing to shade yet. The next arc covers *meshes*, the file formats carrying them, the operations that construct new shapes, and the scene graph arranging them in a world.

Meshes: vertices, indices, and the invariants that matter

A **mesh** is the geometry an object is made of. In practice that means two arrays, plus a set of invariants which, when violated, produce the rendering faults this chapter names.

The indexed triangle mesh

A vertex array holds each distinct position once. An index array names three vertices per triangle. A cube of side 2:

```
# Winding is counter-clockwise seen from OUTSIDE, OpenGL's GL_CCW
# default. Every triangle in the mesh must agree on this.
my @vert = (
    [-1,-1,-1], [ 1,-1,-1], [ 1, 1,-1], [-1, 1,-1],   # 0..3  back
 ↪(z=-1)
    [-1,-1, 1], [ 1,-1, 1], [ 1, 1, 1], [-1, 1, 1],   # 4..7  front
 ↪(z=+1)
);
my @idx = (
    4,5,6,  4,6,7,      # front  (+z)
    1,0,3,  1,3,2,      # back   (-z)
    0,4,7,  0,7,3,      # left   (-x)
    5,1,2,  5,2,6,      # right  (+x)
    3,7,6,  3,6,2,      # top    (+y)
    0,1,5,  0,5,4,      # bottom (-y)
);
```

```
vertices stored: 8
triangles:       12
indices:         36
expanded (no index buffer) would store 36 vertices
position data: 60 floats indexed, 108 floats expanded
```

Indexing is not only about size. A shared vertex is shared: move it once and every triangle using it follows, which is what makes editing tractable. It also lets the GPU cache transformed vertices instead of transforming the same corner five times.

CCW-from-outside is consistent with the *rasteriser*: a counter-clockwise triangle in a right-handed space projects to a clockwise triangle on a y-down screen, which is the positive edge area that chapter treated as front-facing.

Invariant 1: consistent winding

Every triangle must wind the same way seen from outside. The check is the **signed volume**, which for a closed mesh is the sum over triangles of one sixth of `dot(a, cross(b, c))`:

```
sub signed_volume {
    # Requires a CLOSED mesh. On an open surface the result is
    # meaningless rather than merely wrong.
    my ($v, $i) = @_;
    my $vol = 0;
    for (my $k = 0; $k < @$i; $k += 3) {
        my ($a, $b, $c) = @{$v}[ @{$i}[$k, $k+1, $k+2] ];
        $vol += v_dot($a, v_cross($b, $c)) / 6;
    }
    return $vol;
}
```

```
signed volume = 8.0      side 2 cube, so 8 means every face is␣
↪outward
one triangle reversed: 6.6667
```

A side-2 cube has volume 8, and the test returns exactly that. Reverse one triangle and it returns 6.6667: that triangle now subtracts its contribution instead of adding it. A negative total means the whole mesh is inside out.

Symptom: a model renders with holes in it, and the holes move as the camera orbits. Cause: inconsistent winding. The holes are triangles being back-face culled that should not have been.

Invariant 2: the mesh is manifold

In a closed mesh, every **undirected** edge is shared by exactly two triangles, and every **directed** edge occurs exactly once. The second half is the winding check again, expressed combinatorially: two adjacent triangles traverse their shared edge in opposite directions, so a directed edge appearing twice means two neighbours disagree.

```
undirected edges: 18, each shared by exactly 2 faces: yes
directed edges occurring twice (a winding error): 0
same test on the mesh with one triangle reversed: 3 duplicate␣
↪directed edges
```

Both faults are worth testing for on any mesh you load or generate:

- An undirected edge used **once** means the surface has a hole.
- Used **more than twice** means it is non-manifold, and operations like boolean subtraction and subdivision have no defined answer there.
- A duplicated **directed** edge means inconsistent winding, exactly the three the reversed triangle produced above.

Normals, and the trap in averaging them

A face normal is the cross product of two edges. Per-vertex normals come from averaging the normals of the faces meeting at that vertex, which is what makes a faceted surface shade smoothly.

The naive average is weighted by **triangle count**, and that is rarely what you want:

```
vertex 6 at [1, 1, 1] -> normal [0.333, 0.667, 0.667]
```

The corner of a cube should point equally along all three axes, at `[0.577, 0.577, 0.577]`. It does not, because the six quads were each split into two triangles and that split is not symmetric about the corner: two triangles of the front face touch vertex 6, two of the top face, but only one of the right face. The average comes out `[1, 2, 2]` normalised.

Fixes, in increasing order of correctness: weight each face normal by its **area**, or by the **angle** the face subtends at that vertex. Angle weighting is the one that survives retriangulation, because the angles at a corner do not change when you split a quad differently.

Symptom: a smooth-shaded model has a subtle lighting seam along an edge where nothing else is wrong. Cause: unweighted normal averaging across a region that was triangulated asymmetrically.

When one position needs several vertices

Averaging normals is right for a curved surface and wrong for a cube. A cube should look faceted, which means each corner needs a **different** normal per face it belongs to. A vertex is not a position; it is a position **plus its attributes**, and two triangles can share a position while disagreeing about the normal there.

So a smooth-shaded cube has 8 vertices, and a correctly faceted one has 24: each of the 8 corners duplicated once per adjoining face. The same applies to UV coordinates. A **UV seam**, where the texture is cut to be unwrapped flat, forces duplicated positions along the cut, because one position needs two different UVs.

Symptom: an exported model has far more vertices than its corners suggest. Usually not a bug: hard edges and UV seams both split vertices, and a cube legitimately arrives as 24.

Bounding volumes

Cheap approximations used to reject work early: a ray that misses the box cannot hit any triangle in it.

```
AABB min [-1, -1, -1] max [ 1,  1,  1]
sphere centre [ 0,  0,  0] radius 1.7321
```

An **axis-aligned bounding box** is two corners, trivially computed, and tests fast. A **bounding sphere** is a centre and a radius, invariant under rotation, so it does not need recomputing when the object turns. The cube's radius is the half-diagonal, `sqrt(3)`.

Both are used by the culling and ray tests later in this guide, and they are the leaves of the BVH the scene-graph chapter builds.

Welding

Geometry from a file or a generator often repeats positions that should be shared. Welding collapses them by exact or tolerance match:

```
# Exact match on a rounded key. A tolerance-based weld needs a␣
 ↪spatial
# hash, not a string key, or it is quadratic.
my $key = join ",", map { sprintf "%.6f", $_ } @$v;
```

```
raw vertices 6 -> welded 4, indices [0,1,2,1,3,2]
```

Two triangles authored as six independent vertices become four vertices and an index list. Weld before running the manifold check: unwelded geometry reports every edge as used once, because no two triangles share a vertex index even where they visually share an edge.

Half-edge, for editing

The two arrays answer "what are this triangle's vertices" in constant time and "which faces touch this vertex" only by scanning everything. Editing operations need the second question constantly.

A **half-edge** structure stores each edge twice, once per direction, with each half knowing its face, its next half-edge, and its opposite. Walking the opposites around a vertex enumerates its neighbours without a scan. It is what a modelling tool holds in memory; a renderer converts back to vertex and index arrays before drawing. The modelling-operations chapter builds on it.

Next

The mesh here was written as Perl literals. Real geometry comes from files, and the *next chapter* covers the formats that carry it.

Model formats: OBJ, STL, PLY, glTF

No mesh-format module ships with pperl, and there is no maintained CPAN one to fall back on. That is less of a problem than it sounds: OBJ and STL are small enough to parse directly, and this chapter gives working readers and writers for both.

What the four differ on

	Geometry	Materials	Scene tree	Animation	Encoding	Indexed
OBJ	yes	via `.mtl`	no	no	text	yes
STL	yes	no	no	no	text + binary	no
PLY	yes	vertex colour	no	no	text + binary	yes
glTF	yes	PBR	yes	yes	JSON + binary	yes

OBJ for interchange of static geometry, STL for 3D printing, PLY for scanned point and vertex data, glTF for anything with a scene, materials or animation.

OBJ

Line-oriented text. `v` a position, `vt` a texture coordinate, `vn` a normal, `f` a face referring to them.

```
sub write_obj {
    my ($path, $v, $i) = @_;
    open my $fh, '>', $path or die "$path: $!";
    print $fh "# cube, side 2\n";
    printf $fh "v %g %g %g\n", @$_ for @$v;
    for (my $k = 0; $k < @$i; $k += 3) {
        # OBJ indices are 1-BASED
        printf $fh "f %d %d %d\n", map { $_ + 1 } @{$i}[$k, $k+1,
↪$k+2];
    }
    close $fh;
}
```

The cube becomes 21 lines: a comment, 8 positions, 12 faces.

```
# cube, side 2
v -1 -1 -1
v 1 -1 -1
v 1 1 -1
...
f 1 6 5
```

Reading is where the traps are, and there are four:

```
sub read_obj {
    my ($path) = @_;
    open my $fh, '<', $path or die "$path: $!";
    my (@v, @f);
    while (my $line = <$fh>) {
        next if $line =~ /^\s*(#|$)/;
        my @tok = split ' ', $line;
        my $kw = shift @tok;
        if ($kw eq 'v') { push @v, [ @tok[0..2] ] }
        elsif ($kw eq 'f') {
            # A face may have MORE than three vertices. Fan-
↪triangulate.
            # Each token may be v, v/vt, v//vn or v/vt/vn: take the␣
↪first.
            my @vi = map { my ($n) = split m{/}, $_; $n } @tok;
            # Indices are 1-based; a NEGATIVE index counts back from␣
↪the
            # current end of the vertex list.
            @vi = map { $_ < 0 ? @v + $_ : $_ - 1 } @vi;
            push @f, [ $vi[0], $vi[$_], $vi[$_ + 1] ] for 1 .. $#vi␣
↪- 1;
        }
```

(continues on next page)

(continued from previous page)

```
    }
    close $fh;
    return (\@v, \@f);
}
```

1. **Indices are 1-based.** Off-by-one here is the most common OBJ bug.
2. **A negative index is relative**, counting back from the vertices read so far, which is how generators emit streaming output.
3. **A face can have any number of vertices.** Quads are common and n-gons occur; both need triangulating. Fan triangulation is correct for convex faces only, which in practice OBJ faces are.
4. **A face token has up to three slots**, `v/vt/vn`, and `vt` may be empty as `v//vn`. Splitting on `/` and taking the first field is enough when only positions are wanted.

Round-tripping the cube reads back 8 vertices and 12 triangles, and a single 4-vertex face fans correctly:

```
read 8 vertices, 12 triangles
one 4-vertex face -> 2 triangles: [0 1 2] [0 2 3]
```

Symptom: a loaded model is missing every second triangle, or shows stray triangles connecting distant parts. Cause: n-gon faces triangulated as if they were triangles, or the 1-based offset applied twice.

Materials live in a sibling `.mtl` file named by a `mtllib` line, with `usemtl` switching material between faces. Colours, texture map filenames and shininess go there; geometry never does.

STL

The 3D-printing format. Binary STL is a fixed-size record layout, which makes it the clearest `pack` exercise in this guide:

```
sub write_stl {
    my ($path, $v, $i) = @_;
    open my $fh, '>', $path or die "$path: $!";
    binmode $fh;                                # REQUIRED: binary␣
↪format
    print $fh "\0" x 80;                        # 80-byte header,␣
↪ignored
    print $fh pack "V", @$i / 3;                # uint32 triangle␣
↪count
    for (my $k = 0; $k < @$i; $k += 3) {
        my ($a, $b, $c) = @{$v}[ @{$i}[$k, $k+1, $k+2] ];
        my $n = v_norm(v_cross(v_sub($b,$a), v_sub($c,$a)));
        # 12 little-endian floats then a uint16: 50 bytes per␣
↪triangle.
        print $fh pack "f<12 v", @$n, @$a, @$b, @$c, 0;
    }
```

(continues on next page)

(continued from previous page)

```
    close $fh;
}
```

The size is completely determined: 80 header bytes, 4 count bytes, then 50 per triangle.

```
file size: 684 bytes (84 + 50 * 12 = 684)
```

Reading is the same layout in reverse, and the result shows STL's defining property:

```
read 12 triangles
vertex records: 36, distinct positions: 8
```

STL has no index buffer. Every triangle carries its three positions in full, so a cube's 8 corners arrive as 36 vertex records. Weld them (the *meshes chapter* has the code) before doing anything that needs adjacency, or the manifold check will report every edge as used once.

Four more facts worth having:

- The format is **little-endian** regardless of host.
- The per-triangle **normal is frequently zero or wrong**. Recompute it from the winding rather than trusting it.
- The trailing **uint16 is nominally an attribute byte count** and is normally 0. Some tools abuse it to store colour; there is no standard.
- An **ASCII variant** exists (`solid`/`facet normal`/`outer loop`), equivalent in content and larger on disk. How much larger depends entirely on coordinate precision: this cube, whose coordinates are single digits, comes out 1471 bytes against 684, a factor of 2.2, while a scanned model with six significant figures per axis runs far higher. Detect the variant by checking whether the file begins with `solid` **and** its size does not match `84 + 50 *` n; testing the first five bytes alone is not enough, because a binary file's 80-byte header can begin with them too.

PLY

Header-driven: the header declares the elements and their properties, and the body follows that declaration.

```
ply
format ascii 1.0
element vertex 8
property float x
property float y
property float z
element face 12
property list uchar int vertex_indices
end_header
```

The parser reads the header, builds a record description from it, then reads that many records. That indirection is the point: PLY carries arbitrary per-vertex properties, so

scanner output with confidence values, colours and normals per point needs no format extension. Both ASCII and binary bodies are declared by the `format` line.

glTF

The modern interchange format, and the one to target for anything with structure. A `.gltf` is JSON describing nodes, meshes, materials, cameras and animations, referring to binary buffers for the bulk data; a `.glb` packs the same JSON and buffers into one file.

Reading it fully is a project rather than a page, but the shape is worth knowing because it explains the design: `accessors` describe typed views into `bufferViews`, which are ranges of `buffers`. Vertex positions, indices and animation keyframes are all accessors over the same byte range mechanism, so a loader implements the accessor logic once.

glTF is where PBR material parameters, a scene hierarchy and skinned animation actually live. OBJ and STL carry none of those, and no amount of convention makes them.

Choosing

- Writing geometry for another tool to read: **OBJ**, unless it needs materials or a hierarchy.
- Sending a solid to a printer or a slicer: **STL**, welded on the way back in.
- Point clouds or per-vertex measured data: **PLY**.
- A scene, materials, or animation: **glTF**, and nothing else.

Next

Geometry can now be stored and loaded. The *next chapter* constructs it: extrude, revolve, subdivision and boolean operations.

Modelling operations: extrude, revolve, subdivide, combine

Four operations build most geometry that is not scanned or sculpted. Each takes a mesh or a profile and produces a mesh, and each has an analytic answer you can check it against, which is what this chapter does throughout.

Checking a modelling operation

Before the operations themselves: a modelling operation that produces a closed solid can be verified. The `signed_volume` from the *meshes chapter* returns the enclosed volume, and most primitive shapes have a known one. A wrong answer means wrong winding, a hole, or wrong geometry, and the three are distinguishable.

Treat it as the unit test for every operation below. It catches the faults that are invisible in a render until you rotate the camera.

Extrude

Take a closed polygon, sweep it along a direction, cap both ends. This is the push/pull of a direct-modelling tool and the single most useful operation in the set.

```
# $poly is a closed polygon in the xy plane, counter-clockwise seen
# from +z. Extrudes along +z by $h. Returns ($verts, $indices).
sub extrude {
    my ($poly, $h) = @_;
    my $n = @$poly;
    # The outer parens are load-bearing: `my @v = (A), (B)` parses
↪as
    # `(my @v = A), B` and silently stores only the first list.
    my @v = ( (map { [ $_->[0], $_->[1], 0  ] } @$poly),
              (map { [ $_->[0], $_->[1], $h ] } @$poly) );
    my @i;
    # top cap: fan over the upper ring, already CCW seen from +z
    push @i, ($n, $n + $_, $n + $_ + 1) for 1 .. $n - 2;
    # bottom cap: same fan reversed, so it faces -z
    push @i, (0, $_ + 1, $_) for 1 .. $n - 2;
    # sides: one quad per edge, wound outward
    for my $e (0 .. $n - 1) {
        my $f = ($e + 1) % $n;
        push @i, ($e, $f, $n + $f), ($e, $n + $f, $n + $e);
    }
    return (\@v, \@i);
}
```

An n-gon extrudes to 2n vertices and 4n - 4 triangles: n - 2 per cap and two per side quad.

```
unit square extruded by 2: 8 vertices, 12 triangles
signed volume = 2.0000  (analytic: area 1 * height 2 = 2)
unit hexagon extruded by 1: 20 triangles, volume 2.5981 (analytic 2.
↪5981)
```

Extruding a unit square produces exactly the cube from the meshes chapter: 8 vertices, 12 triangles. The hexagon lands on `3*sqrt(3)/2` to four decimal places.

Fan-capping is only valid for a **convex** polygon. A concave outline needs ear clipping, and a polygon with holes needs a real triangulator; both produce caps that self-overlap if fanned.

Symptom: an extruded shape looks right from outside but its volume test returns a small or negative number. Cause: cap winding, usually the bottom cap not reversed.

Revolve

Sweep a profile around an axis. The profile is a list of `[radius, y]` points; each is rotated into `seg` positions around the Y axis and consecutive rings are joined.

```
for my $k (0 .. $p - 2) {
    my ($q0, $q1) = ($s * $p + $k,     $t * $p + $k);
    my ($q2, $q3) = ($t * $p + $k + 1, $s * $p + $k + 1);
    # wound so the normal points away from the axis
    push @i, ($q0, $q3, $q2), ($q0, $q2, $q1);
}
```

Unlike extrude, revolve only **approximates** its analytic shape, and the segment count is the knob. A cylinder of radius 1 and height 2:

```
cylinder r=1 h=2, analytic volume 6.2832
    8 segments:   32 triangles, volume 5.6569
   16 segments:   64 triangles, volume 6.1229
   32 segments:  128 triangles, volume 6.2429
   64 segments:  256 triangles, volume 6.2731
  256 segments: 1024 triangles, volume 6.2826
```

The volume approaches `pi * r^2 * h` from **below**, because an inscribed prism is contained by the cylinder it approximates. Every doubling of segments cuts the error by roughly four, so accuracy is cheap in geometry: halving the error takes only about 41 percent more segments. 32 segments is within 0.7 percent and is the usual default in modelling tools for a reason.

A profile that reaches the axis (`radius = 0`) needs no cap at that end, and joining a full ring to a single axis point there produces degenerate zero-area triangles unless it is special-cased.

Subdivision

Split each triangle into four by its edge midpoints. On its own this adds geometry without adding shape; the operation becomes useful when the new vertices are **moved**, which is what a subdivision surface does. Projecting them onto a sphere is the clearest case:

```
my $ab = $midpoint->($a, $b);
my $bc = $midpoint->($b, $c);
my $ca = $midpoint->($c, $a);
push @ni, ($a,$ab,$ca), ($ab,$b,$bc), ($ca,$bc,$c), ($ab,$bc,$ca);
```

The midpoint helper must **cache by edge**, or the two triangles sharing an edge each create their own copy of the midpoint and the mesh comes apart into unwelded fragments.

Starting from an octahedron and normalising every new vertex:

```
unit sphere, analytic volume 4.18879
  level 0:    8 triangles,    6 vertices, volume 1.33333
  level 1:   32 triangles,   18 vertices, volume 2.94281
  level 2:  128 triangles,   66 vertices, volume 3.81773
  level 3:  512 triangles,  258 vertices, volume 4.09160
  level 4: 2048 triangles, 1026 vertices, volume 4.16420
```

Triangles multiply by four per level. Vertices do not: each level yields `4V - 6`, which the table confirms at every step (18, 66, 258, 1026), because the midpoints on shared edges

are created once and used twice. The volume converges from below again, and level 4 is within 0.6 percent at 2048 triangles.

This is the simplest subdivision scheme. **Catmull-Clark**, the one modelling tools use, works on quads, weights each new vertex by its neighbours instead of projecting onto a known surface, and converges to a smooth limit surface with no target shape needed. The mechanism here is the same: insert vertices, then move them.

Boolean operations

Union, difference and intersection of two solids. This is the operation users reach for most and the one that is hardest to implement, so the honest position matters: **this chapter does not implement it.** A correct constructive solid geometry (CSG) implementation is a project, not a page.

The reason is that it is not really a geometry problem, it is a robustness problem. The algorithm is:

1. Split every triangle of A against the planes of B, and the reverse.
2. Classify each resulting fragment as inside, outside, or on the boundary of the other solid.
3. Keep the fragments the operation calls for, and flip the winding of B's kept fragments for a difference.

Step 1 in floating point is where implementations fail. A vertex landing exactly on a plane, two coplanar faces, a nearly-degenerate sliver triangle: each needs an explicit decision, and getting one wrong yields a mesh with a hole rather than an error.

The classification in step 2 rests on one primitive that **is** worth having on its own, since picking and collision need it too: is a point inside a closed mesh? Cast a ray and count crossings. Odd means inside.

```
# FRAGMENT: the ray/triangle body is elided. Requires a CLOSED,␣
 ↪manifold
# mesh. The ray direction is nudged off axis because a ray hitting␣
 ↪an
# edge or vertex exactly is counted twice or not at all, and nudging␣
 ↪is
# cheaper than special-casing every degenerate hit.
sub point_inside {
    my ($v, $i, $p) = @_;
    my $hits = 0;
    my $dir = v_norm([1, 0.0001, 0.0002]);
    for (my $k = 0; $k < @$i; $k += 3) {
        # ... Möller-Trumbore ray/triangle test, yielding $t ...
        $hits++ if $t > 1e-9;          # count forward hits only
    }
    return $hits % 2;
}
```

```
  [0,0,0         ] inside
  [0.9,0.9,0.9   ] inside
```

(continues on next page)

(continued from previous page)

```
  [1.1,0,0       ] outside
  [2,2,2         ] outside
  [-0.5,0.5,0    ] inside
```

The Möller-Trumbore test in the loop returns the barycentric coordinates as a side effect of solving for the intersection, which is why the same routine serves picking in the scene-graph chapter.

For real boolean work, model as a **tree of operations** rather than committing to a result mesh: keep the primitives and the operations that combine them, and evaluate to triangles only for display or export. That is what CAD tools store, and it keeps the operations reversible.

Next

Objects can now be built. The *next chapter* arranges them: transform hierarchies, cameras, picking, and the spatial structures that keep scenes fast.

Scene graphs: hierarchy, cameras, picking

A scene is more than a pile of meshes. Objects are attached to other objects, a camera has to be steered, the user clicks on something and expects the right thing selected, and none of it may cost a pass over every triangle. This chapter covers those four.

The hierarchy

Each node holds a **local** transform, relative to its parent, and a list of children. Its **world** transform is the parent's world times its own local, which means one top-down walk computes the whole scene.

```
sub walk {
    # $parent_world is the identity at the root. Multiplication
↪order
    # is parent * local: the local transform applies FIRST, then the
    # parent's, which is the right-to-left rule from foundations.
    my ($node, $parent_world, $out) = @_;
    my $world = m_mul($parent_world, $node->{local});
    $out->{ $node->{name} } = $world;
    walk($_, $world, $out) for @{ $node->{children} };
    return $out;
}
```

A turntable yawed 90 degrees, an arm translated 4 along its own x, and a hand raised 2 above the arm:

```
turntable  origin at [ 0.00,  0.00,  0.00]
arm        origin at [ 0.00,  0.00, -4.00]
hand       origin at [ 0.00,  2.00, -4.00]
```

The arm was placed 4 along **its own** x, and the turntable's yaw put it on world -z. The hand rose 2 in y, which the yaw leaves alone. That is the entire value of a hierarchy: the arm's author never thinks about the turntable's angle.

Symptom: rotating a parent makes a child orbit when it should spin in place, or vice versa. Cause: the multiplication order at this line. It is the *foundations* order rule appearing where it costs the most.

Recomputing the whole tree every frame is normal and cheap. Caching world transforms is an optimisation that needs invalidation on every parent change, and getting that wrong is a much worse bug than the recompute is a cost.

The orbit camera

Two angles and a distance. This is the camera every modelling tool uses, and the one the *modeller capstone* needs.

```
sub orbit_eye {
    # $yaw and $pitch in radians, $dist from the target. Clamp pitch
    # short of +/- 90 degrees: at exactly vertical the view␣
↪direction
    # is parallel to `up` and look_at degenerates.
    my ($target, $yaw, $pitch, $dist) = @_;
    return [ $target->[0] + $dist * cos($pitch) * sin($yaw),
             $target->[1] + $dist * sin($pitch),
             $target->[2] + $dist * cos($pitch) * cos($yaw) ];
}
```

The camera has an invariant worth testing against, because it holds at every angle: the target must land on the view-space -z axis at exactly the orbit distance.

```
yaw   0: eye [ 0.00,  3.42,  9.40]  target in view space [ 0.00,  0.
↪00, -10.00]
yaw  45: eye [ 6.64,  3.42,  6.64]  target in view space [ 0.00, -0.
↪00, -10.00]
yaw  90: eye [ 9.40,  3.42,  0.00]  target in view space [ 0.00,  0.
↪00, -10.00]
```

The eye swings around, the target stays put. Any camera bug shows up here immediately as a target that drifts off the axis or changes depth.

A **fly camera** is the other common kind: position plus a forward direction, moved with the keyboard and turned with the mouse. It stores orientation rather than deriving it, which is where the *quaternion* advice applies, since accumulating Euler angles across thousands of frames drifts and eventually gimbal-locks.

Picking

The user clicks a pixel and means an object. Turning the first into the second is a **ray** through that pixel, then an intersection test.

The ray could be built by inverting the view-projection matrix, but for a camera it is cheaper to construct it directly from the camera axes: the pixel's normalised device coordinates scale the right and up vectors, and forward supplies the rest.

```
sub pick_ray {
    my ($eye, $target, $up, $fovy, $aspect, $sx, $sy, $W, $H) = @_;
    my $f = v_norm(v_sub($target, $eye));
    my $s = v_norm(v_cross($f, $up));
    my $u = v_cross($s, $f);
    my $ndc_x = 2 * ($sx + 0.5) / $W - 1;
    my $ndc_y = 1 - 2 * ($sy + 0.5) / $H;        # screen y is down
    my $th = tan_half($fovy);
    my @d = map {
        $f->[$_] + $s->[$_] * $ndc_x * $th * $aspect + $u->[$_] *
↪$ndc_y * $th
    } 0..2;
    return ($eye, v_norm(\@d));
}
```

The `+ 0.5` samples the pixel centre and the `1 -` flips screen y, both for the same reasons as in the *rasteriser*. The aspect multiplies x because the field of view argument is vertical.

```
centre pixel ray dir [ 0.00, -0.00, -1.00]
right-of-centre  dir [ 0.50, -0.00, -0.87]
unit sphere at origin: hit at t=9.000 (eye is 10 away, r=1)
same sphere from the off-centre ray: miss
```

The centre pixel gives exactly `-z`, and a unit sphere at the origin is hit at `t = 9` from an eye 10 away. Both are checkable by hand, which is what makes them useful tests.

Test against **bounding volumes first**, and only then against triangles. A sphere test is a handful of operations:

```
sub ray_sphere {
    # Returns the near intersection distance, or undef. $d MUST be
    # normalised or $t is in units of $d's length rather than␣
↪distance.
    my ($o, $d, $c, $r) = @_;
    my $oc = v_sub($o, $c);
    my $b = 2 * v_dot($oc, $d);
    my $cc = v_dot($oc, $oc) - $r * $r;
    my $disc = $b*$b - 4*$cc;
    return undef if $disc < 0;
    my $t = (-$b - sqrt($disc)) / 2;
    return $t > 0 ? $t : undef;
}
```

The `$t > 0` check rejects hits behind the eye. Without it, clicking empty space selects whatever is behind the camera.

The bounding volume hierarchy

Testing every object per ray is fine for ten objects and hopeless for ten thousand. A **BVH** is a binary tree of nested boxes: if the ray misses a node's box, nothing inside it needs testing.

Building it is a recursive median split along the widest axis:

```
sub build_bvh {
    my ($items) = @_;
    my ($lo, $hi) = bounds($items);
    return { lo => $lo, hi => $hi, items => $items } if @$items <=
 ↪4;
    my $ax = 0;
    my $best = $hi->[0] - $lo->[0];
    for my $a (1, 2) {
        my $e = $hi->[$a] - $lo->[$a];
        if ($e > $best) { $best = $e; $ax = $a }
    }
    my @sorted = sort { $a->{c}[$ax] <=> $b->{c}[$ax] } @$items;
    my $mid = int(@sorted / 2);
    return { lo => $lo, hi => $hi,
             left  => build_bvh([ @sorted[0 .. $mid-1] ]),
             right => build_bvh([ @sorted[$mid .. $#sorted] ]) };
}
```

Splitting the **widest** axis is what keeps the boxes compact; always splitting x produces long thin nodes that overlap and defeat the pruning. The leaf size of 4 is a tuning constant: smaller leaves prune better and cost more nodes to traverse.

The ray/box test is the slab method, intersecting the ray with each pair of parallel planes and keeping the overlap:

```
my $inv = 1 / $d->[$ax];
my ($t1, $t2) = (($lo->[$ax]-$o->[$ax])*$inv, ($hi->[$ax]-$o->[
 ↪$ax])*$inv);
($t1, $t2) = ($t2, $t1) if $t1 > $t2;
$tmin = $t1 if $t1 > $tmin;
$tmax = $t2 if $t2 < $tmax;
return 0 if $tmin > $tmax;
```

A zero component in the direction needs handling before the division, or the reciprocal is an infinity and the comparisons stop meaning what they should.

Over 512 objects on a grid, with one ray:

```
512 objects, 255 BVH nodes
BVH:    hit object 0 at t=9.00, 39 nodes visited
linear: hit object 0 at t=9.00, 512 objects tested
same answer: yes
```

Thirty-nine node visits against 512 object tests, for the same answer. The 255 nodes are 128 leaves of four objects each plus 127 internal nodes, which is where the memory goes.

The same structure serves collision detection and ray tracing. In a renderer its sibling is **frustum culling**: test each node's box against the six planes of the view volume and skip whole subtrees that fall outside, which is the same pruning with a different predicate.

Next

This closes the geometry half of the guide. Everything so far has been plain Perl, computing pixels or coordinates in software.

The *next arc* hands the work to the graphics card, through `Peta::WebGPU`.

The GPU: a device, a window, and the frame loop

Everything so far computed pixels in Perl. From here the graphics card computes them. pperl's GPU binding is `Peta::WebGPU`: WebGPU as implemented by the wgpu library, which drives Vulkan, Metal, Direct3D 12 or OpenGL underneath, whichever the machine has. The programs that run on the card are written in WGSL, WebGPU's shading language. The *world generator* used the binding's compute half; this arc uses the half that draws.

The window, the events and the clock come from the native `SDL2` binding, as in the *2D guide's toolkit chapter*. The two meet at exactly one point: wgpu builds its surface on the handles of the SDL2 window. There is no context to create and no swap call. This chapter covers what takes their place.

Is there a GPU at all?

Two different things can be missing, and they fail differently.

The pperl binary may be built without the GPU option. Loading the module then dies, naming the fix:

```
Peta::WebGPU requires the 'webgpu' capability, which this pperl␣
 ↪binary
was built without. Rebuild with: cargo build --features webgpu
```

Or the build has it and the machine has no usable adapter, as in a container with no GPU driver. `have_gpu` answers that without dying, so one line covers both cases and a program can fall back to the *software rasteriser*:

```
my $gpu_ok = eval { require Peta::WebGPU; 1 } && Peta::WebGPU::have_
 ↪gpu();
```

`adapters` lists what the machine offers:

```
for my $a (Peta::WebGPU::adapters()) {
    printf "%-8s %-14s %s\n", $a->{backend}, $a->{device_type}, $a->
 ↪{name};
}
```

```
Vulkan   IntegratedGpu  Intel(R) HD Graphics P530 (SKL GT2)
Vulkan   DiscreteGpu    Quadro M2000M
Gl       Other          Quadro M2000M/PCIe/SSE2
```

One card appears twice, once per backend that can reach it. The list is for reporting; `Peta::WebGPU::Device->new` makes the choice, steered by its `power` option. On this

machine `low-power` opens the Intel chip and `high-performance` opens the Quadro. `$gpu->info` says which one you got, with the same keys.

Drawing without a window

Start with the render target that needs no display at all, because it is how every example in this arc is checked, and how you check yours.

```
use strict;
use warnings;
use Peta::WebGPU;

my $gpu = Peta::WebGPU::Device->new;           # no window
my ($W, $H) = (320, 240);

my $img = $gpu->texture(width => $W, height => $H);
$gpu->render_pass($img, clear => [0.2, 0.4, 0.6, 1])->end;

my $rgba = $img->read;                         # $W * $H * 4 bytes
sub pixel {
    my ($x, $y) = @_;
    return unpack 'C4', substr($rgba, ($y * $W + $x) * 4, 4);
}
printf "%d bytes, centre pixel %s\n", length $rgba, join ' ', pixel(
 ↪$W / 2, $H / 2);
# 307200 bytes, centre pixel 51 102 153 255
```

`texture` makes an image on the card; its format defaults to `rgba8unorm`, four bytes per pixel. `render_pass` starts recording into it, clearing to the given colour, components from 0 to 1. **Nothing happens until end**: a pass records, and end hands the recording to the card. `read` waits for the card to finish and copies the image back, rows top first, bytes in R, G, B, A order: `0.2, 0.4, 0.6` arrive as `51 102 153`.

Reading crosses the bus and waits, so it belongs at the end of a job, not inside a loop.

The same program run with `DISPLAY` and `WAYLAND_DISPLAY` removed from the environment prints the same line. It needs no display server, no window manager and no compositor, so it runs on a server or in a test suite.

To look at the result, write it out as a PNG through `GD`, which the *2D guide* covers:

```
use GD;

sub save_png {
    my ($file, $rgba, $w, $h) = @_;
    my $im = GD::Image->new($w, $h, 1);        # truecolor
    for my $y (0 .. $h - 1) {
        for my $x (0 .. $w - 1) {
            my ($r, $g, $b) = unpack 'C3', substr($rgba, ($y * $w +
 ↪$x) * 4, 3);
            $im->setPixel($x, $y, $im->colorAllocate($r, $g, $b));
        }
```

(continues on next page)

(continued from previous page)

```
    }
    open my $fh, '>', $file or die "$file: $!\n";
    binmode $fh;
    print $fh $im->png;
    close $fh;
}

save_png('clear.png', $rgba, $W, $H);
```

A loop over every pixel is slow in Perl, but this is a checking tool, run once, and it keeps the picture exactly as the card produced it.

A window, and a device opened for it

```
use strict;
use warnings;
use SDL2 qw(:all);
use Peta::WebGPU;

my $win  = SDL2::Window->new(title => 'pperl gpu', w => 800, h =>␣
↪600, resizable => 1);
my $gpu  = Peta::WebGPU::Device->new(window => $win);
my $surf = $gpu->surface;
```

The window goes to `Device->new`, not to a later call. The adapter has to be one that can present to that window, and wgpu can only answer that question for a real surface, so the device is chosen with the window in hand. A machine with two cards could otherwise open the one that does not drive the screen.

`Peta::WebGPU` opens no windows itself. It asks the window object two questions, `$win->wm_info` for the window system's own handles and `$win->size`, and draws on anything that answers them. `SDL2::Window` does; under X11 its `wm_info` carries `driver`, `display` and `window`, under Wayland `display` and `surface`. Those two window systems are the ones a surface can be made on.

`$gpu->surface` is the window's chain of images to draw into, configured at the window's size. On the machine these examples were checked on:

```
print $surf->format, ' ', join('x', $surf->size), "\n";
# bgra8unorm-srgb 800x600
```

The format is the compositor's preference. `surface(format => ...)` can ask for another the window offers, and one it does not offer is refused with the list of those it does. Whichever it is, anything that draws into the window has to be built for it, which the *next chapter* returns to.

The import trap

This one costs an afternoon, and it produces a wrong result rather than an error.

```
use SDL2;                       # no import list
SDL2->import(qw(:all));         # WRONG: runs at runtime

my $win = SDL2::Window->new(title => 'trap', w => 160, h => 120);
SDL_PushEvent({ type => SDL_QUIT() });   # parentheses: resolved at␣
↪runtime

my $quit = 0;
while (my $ev = SDL_PollEvent()) {
    $quit = 1 if $ev->{type} == SDL_QUIT;
}
print "SDL_QUIT is: ", SDL_QUIT, "\n";
print "quit noticed: ", ($quit ? "yes" : "no"), "\n";
```

`use SDL2 qw(:all)` imports at **compile time**, so the constants exist before the rest of the file is parsed. A separate `SDL2->import` runs after the whole file has been compiled, so while it was being compiled `SDL_QUIT` without parentheses was not a known function. It parsed as a bareword and became the string `"SDL_QUIT"`:

```
SDL_QUIT is: SDL_QUIT
quit noticed: no
```

The quit event was on the queue and the comparison, against a string that numifies to 0, never matched it. In a real program the window's close button does nothing. `use warnings` adds one line, `Argument "SDL_QUIT" isn't numeric in numeric eq (==)`, and the program still runs on.

`use strict` converts this from a silent wrong answer into a compile error naming the constant. The same file with `use strict;` as its first line:

```
Bareword "SDL_QUIT" not allowed while "strict subs" in use at trap.
↪pl line 9.
Bareword "SDL_QUIT" not allowed while "strict subs" in use at trap.
↪pl line 12.
Execution of trap.pl aborted due to compilation errors.
```

Line 9 is the statement modifier inside the `while` body, reported at the line where that statement starts, as perl5 reports it.

That is the whole argument for `use strict` in one example. Write `use SDL2 qw(:all);` and keep `use strict` on.

The frame loop

A frame is three steps: take an image from the surface, draw into it, present it.

```
my $frame = $surf->next_frame or next;          # no image this time:␣
↪skip
```

(continues on next page)

(continued from previous page)

```
$gpu->render_pass($frame, clear => [0.2, 0.3, 0.5, 1])->end;
$frame->present;
```

A frame is a render target exactly as a texture is: the same `render_pass`, the same recording calls. That is what lets one drawing routine serve both the screen and the checking path above.

`next_frame` returns undef rather than dying when wgpu reports the surface occluded or timed out. That is not an error; skip the frame and come back. When wgpu reports the surface out of date, `next_frame` configures it again with the size it was last given and retries once, which is why a missed resize, below, produces no error at all.

End the pass before presenting. Presenting first is accepted, and the `end` that follows dies, because the image it recorded into is gone:

```
Peta::WebGPU::Pass::end: Validation Error

Caused by:
  In Queue::submit
    In a pass parameter
      Texture with '<Surface Texture>' label has been destroyed
```

Presenting the same frame twice is refused with `this frame has already been presented`.

Pacing

A surface's present mode decides how a frame reaches the screen. The default is `fifo`: it queues each frame for the display's next refresh and never tears, so the loop runs at the display's rate with no sleep anywhere in the program. It is also the one mode WebGPU guarantees every surface supports, so the default never fails. Measured over 120 frames on a 60 Hz display, a surface made as above averaged 16.68 ms per frame, and one made with `present_mode => 'fifo'` spelled out 16.67 ms.

Another mode is asked for by name when the surface is made:

```
my $surf = $gpu->surface(present_mode => 'mailbox');
```

`mailbox` keeps only the newest frame and `immediate` does not wait at all, which can tear. Both returned as fast as the loop could go, under a millisecond per frame. `fifo-relaxed` waits like `fifo` but lets a late frame through at once, at the risk of a tear. A mode the window does not offer is refused, with the ones it does offer in the message.

Colour arrives in sRGB

The surface format above ends in `-srgb`, and that suffix changes the numbers. An `-srgb` target treats what is written to it as linear light and encodes it on the way in, the encoding image files and displays use. The same clear into two formats:

```
format            clear [0.2, 0.4, 0.6]   stored as
rgba8unorm        written as given       51 102 153 255
rgba8unorm-srgb   encoded                124 170 203 255
```

Lighting arithmetic is only correct in linear light, so rendering into an `-srgb` target is the right thing. The consequence is for colours you pick by eye, from an image editor or a web page: those are already encoded, and fed straight in they come out washed out. Decode them first. Raising each component to the power 2.2 is close enough for picking colours: the clear `[ map { $_ ** 2.2 } 0.2, 0.4, 0.6 ]` stores as `48 102 154`.

For the checking path, render into `rgba8unorm-srgb` whenever the image should match what the window shows. `bgra8unorm-srgb` would match too, but `read` returns its bytes in B, G, R, A order.

Window events

The event loop from the *2D guide* is unchanged. Three events matter from the first program on:

```
while (my $ev = SDL_PollEvent()) {
    if ($ev->{type} == SDL_QUIT) { $running = 0 }
    elsif ($ev->{type} == SDL_KEYDOWN && $ev->{sym} == SDLK_ESCAPE)
 ↪{ $running = 0 }
    elsif ($ev->{type} == SDL_WINDOWEVENT
           && $ev->{event} == SDL_WINDOWEVENT_SIZE_CHANGED) {
        $surf->configure($ev->{data1}, $ev->{data2});
    }
}
```

Symptom: after a resize the picture is the wrong size, and nothing reported an error. Cause: the surface was never reconfigured. A window resized from 320x240 to 400x300 kept a 320x240 surface, every frame presented without complaint, and `$surf->size` said `320x240` against the window's `400x300`.

Key the reconfigure on `SDL_WINDOWEVENT_SIZE_CHANGED`, not `SDL_WINDOWEVENT_RESIZED`. A resize made by the program with `$win->set_size(400, 300)` delivered `SIZE_CHANGED` with `data1 = 400`, `data2 = 300`, and no `RESIZED` at all.

`SDL_PushEvent` puts an event on the queue exactly as if a person had caused it, which is how a test drives a program that normally waits for input. Pushed and read back, an Escape key press arrives as `sym = 27, scancode = 41, key_name = Escape`, and a mouse drag as `SDL_MOUSEMOTION` with `x`, `y`, the movement since the last event in `xrel`, `yrel`, and the held buttons in `state`, which is what an orbit camera reads.

Timing: a fixed step

`fifo` paces the frames, but a simulation must not depend on the frame rate: a faster display would make everything move faster. Accumulate the real elapsed time and advance the simulation in fixed steps:

```
use constant { STEP => 16, MAX_STEPS => 5 };   # milliseconds, steps
 ↪per frame
```

(continues on next page)

(continued from previous page)

```
my $now = SDL_GetTicks();
$acc += $now - $last;
$last = $now;
my $steps = 0;
while ($acc >= STEP) {
    update();
    $acc -= STEP;
    if (++$steps >= MAX_STEPS) { $acc = 0; last }
}
```

In a window paced by `fifo`, this ran 205 updates in 3307 ms of real time, and 3307 / 16 is 206.7: the simulation keeps to the clock whatever the frame rate.

The step cap is not optional, and neither is dropping the backlog when the cap is hit. What each part prevents is clearest on a simulated clock, where every frame costs 6 ms of rendering plus the time its updates take. The figures are updates run per frame, in consecutive frames:

```
each update costs 20 ms, more than the 16 ms it simulates:
  no cap              7 9 12 15 19 24 31 39 49 62 77 97 122 153 ...
  cap, keep backlog   6 6 6 6 ...   owed 22, 52, 82 ... 412 ms, and␣
↪growing
  cap, drop backlog   6 6 6 6 ...   owed 0

a 500 ms stall, then updates costing 12 ms:
  no cap              33 25 19 15 11 9 7 6 5 4 3 3 2 ...
  cap, keep backlog   6 6 6 6 ...   owed 436 ms, still 220 ms 12␣
↪frames later
  cap, drop backlog   6 4 4 3 3 2 2 2 2 ...
```

Without a cap, an update slower than its step is a **spiral**: each frame takes longer, owes more steps, and takes longer still, until the program stops responding. A stall on its own is survivable when updates are cheaper than their step, but the first frame after it runs 33 updates, which is a second freeze of almost 400 ms. With a cap but keeping the backlog, frame time is bounded but the simulation fast-forwards for seconds afterwards, working off time that has already passed. Dropping the backlog is the honest answer: after a stall, a breakpoint or a window drag, the simulation loses that time and carries on at normal speed. When every update is slower than its step, it runs slower than real time, which is the only possible outcome, and the window keeps responding.

The whole program

Put together, with an optional frame count so it can run unattended:

```
use strict;
use warnings;
use SDL2 qw(:all);
use Peta::WebGPU;
```

(continues on next page)

(continued from previous page)

```
use constant { STEP => 16, MAX_STEPS => 5 };

my $max_frames = shift // 0;          # 0: run until the window is
↪closed

my $win  = SDL2::Window->new(title => 'pperl gpu', w => 800, h =>
↪600, resizable => 1);
my $gpu  = Peta::WebGPU::Device->new(window => $win);
my $surf = $gpu->surface;
printf "%s via %s, surface %s %s\n", $gpu->info->{name}, $gpu->info-
↪>{backend},
    $surf->format, join('x', $surf->size);

my $phase = 0;                        # the whole simulation state
sub update { $phase += 0.001 * STEP }  # radians per step: one cycle
↪in 6.3 s

sub colour {
    return [ 0.2 + 0.2 * sin($phase), 0.3, 0.5 + 0.2 * cos($phase),
↪1 ];
}

my ($last, $acc, $frames, $running) = (SDL_GetTicks(), 0, 0, 1);
while ($running) {
    while (my $ev = SDL_PollEvent()) {
        if ($ev->{type} == SDL_QUIT) { $running = 0 }
        elsif ($ev->{type} == SDL_KEYDOWN && $ev->{sym} == SDLK_
↪ESCAPE) { $running = 0 }
        elsif ($ev->{type} == SDL_WINDOWEVENT
               && $ev->{event} == SDL_WINDOWEVENT_SIZE_CHANGED) {
            $surf->configure($ev->{data1}, $ev->{data2});
        }
    }

    my $now = SDL_GetTicks();
    $acc += $now - $last;
    $last = $now;
    my $steps = 0;
    while ($acc >= STEP) {
        update();
        $acc -= STEP;
        if (++$steps >= MAX_STEPS) { $acc = 0; last }
    }

    my $frame = $surf->next_frame or next;    # no image this time:
↪skip
    $gpu->render_pass($frame, clear => colour())->end;
    $frame->present;
```

(continues on next page)

(continued from previous page)

```
    $running = 0 if $max_frames && ++$frames >= $max_frames;
}
printf "%d frames, phase %.2f\n", $frames, $phase;
```

```
Intel(R) HD Graphics P530 (SKL GT2) via Vulkan, surface bgra8unorm-
↪srgb 800x600
120 frames, phase 1.94
```

A window whose colour drifts slowly, closing on Escape or the close button. 1.94 radians is 121 steps of 0.016, which is 1936 ms: two seconds of clock for 120 frames at 60 Hz. The simulation followed the clock, not the frame count.

Next

The loop runs and the checking path works. The *next chapter* puts geometry through it: a shader, a vertex buffer, a pipeline and a draw call.

The pipeline: shaders, buffers, and the first triangle

The *software rasteriser* did three jobs per triangle: move the vertices to the screen, decide which pixels are covered, and choose each pixel's colour. On the graphics card the middle job is fixed hardware. The first and last are programs you write, the **vertex shader** and the **fragment shader**, and a **pipeline** binds them to a description of the data they are fed.

This chapter draws one triangle, then the same data through an index buffer, and ends with what the card says when something is wrong.

The shader

Both programs live in one WGSL source:

```
struct VOut {
    @builtin(position) pos: vec4<f32>,
    @location(0) col: vec3<f32>,
};

@vertex
fn vs(@location(0) p: vec2<f32>, @location(1) c: vec3<f32>) -> VOut
↪{
    var o: VOut;
    o.pos = vec4<f32>(p, 0.0, 1.0);
    o.col = c;
    return o;
}

@fragment
fn fs(i: VOut) -> @location(0) vec4<f32> {
    return vec4<f32>(i.col, 1.0);
}
```

`vs` runs once per vertex. Its inputs carry `@location` numbers, which is how the vertex data below finds them, and its output must include a `@builtin(position)`: the vertex in *clip space*, `x, y, z, w`. With `z = 0` and `w = 1` the divide changes nothing, so `x` and `y` here are already normalised device coordinates, -1 to +1 with y up.

`fs` runs once per covered pixel. What it receives is `vs`'s output **interpolated** across the triangle with barycentric weights, exactly as the rasteriser chapter computed them by hand, and perspective-correct without being asked. Its `@location(0)` output is the colour written to the target.

WGSL is strictly typed: `vec2<f32>` is two 32-bit floats, `let` names a value, `var` a variable. There is no double precision; everything the card computes here is `f32`.

`$gpu->shader($source)` compiles it, and a mistake in it is a Perl exception carrying the compiler's message, shown below.

Vertex data

Three vertices, each a position and a colour, packed as 32-bit floats:

```
#                  x     y      r  g  b
my @tri = ( -0.8, -0.8,   1, 0, 0,
             0.8, -0.8,   0, 1, 0,
             0.0,  0.8,   0, 0, 1 );
my $verts = $gpu->vertex_buffer(pack 'f*', @tri);
print $verts->size;                              # 60
```

Fifteen floats of four bytes each. The buffer is bytes on the card and knows nothing of their meaning; the pipeline supplies that.

The pipeline

```
my $pipe = $gpu->render_pipeline($gpu->shader($WGSL),
    vertex => 'vs', fragment => 'fs',
    vertex_buffers => [ { attributes => ['float32x2', 'float32x3'] }␣
↪],
);
```

`vertex_buffers` describes each buffer as a list of attribute formats. The short form above lays them end to end: `float32x2` at `@location(0)`, offset 0; `float32x3` at `@location(1)`, offset 8; and a **stride** of 20 bytes from one vertex to the next. Written out in full, which is necessary only when the layout is not that simple:

```
vertex_buffers => [ {
    stride     => 20,
    attributes => [ { format => 'float32x2', location => 0, offset␣
↪=> 0 },
                    { format => 'float32x3', location => 1, offset␣
↪=> 8 } ],
} ],
```

The format names are WebGPU's own: `float32` to `float32x4`, the `uint32` and `sint32` families, `unorm8x4`, `float16x2`. A name that is not one of them is refused with the

whole list.

The pipeline also fixes the **format** of the target it draws into, `rgba8unorm` unless told otherwise. For a window that must be the surface's format:

```
my $pipe = $gpu->render_pipeline($shader, format => $surf->format, .
↪..);
```

Compiling a pipeline is the expensive step. Build it once, before the frame loop, and use it every frame.

The draw

```
my ($W, $H) = (256, 256);
my $img = $gpu->texture(width => $W, height => $H);
$gpu->render_pass($img, clear => [0.1, 0.1, 0.1, 1])
    ->pipeline($pipe)->vertices($verts)->draw(3)->end;
```

Every recording call returns the pass, so a frame reads as one sentence: this pipeline, these vertices, three of them, done. The order matters in one place only: `pipeline` comes first, because what follows is checked against it.

Read back, the pixels say what was drawn:

```
centre                  63 65 127 255
corner                  26 26 26 255
near bottom-left vertex  246 3 6 255
near bottom-right vertex 3 246 6 255
near top vertex          2 4 249 255
```

The corner is the clear colour, 0.1 as a byte. Each vertex's neighbourhood is almost pure red, green or blue, and the centre is a mix of all three, which is the interpolation at work. Written out with `save_png` from the *previous chapter*, it is the familiar colour-blended triangle, point upwards: NDC's y points up and the texture's top row comes first, the same flip *foundations* made in its screen mapping.

The same pixels as the software rasteriser

The rasteriser chapter filled a triangle with corners at pixel positions `(1,1)`, `(17,2)` and `(6,8)` on a 20 by 10 canvas and printed 54 pixels. Converted to NDC and drawn by the card into a 20 by 10 texture, printing # for every pixel that was written:

```
my @ndc = map { (2 * $_->[0] / 20 - 1, 1 - 2 * $_->[1] / 10) } [1,
↪1], [17,2], [6,8];
```

```
....................
.########...........
..##############....
...###########......
...#########........
....#######.........
```

(continues on next page)

(continued from previous page)

```
.....####..........
......#............
..................
..................
```

54 pixels, identical to the figure the edge function produced, and identical again with the vertex order reversed. The card is running the same inside test at the same pixel centres.

Index buffers

A quad is two triangles sharing an edge. Listing six vertices repeats two of them; an **index buffer** lists each vertex once and says which three make each triangle:

```
my $corners = $gpu->vertex_buffer(pack 'f*',
    -0.5, -0.5,   1, 0.8, 0.2,
     0.5, -0.5,   1, 0.8, 0.2,
     0.5,  0.5,   1, 0.8, 0.2,
    -0.5,  0.5,   1, 0.8, 0.2);
my $quad = $gpu->index_buffer(pack 'L*', 0, 1, 2,  0, 2, 3);

my $sq = $gpu->texture(width => 64, height => 64);
$gpu->render_pass($sq, clear => [0, 0, 0, 1])
    ->pipeline($pipe)->vertices($corners)->indices($quad)->draw_
↪indexed(6)->end;
```

Same pipeline, same vertex layout; only the drawing call changed.

The quad covers 1024 pixels of the 64 by 64 target, which is 32 by 32: half the width and half the height, exactly. `draw_indexed` counts indices, not vertices.

This is the *meshes* chapter's indexed triangle mesh, a vertex array and an index array, going to the card as one buffer each.

Indices are 32-bit by default. Sixteen-bit indices halve the buffer and must be declared, because the bytes alone cannot say which they are:

```
my $q16 = $gpu->index_buffer(pack 'S*', 0, 1, 2, 0, 2, 3);   # 12␣
↪bytes
$gpu->render_pass($sq, clear => [0, 0, 0, 1])
    ->pipeline($pipe)->vertices($corners)->indices($q16, 'uint16')->
↪draw_indexed(6)->end;
```

The same 12 bytes without the `'uint16'` read as three 32-bit indices, and the draw is refused at end: `Index 6 extends beyond limit 3. Did you bind the correct index buffer?`

When something is wrong

Every failure is a Perl exception carrying wgpu's own text, so `eval` traps it and `$@` explains it. A call that succeeds returns its object, which is what makes the chaining above work and `... ->end or die` behave. The exceptions arrive at three places, and knowing which saves time.

At `shader`: the WGSL does not compile. The message points into the source by line and column:

```
shader compilation failed: Validation Error

Caused by:
  In Device::create_shader_module

Shader '' parsing error: expected `;`, found "return"
   ┌─ wgsl:11:5
   │
11 │     return o;
   │     ^^^^^^ expected `;`
```

The missing semicolon was at the end of `o.col = c` on line 10; the parser noticed on line 11. Line numbers count from the first line of the WGSL string, not of the Perl file. A type error names the entry point, here `fs` returning a `vec3` where it declared a `vec4`:

```
Shader validation error: Entry point fs at Fragment is invalid
```

At `render_pipeline`: the shader does not fit the description. An entry point name that is not in the shader, or a shader input no buffer supplies:

```
Unable to find entry point 'vertex_main'
```

```
Location[1] Float32x3 interpolated as Some(Perspective) with␣
↪sampling
Some(Center) is not provided by the previous stage outputs
```

The second is what a `vertex_buffers` list with only `float32x2` in it produces for this shader: nothing feeds `@location(1)`.

At `end`: the recording was inconsistent. Mistakes made while recording are collected and reported when the pass is submitted, so the exception comes from `end` even though the call at fault was earlier. The message names the command:

mistake	the message ends with
draw with no vertices call	`requires vertex buffer 0 to be set`
draw(6) on a three-vertex buffer	`Vertex 6 extends beyond limit 3`
pipeline format differs from the target	`the RenderPass uses textures with formats [Some(Bgra8UnormSrgb)] but the RenderPipeline ... uses attachments with formats [Some(Rgba8Unorm)]`
a shader var<uniform> never bound	`expects a BindGroup to be set at index 0`

The format row is the one every windowed program hits once: a pipeline built with the default `rgba8unorm`, drawn into a frame whose surface is `bgra8unorm-srgb`.

The one that says nothing

Symptom: the triangle is the wrong shape, or its colours are wrong, and nothing raised an exception. Cause: a vertex format with fewer components than the shader's input. WebGPU allows it deliberately and fills in the missing components, so the layout `['float32x2', 'float32x2']` for this shader's `vec2` and `vec3` inputs is accepted. Its computed stride is 16 bytes instead of 20, so every vertex after the first is read from the wrong place. On a 64 by 64 target the correct layout covered 1352 pixels with a blue-grey centre, `63 67 125`; the short one covered 1146 pixels with a green centre, `22 62 0`.

The layout and the `pack` template describe the same bytes twice. When a picture is nearly right, compare the two before suspecting anything else.

Next

The triangle's corners were already in clip space. The *next chapter* feeds the card model coordinates and the matrices from *foundations*, and brings back the depth buffer, culling and lighting of the software arc.

A 3D scene on the GPU: matrices, depth, and light

The *previous chapter* handed the card positions already in clip space. This one hands it model coordinates and the matrices from *foundations*, and brings back, on the card, everything the software arc built by hand: the *depth buffer*, back-face culling, and the lighting model from *shading*, evaluated per pixel.

The examples use the vector and matrix subs from foundations unchanged: `v_add`, `v_sub`, `v_scale`, `v_dot`, `v_cross`, `v_norm`, `m_mul`, `m_translate`, `m_scale`, `m_rotate_y`, `m_look_at`, `m_perspective` and `xform`. The complete program at the end loads them from a file.

Matrices as uniforms

A value that is the same for every vertex of a draw is a **uniform**. The shader declares it, and a uniform buffer supplies it:

```
@group(0) @binding(0) var<uniform> mvp: mat4x4<f32>;

@vertex fn vs(@location(0) p: vec3<f32>) -> @builtin(position) vec4
 ↪<f32> {
    return mvp * vec4<f32>(p, 1.0);
}
```

```
my $mvp = m_mul($proj, m_mul($view, $model));
my $ub  = $gpu->uniform_buffer(pack 'f16', @$mvp);

$gpu->render_pass($img, clear => [0, 0, 0, 1])
    ->pipeline($pipe)->bind([$ub])->vertices($vb)->draw($n)->end;
```

`bind` takes a list of buffers for `@group(0)`, in binding order: `[$a, $b]` supplies `@binding(0)` and `@binding(1)`. It comes after `pipeline`, because the pipeline read the shader and knows what each binding has to be.

No transposing. Foundations stores matrices column by column, element `m[col * 4 + row]`, and WGSL's `mat4x4<f32>` is column-major too, so `pack 'f16', @$m` is already the layout the shader reads, and `mvp * v` in WGSL is `xform($mvp, $v)` in Perl.

That is checkable to the pixel. Five points drawn with `topology => 'point-list'`, each lighting one pixel of an 800x600 target, against foundations' own screen mapping of the same points, with the camera at `[0, 0, 5]`:

```
point           Perl screen position   pixel the card lit
[0,0,0]         (400.00, 300.00)       (399,299)
[1,0,-1]        (486.60, 300.00)       (486,299)
[1,0,-8]        (439.97, 300.00)       (439,299)
[-1.5,1,0]      (244.12, 196.08)       (244,196)
[0.7,-1.2,2]    (521.24, 507.85)       (521,507)
```

Every coordinate strictly inside a pixel matches. The three points whose predicted y is exactly 300.00 sit on the boundary between rows 299 and 300, and the card chose 299: a tie on a pixel edge, not an error. The first point is foundations' own "dead centre of an 800x600 window".

That check needed one more matrix than foundations used, for the reason below.

WebGPU's depth range

OpenGL, whose conventions foundations follows, keeps a point when `-w <= z <= w` in clip space, so normalised depth runs from -1 to +1. WebGPU keeps it when `0 <= z <= w`: depth runs from 0 to 1. Everything foundations' projection sends to negative depth, the card throws away.

Symptom: objects close to the camera vanish, well before they reach the near plane. Cause: an OpenGL-style projection matrix on WebGPU. With `near = 0.1` and

`far = 100`, a small square drawn at decreasing distances, its size on screen held constant:

```
distance   foundations' ndc.z   pixels drawn   remapped ndc.z   pixels drawn
5            0.962                100            0.981            100
0.5          0.602                100            0.801            100
0.21         0.049                100            0.524            100
0.19        -0.052                  0            0.474            100
0.15        -0.333                  0            0.334            100
0.12        -0.666                  0            0.167            100
```

The cut-off is where `ndc.z` crosses zero, at `2 * far * near / (far + near)`, which is 0.1998: about **twice** the near distance. Geometry between the near plane and there is silently lost, and for a camera walking through a scene that is everything within arm's reach.

Keep foundations' projection and convert its output with one more matrix, which maps `z` to `(z + w) / 2`:

```
# OpenGL clip space keeps -w <= z <= w, WebGPU keeps 0 <= z <= w.
my $gl_to_wgpu = [1,0,0,0,  0,1,0,0,  0,0,0.5,0,  0,0,0.5,1];

my $proj = m_mul($gl_to_wgpu, m_perspective(60, 4/3, 0.1, 100));
```

Only depth changes. `x`, `y` and `w` are untouched, so every screen position in foundations still holds, as the point table above shows. Foundations' example point at depth 5 goes from `ndc.z = 0.962` to `0.981`.

Changing uniforms

`$buf->write($bytes)` replaces a buffer's contents, and the next pass sees the new values. Writing once per frame, before the pass, is how a camera moves.

Symptom: several objects drawn in one pass all appear where the last one should be. Cause: one uniform buffer rewritten between the draws. The writes are applied before the pass runs, so every draw in it sees the last value. Two squares, one uniform, rewritten between two draws:

```
one buffer rewritten:  left half 0 px, right half 144 px
one buffer per object: left half 144 px, right half 144 px
```

Give each object its own buffer and `bind` it before its draw, or use instancing, below, which is one draw for all of them.

Symptom: a uniform value is shifted, or reads as zero. Cause: WGSL aligns a `vec3<f32>` to 16 bytes, not 12. In `struct U { a: vec3<f32>, b: vec3<f32> }`, `b` starts at byte 16, so `pack 'f6'` puts b's first component into padding:

```
packed tight   pack 'f8', a, b, 0, 0        b reads as 0 0 0
padded         pack 'f8', a, 0, b, 0        b reads as 255 0 0
```

A buffer smaller than the struct is refused at `bind`, with `Binding size 24 of Buffer ... is less than minimum 32`, but a misaligned one of the right size is accepted. The

reliable rule is to use `vec4<f32>` in uniform structs and leave the fourth component unused, as every struct in this chapter does.

The depth buffer

The *rasteriser chapter* kept the nearest depth per pixel and skipped anything further. On the card that is two settings, and both are needed:

```
my $pipe  = $gpu->render_pipeline($shader, depth => 1, ...);
my $depth = $gpu->depth_buffer($W, $H);

$gpu->render_pass($target, clear => [0.05, 0.05, 0.08, 1], depth =>
 ↪$depth)
    ->pipeline($pipe) ... ->end;
```

`depth => 1` on the pipeline makes it test and write depth; `depth => $depth` on the pass supplies the buffer. The pass clears it to 1.0, the far end of WebGPU's range, and the test keeps a fragment only when it is nearer than what is there: the rasteriser's `next if $z >= $t->{depth}[$y][$x]`, in hardware.

Either half alone is refused at end, naming the mismatch:

```
Incompatible depth-stencil attachment format: the RenderPass uses a
texture with format None but the RenderPipeline with '' label uses␣
 ↪an
attachment with format Some(Depth32Float)
```

The depth buffer must be the size of the target. After a resize, make a new one; the old one is refused with `Attachments have differing sizes`.

Nine cubes in three rows, rendered into a 320x240 texture, show what it buys. Drawn back row first, the scene comes out identical with the depth buffer and without it, because that order happens to be the painter's algorithm done right. Drawn front row first:

```
front row first, depth buffer on    identical to back row first
front row first, depth buffer off   3007 pixels differ
```

The far cubes paint over the near ones, the *rasteriser chapter's* second figure in full colour. With the depth buffer the order of the draws stops mattering, which is the whole point.

A mesh: the cube

A cube has 8 corners but needs 24 vertices here: each face has its own normal, and a vertex carries one normal, so a corner shared by three faces is three vertices. That is the flat-faces case from *shading*; a smooth surface shares its vertices instead.

Each face is described by its normal and two in-plane axes, chosen so that `u x v` is the normal. Then its corners, taken in the order below, wind counter-clockwise seen from outside, and the cross product checks it while building:

```
my @FACES = (
    # normal       u (right)     v (up)      u x v == normal
    [[ 1, 0, 0], [ 0, 0,-1], [0, 1, 0]],
    [[-1, 0, 0], [ 0, 0, 1], [0, 1, 0]],
    [[ 0, 1, 0], [ 1, 0, 0], [0, 0,-1]],
    [[ 0,-1, 0], [ 1, 0, 0], [0, 0, 1]],
    [[ 0, 0, 1], [ 1, 0, 0], [0, 1, 0]],
    [[ 0, 0,-1], [-1, 0, 0], [0, 1, 0]],
);
my (@vert, @idx);
for my $f (@FACES) {
    my ($n, $u, $v) = @$f;
    die "face winds the wrong way\n" unless v_dot(v_cross($u, $v),
 ↪$n) > 0;
    my $base = @vert / 6;
    for my $c ([-1,-1], [1,-1], [1,1], [-1,1]) {
        my $p = v_add(v_scale($n, 0.5),
                      v_add(v_scale($u, 0.5 * $c->[0]), v_scale($v,␣
 ↪0.5 * $c->[1])));
        push @vert, @$p, @$n;
    }
    push @idx, map { $base + $_ } 0, 1, 2, 0, 2, 3;
}
# 24 vertices of six floats, 36 indices

my $verts = $gpu->vertex_buffer(pack 'f*', @vert);
my $index = $gpu->index_buffer(pack 'L*', @idx);
```

The vertex layout is `['float32x3', 'float32x3']`: position at `@location(0)`, normal at `@location(1)`.

Back-face culling

```
my $pipe = $gpu->render_pipeline($shader, cull => 'back', ...);
```

`cull => 'back'` discards every triangle whose front faces away. Which side is the front is `front_face`: `ccw`, the default, means counter-clockwise as the viewer sees it, which is how the cube above was built. This is the rasteriser chapter's signed-area test, done by the card before any pixel work.

On the nine-cube scene:

```
cull => 'none'                          identical to cull => 'back'
cull => 'back', front_face => 'cw'   22508 of 76800 pixels differ
```

On a closed mesh behind a depth buffer, culling changes nothing visible: the back faces it removes would have lost the depth test anyway. That identity is the test the *world generator* applies with its `--no-cull` option. It is also why a wrong winding convention is so visible: with `front_face => 'cw'` the card keeps exactly the faces that point away, and each cube renders as its own dark interior.

Symptom: a model renders dark and inside out. Cause: its winding and the pipeline's

`front_face` disagree. The model may have been authored the other way round, or its model matrix may contain a reflection: the same scene with every cube mirrored by `m_scale(-1, 1, 1)` differed in the same 22508 pixels under `cull => 'back'`, and `front_face => 'cw'` restored it exactly. A reflection reverses winding, the same fact the world generator's y-flip ran into.

Instancing

Nine cubes are the same 36 indices nine times with a different model matrix. Drawing them as nine draws works, but one draw can do it, with an instance count after the index count:

```
$gpu->render_pass($target, clear => [0.05, 0.05, 0.08, 1], depth =>
 ↪$depth)
    ->pipeline($pipe)->bind([$scene, $instances])
    ->vertices($verts)->indices($index)->draw_indexed(36, 9)->end;
```

The shader tells the instances apart by `@builtin(instance_index)`, 0 to 8, and reads per-instance data from a storage buffer:

```
struct Instance {
    model:  mat4x4<f32>,
    colour: vec4<f32>,
};
@group(0) @binding(1) var<storage, read> inst: array<Instance>;

@vertex
fn vs(@location(0) p: vec3<f32>, @location(1) n: vec3<f32>,
      @builtin(instance_index) i: u32) -> VOut {
    let m = inst[i].model;
    ...
}
```

```
my @inst;
for my $k (0 .. 8) {
    my ($gx, $gz) = ($k % 3 - 1, int($k / 3) - 1);
    my $model = m_mul(m_translate(1.8 * $gx, 0, 1.8 * $gz), m_
 ↪rotate_y(0.35 * $k));
    push @inst, @$model, @{ $COLOUR[$k % 3] }, 1;
}
my $instances = $gpu->storage_buffer(pack 'f*', @inst);    # 9 x 80␣
 ↪bytes
```

Each instance is 20 floats, a matrix and a `vec4` colour, 80 bytes, with no padding to think about because both members are multiples of 16 bytes. Moving the cubes is one `write` of 720 bytes per frame.

Declare the storage buffer `read`. A vertex shader may not write to storage, and `read_write` there is refused when the pipeline is built: `Features ... (VERTEX_WRITABLE_STORAGE) are required but not enabled`.

Lighting per fragment

Shading ended on its rule: interpolate the inputs, not the answer. The vertex shader passes the fragment shader a world position and a normal, the card interpolates both, and the lighting runs once per pixel:

```
@fragment
fn fs(v: VOut) -> @location(0) vec4<f32> {
    let n = normalize(v.normal);
    let l = normalize(scene.light_dir.xyz);
    let e = normalize(scene.eye.xyz - v.world);
    let lambert = max(dot(n, l), 0.0);
    let h = normalize(l + e);
    let spec = pow(max(dot(n, h), 0.0), 32.0);
    let c = v.colour * (0.1 + 0.9 * lambert) + vec3<f32>(0.4) *
↪spec;
    return vec4<f32>(min(c, vec3<f32>(1.0)), 1.0);
}
```

Line for line it is shading's model: the Lambert term clamped at zero, the ambient `0.1 + 0.9 * lambert`, the Blinn-Phong halfway vector with the dot product clamped before the power. `light_dir` points from the surface toward the light and `e` from the surface toward the eye, the conventions shading fixed.

The first `normalize` is not optional. Interpolating between two unit vectors gives a vector shorter than 1 in between, and an unnormalised normal scales the lighting silently.

To check the arithmetic rather than the picture, render into a floating-point target and read the numbers themselves. A pipeline and texture with `format => 'rgba32float'` store four `f32` per pixel, and `unpack 'f4'` reads them. A fragment shader writing `vec4<f32>(lambert, spec, 0.0, 1.0)` for a surface facing the viewer reproduces shading's tables:

```
light off the normal   lambert        shininess   specular   Perl,
↪double
0 deg                  1.000          1           0.965926   0.
↪965926
30 deg                 0.866          8           0.757793   0.
↪757792
60 deg                 0.500          32          0.329763   0.
↪329762
90 deg                 0.000          128         0.011825   0.
↪011825
180 deg                0.000
```

The specular column has the light 30 degrees off and the viewer straight on, so the halfway vector is 15 degrees off the normal. The card and Perl's `cos(15 deg) ** $shininess` agree to the sixth decimal, and differ there because WGSL computes in `f32`.

Normals and the model matrix

The vertex shader turns the normal with `(m * vec4<f32>(n, 0.0)).xyz`. The `0.0` makes it a direction: the translation column is multiplied by zero, so moving an object does not bend its normals.

That is correct for rotation, translation and uniform scale, which is all this chapter uses. Under a **non-uniform** scale it is wrong. A face with normal `(1, 1, 0)` and in-plane direction `(1, -1, 0)`, scaled by `(1, 3, 1)`: the in-plane direction becomes `(1, -3, 0)` and the transformed normal `(1, 3, 0)`, and their dot product is -8, so the "normal" no longer stands at right angles to its own surface. Turned by the inverse transpose of the scale instead, the normal is `(1, 1/3, 0)`, and the dot product is 0. A model matrix with non-uniform scale needs that inverse-transpose matrix passed alongside it.

Colours in linear light

The window's surface is an `-srgb` format, as the *device chapter* measured, so what the fragment shader writes is encoded on the way to the screen. The lighting above is therefore computed in linear light, which is correct, and the base colours, picked by eye, are decoded before upload:

```
my @COLOUR = map { [ map { $_ ** 2.2 } @$_ ] }       # picked by eye: decode
             [0.9, 0.3, 0.2], [0.3, 0.8, 0.3], [0.3, 0.4, 0.9];
```

Left encoded, the same scene renders pale and washed out. The headless renders in this chapter use an `rgba8unorm-srgb` texture for the same reason, so the image read back is the image the window shows.

The whole scene

Nine lit cubes turning on a grid, in a resizable window, with the fixed step from the *device chapter*. The subs from foundations are in `foundations.pl` beside it, followed by a line `1;`.

```
use strict;
use warnings;
use SDL2 qw(:all);
use Peta::WebGPU;
BEGIN { require './foundations.pl' }   # the v_* and m_* subs, then 1;

use constant { STEP => 16, MAX_STEPS => 5 };
my $max_frames = shift // 0;

my $WGSL = <<'WGSL';
struct Scene {
    view_proj: mat4x4<f32>,
    light_dir: vec4<f32>,     // xyz: toward the light, world space
    eye:       vec4<f32>,     // xyz: camera position, world space
};
```

(continues on next page)

(continued from previous page)

```
struct Instance {
    model:  mat4x4<f32>,
    colour: vec4<f32>,
};
@group(0) @binding(0) var<uniform> scene: Scene;
@group(0) @binding(1) var<storage, read> inst: array<Instance>;

struct VOut {
    @builtin(position) pos: vec4<f32>,
    @location(0) world:  vec3<f32>,
    @location(1) normal: vec3<f32>,
    @location(2) colour: vec3<f32>,
};

@vertex
fn vs(@location(0) p: vec3<f32>, @location(1) n: vec3<f32>,
      @builtin(instance_index) i: u32) -> VOut {
    let m = inst[i].model;
    let w = m * vec4<f32>(p, 1.0);
    var o: VOut;
    o.pos    = scene.view_proj * w;
    o.world  = w.xyz;
    o.normal = (m * vec4<f32>(n, 0.0)).xyz;
    o.colour = inst[i].colour.rgb;
    return o;
}

@fragment
fn fs(v: VOut) -> @location(0) vec4<f32> {
    let n = normalize(v.normal);
    let l = normalize(scene.light_dir.xyz);
    let e = normalize(scene.eye.xyz - v.world);
    let lambert = max(dot(n, l), 0.0);
    let h = normalize(l + e);
    let spec = pow(max(dot(n, h), 0.0), 32.0);
    let c = v.colour * (0.1 + 0.9 * lambert) + vec3<f32>(0.4) *␣
↪spec;
    return vec4<f32>(min(c, vec3<f32>(1.0)), 1.0);
}
WGSL

my @FACES = (
    # normal       u (right)     v (up)      u x v == normal
    [[ 1, 0, 0], [ 0, 0,-1], [0, 1, 0]],
    [[-1, 0, 0], [ 0, 0, 1], [0, 1, 0]],
    [[ 0, 1, 0], [ 1, 0, 0], [0, 0,-1]],
    [[ 0,-1, 0], [ 1, 0, 0], [0, 0, 1]],
    [[ 0, 0, 1], [ 1, 0, 0], [0, 1, 0]],
    [[ 0, 0,-1], [-1, 0, 0], [0, 1, 0]],
```

(continues on next page)

(continued from previous page)

```
);
my (@vert, @idx);
for my $f (@FACES) {
    my ($n, $u, $v) = @$f;
    die "face winds the wrong way\n" unless v_dot(v_cross($u, $v),
↪$n) > 0;
    my $base = @vert / 6;
    for my $c ([-1,-1], [1,-1], [1,1], [-1,1]) {
        my $p = v_add(v_scale($n, 0.5),
                      v_add(v_scale($u, 0.5 * $c->[0]), v_scale($v,␣
↪0.5 * $c->[1])));
        push @vert, @$p, @$n;
    }
    push @idx, map { $base + $_ } 0, 1, 2, 0, 2, 3;
}

my $win  = SDL2::Window->new(title => 'cubes', w => 800, h => 600,␣
↪resizable => 1);
my $gpu  = Peta::WebGPU::Device->new(window => $win);
my $surf = $gpu->surface;
my ($W, $H) = $surf->size;

my $pipe = $gpu->render_pipeline($gpu->shader($WGSL),
    format => $surf->format, depth => 1, cull => 'back',
    vertex_buffers => [ { attributes => ['float32x3', 'float32x3'] }␣
↪]);
my $verts = $gpu->vertex_buffer(pack 'f*', @vert);
my $index = $gpu->index_buffer(pack 'L*', @idx);
my $depth = $gpu->depth_buffer($W, $H);

my @COLOUR = map { [ map { $_ ** 2.2 } @$_ ] }       # picked by␣
↪eye: decode
             [0.9, 0.3, 0.2], [0.3, 0.8, 0.3], [0.3, 0.4, 0.9];
my $instances = $gpu->storage_buffer(9 * 20 * 4);
my $scene     = $gpu->uniform_buffer(24 * 4);
my $gl_to_wgpu = [1,0,0,0,  0,1,0,0,  0,0,0.5,0,  0,0,0.5,1];

my $angle = 0;
sub update { $angle += 0.0015 * STEP }

sub upload {
    my @inst;
    for my $k (0 .. 8) {
        my ($gx, $gz) = ($k % 3 - 1, int($k / 3) - 1);
        my $model = m_mul(m_translate(1.8 * $gx, 0, 1.8 * $gz),
                          m_rotate_y($angle + 0.35 * $k));
        push @inst, @$model, @{ $COLOUR[$k % 3] }, 1;
    }
    $instances->write(pack 'f*', @inst);
```

(continues on next page)

(continued from previous page)

```
    my $eye  = [0, 4, 6];
    my $proj = m_mul($gl_to_wgpu, m_perspective(45, $W / $H, 0.1,␣
↪100));
    my $vp   = m_mul($proj, m_look_at($eye, [0, 0, 0], [0, 1, 0]));
    $scene->write(pack 'f*', @$vp, @{ v_norm([0.6, 1.0, 0.8]) }, 0,␣
↪@$eye, 1);
}

sub draw {
    my ($target) = @_;
    $gpu->render_pass($target, clear => [0.05, 0.05, 0.08, 1], depth␣
↪=> $depth)
        ->pipeline($pipe)->bind([$scene, $instances])
        ->vertices($verts)->indices($index)->draw_indexed(scalar␣
↪@idx, 9)->end;
}

my ($last, $acc, $frames, $running) = (SDL_GetTicks(), 0, 0, 1);
while ($running) {
    while (my $ev = SDL_PollEvent()) {
        if ($ev->{type} == SDL_QUIT) { $running = 0 }
        elsif ($ev->{type} == SDL_KEYDOWN && $ev->{sym} == SDLK_
↪ESCAPE) { $running = 0 }
        elsif ($ev->{type} == SDL_WINDOWEVENT
               && $ev->{event} == SDL_WINDOWEVENT_SIZE_CHANGED) {
            ($W, $H) = @$ev{qw(data1 data2)};
            $surf->configure($W, $H);
            $depth = $gpu->depth_buffer($W, $H);     # must match␣
↪the frame
        }
    }
    my $now = SDL_GetTicks();
    $acc += $now - $last;
    $last = $now;
    my $steps = 0;
    while ($acc >= STEP) {
        update();
        $acc -= STEP;
        if (++$steps >= MAX_STEPS) { $acc = 0; last }
    }

    my $frame = $surf->next_frame or next;
    upload();
    draw($frame);
    $frame->present;
    $running = 0 if $max_frames && ++$frames >= $max_frames;
}
printf "%d frames at %dx%d\n", $frames, $W, $H;
```

`draw` takes its target as an argument, so the same routine renders into a texture for checking. A texture with `format => $surf->format` and the same size as the depth buffer is all it needs; with `bgra8unorm-srgb`, remember that `read` returns B, G, R, A. Run in a window that was resized from 800x600 to 640x480 while it ran, the depth buffer was recreated at the new size and the frames carried on.

What this arc covers

The card receives meshes as vertex and index buffers, transforms them with foundations' matrices, resolves visibility with a depth buffer, culls back faces, draws many objects in one call, and lights every pixel with shading's model. Every stage of the software arc has its hardware counterpart, and each was checked against the software version: the same 54 pixels, the same screen positions, the same lighting numbers.

The capstones put it to work on complete programs: an *interactive modeller*, which adds the orbit camera and picking from *scene graphs*, and a *small 3D game*.

Capstone: a procedural world generator

A program that invents a landscape and draws it. Three stages turn noise into terrain, and each is written three times: as plain Perl loops, as PDL whole-array expressions, and as WGSL compute shaders. `--mesh all` runs all three on every launch and prints how far they disagree, so the comparison is known to be measuring one algorithm rather than three.

Everything after those stages is the plain Perl from this guide's earlier chapters: the mesh from *meshes*, the camera and rasteriser from *software-rasteriser*, the Lambert term from *shading*, the writer from *model-formats*.

```
world   512x512 grid, 6 octaves, 12 erosion passes, seed 1234
gpu     Quadro M2000M (Vulkan)

  backend   terrain   erosion   normals  readback     total  speedup
  perl      1.033s    4.945s    0.221s    0.000s    6.198s     1.0x
  pdl       0.253s    0.304s    0.017s    0.033s    0.607s    10.2x
  gpu       0.002s    0.004s    0.004s    0.034s    0.043s   145.2x

  agree   perl vs pdl   max dh = 0.00e+00   max dn.x = 0.00e+00
  agree   perl vs gpu   max dh = 2.95e-07   max dn.x = 1.26e-05
```

PDL and plain Perl agree to the last bit, because both compute in `double`. The GPU differs in the seventh decimal, because WGSL storage buffers here are `f32`. That is the expected difference, and it is worth printing rather than assuming: a backend that has silently diverged shows up here as a large number instead of a small one.

Three stages, one shape

Stage	Input	Output	Per cell
terrain	noise lattice	height	sum N octaves of bilinear samples
erosion	height	height	compare 8 neighbours
normals	height	3 components	4 neighbours, one `sqrt`

Each produces one value per grid cell from a fixed neighbourhood, with no cell depending on another cell's result within the same pass. That is the shape a compute shader exists for, the shape PDL expresses without a loop, and the shape the JIT compiles well. It is why one algorithm can have three implementations that agree to the bit.

The renderer is deliberately not in that list. Rasterising is a scatter: triangles write to overlapping pixels and the Z-buffer arbitrates. It stays in Perl, and at 512 it costs `4.10s` against the field stages' `0.043s`.

One lattice, three backends

The noise lattice is built once, in Perl, and handed to whichever backend runs:

```
srand $o{seed};
my @LAT = map { rand() } 0 .. $R * $R - 1;
```

Invariant: every backend consumes byte-identical input. A shader that generated its own noise from an integer hash would produce a different field from the Perl one, and "the three agree" would be untestable. This also removes the portability trap in hash-based noise: `sin`-based hashes differ between drivers, so terrain generated on one GPU would not reproduce on another.

Terrain: octaves of value noise

Good terrain sums several **octaves**, each at twice the frequency and half the amplitude of the last, which is **fractal Brownian motion**. Each octave bilinearly samples the lattice with a smoothstep fade.

```
struct P { n: u32, r: u32, oct: u32, pad: u32, base: f32, p1: f32,␣
 ↪p2: f32, p3: f32 };
@group(0) @binding(0) var<storage, read_write> lat: array<f32>;
@group(0) @binding(1) var<uniform> p: P;
@group(0) @binding(2) var<storage, read_write> out: array<f32>;

// NOT `smooth`: that is a reserved word in WGSL and the shader will␣
 ↪not compile.
fn fade(t: f32) -> f32 { return t * t * (3.0 - 2.0 * t); }

fn cell(ix: i32, iy: i32, r: i32) -> f32 {
    return lat[u32((iy % r) * r + (ix % r))];
}
```

(continues on next page)

(continued from previous page)

```
@compute @workgroup_size(8, 8)
fn main(@builtin(global_invocation_id) id: vec3<u32>) {
    if (id.x >= p.n || id.y >= p.n) { return; }   // grid need not␣
↪divide by 8
    let r = i32(p.r);
    var sum: f32 = 0.0; var amp: f32 = 1.0; var freq: f32 = 1.0; var␣
↪norm: f32 = 0.0;
    for (var o: u32 = 0u; o < p.oct; o = o + 1u) {
        // The per-octave offset decorrelates the octaves. Without␣
↪it they
        // sample the same lattice cells and the sum has visible␣
↪structure.
        let px = f32(id.x) / f32(p.n) * p.base * freq + f32(o) * 17.
↪0;
        let py = f32(id.y) / f32(p.n) * p.base * freq + f32(o) * 17.
↪0;
        let ix = i32(floor(px)); let iy = i32(floor(py));
        let a = cell(ix, iy, r);         let b = cell(ix + 1, iy,␣
↪r);
        let c = cell(ix, iy + 1, r);     let d = cell(ix + 1, iy +␣
↪1, r);
        let ux = fade(px - floor(px));   let uy = fade(py -␣
↪floor(py));
        let lo = a + (b - a) * ux;
        let hi = c + (d - c) * ux;
        // Ridged: fold EACH octave, never the sum. Squaring␣
↪sharpens crests.
        let r = 1.0 - abs(2.0 * (lo + (hi - lo) * uy) - 1.0);
        sum = sum + amp * r * r;
        norm = norm + amp;
        amp = amp * 0.5; freq = freq * 2.0;
    }
    out[id.y * p.n + id.x] = sum / norm;
}
```

The bounds check is not optional. A dispatch covers whole workgroups, so a 100-cell grid with `workgroup_size(8)` runs 104 invocations per axis and the last four would write past the buffer.

The fold is what makes ridges, and it must be applied per octave. `1 - |2v - 1|` reflects a noise value about its midpoint, turning a smooth hill into a crease; squaring it sharpens the crest. Doing that to each octave and then summing gives ridgelines at every scale.

Folding the **sum** instead is a trap worth naming, because the code looks almost identical and the output looks plausible until you measure it. The sum of six octaves, normalised by their total amplitude, is a weighted average of six independent samples, so by the central limit theorem it clusters tightly around 0.5, and 0.5 is exactly where the fold peaks. The commonest value therefore maps to the *maximum* height. The result is a plateau with dips punched in it rather than mountains, and the height histogram gives it away at once:

```
folding the sum      0:0%  1:2%  2:3%  3:6%  4:9% 5:12% 6:18% 7:22%
 ↪8:19% 9:8%
folding each octave  0:3% 1:10% 2:15% 3:17% 4:17% 5:15% 6:12%  7:7%
 ↪8:3% 9:1%
```

The second is what terrain looks like: much low ground, a thin tail of peaks. `--histogram` prints this for any run.

Erosion: the same pass three ways

Raw fractal noise looks synthetic because real landscapes have been worn. **Thermal erosion** is the cheapest correction: material sitting at a steeper angle than the rock holds slides to lower neighbours. The talus angle is that steepest held slope.

Plain Perl states it directly:

```
for my $pass (1 .. $EROSION) {
    my @n = @h;
    for my $y (0 .. $N - 1) {
        my $ym = $y > 0 ? $y - 1 : 0;
        my $yp = $y < $N - 1 ? $y + 1 : $N - 1;
        for my $x (0 .. $N - 1) {
            my $xm = $x > 0 ? $x - 1 : 0;
            my $xp = $x < $N - 1 ? $x + 1 : $N - 1;
            my $c = $h[$y * $N + $x];
            my $moved = 0;
            for my $yy ($ym, $y, $yp) {
                for my $xx ($xm, $x, $xp) {
                    next if $xx == $x && $yy == $y;
                    my $d = $c - $h[$yy * $N + $xx];
                    $moved += ($d - $TALUS) * $RATE if $d > $TALUS;
                }
            }
            $n[$y * $N + $x] = $c - $moved;
        }
    }
    @h = @n;
}
```

PDL replaces the two inner loops with eight whole-array expressions. The neighbour lookup is the part worth getting right: pad the field once with edge replication, then read each neighbour as a **lazy slice** of the padded copy. A slice is a view, so a pass costs one padded copy rather than one copy per neighbour.

```
sub padded {
    my ($P) = @_;
    my $E = $N + 1;
    my $Q = zeroes($N + 2, $N + 2);
    $Q->slice("1:$N,1:$N")   .= $P;
    $Q->slice("0:0,1:$N")    .= $P->slice("0:0,:");        # left
 ↪column
```

(continues on next page)

(continued from previous page)

```
    $Q->slice("$E:$E,1:$N")  .= $P->slice("-1:-1,:");      # right
↪column
    # Rows 1 and N already carry the correct left and right edges,
↪so
    # copying them outward fills the top and bottom rows AND the
↪corners.
    $Q->slice("0:$E,0:0")    .= $Q->slice("0:$E,1:1");
    $Q->slice("0:$E,$E:$E")  .= $Q->slice("0:$E,$N:$N");
    return $Q;
}

sub nbr {
    my ($Q, $dx, $dy) = @_;
    return $Q->slice(sprintf "%d:%d,%d:%d", 1+$dx, $N+$dx, 1+$dy,
↪$N+$dy);
}

for my $pass (1 .. $EROSION) {
    my $Q = padded($Hp);
    my $moved = zeroes($N, $N);
    for my $dy (-1, 0, 1) {
        for my $dx (-1, 0, 1) {
            next if $dx == 0 && $dy == 0;
            # lclip(0) is the branchless `if (d > talus)`:
↪everything at or
            # below the talus angle contributes exactly zero.
            $moved += (($Hp - nbr($Q, $dx, $dy)) - $TALUS)->lclip(0)
↪* $RATE;
        }
    }
    $Hp = $Hp - $moved;
}
```

The two Perl-level loops that remain run 8 times per pass, not `8 * N * N` times. That is the whole PDL technique: the loop nest moves from Perl into one C pass per expression.

The shader keeps the neighbour loop, because on the GPU it runs inside a single invocation that is already one of `N * N` running at once:

```
fn at(x: i32, y: i32, n: i32) -> f32 {
    return src[u32(clamp(y, 0, n - 1) * n + clamp(x, 0, n - 1))];
}

@compute @workgroup_size(8, 8)
fn main(@builtin(global_invocation_id) id: vec3<u32>) {
    let n = i32(p.n);
    if (id.x >= p.n || id.y >= p.n) { return; }
    let x = i32(id.x); let y = i32(id.y);
    let h = at(x, y, n);
    var moved: f32 = 0.0;
```

(continues on next page)

(continued from previous page)

```
    for (var dy: i32 = -1; dy <= 1; dy = dy + 1) {
        for (var dx: i32 = -1; dx <= 1; dx = dx + 1) {
            if (dx == 0 && dy == 0) { continue; }
            let d = h - at(x + dx, y + dy, n);
            if (d > p.talus) { moved = moved + (d - p.talus) * p.
↪rate; }
        }
    }
    dst[u32(y * n + x)] = h - moved;
}
```

Invariant: `src` and `dst` must be different buffers. Invocations run in an undefined order, so a shader that updated the grid in place would have each cell reading a mixture of this pass's and last pass's neighbours. The result is not merely noisy, it is non-deterministic between runs. Note that the Perl version obeys the same rule for the same reason, with `my @n = @h`.

The Perl side alternates them, which is the standard **ping-pong**:

```
for my $i (1 .. $EROSION) {
    my ($s, $d) = ($i % 2) ? ($bufA, $bufB) : ($bufB, $bufA);
    $gpu->compute($erode, 'main', [$s, $pe, $d], $g, $g);
}
my $final = ($EROSION % 2) ? $bufB : $bufA;      # parity decides
↪which holds it
```

Buffers bind in the order given, so `[$s, $pe, $d]` supplies bindings 0, 1 and 2. The dispatch is two-dimensional because the workgroup is: `$g` is `ceil(N/8)` on each axis.

Symptom: the terrain differs slightly between identical runs. Cause: a compute shader reading and writing one buffer.

Normals: analytic, not accumulated

The *meshes chapter* computes vertex normals by accumulating face normals into the vertices that share them. That is correct for arbitrary geometry, but it is a **scatter**: several faces add into one vertex, so the order matters and the operation does not parallelise without atomics.

A heightfield does not need it. The surface is a function of `x` and `z`, so the normal follows from the slope by central differences:

```
n = normalize(-dh/dx, 1, -dh/dz)
```

That is a **gather**: each vertex reads its four neighbours and writes only itself. Choosing this formulation is what lets the third stage be the same algorithm in all three backends.

```
my $dx = ($h[$y * $N + $xp] - $h[$y * $N + $xm]) / (($xp - $xm) *
↪$S);
my $dz = ($h[$yp * $N + $x] - $h[$ym * $N + $x]) / (($yp - $ym) *
↪$S);
```

(continues on next page)

(continued from previous page)

```
my $l = sqrt($dx * $dx + 1 + $dz * $dz);
my $i = $y * $N + $x;
$nx[$i] = -$dx / $l; $ny[$i] = 1 / $l; $nz[$i] = -$dz / $l;
```

Interior cells span `2 * S` between neighbours; clamped edge cells span only `S`. Rather than special-case the border, PDL builds the divisor as a field so the expression stays branchless:

```
my $wx = zeroes($N, $N) + 2 * $S;
$wx->slice("0:0,:") .= $S; $wx->slice("-1:-1,:") .= $S;
```

Getting the answer back

At 512 the GPU spends `0.010s` computing and `0.034s` returning the result. **Readback, not compute, is the cost of using a GPU for this.** Three rules follow, and they are what makes that `0.034s` rather than seconds.

Transfer in bulk, never per element. One `unpack` over the whole buffer, never a loop of accessors:

```
my @nx = unpack 'f*', $nb[0]->read;
```

The same rule applies coming out of PDL. A piddle's data slab is reachable as a string, and reading it in one go beats looping with `at` by 7.6x at `512 * 512`:

```
sub unpiddle { return unpack 'd*', ${ $_[0]->sever->get_dataref } }
```

Give each output its own buffer. The shader writes `nx`, `ny` and `nz` to three storage buffers rather than one interleaved `xyzxyz` buffer, so each comes back as one `unpack`. De-interleaving in Perl is an `N * N` loop, and at `1024 * 1024` it costs `0.232s` against a `0.002s` dispatch.

The same trick works in the other direction. This program's renderer wants Perl arrays, so it stops at the `unpack`. When the destination is a piddle instead, assign the raw buffer straight into its data slab rather than building a list on the way; that moves 3 MB in `0.003s`:

```
my $G = zeroes(float, 3, $N, $N);
my $ref = $G->get_dataref;
$$ref = $raw;
$G->upd_data;
```

Materials and light

Picking a colour by height alone produces a relief map, not a landscape. Three corrections turn one into the other, and each is visible the moment it is missing.

Blend across a band, do not switch at a point. A hard threshold draws a contour line wherever two materials meet. Interpolating with a smoothstep over a narrow band removes it:

```
my $t = smoothstep(($h - $b + $BLEND) / (2 * $BLEND));
@c = map { $MATC[$i][$_] + ($MATC[$i+1][$_] - $MATC[$i][$_]) * $t }
 ↪0..2;
```

Let steepness override altitude. A cliff is bare rock at any height. Without this every band follows an exact contour and the terrain reads as stacked cut-outs. The surface normal already carries the answer: `n.y` is 1 on flat ground and 0 on a vertical face.

```
my $steep = smoothstep((0.80 - $ny) / 0.35);
return map { $c[$_] + ($ROCK->[$_] - $c[$_]) * $steep } 0..2;
```

Put the boundaries at percentiles, not at fixed heights. Which heights count as "high" depends on the terrain, and a fixed cut point that suits one seed strands another entirely above or below it. Taking the 15th, 50th and 85th percentiles of the actual field keeps the palette balanced whatever the generator produces:

```
my @bin = (0) x 256;
$bin[ int((($_ - $lo) / $span) * 255.999) ]++ for @h;
my ($acc, $j) = (0, 0);
for my $i (0 .. 255) {
    $acc += $bin[$i];
    while ($j <= $#PCT && $acc >= $PCT[$j] * @h) { $BOUND[$j++] = $i
 ↪/ 255 }
}
```

For light, **a low sun is what gives a landscape its relief.** Near the zenith every upward-facing triangle returns almost the same Lambert term and the terrain flattens into a white sheet; `--sun-elevation 80` demonstrates this. A dim cool fill from the sky hemisphere then keeps faces turned away from the sun as blue shadow rather than flat grey:

```
my $sky = 0.5 + 0.5 * $n->[1];
my @col = map {
    my $c = $base[$_] * (0.10 + 0.78 * $lam) + $SKY[$_] * 0.20 *
 ↪$sky;
    $c > 255 ? 255 : int $c;
} 0 .. 2;
```

Choosing a backend

Situation	Use
the grid is small, or the code must run anywhere	`perl`
the work is array arithmetic and stays in the process	`pdl`
the same small operation runs over a very large grid	`gpu`
the result is consumed cell by cell in Perl	`pdl`, and keep it in the piddle

The `145.2x` in the opening table is over plain Perl. Against PDL the GPU is `14.1x`, and `0.034s` of its `0.043s` is transfer.

The crossover is lower than it looks, because the GPU's cost is almost all fixed:

grid	perl	pdl	gpu
32	0.086s	**0.006s**	0.014s
64	0.170s	**0.009s**	0.014s
96	0.302s	0.018s	**0.015s**
128	0.474s	0.040s	**0.018s**
192	0.938s	0.073s	**0.017s**
512	6.198s	0.607s	**0.043s**

The GPU column barely moves from 32 to 192 while the grid grows 36-fold: up to a few hundred thousand cells the dispatch is free and the time is latency and transfer. PDL crosses it at about 80 cells on a side. Below that the card is pure overhead.

The card is not a faster computer; it is a machine for doing the same small thing to a great many values, and the skill is recognising which part of a problem has that shape.

Running it

The stages above, plus the mesh builder from *meshes* and the rasteriser from *software-rasteriser*, assemble into one file:

```
pperl worldgen.pl --size 512 --octaves 6 --erosion 12 --seed 1234 \
                  --mesh all --width 800 --height 600 \
                  --out world.png --obj world.obj --show
```

Option	Effect
`--size`	grid edge in cells
`--seed`	selects the landscape
`--octaves`	how many frequencies of noise to sum
`--erosion`	how many passes to weather it; 12 keeps the three-way run quick, 40 gives noticeably smoother ridges
`--mesh`	`perl`, `pdl`, `gpu`, `all`, or `auto` for the best available
`--azimuth`, `--elevation`, `--distance`	orbit camera; the terrain spans 6 units
`--fov`, `--look`	field of view, and the height the camera aims at
`--sun-azimuth`, `--sun-elevation`	sun position; low elevation gives the most relief
`--obj`	additionally write the mesh as Wavefront OBJ
`--show`	display the result in an SDL2 window
`--histogram`	print the height distribution and the material boundaries
`--no-cull`	disable back-face culling
`--power`	`high-performance` or `low-power` adapter preference

`--mesh pdl` needs a binary built with `--features pdl`, and `--mesh gpu` needs `--features webgpu` plus a usable adapter. `--mesh auto` picks the fastest that is present.

`--show` opens a window with the finished PNG through the native SDL2 binding of *the 2D guide's toolkit chapter*, so a run ends with the picture rather than a filename:

```
my $win = SDL2::Window->new(title => "worldgen $N x $N", w => $W, h
 ↪=> $H);
my $ren = $win->renderer;
my $tex = SDL2::Image::load_texture($ren, $o{out});
$ren->clear;
$ren->copy($tex);
$ren->present;
```

`--obj` writes ASCII Wavefront OBJ, one line per vertex and one per face, so it grows fast: 4 MB at 256, 17 MB at 512. It is there to open the terrain in a modelling tool, not as a transport format for large grids.

`--no-cull` should produce a byte-identical image, because the Z-buffer already resolves what the cull merely avoids computing. If it does not, the winding is wrong.

Winding, and the one line that decides it

```
next if $area == 0;                          # degenerate
next if $o{cull} && $area >= 0;              # back-face cull (screen y
 ↪is down)
my $sgn = $area < 0 ? -1 : 1;                # inside test follows the
 ↪winding
```

The projection flips y, because the screen's origin is top-left and the mathematics has y going up. **A flip is a reflection, and a reflection reverses orientation**, so every front-facing triangle arrives with *negative* signed area. Culling on `$area <= 0` therefore discards exactly the triangles it should keep, and an inside test written to accept only positive weights then rejects every interior pixel of whatever survived.

Decide the sign once per triangle and let the coverage test follow it.

Symptom: geometry is sparse or inside out, and inverting the cull changes which half is missing rather than fixing it. Cause: the cull and the coverage test disagreeing about winding.

What this demonstrates

The division of labour is the lesson. Three stages have the one shape a GPU is built for, and on a 512 grid they drop from `6.2s` to `0.043s`. The parts that do not have that shape stay in Perl, where they are readable, and the renderer among them is now the program's bottleneck at `4.10s`.

Writing the same algorithm three times is not redundancy here. It is what makes the number in the table trustworthy: the agreement line proves the three are one algorithm, so the speedup column compares implementations rather than problems.

Capstone: an interactive modeller

A small modelling program you can run. It opens a window on a floor with three objects standing on it; you orbit the camera by dragging, click an object to select it, and change it from the keyboard: extrude the face you clicked, subdivide, scale, turn, move, add and delete. `W` saves the scene as an OBJ file and reads it straight back to prove the file is sound.

Almost nothing in it is new. The meshes come from *meshes* and *modelling-operations*, the camera and the picking from *scene-graphs*, the file format from *model-formats*, and the drawing is the lit, depth-tested pipeline of the *GPU scene chapter*, shader included. What this chapter adds is the part that joins them: one mesh representation that the operations edit, the picker searches, the card draws and the writer saves.

A session, with the terminal beside the window:

```
added      cube1      12 triangles    8 vertices  volume 1.0000
added      prism2     20 triangles   12 vertices  volume 0.8660
added      sphere3   128 triangles   66 vertices  volume 0.4772
selected   cube1 (triangle 0)
extruded   cube1      20 triangles   12 vertices  volume 1.5000
extruded   cube1      28 triangles   16 vertices  volume 2.0000
selected   sphere3 (triangle 9)
subdivided sphere3   512 triangles  258 vertices  volume 0.5115
scaled     sphere3   512 triangles  258 vertices  volume 0.9989
added      prism4     20 triangles   12 vertices  volume 0.8660
wrote model.obj: 298 vertices, 580 triangles, volume 4.7310
read  model.obj: 298 vertices, 580 triangles, volume 4.7310  ok
```

Every operation prints the object's volume, using the `signed_volume` test from *meshes*. That is the modelling chapter's rule carried into an interactive program: a closed mesh has a volume you can predict, and an operation that breaks the mesh shows up as a number that is wrong.

Controls

Input	Effect
left click	select the object under the pointer, and remember which face was hit
left drag	orbit the camera
right drag	zoom: down moves the camera away, up brings it closer
`-`, `=`	zoom out, zoom in
1, 2, 3	add a cube, a hexagonal prism, a sphere
`E`	extrude the clicked face outward by 0.5
`D`	subdivide the selection
`[`, `]`	scale the selection by 0.8, by 1.25
`R`	turn the selection 15 degrees
arrow keys	move the selection 0.25 across the floor
`F`	orbit around the selection instead of the origin
`Delete`	remove the selection
`W`	write `model.obj` and read it back
`P`	save the current view as `shot1.png`, `shot2.png`, ...
`Escape`	quit

One mesh, four users

Each object keeps its geometry as the *indexed triangle mesh*: an array of positions and a flat list of indices, three per triangle, wound counter-clockwise seen from outside. The operations change that mesh in place; the picker tests rays against it; the writer prints it; and a vertex buffer made from it is what the card draws. Where the object sits in the world is kept apart from the mesh, as a position, a turn about y and a uniform scale, combined into a model matrix when needed.

The program starts with the same header as the *GPU scene chapter*, and the same `foundations.pl` beside it holding the vector and matrix subs from *foundations*, followed by a line `1;`:

```
use strict;
use warnings;
use SDL2 qw(:all);
use Peta::WebGPU;
use SDL2::Image;
BEGIN { require './foundations.pl' }   # the v_* and m_* subs, then 1;

my $max_frames = shift // 0;           # 0: run until the window is closed
my $PI = 4 * atan2(1, 1);
```

`use SDL2 qw(:all)`, not a later `import`, for the reason the *device chapter* spent a section on: every `SDL_*` constant below must exist while the file is compiled.

Building shapes

Three routines come from earlier chapters. `extrude` is *modelling-operations'* unchanged. `subdivide` is that chapter's subdivision written out whole, with the midpoint cache it insists on, and with the sphere projection made optional. `signed_volume` is the *meshes* winding check.

```
# modelling-operations: a convex polygon in the xy plane, counter-
 ↪clockwise
# seen from +z, swept along +z by $h and capped at both ends.
sub extrude {
    my ($poly, $h) = @_;
    my $n = @$poly;
    # The outer parens are load-bearing: `my @v = (A), (B)` parses
 ↪as
    # `(my @v = A), B` and silently stores only the first list.
    my @v = ( (map { [ $_->[0], $_->[1], 0  ] } @$poly),
              (map { [ $_->[0], $_->[1], $h ] } @$poly) );
    my @i;
    # top cap: fan over the upper ring, already CCW seen from +z
    push @i, ($n, $n + $_, $n + $_ + 1) for 1 .. $n - 2;
    # bottom cap: same fan reversed, so it faces -z
    push @i, (0, $_ + 1, $_) for 1 .. $n - 2;
    # sides: one quad per edge, wound outward
    for my $e (0 .. $n - 1) {
        my $f = ($e + 1) % $n;
        push @i, ($e, $f, $n + $f), ($e, $n + $f, $n + $e);
    }
    return (\@v, \@i);
}

# modelling-operations: split every triangle into four at its edge
# midpoints. With $radius, each new vertex is pushed out onto the
 ↪sphere
# of that radius. The mesh is changed in place.
sub subdivide {
    my ($v, $i, $radius) = @_;
    my %mid;
    my $midpoint = sub {
        my ($a, $b) = @_;
        # Cache by edge: the two triangles sharing it must get ONE
 ↪vertex.
        my $key = $a < $b ? "$a,$b" : "$b,$a";
        return $mid{$key} //= do {
            my $m = v_scale(v_add($v->[$a], $v->[$b]), 0.5);
            $m = v_scale(v_norm($m), $radius) if $radius;
            push @$v, $m;
            $#$v;
        };
    };
    my @ni;
```

(continues on next page)

(continued from previous page)

```
    for (my $k = 0; $k < @$i; $k += 3) {
        my ($a, $b, $c) = @{$i}[$k, $k + 1, $k + 2];
        my $ab = $midpoint->($a, $b);
        my $bc = $midpoint->($b, $c);
        my $ca = $midpoint->($c, $a);
        push @ni, ($a,$ab,$ca), ($ab,$b,$bc), ($ca,$bc,$c), ($ab,
↪$bc,$ca);
    }
    @$i = @ni;
}

# meshes: the enclosed volume of a closed mesh, positive when every
# triangle winds counter-clockwise seen from outside.
sub signed_volume {
    my ($v, $i) = @_;
    my $vol = 0;
    for (my $k = 0; $k < @$i; $k += 3) {
        my ($a, $b, $c) = @{$v}[ @{$i}[$k, $k+1, $k+2] ];
        $vol += v_dot($a, v_cross($b, $c)) / 6;
    }
    return $vol;
}
```

The primitives are built from those. A cube is a square extruded, exactly as modelling-operations showed; a hexagonal prism is a hexagon extruded; a sphere is an octahedron subdivided twice with every new vertex pushed onto the sphere.

```
# A prism of unit height whose cross-section has an apothem of 0.5,␣
↪so
# four sides make the unit cube. Centred on the origin, standing on␣
↪y.
sub prism {
    my ($sides) = @_;
    my $r = 0.5 / cos($PI / $sides);
    my @poly = map { my $t = 2 * $PI * ($_ + 0.5) / $sides; [ $r *␣
↪cos $t, $r * sin $t ] }
               0 .. $sides - 1;
    my ($v, $i) = extrude(\@poly, 1);
    # (x, y, z) -> (x, z, -y) is a rotation, so the winding survives␣
↪it.
    return ([ map { [ $_->[0], $_->[2] - 0.5, -$_->[1] ] } @$v ],
↪$i);
}

sub sphere {
    my ($r, $levels) = @_;
    my @v = ([$r,0,0], [-$r,0,0], [0,$r,0], [0,-$r,0], [0,0,$r], [0,
↪0,-$r]);
    my @i = (0,2,4, 2,1,4, 1,3,4, 3,0,4, 2,0,5, 1,2,5, 3,1,5, 0,3,
```

(continues on next page)

(continued from previous page)

```
↪5);
    subdivide(\@v, \@i, $r) for 1 .. $levels;
    return (\@v, \@i);
}
```

`extrude` sweeps along z, and a modeller wants its objects standing on a floor with y up, so `prism` turns the result over. **The turn must be a rotation, not a swap of two axes.** Swapping y and z is a reflection, and a reflection reverses every triangle's winding, the fact the *GPU scene chapter* met as a model rendering inside out. Mapping `(x, y, z)` to `(x, z, -y)` is a quarter turn about x, and the volume check confirms it: the cube reports `1.0000`, not `-1.0000`.

The numbers the program starts with are the analytic ones: the unit cube's 1, the hexagonal prism's `sqrt(3)/2 = 0.8660`, and the level-2 sphere's `0.4772`, which is modelling-operations' `3.81773` for a unit sphere scaled by `0.5 ** 3`.

The window and the pipeline

The window, the device, the surface and the shader are the *GPU scene chapter's*, unchanged. The shader lights each pixel with shading's model and reads each instance's model matrix and colour from a storage buffer.

```
my $win  = SDL2::Window->new(title => 'pperl modeller', w => 800, h
↪=> 600, resizable => 1);
my $gpu  = Peta::WebGPU::Device->new(window => $win);
my $surf = $gpu->surface;
my ($W, $H) = $surf->size;

my $WGSL = <<'WGSL';
struct Scene {
    view_proj: mat4x4<f32>,
    light_dir: vec4<f32>,     // xyz: toward the light, world space
    eye:       vec4<f32>,     // xyz: camera position, world space
};
struct Instance {
    model:  mat4x4<f32>,
    colour: vec4<f32>,
};
@group(0) @binding(0) var<uniform> scene: Scene;
@group(0) @binding(1) var<storage, read> inst: array<Instance>;

struct VOut {
    @builtin(position) pos: vec4<f32>,
    @location(0) world:  vec3<f32>,
    @location(1) normal: vec3<f32>,
    @location(2) colour: vec3<f32>,
};

@vertex
fn vs(@location(0) p: vec3<f32>, @location(1) n: vec3<f32>,
```

(continues on next page)

(continued from previous page)

```
      @builtin(instance_index) i: u32) -> VOut {
    let m = inst[i].model;
    let w = m * vec4<f32>(p, 1.0);
    var o: VOut;
    o.pos    = scene.view_proj * w;
    o.world  = w.xyz;
    o.normal = (m * vec4<f32>(n, 0.0)).xyz;
    o.colour = inst[i].colour.rgb;
    return o;
}

@fragment
fn fs(v: VOut) -> @location(0) vec4<f32> {
    let n = normalize(v.normal);
    let l = normalize(scene.light_dir.xyz);
    let e = normalize(scene.eye.xyz - v.world);
    let lambert = max(dot(n, l), 0.0);
    let h = normalize(l + e);
    let spec = pow(max(dot(n, h), 0.0), 32.0);
    let c = v.colour * (0.1 + 0.9 * lambert) + vec3<f32>(0.4) *␣
 ↪spec;
    return vec4<f32>(min(c, vec3<f32>(1.0)), 1.0);
}
WGSL

my $pipe = $gpu->render_pipeline($gpu->shader($WGSL),
    format => $surf->format, depth => 1, cull => 'back',
    vertex_buffers => [ { attributes => ['float32x3', 'float32x3'] }␣
 ↪]);
my $depth = $gpu->depth_buffer($W, $H);
my $scene = $gpu->uniform_buffer(24 * 4);
```

The shader turns normals with the model matrix, which the GPU scene chapter showed is correct only for rotation, translation and **uniform** scale. That is why `[` and `]` scale all three axes together: a non-uniform scale would need the inverse-transpose matrix passed alongside, and without it the lighting of a stretched object would be quietly wrong.

Flat faces, and the floor as instances

The meshes share vertices between triangles, which the operations need: an extruded face must drag its neighbours' corners with it. The card needs the opposite. A modeller shows its faces flat, so each triangle is lit by its own normal, and a vertex carries one normal, so a corner shared by several faces has to become several vertices. That is the *meshes chapter's* cube with 24 vertices instead of 8, taken to its end: every triangle gets three vertices of its own.

```
# Flat shading: each triangle gets its own three vertices carrying␣
 ↪the
```

(continues on next page)

(continued from previous page)

```
# face normal, so nothing is shared and there is no index buffer.
sub flat_vertices {
    my ($v, $i) = @_;
    my @data;
    for (my $k = 0; $k < @$i; $k += 3) {
        my ($a, $b, $c) = @{$v}[ @{$i}[$k, $k+1, $k+2] ];
        my $n = v_norm(v_cross(v_sub($b, $a), v_sub($c, $a)));
        push @data, @$a, @$n, @$b, @$n, @$c, @$n;
    }
    return $gpu->vertex_buffer(pack 'f*', @data);
}

sub srgb_decode { return [ map { $_ ** 2.2 } @_ ] }   # colours
 ↪picked by eye

# The floor: one quad, drawn 400 times by instancing, as a
 ↪checkerboard.
my $TILES = 20;
my $floor_quad = flat_vertices([[-0.5,0,0.5], [0.5,0,0.5], [0.5,0,-
 ↪0.5], [-0.5,0,-0.5]],
                               [0, 1, 2,  0, 2, 3]);
my @TILE_COLOUR = (srgb_decode(0.30, 0.30, 0.33), srgb_decode(0.38,
 ↪0.38, 0.41));
my @tiles;
for my $k (0 .. $TILES * $TILES - 1) {
    my ($tx, $tz) = ($k % $TILES, int($k / $TILES));
    push @tiles, @{ m_translate($tx - $TILES / 2 + 0.5, 0, $tz -
 ↪$TILES / 2 + 0.5) },
                 @{ $TILE_COLOUR[($tx + $tz) % 2] }, 1;
}
my $floor = $gpu->storage_buffer(pack 'f*', @tiles);
```

With nothing shared, an index buffer would only count `0, 1, 2, ...`, so the objects are drawn with `draw`, not `draw_indexed`.

Flat shading is also what makes the operations visible: a subdivided sphere shows its new, smaller facets at once.

Instancing is used where one mesh repeats, and in this program that is the floor: one quad and 400 instances, each a translation and one of two greys, drawn by a single `draw(6, 400)`. The storage buffer has the layout the shader's `Instance` expects, 20 floats each, as in the *GPU scene chapter*. The objects are the opposite case, each with its own mesh, so each is one draw with its own one-instance buffer, and the same shader serves both.

The colours are picked by eye and decoded with the power 2.2, because the window's surface is an sRGB format, as the *device chapter* measured.

Objects

```
my %COLOUR = (cube   => srgb_decode(0.55, 0.65, 0.80),
              prism  => srgb_decode(0.60, 0.78, 0.55),
              sphere => srgb_decode(0.85, 0.80, 0.60));
my $HIGHLIGHT = srgb_decode(1.00, 0.55, 0.10);

my (@objects, $sel, $sel_tri);
my $serial = 0;

sub model_matrix {
    my ($o) = @_;
    # Read right to left: scale, then turn, then place.
    return m_mul(m_translate(@{ $o->{pos} }),
                 m_mul(m_rotate_y($o->{yaw}), m_scale(($o->{scale})
 ↪x 3)));
}

# Rebuild what the card and the picker hold after the mesh changed.
sub refresh {
    my ($o) = @_;
    my ($v, $i) = @{ $o->{mesh} }{qw(v i)};
    $o->{vb}    = flat_vertices($v, $i);
    $o->{count} = @$i;
    # Bounding sphere in object space: the box's centre, the
 ↪farthest vertex.
    my @lo = @{ $v->[0] };
    my @hi = @lo;
    for my $p (@$v) {
        for (0 .. 2) {
            $lo[$_] = $p->[$_] if $p->[$_] < $lo[$_];
            $hi[$_] = $p->[$_] if $p->[$_] > $hi[$_];
        }
    }
    $o->{centre} = v_scale(v_add(\@lo, \@hi), 0.5);
    $o->{radius} = 0;
    for my $p (@$v) {
        my $d = v_len(v_sub($p, $o->{centre}));
        $o->{radius} = $d if $d > $o->{radius};
    }
}

sub report {
    my ($what, $o) = @_;
    my ($v, $i) = @{ $o->{mesh} }{qw(v i)};
    printf "%-10s %-8s %4d triangles %4d vertices  volume %.4f\n",
 ↪$what, $o->{name},
        @$i / 3, scalar @$v, signed_volume($v, $i) * $o->{scale} **
 ↪3;
}
```

(continues on next page)

(continued from previous page)

```
sub add_object {
    my ($kind) = @_;
    my ($v, $i) = $kind eq 'cube'   ? prism(4)
                : $kind eq 'prism' ? prism(6)
                :                    sphere(0.5, 2);
    my $n = $serial++;
    my $o = { name => $kind . ($n + 1), mesh => { v => $v, i => $i }
↪,
              pos => [ 1.6 * ($n % 5 - 1), 0.5, -1.6 * int($n / 5)␣
↪],
              yaw => 0, scale => 1, colour => $COLOUR{$kind},
              round => $kind eq 'sphere',
              inst => $gpu->storage_buffer(20 * 4) };
    refresh($o);
    push @objects, $o;
    ($sel, $sel_tri) = ($o, undef);
    report('added', $o);
}
```

`model_matrix` is the *foundations* order rule applied once: `T * R * S` scales first, then turns, then places, so an object turns about its own centre wherever it stands. Written `R * T * S` it would swing round the origin instead.

`refresh` is the one place a mesh change is made visible. It rebuilds the vertex buffer and recomputes the bounding sphere the picker tests first, the *meshes chapter's* cheap approximation. Every operation calls it; forgetting to is the bug where the picture and the clicks disagree.

The volume `report` prints is multiplied by the cube of the scale, because `signed_volume` measures the mesh in its own space and the scale lives in the model matrix.

New objects are placed along a row of slots so they do not appear inside one another, and each new object becomes the selection.

The camera

The orbit camera from *scene-graphs*, unchanged, and the per-frame upload of the GPU scene chapter:

```
my $TARGET = [0, 0.5, 0];
my ($yaw, $pitch, $dist) = (0.6, 0.45, 7);
my $FOVY = 45;
my $GL_TO_WGPU = [1,0,0,0,  0,1,0,0,  0,0,0.5,0,  0,0,0.5,1];

# scene-graphs: two angles and a distance around a target.
sub orbit_eye {
    # $yaw and $pitch in radians, $dist from the target. Clamp pitch
    # short of +/- 90 degrees: at exactly vertical the view␣
↪direction
    # is parallel to `up` and look_at degenerates.
```

(continues on next page)

(continued from previous page)

```
    my ($target, $yaw, $pitch, $dist) = @_;
    return [ $target->[0] + $dist * cos($pitch) * sin($yaw),
             $target->[1] + $dist * sin($pitch),
             $target->[2] + $dist * cos($pitch) * cos($yaw) ];
}

sub upload {
    my $eye  = orbit_eye($TARGET, $yaw, $pitch, $dist);
    my $proj = m_mul($GL_TO_WGPU, m_perspective($FOVY, $W / $H, 0.1,
↪100));
    my $vp   = m_mul($proj, m_look_at($eye, $TARGET, [0, 1, 0]));
    # The light orbits with the camera, high and off to one side, so
↪the
    # faces in view are always lit and a box's three show three
↪shades.
    my $light = v_sub(orbit_eye($TARGET, $yaw + 0.7, 1.0, 1),
↪$TARGET);
    $scene->write(pack 'f*', @$vp, @$light, 0, @$eye, 1);
    for my $o (@objects) {
        my $colour = $sel && $o == $sel ? $HIGHLIGHT : $o->{colour};
        $o->{inst}->write(pack 'f*', @{ model_matrix($o) }, @
↪$colour, 1);
    }
}

sub draw {
    my ($target) = @_;
    my $pass = $gpu->render_pass($target, clear => [0.05, 0.05, 0.
↪08, 1], depth => $depth)
                   ->pipeline($pipe);
    $pass->bind([$scene, $floor])->vertices($floor_quad)->draw(6,
↪$TILES * $TILES);
    $pass->bind([$scene, $_->{inst}])->vertices($_->{vb})->draw($_->
↪{count}) for @objects;
    $pass->end;
}
```

`$GL_TO_WGPU` is the depth-range conversion from the *GPU scene chapter*; without it the near half of the view volume is clipped away and anything close to the camera vanishes.

The light is placed with the same `orbit_eye`, 0.7 radians round from the camera and 1 radian up, at distance 1, so subtracting the target leaves a unit vector. A light fixed in the world leaves the far side of every object at the ambient level once you orbit round to it. This one moves with the camera, so the faces in view are always lit, and because it stands off to one side and above, a box's three visible faces get three different Lambert terms: from the starting view, 0.84 for the top, 0.52 for the side facing +x and 0.14 for the side facing +z.

The selection is a colour, nothing more. `upload` writes each object's instance buffer every frame, and the selected object's gets the highlight instead of its own colour. The mesh and the pipeline are untouched, so selecting costs one 80-byte write.

Every object has its own instance buffer and binds it before its own draw. One buffer rewritten between draws would put every object where the last one is, the symptom the GPU scene chapter measured.

`draw` takes its target as an argument, so the same routine draws into a frame for the window and into a texture for a screenshot.

Clicking on an object

A click becomes a ray through the pixel, exactly as *scene-graphs* builds it, and the ray is tested against every object's bounding sphere, then against the triangles of the objects whose sphere it passes through. `pick_ray` and `ray_sphere` are that chapter's, unchanged; `tan_half` is the tangent of half the vertical field of view, given in degrees as `m_perspective` takes it.

```
# scene-graphs: a ray through the centre of pixel ($sx, $sy).
sub tan_half { my $h = $_[0] * $PI / 360; return sin($h) / cos($h) }

sub pick_ray {
    my ($eye, $target, $up, $fovy, $aspect, $sx, $sy, $W, $H) = @_;
    my $f = v_norm(v_sub($target, $eye));
    my $s = v_norm(v_cross($f, $up));
    my $u = v_cross($s, $f);
    my $ndc_x = 2 * ($sx + 0.5) / $W - 1;
    my $ndc_y = 1 - 2 * ($sy + 0.5) / $H;        # screen y is down
    my $th = tan_half($fovy);
    my @d = map {
        $f->[$_] + $s->[$_] * $ndc_x * $th * $aspect + $u->[$_] *
 ↪$ndc_y * $th
    } 0..2;
    return ($eye, v_norm(\@d));
}

sub ray_sphere {
    # Returns the near intersection distance, or undef. $d MUST be
    # normalised or $t is in units of $d's length rather than␣
 ↪distance.
    my ($o, $d, $c, $r) = @_;
    my $oc = v_sub($o, $c);
    my $b = 2 * v_dot($oc, $d);
    my $cc = v_dot($oc, $oc) - $r * $r;
    my $disc = $b*$b - 4*$cc;
    return undef if $disc < 0;
    my $t = (-$b - sqrt($disc)) / 2;
    return $t > 0 ? $t : undef;
}

# Möller-Trumbore: the distance along the ray to triangle ($a, $b,
 ↪$c),
# or undef. Both sides count; the nearest hit decides.
sub ray_triangle {
```

(continues on next page)

(continued from previous page)

```
    my ($o, $d, $a, $b, $c) = @_;
    my $e1  = v_sub($b, $a);
    my $e2  = v_sub($c, $a);
    my $p   = v_cross($d, $e2);
    my $det = v_dot($e1, $p);
    return undef if abs($det) < 1e-12;          # ray parallel to
↪the plane
    my $s = v_sub($o, $a);
    my $u = v_dot($s, $p) / $det;
    return undef if $u < 0 || $u > 1;
    my $q = v_cross($s, $e1);
    my $w = v_dot($d, $q) / $det;
    return undef if $w < 0 || $u + $w > 1;
    my $t = v_dot($e2, $q) / $det;
    return $t > 1e-9 ? $t : undef;
}
```

`ray_triangle` is the Möller-Trumbore test that *modelling-operations* named and left elided. It solves for the hit distance `t` and the barycentric weights `u` and `w` at once; the ray hits the triangle when both weights and their sum lie between 0 and 1, the same inside test the *rasteriser* applies to a pixel centre.

The picker needs the triangle, not only the object. Extrude acts on the face you clicked, and which face that was is exactly what the nearest triangle hit says. That is why picking runs here in Perl against the very meshes the operations edit.

```
sub pick {
    my ($sx, $sy) = @_;
    my ($o, $d) = pick_ray(orbit_eye($TARGET, $yaw, $pitch, $dist),
↪$TARGET, [0, 1, 0],
                           $FOVY, $W / $H, $sx, $sy, $W, $H);
    my ($best, $best_t, $best_tri);
    # The floor is drawn, so it hides whatever is below it. It is
↪not
    # selectable, but a ray that reaches it first has hit nothing.
    if ($o->[1] > 0 && $d->[1] < 0) {
        my $t = -$o->[1] / $d->[1];
        $best_t = $t if abs($o->[0] + $t * $d->[0]) <= $TILES / 2
                     && abs($o->[2] + $t * $d->[2]) <= $TILES / 2;
    }
    for my $obj (@objects) {
        my $m = model_matrix($obj);
        my $c = [ @{ xform($m, [ @{ $obj->{centre} }, 1 ]) }[0 .. 2]
↪];
        my $r = $obj->{radius} * $obj->{scale};
        # Bounding sphere first. ray_sphere ignores hits behind the
↪eye,
        # so an eye INSIDE the sphere must skip the test, not fail
↪it.
        next unless v_len(v_sub($o, $c)) < $r || defined ray_sphere(
```

(continues on next page)

(continued from previous page)

```
↪$o, $d, $c, $r);
        my @w = map { [ @{ xform($m, [ @$_, 1 ]) }[0 .. 2] ] } @{
↪$obj->{mesh}{v} };
        my $i = $obj->{mesh}{i};
        for (my $k = 0; $k < @$i; $k += 3) {
            my $t = ray_triangle($o, $d, @w[ @{$i}[$k, $k+1, $k+2]␣
↪]) // next;
            ($best, $best_t, $best_tri) = ($obj, $t, $k / 3)
                if !defined $best_t || $t < $best_t;
        }
    }
    return ($best, $best_tri);
}
```

The ray is built from the same eye, target, field of view and aspect ratio that `upload` gives the card, so the ray through a pixel is the line of sight the card rendered through it. That is checkable. Rendering each object alone into a texture shows which objects cover each pixel; rendering all of them shows which one is in front. Over a grid of 30000 pixels of the session above, the picker returned nothing wherever no object was drawn, and the front object wherever one was. From a view along the row, 669 of 13400 sampled pixels showed two or more objects overlapping, and it returned the front one at every one of them.

Symptom: clicking the floor beside an object selects the object. Cause: the picker ignores something the renderer draws. The sphere scaled up about its centre sinks below the floor; the floor hides that part, but a ray that passes through the floor still reaches it. Without the floor test above, 3 of those 30000 pixels pick the sphere where the frame shows only floor. The rule is general: the picker must test everything that is drawn, even what cannot be selected, because what cannot be selected can still be in the way. Below the floor the test does not apply, because from there the floor is a back face and culled.

Symptom: an object cannot be clicked once the camera is close to it. Cause: the eye is inside the object's bounding sphere, where `ray_sphere` answers undef because its only hit is behind the eye. That check is right for the question scene-graphs asked, which was where a ray first meets a sphere; the bounding test asks a different one, whether the ray can reach anything inside. A prism scaled five times and orbited from 2 units away put the eye 2.000 from the centre of a sphere of radius 2.331: `ray_sphere` said undef, and `pick` still found the prism.

For a handful of objects the linear loop over objects is fine; a scene of thousands would put the bounding spheres into the *BVH* that chapter builds.

Push and pull: extruding a face

Modelling-operations' `extrude` sweeps a polygon into a solid. A modeller needs the same operation applied to a face of a solid that already exists, which is the push/pull that chapter called the most useful operation in the set.

A click hits one triangle, but a face of a cube is two triangles, and of a subdivided cube eight. The face the click means is the hit triangle and everything connected to it across shared edges that lies in the same plane:

```
sub tri_normal {
    my ($v, $i, $t) = @_;
    my ($a, $b, $c) = @{$v}[ @{$i}[3 * $t .. 3 * $t + 2] ];
    return v_norm(v_cross(v_sub($b, $a), v_sub($c, $a)));
}

# The face a click means: the hit triangle and every triangle
↪reachable
# from it across shared edges that lies in the same plane.
sub face_of {
    my ($v, $i, $t0) = @_;
    my $edges_of = sub {                        # triangle $t's three
↪undirected edges
        my @c = @{$i}[3 * $_[0] .. 3 * $_[0] + 2];
        return map { my ($p, $q) = ($c[$_], $c[($_ + 1) % 3]); $p <
↪$q ? "$p,$q" : "$q,$p" } 0 .. 2;
    };
    my %by_edge;
    for my $t (0 .. @$i / 3 - 1) { push @{ $by_edge{$_} }, $t for
↪$edges_of->($t) }
    my $n = tri_normal($v, $i, $t0);
    my %in = ($t0 => 1);
    my @todo = ($t0);
    while (defined(my $t = shift @todo)) {
        for my $u (map { @{ $by_edge{$_} } } $edges_of->($t)) {
            next if $in{$u} || v_dot(tri_normal($v, $i, $u), $n) < 1
↪- 1e-9;
            $in{$u} = 1;
            push @todo, $u;
        }
    }
    return sort { $a <=> $b } keys %in;
}

# Push/pull: move a face along its normal and wall in the gap. The
↪face's
# triangles are re-pointed at moved copies of its vertices; every
↪boundary
# edge a->b gains the quad a, b, b', a', wound like extrude's sides.
sub extrude_face {
    my ($o, $t0, $h) = @_;
    my ($v, $i) = @{ $o->{mesh} }{qw(v i)};
    my @face = face_of($v, $i, $t0);
    my $n = tri_normal($v, $i, $t0);
    my (@edges, %directed);
    for my $t (@face) {
        my @c = @{$i}[3 * $t .. 3 * $t + 2];
        for (0 .. 2) {
            push @edges, [ $c[$_], $c[($_ + 1) % 3] ];
            $directed{"$c[$_],$c[($_ + 1) % 3]"} = 1;
```

(continues on next page)

(continued from previous page)

```
        }
    }
    my %moved;
    for my $t (@face) {
        for my $k (3 * $t .. 3 * $t + 2) {
            my $old = $i->[$k];
            $moved{$old} //= do { push @$v, v_add($v->[$old], v_
↪scale($n, $h)); $#$v };
            $i->[$k] = $moved{$old};
        }
    }
    for my $e (@edges) {
        my ($a, $b) = @$e;
        next if $directed{"$b,$a"};            # inside the face:␣
↪both sides moved
        push @$i, ($a, $b, $moved{$b}), ($a, $moved{$b}, $moved{$a}
↪);
    }
    $o->{round} = 0;                           # no longer a sphere␣
↪to project onto
}
```

`face_of` is a flood fill over the edge adjacency that the *meshes chapter* says a half-edge structure keeps ready; here it is rebuilt from the index list on each call, which for a few hundred triangles is instant. The same-plane test compares normals, and connectivity across shared edges keeps two separate faces that happen to be parallel, such as the tops of two steps, apart.

The extrusion itself is the *manifold* invariant used as a recipe. An edge of the face whose reverse is also in the face is interior: both triangles beside it move, and nothing needs adding. An edge whose reverse is not in the face is boundary: the triangle beyond it stays behind, so the gap between the old edge and the moved one needs a wall. The wall quad takes the edge in the face's own direction, `a` to `b`, because its neighbour beyond runs it `b` to `a`; along the top it runs `b'` to `a'`, opposite to the moved face. Every shared edge is then traversed once each way, which is exactly the condition for a closed, consistently wound mesh. The same reasoning gives `extrude`'s side quads their order.

The numbers say whether it worked. The cube's top is two triangles with four boundary edges, so an extrusion adds 4 vertices and 8 triangles and, at height 0.5, half a unit of volume:

```
extruded   cube1      20 triangles   12 vertices  volume 1.5000
extruded   cube1      28 triangles   16 vertices  volume 2.0000
```

The second `E` extrudes the same face again, because the face's triangles keep their numbers when they move, so the remembered click still names them. On a subdivided cube the top is eight triangles and nine vertices, and one extrusion took it from 48 triangles and 26 vertices to 64 and 35, volume 1.0000 to 1.5000. Extruding a single triangle of the sphere grew its volume by `0.009508`, which is that triangle's area times 0.5 to the sixth decimal. After each of these, every edge of every mesh was shared by exactly two triangles and no directed edge occurred twice.

Subdividing is modelling-operations' routine applied to the selection. A sphere keeps its round flag, so its new vertices go onto the sphere and it becomes rounder: `D` on the level-2 sphere gave 512 triangles, 258 vertices and volume `0.5115`, modelling-operations' level-3 `4.09160` scaled by `0.5 ** 3`. Any other object is split flat, adding triangles without changing its shape or its volume, which is the preparation for extruding a smaller face. Subdividing renumbers every triangle, so it forgets the clicked face; `E` then asks for a new click.

Saving to OBJ

```
# model-formats: read_obj, unchanged.
sub read_obj {
    my ($path) = @_;
    open my $fh, '<', $path or die "$path: $!";
    my (@v, @f);
    while (my $line = <$fh>) {
        next if $line =~ /^\s*(#|$)/;
        my @tok = split ' ', $line;
        my $kw = shift @tok;
        if ($kw eq 'v') { push @v, [ @tok[0..2] ] }
        elsif ($kw eq 'f') {
            # A face may have MORE than three vertices. Fan-
 ↪triangulate.
            # Each token may be v, v/vt, v//vn or v/vt/vn: take the␣
 ↪first.
            my @vi = map { my ($n) = split m{/}, $_; $n } @tok;
            # Indices are 1-based; a NEGATIVE index counts back from␣
 ↪the
            # current end of the vertex list.
            @vi = map { $_ < 0 ? @v + $_ : $_ - 1 } @vi;
            push @f, [ $vi[0], $vi[$_], $vi[$_ + 1] ] for 1 .. $#vi␣
 ↪- 1;
        }
    }
    close $fh;
    return (\@v, \@f);
}

# model-formats' write_obj, for a whole scene: one `o` line per␣
 ↪object,
# vertices baked into world space, face indices counted across the␣
 ↪file.
sub save_obj {
    my ($path) = @_;
    open my $fh, '>', $path or die "$path: $!";
    print $fh "# pperl modeller, ", scalar @objects, " objects\n";
    my $base = 0;
    for my $o (@objects) {
        my $m = model_matrix($o);
        my ($v, $i) = @{ $o->{mesh} }{qw(v i)};
```

(continues on next page)

(continued from previous page)

```
        print $fh "o $o->{name}\n";
        printf $fh "v %.6f %.6f %.6f\n", @{ xform($m, [ @$_, 1 ]) }
↪[0 .. 2] for @$v;
        for (my $k = 0; $k < @$i; $k += 3) {
            # OBJ indices are 1-BASED, and count every vertex␣
↪written so far
            printf $fh "f %d %d %d\n", map { $base + $_ + 1 } @{$i}[
↪$k, $k+1, $k+2];
        }
        $base += @$v;
    }
    close $fh;

    my ($nv, $nt, $vol) = (0, 0, 0);
    for my $o (@objects) {
        my ($v, $i) = @{ $o->{mesh} }{qw(v i)};
        $nv += @$v;
        $nt += @$i / 3;
        $vol += signed_volume($v, $i) * $o->{scale} ** 3;
    }
    my ($rv, $rf) = read_obj($path);
    my $rvol = signed_volume($rv, [ map { @$_ } @$rf ]);
    printf "wrote %s: %d vertices, %d triangles, volume %.4f\n",
↪$path, $nv, $nt, $vol;
    printf "read  %s: %d vertices, %d triangles, volume %.4f  %s\n",␣
↪$path,
        scalar @$rv, scalar @$rf, $rvol,
        (@$rv == $nv && @$rf == $nt && abs($rvol - $vol) < 1e-4 ?
↪'ok' : 'MISMATCH');
}
```

OBJ has no scene tree, as the *model-formats* table says, so each object's model matrix is applied to its vertices on the way out and the file holds world positions. An `o` line names each object; readers that understand it keep the objects apart, and `read_obj` skips it like any other line it does not know.

Face indices count across the whole file, not per object. The second object's first vertex is number 9 if the first object wrote 8, which is what `$base` carries. Restart the count per object and every object after the first is built from the first one's vertices: the counts still match, and only the volume gives it away. That is why the check reads the file back with model-formats' own reader and compares three numbers, not two. The volume of everything read back must equal the sum of the objects' volumes, and it can only do so if every face refers to the vertices it was written with.

The session at the top wrote four objects in one file, 298 vertices and 580 triangles, and read back the same counts and the same volume, `4.7310`.

Screenshots

P renders the current view into a texture and writes it as a PNG: the checking path of the *device chapter*, in the program.

```
my $shots = 0;
sub screenshot {
    my $tex = $gpu->texture(width => $W, height => $H, format =>
↪$surf->format);
    upload();
    draw($tex);
    # read returns the texture's own byte order, B, G, R, A for a␣
↪bgra
    # format; the surface is told which, so nothing is swapped in␣
↪Perl.
    my $order = $surf->format =~ /^bgra/ ? SDL_PIXELFORMAT_BGRA32 :␣
↪SDL_PIXELFORMAT_RGBA32;
    my $file  = sprintf 'shot%d.png', ++$shots;
    SDL2::Image::save_png(SDL2::Surface->from_pixels($tex->read, $W,␣
↪$H, $order), $file) == 0
        or die "$file: ", SDL_GetError(), "\n";
    print "saved $file\n";
}
```

The texture has the surface's format, so the pipeline built for the window draws into it unchanged, and the depth buffer, already the window's size, fits it. On the machine this was checked on that format is `bgra8unorm-srgb`, whose bytes come back as B, G, R, A. Rather than swap them in Perl, the surface is created with `SDL_PIXELFORMAT_BGRA32`, which names that byte order, and `SDL2::Image` writes the file.

The screenshot runs inside the frame loop, so it has to be quick. The device chapter's `save_png` sets every pixel from Perl through `GD`, which is the right tool for checking an image once and the wrong one here: while it runs, the loop neither polls events nor presents frames, and the window stops responding until it finishes. Handing the whole buffer to `SDL2::Image` is one call.

Keys and the mouse

The keyboard is a table from key to action:

```
sub with_selection {
    my ($code) = @_;
    return print "nothing selected\n" unless $sel;
    $code->($sel);
}

my %KEY = (
    SDLK_1() => sub { add_object('cube') },
    SDLK_2() => sub { add_object('prism') },
    SDLK_3() => sub { add_object('sphere') },
    SDLK_e() => sub {
```

(continues on next page)

(continued from previous page)

```
        with_selection(sub {
            return print "click a face of $_[0]{name} first\n"
↪unless defined $sel_tri;
            extrude_face($_[0], $sel_tri, 0.5);
            refresh($_[0]);
            report("extruded", $_[0]);
        });
    },
    SDLK_d() => sub {
        with_selection(sub {
            subdivide($_[0]{mesh}{v}, $_[0]{mesh}{i}, $_[0]{round} ?
↪0.5 : undef);
            $sel_tri = undef;                   # triangle numbers
↪have changed
            refresh($_[0]);
            report("subdivided", $_[0]);
        });
    },
    SDLK_RIGHTBRACKET() => sub { with_selection(sub { $_[0]{scale}
↪*= 1.25; report('scaled', $_[0]) }) },
    SDLK_LEFTBRACKET()  => sub { with_selection(sub { $_[0]{scale}
↪*= 0.8;  report('scaled', $_[0]) }) },
    SDLK_r()     => sub { with_selection(sub { $_[0]{yaw} += $PI /
↪12 }) },
    SDLK_LEFT()  => sub { with_selection(sub { $_[0]{pos}[0] -= 0.25
↪}) },
    SDLK_RIGHT() => sub { with_selection(sub { $_[0]{pos}[0] += 0.25
↪}) },
    SDLK_UP()    => sub { with_selection(sub { $_[0]{pos}[2] -= 0.25
↪}) },
    SDLK_DOWN()  => sub { with_selection(sub { $_[0]{pos}[2] += 0.25
↪}) },
    SDLK_f()     => sub { with_selection(sub { $TARGET = [ @{ $_[0]
↪{pos} } ] }) },
    SDLK_DELETE() => sub {
        with_selection(sub {
            @objects = grep { $_ != $_[0] } @objects;
            print "deleted $_[0]{name}\n";
            ($sel, $sel_tri) = ();
        });
    },
    SDLK_EQUALS() => sub { $dist = $dist * 0.8 < 2 ? 2 : $dist * 0.8
↪},
    SDLK_MINUS()  => sub { $dist = $dist * 1.25 > 40 ? 40 : $dist *
↪1.25 },
    SDLK_w() => sub { save_obj('model.obj') },
    SDLK_p() => sub { screenshot() },
);
```

The `()` after each constant is required. `=>` quotes a bareword on its left, so `SDLK_1`

`=> sub { ... }` makes the key the string `"SDLK_1"`, not 49. It compiles under `use strict`, because a quoted word is not a bareword, and the key simply never matches. It is the *import trap* in another form, and this time strict does not catch it.

Moving and turning change only the model matrix, which `upload` sends every frame, so they need no `refresh`. The operations that change the mesh call it.

The mouse needs one decision: a left press may be the start of a click or of a drag. The handler counts how far the pointer moves while the button is down, and only a release after less than 4 pixels of movement is a click:

```
my ($dragged, $running) = (0, 1);
sub handle {
    my ($ev) = @_;
    my $type = $ev->{type};
    if ($type == SDL_QUIT) { $running = 0 }
    elsif ($type == SDL_KEYDOWN) {
        if ($ev->{sym} == SDLK_ESCAPE) { $running = 0 }
        elsif (my $action = $KEY{ $ev->{sym} }) { $action->() }
    }
    elsif ($type == SDL_MOUSEBUTTONDOWN && $ev->{button} == SDL_
↪BUTTON_LEFT) { $dragged = 0 }
    elsif ($type == SDL_MOUSEMOTION && $ev->{state} & SDL_BUTTON_
↪LMASK) {
        $dragged += abs($ev->{xrel}) + abs($ev->{yrel});
        $yaw   -= 0.01 * $ev->{xrel};
        $pitch += 0.01 * $ev->{yrel};
        $pitch = 1.4 if $pitch > 1.4;
        $pitch = -1.4 if $pitch < -1.4;
    }
    elsif ($type == SDL_MOUSEMOTION && $ev->{state} & SDL_BUTTON_
↪RMASK) {
        $dist *= exp(0.01 * $ev->{yrel});
        $dist = 2 if $dist < 2;
        $dist = 40 if $dist > 40;
    }
    elsif ($type == SDL_MOUSEBUTTONUP && $ev->{button} == SDL_
↪BUTTON_LEFT && $dragged < 4) {
        ($sel, $sel_tri) = pick($ev->{x}, $ev->{y});
        print $sel ? "selected   $sel->{name} (triangle $sel_tri)\n"␣
↪: "selected   nothing\n";
    }
    elsif ($type == SDL_WINDOWEVENT && $ev->{event} == SDL_
↪WINDOWEVENT_SIZE_CHANGED) {
        ($W, $H) = @$ev{qw(data1 data2)};
        $surf->configure($W, $H);
        $depth = $gpu->depth_buffer($W, $H);   # must match the␣
↪frame
    }
}
```

The held buttons arrive in the motion event's `state`, as the *device chapter* showed, so

the handler needs no memory of which button went down. Dragging right moves the eye the other way round, so the scene turns with the pointer as if held. The pitch clamp at 1.4 radians keeps the eye short of vertical, where *scene-graphs* warns that `m_look_at` degenerates. Zoom by dragging is multiplicative, `exp` of the movement, so the same drag changes the view by the same proportion near and far: 50 pixels down multiplies the distance by `e ** 0.5 = 1.6487`.

A click on nothing clears the selection, which is how to deselect.

The resize branch is the GPU scene chapter's: after `$win->set_size(640, 480)` the handler reconfigured the surface to `640x480`, made a matching depth buffer, and frames carried on drawing.

The frame loop

```
add_object($_) for qw(cube prism sphere);
($sel, $sel_tri) = ();

my $frames = 0;
while ($running) {
    while (my $ev = SDL_PollEvent()) { handle($ev) }
    my $frame = $surf->next_frame or next;
    upload();
    draw($frame);
    $frame->present;
    $running = 0 if $max_frames && ++$frames >= $max_frames;
}
```

Nothing in a modeller moves by itself, so unlike the GPU scene chapter's cubes there is no simulation clock and no fixed step: every change is the answer to an event, and the frame shows the state after the events it drained. The surface's default `fifo` mode paces the loop to the display, as the *device chapter* measured.

Every pass of the loop must reach `present` quickly. The window shows only what the loop presents, and the window system judges the program alive by whether it keeps taking events off the queue. Anything that holds the loop for seconds, such as a slow screenshot or a long computation in a key handler, leaves the window frozen on its last frame and, after a while, marked as not responding. Work that takes that long belongs outside the frame loop or split across frames.

Running it

The blocks above, in order, are the whole program. Save them as `modeller.pl` beside `foundations.pl` and run:

```
pperl modeller.pl
```

`pperl modeller.pl 600` stops after 600 frames, for unattended runs. It needs a binary built with `--features webgpu`, a usable adapter, and an X11 or Wayland display for the window.

The program was checked with input pushed onto SDL's event queue while its frame loop ran: a click on the cube's top selected `cube1` at triangle 0, the top cap; two E presses

extruded it; a click on the sphere, `D` and `]` subdivided and scaled it; a drag of 60 pixels turned the camera 0.6 radians without selecting anything; a click on empty sky cleared the selection; `W` wrote and re-read the file. The window's own contents, captured from the screen as it ran, showed each step: the orange highlight moving from the cube to the sphere, the column rising from the cube, the finer facets of the sphere, and the view turning. The PNG that `P` wrote was identical, in all 480000 pixels, to the window's contents captured at the same moment.

What this demonstrates

Every chapter of the guide appears in one program, and the program works because they agree on one representation: counter-clockwise triangles in an indexed mesh, column-major matrices, a right-handed world with y up. The operations, the picker, the renderer and the writer each read that representation, and none of them converts it for another.

The checks carry over as well. The volume after each operation, the pick compared against the rendered pixels, and the file read back and compared against the scene are the same kind of test the earlier chapters used: a number known in advance, computed a second way.

Capstone: a 3D game

A complete small game: a craft flies down a lane, blocks stand in its way, gems hang between them, and the lane gets faster the longer the run lasts. The arrow keys steer, each gem scores, and touching a block ends the run. It is the *2D game* with a third dimension, and it keeps that game's shape: the world is plain data, an update step moves it, a draw step shows it, and a loop runs both at a fixed rate.

Nothing in it is new. Every piece was built in an earlier chapter, and this one puts them together:

Piece	Taught in
vectors, matrices, `m_look_at`, `m_perspective`	*foundations*
boxes and spheres as bounding volumes	*meshes*
the node hierarchy and `walk`	*scene graphs*
the window, the surface, the fixed step with its cap	*the device chapter*
the pipeline, vertex and index buffers	*the pipeline chapter*
the cube, depth, instancing, per-fragment light	*the GPU scene chapter*
`key_pressed` and scancodes	*the 2D game*

The blocks of code from here to the loop, in order, are the whole program. It loads the subs from foundations from `foundations.pl` beside it, exactly as the *GPU scene chapter* does.

The world is plain data

```
use strict;
use warnings;
```

(continues on next page)

(continued from previous page)

```
use SDL2 qw(:all);
use Peta::WebGPU;
BEGIN { require './foundations.pl' }   # the v_* and m_* subs, then 1;

use constant { STEP => 16, MAX_STEPS => 5 };   # milliseconds, steps per frame
use constant { LANE => 4, AHEAD => 60, N_BLOCKS => 24, N_GEMS => 8 };
my $max_frames = shift // 0;          # 0: play until the window is closed

my %game;

sub new_game {
    %game = (
        craft  => { x => 0, z => 0, vx => 0 },
        speed  => 0.12,                  # units per step, along -z
        blocks => [ map { place_block({}, -12 - $_ * AHEAD / N_BLOCKS) } 0 .. N_BLOCKS - 1 ],
        # -13.25 is halfway between the first two rows of blocks.
        gems   => [ map { place_gem({}, -13.25 - $_ * AHEAD / N_GEMS) } 0 .. N_GEMS - 1 ],
        score  => 0,
        over   => 0,
        steps  => 0,
    );
}

sub place_block {
    my ($b, $z) = @_;
    my $w = 0.8 + rand 0.8;
    %$b = (x => $w / 2 - LANE + rand(2 * LANE - $w), z => $z, w => $w, h => 0.8 + rand 1.2);
    return $b;
}

sub place_gem {
    my ($g, $z) = @_;
    %$g = (x => 0.5 - LANE + rand(2 * LANE - 1), z => $z, taken => 0);
    return $g;
}
```

The craft flies toward -z, the direction a camera looks in *foundations*, and steers along x. The lane runs from `x = -4` to `x = 4`, with the floor at `y = 0`.

The field looks endless and is not. There are always 24 blocks, one row every 2.5 units over the 60 units ahead, and 8 gems, one every 7.5 units, each halfway between two rows of blocks so that no gem is ever inside a block. When one falls behind the camera,

`update` moves it 60 units further on and gives it a new position and size. **The number of objects never changes**, which is what lets the draw step below fill one buffer of fixed size every frame.

`rand` makes each run different. Calling `srand` with a fixed number before `new_game` makes it the same every time, which is what the checks at the end of this chapter do.

Collision: boxes and a sphere

```
# Boxes are [min corner], [max corner]: the axis-aligned form from meshes.
sub craft_box {
    my $c = $game{craft};
    return ([ $c->{x} - 0.8, 0.3, $c->{z} - 0.5 ], [ $c->{x} + 0.8, 0.75, $c->{z} + 0.5 ]);
}

sub block_box {
    my ($b) = @_;
    my $r = $b->{w} / 2;
    return ([ $b->{x} - $r, 0, $b->{z} - $r ], [ $b->{x} + $r, $b->{h}, $b->{z} + $r ]);
}

sub boxes_overlap {
    my ($alo, $ahi, $blo, $bhi) = @_;
    for my $i (0 .. 2) {
        # A gap along any one axis separates them; they touch only when
        # they overlap along all three.
        return 0 if $ahi->[$i] < $blo->[$i] || $bhi->[$i] < $alo->[$i];
    }
    return 1;
}

sub sphere_touches_box {
    my ($c, $r, $lo, $hi) = @_;
    my $d2 = 0;
    for my $i (0 .. 2) {
        # The point of the box nearest the centre: clamp each coordinate.
        my $p = $c->[$i] < $lo->[$i] ? $lo->[$i]
              : $c->[$i] > $hi->[$i] ? $hi->[$i] : $c->[$i];
        $d2 += ($c->[$i] - $p) ** 2;
    }
    return $d2 <= $r * $r;
}
```

These are the two *bounding volumes* the meshes chapter introduced, the axis-aligned box and the sphere, and the *scene-graph chapter* tested rays against the same two shapes.

There they were a cheap first test before an exact one. Here nothing exact follows, because the volumes are close enough to the shapes: a block *is* a box, and a gem spinning in place stays inside the same sphere at every angle, which is the reason to give a turning object a sphere rather than a box.

The craft's box ignores its bank. It is the box of the level craft, and banking narrows the craft by up to a tenth of a unit, so the test errs slightly on the side of a crash. Its height never decides anything: every block rises at least 0.8 from the floor and the box tops out at 0.75, so the vertical ranges always overlap and the other two axes decide.

Twenty-four box tests per step cost nothing. The *bounding volume hierarchy* is for when there are thousands; it would answer the same question with fewer tests, and nothing else would change.

Update

```
sub update {
    my ($steer) = @_;                    # -1 left, 0 straight on,
↪+1 right
    return if $game{over};
    my $c = $game{craft};

    # Ease the sideways speed toward the steering rather than
↪setting it:
    # the craft drifts into a turn and out of it, and its bank
↪follows vx.
    $c->{vx} += (0.18 * $steer - $c->{vx}) * 0.2;
    $c->{x}  += $c->{vx};
    my $edge = LANE - 0.8;
    if ($c->{x} < -$edge) { $c->{x} = -$edge; $c->{vx} = 0 }
    if ($c->{x} >  $edge) { $c->{x} =  $edge; $c->{vx} = 0 }
    $c->{z} -= $game{speed};
    $game{speed} += 0.00004 if $game{speed} < 0.3;
    $game{steps}++;

    my ($lo, $hi) = craft_box();
    for my $b (@{ $game{blocks} }) {
        # Behind the camera nobody can see it: reuse it at the far
↪end.
        place_block($b, $b->{z} - AHEAD) if $b->{z} > $c->{z} + 8;
        if (boxes_overlap($lo, $hi, block_box($b))) { $game{over} =
↪1; return }
    }
    for my $g (@{ $game{gems} }) {
        place_gem($g, $g->{z} - AHEAD) if $g->{z} > $c->{z} + 8;
        next if $g->{taken};
        if (sphere_touches_box([ $g->{x}, 0.6, $g->{z} ], 0.35, $lo,
↪$hi)) {
            $g->{taken} = 1;
            $game{score}++;
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

`update` takes the steering as an argument instead of reading the keyboard itself. That one decision is what makes the game testable: a list of numbers fed to `update` plays a run exactly as a player would, with no window and no keyboard.

Two things in it are load-bearing, and both are the 2D game's lessons again.

The `taken` flag. The craft overlaps a gem for several steps while it passes through it, and every one of those steps would score without the flag. It is the 2D game's `dy > 0` test in another form: count the event, not the overlap.

The clamp at the lane edges also zeroes `vx`. Clamping the position alone leaves the craft pressing on the wall at full sideways speed, banked hard over while not moving, and the first frame of steering back has to cancel that speed before anything happens.

The speed rises from 0.12 units per step to 0.3, reaching it after 4500 steps, which is 72 seconds of play. At 0.3 a row of blocks arrives about every eight steps.

Input

```
sub input {
    SDL_PumpEvents();
    my $steer = 0;
    $steer -= 1 if SDL2::key_pressed(SDL_SCANCODE_LEFT);
    $steer += 1 if SDL2::key_pressed(SDL_SCANCODE_RIGHT);
    new_game() if $game{over} && SDL2::key_pressed(SDL_SCANCODE_
↪SPACE);
    return $steer;
}
```

This is the *2D game's* input routine: `key_pressed` asks whether a key is held now, rather than waiting for events that report changes, so steering is smooth for as long as the key is down. Holding both arrows cancels out. Space starts a new run, but only once the current one has ended, so a player cannot restart by accident.

The loop reads the keyboard once per frame and passes the answer to every fixed step that frame runs. Reading it inside `update` would tie the simulation to SDL again, which is what the argument above avoids.

`SDL_PushEvent`, which the *device chapter* uses to drive a program from a test, does not help here. A pushed `SDL_KEYDOWN` for the left arrow arrives on the event queue with `scancode = 80` as expected, but `key_pressed(SDL_SCANCODE_LEFT)` stays false: the key state is SDL's record of the physical keyboard, and a pushed event does not change it. Tests drive `update` instead.

The craft is a small hierarchy

```
sub m_rotate_z {                              # the pattern of m_rotate_
↪y, about z
```

(continues on next page)

(continued from previous page)

```
    my ($rad) = @_;
    my ($c, $s) = (cos $rad, sin $rad);
    return [ $c,$s,0,0,  -$s,$c,0,0,  0,0,1,0,  0,0,0,1 ];
}

sub walk {
    # $parent_world is the identity at the root. Multiplication
↪order
    # is parent * local: the local transform applies FIRST, then the
    # parent's, which is the right-to-left rule from foundations.
    my ($node, $parent_world, $out) = @_;
    my $world = m_mul($parent_world, $node->{local});
    $out->{ $node->{name} } = $world;
    walk($_, $world, $out) for @{ $node->{children} };
    return $out;
}

# Each part is the unit cube, sized and placed relative to the body.
my %PART = (
    hull    => m_scale(0.5, 0.25, 1.0),
    cockpit => m_mul(m_translate(0, 0.17, -0.05), m_scale(0.25, 0.
↪12, 0.4)),
    lwing   => m_mul(m_translate(-0.5, 0, 0.15), m_scale(0.6, 0.06,
↪0.5)),
    rwing   => m_mul(m_translate( 0.5, 0, 0.15), m_scale(0.6, 0.06,
↪0.5)),
);

sub craft_node {
    my $c = $game{craft};
    # Moving right, the craft banks right: clockwise seen from
↪behind,
    # which is a negative angle about z.
    return { name     => 'craft',
             local    => m_mul(m_translate($c->{x}, 0.5, $c->{z}),
↪m_rotate_z(-2.5 * $c->{vx})),
             children => [ map { +{ name => $_, local => $PART{$_},
↪children => [] } }
                           sort keys %PART ] };
}
```

`walk` is the *scene-graph chapter's*, unchanged. The body node carries the craft's position and its bank; the four parts carry only their offset and size relative to the body. Banking rotates the body, and the wings tip with it, one down and one up, without any part knowing the angle. That is the turntable and arm from that chapter again.

The parts are cubes stretched by different amounts along each axis, and the *GPU scene chapter* warned that a non-uniform scale bends normals. It does not bend these: every face normal of a box lies along one of the box's own axes, a scale only lengthens such a vector, and the fragment shader normalises it back. The warning is for slanted faces.

The chase camera

```
sub camera {
    my $c = $game{craft};
    # Eye and target slide sideways together, by less than the craft
    # moves: the view never turns, and the craft drifts off centre
 ↪in a
    # turn, which is what makes the steering visible.
    my $x = 0.6 * $c->{x};
    return ([ $x, 2.5, $c->{z} + 6 ], [ $x, 0.5, $c->{z} - 6 ]);
}
```

The eye is 6 units behind the craft and 2.5 up, looking at a point 6 units ahead of it on the floor. Both move with the craft, so the view matrix from `m_look_at` changes every step while the craft's place in the picture barely does. The up hint is `[0, 1, 0]`, safely away from the view direction, which never points straight down.

One draw for everything

```
sub decode { return [ map { $_ ** 2.2 } @_ ] }   # picked by eye: to
 ↪linear light

my %COLOUR = (
    hull  => decode(0.85, 0.85, 0.90),  cockpit => decode(0.20, 0.
 ↪50, 0.95),
    wing  => decode(0.95, 0.50, 0.10),  crashed => decode(0.90, 0.
 ↪10, 0.10),
    block => decode(0.55, 0.35, 0.75),  gem     => decode(1.00, 0.
 ↪80, 0.10),
    rail  => decode(0.85, 0.85, 0.90),  sky     => decode(0.50, 0.
 ↪62, 0.80),
    tile0 => decode(0.22, 0.24, 0.28),  tile1   => decode(0.30, 0.
 ↪32, 0.36),
);
use constant MAX_INSTANCES => 4 + N_BLOCKS + N_GEMS + 30 + 2 + 20;

sub instances {
    my @inst;
    my $add = sub { my ($m, $colour) = @_; push @inst, @$m, @
 ↪$colour, 1 };
    my $c = $game{craft};

    my $world = walk(craft_node(), m_identity(), {});
    for my $name (sort keys %PART) {
        my $colour = $game{over}      ? 'crashed'
                   : $name =~ /wing$/ ? 'wing' : $name;
        $add->($world->{$name}, $COLOUR{$colour});
    }
    for my $b (@{ $game{blocks} }) {
        $add->(m_mul(m_translate($b->{x}, $b->{h} / 2, $b->{z}),
```

(continues on next page)

(continued from previous page)

```
                    m_scale($b->{w}, $b->{h}, $b->{w})), $COLOUR
↪{block});
    }
    my $spin = 0.05 * $game{steps};
    for my $g (grep { !$_->{taken} } @{ $game{gems} }) {
        $add->(m_mul(m_translate($g->{x}, 0.6, $g->{z}),
                m_mul(m_rotate_y($spin), m_mul(m_rotate_z(0.785), m_
↪scale(0.4, 0.4, 0.4)))),
                $COLOUR{gem});
    }
    # The floor: 30 strips two units deep, from behind the camera to␣
↪the
    # far end. A strip's colour comes from its place in the world,␣
↪not
    # its place in this list, so the stripes stay put as the craft␣
↪passes.
    my $z0 = 2 * int($c->{z} / 2) + 8;
    for my $k (0 .. 29) {
        $add->(m_mul(m_translate(0, -0.05, $z0 - 2 * $k - 1), m_
↪scale(2 * LANE, 0.1, 2)),
                $COLOUR{ ($z0 / 2 - $k) % 2 ? 'tile1' : 'tile0' });
    }
    for my $side (-1, 1) {                          # rails along the␣
↪lane edges
        $add->(m_mul(m_translate($side * (LANE + 0.15), 0.15, $c->
↪{z} - 22),
                    m_scale(0.3, 0.3, 60)), $COLOUR{rail});
    }
    # The score: one small gem per point, in a row above the lane.
    my $shown = $game{score} < 20 ? $game{score} : 20;
    for my $k (0 .. $shown - 1) {
        $add->(m_mul(m_translate(0.6 * $c->{x} + 0.4 * ($k - ($shown␣
↪- 1) / 2), 3.0, $c->{z} - 4),
                    m_mul(m_rotate_z(0.785), m_scale(0.2, 0.2, 0.
↪2))), $COLOUR{gem});
    }
    return @inst;
}
```

Everything on the screen is the one cube from the *GPU scene chapter*: the craft's four parts, the blocks, the gems turned onto a corner and spun, the floor strips, the rails and the score. Each is an instance, a model matrix and a colour, 20 floats, and the whole frame is **one** `draw_indexed` call with however many instances this frame has. The count varies, since taken gems are not drawn and the score row grows, but never beyond `MAX_INSTANCES`, 88, which sizes the buffer once.

The floor strips show why instancing and plain data suit each other. There is no floor object to move: each frame computes where the 30 strips are from the craft's position, and the pattern stays fixed to the world because a strip's colour depends on which two-unit band it covers. `%` of a negative number is non-negative in Perl when the right-hand side

is positive, so the bands alternate correctly on both sides of zero.

The colours are picked by eye and decoded into linear light before upload, for the reason the *GPU scene chapter* gives: the surface stores sRGB and the lighting is computed in linear light.

The shader: per-fragment light, and fog

```
my $WGSL = <<'WGSL';
struct Scene {
    view_proj: mat4x4<f32>,
    light_dir: vec4<f32>,     // xyz: toward the light, world space
    eye:       vec4<f32>,     // xyz: camera position, world space
    fog:       vec4<f32>,     // rgb: the colour distance fades to
};
struct Instance {
    model:  mat4x4<f32>,
    colour: vec4<f32>,
};
@group(0) @binding(0) var<uniform> scene: Scene;
@group(0) @binding(1) var<storage, read> inst: array<Instance>;

struct VOut {
    @builtin(position) pos: vec4<f32>,
    @location(0) world:  vec3<f32>,
    @location(1) normal: vec3<f32>,
    @location(2) colour: vec3<f32>,
};

@vertex
fn vs(@location(0) p: vec3<f32>, @location(1) n: vec3<f32>,
      @builtin(instance_index) i: u32) -> VOut {
    let m = inst[i].model;
    let w = m * vec4<f32>(p, 1.0);
    var o: VOut;
    o.pos    = scene.view_proj * w;
    o.world  = w.xyz;
    o.normal = (m * vec4<f32>(n, 0.0)).xyz;
    o.colour = inst[i].colour.rgb;
    return o;
}

@fragment
fn fs(v: VOut) -> @location(0) vec4<f32> {
    let n = normalize(v.normal);
    let l = normalize(scene.light_dir.xyz);
    let e = normalize(scene.eye.xyz - v.world);
    let lambert = max(dot(n, l), 0.0);
    let h = normalize(l + e);
    let spec = pow(max(dot(n, h), 0.0), 32.0);
    let c = min(v.colour * (0.1 + 0.9 * lambert) + vec3<f32>(0.4) *
```

(continues on next page)

(continued from previous page)

```
↪spec, vec3<f32>(1.0));
    // Fade to the fog colour from 25 to 55 units away, so that a␣
↪block
    // placed at the far end appears out of the haze instead of␣
↪popping in.
    let f = clamp((distance(scene.eye.xyz, v.world) - 25.0) / 30.0,␣
↪0.0, 1.0);
    return vec4<f32>(mix(c, scene.fog.rgb, f), 1.0);
}
WGSL
```

Everything above the fog is the *GPU scene chapter's* shader: instanced model matrices from a storage buffer, the normal turned with `w = 0`, and *shading's* Lambert and Blinn-Phong terms evaluated per fragment.

The fog is two lines and does a real job. A recycled block reappears 58 units from the eye, and without the fog it would appear there in one frame, fully drawn. With it, anything beyond 55 units is exactly the fog colour, and the fog colour is the clear colour, so the block is invisible until it comes within range and then fades in. Both colours are linear, so the `mix` is in linear light like the rest of the lighting.

The `fog` member is a `vec4` for the alignment reason the *GPU scene chapter* gives, and it makes the uniform 28 floats: 16 for the matrix and three `vec4`s.

The device and the buffers

The cube is the *GPU scene chapter's*, unchanged:

```
my @FACES = (
    # normal       u (right)     v (up)      u x v == normal
    [[ 1, 0, 0], [ 0, 0,-1], [0, 1, 0]],
    [[-1, 0, 0], [ 0, 0, 1], [0, 1, 0]],
    [[ 0, 1, 0], [ 1, 0, 0], [0, 0,-1]],
    [[ 0,-1, 0], [ 1, 0, 0], [0, 0, 1]],
    [[ 0, 0, 1], [ 1, 0, 0], [0, 1, 0]],
    [[ 0, 0,-1], [-1, 0, 0], [0, 1, 0]],
);
my (@vert, @idx);
for my $f (@FACES) {
    my ($n, $u, $v) = @$f;
    die "face winds the wrong way\n" unless v_dot(v_cross($u, $v),
↪$n) > 0;
    my $base = @vert / 6;
    for my $c ([-1,-1], [1,-1], [1,1], [-1,1]) {
        my $p = v_add(v_scale($n, 0.5),
                      v_add(v_scale($u, 0.5 * $c->[0]), v_scale($v,␣
↪0.5 * $c->[1])));
        push @vert, @$p, @$n;
    }
    push @idx, map { $base + $_ } 0, 1, 2, 0, 2, 3;
}
```

The window, the device and the surface, from the *device chapter*:

```
my $win    = SDL2::Window->new(title => 'pperl run', w => 800, h =>
 ↪600, resizable => 1);
my $gpu    = Peta::WebGPU::Device->new(window => $win);
my $surf   = $gpu->surface;
my $FORMAT = $surf->format;
my ($W, $H) = $surf->size;
```

These five lines are the only ones that need a window. Everything after them uses `$gpu`, `$FORMAT`, `$W` and `$H`, so replacing them with a device opened without a window and a texture format renders the same game into a texture, which is how it is checked below.

```
my $pipe = $gpu->render_pipeline($gpu->shader($WGSL),
    format => $FORMAT, depth => 1, cull => 'back',
    vertex_buffers => [ { attributes => ['float32x3', 'float32x3'] }
 ↪]);
my $verts      = $gpu->vertex_buffer(pack 'f*', @vert);
my $index      = $gpu->index_buffer(pack 'L*', @idx);
my $depth      = $gpu->depth_buffer($W, $H);
my $instances  = $gpu->storage_buffer(MAX_INSTANCES * 20 * 4);
my $scene      = $gpu->uniform_buffer(28 * 4);
my $gl_to_wgpu = [1,0,0,0,  0,1,0,0,  0,0,0.5,0,  0,0,0.5,1];

sub draw {
    my ($target) = @_;
    my @inst = instances();
    $instances->write(pack 'f*', @inst);

    my ($eye, $look) = camera();
    my $proj = m_mul($gl_to_wgpu, m_perspective(60, $W / $H, 0.1,
 ↪100));
    my $vp   = m_mul($proj, m_look_at($eye, $look, [0, 1, 0]));
    $scene->write(pack 'f*', @$vp, @{ v_norm([-0.4, 1.0, 0.6]) }, 0,
 ↪@$eye, 1,
                  @{ $COLOUR{sky} }, 1);

    $gpu->render_pass($target, clear => [ @{ $COLOUR{sky} }, 1 ],
 ↪depth => $depth)
        ->pipeline($pipe)->bind([$scene, $instances])
        ->vertices($verts)->indices($index)->draw_indexed(scalar
 ↪@idx, @inst / 20)->end;
}
```

The storage buffer is created at its largest size, and each frame writes only as many instances as it has. `draw_indexed`'s second argument says how many to draw, so whatever lies beyond them in the buffer is never read.

`$gl_to_wgpu` is the depth-range correction from the *GPU scene chapter*: without it, anything closer to the eye than about twice the near distance is silently cut away.

Score and game over, without text

```
my $title = '';
sub show_score {
    my $text = "pperl run - score $game{score}";
    $text .= " - crashed, Space to fly again" if $game{over};
    return if $text eq $title;           # only when it changes
    $win->set_title($text);
    $title = $text;
}
```

There is no text on the screen, and the game does not need any. The score is in two places: the window's title bar, through the SDL2 window's `set_title`, and the scene itself, as the row of small gems that grows over the lane. A crash shows the same way: the title says so, the craft turns red, and the world stops, because `update` returns at once when `over` is set.

The title changes only when the text does, rather than sending the window system the same string every frame.

The loop

```
new_game();
my ($last, $acc, $frames, $running) = (SDL_GetTicks(), 0, 0, 1);
while ($running) {
    while (my $ev = SDL_PollEvent()) {
        if ($ev->{type} == SDL_QUIT) { $running = 0 }
        elsif ($ev->{type} == SDL_KEYDOWN && $ev->{sym} == SDLK_
↪ESCAPE) { $running = 0 }
        elsif ($ev->{type} == SDL_WINDOWEVENT
               && $ev->{event} == SDL_WINDOWEVENT_SIZE_CHANGED) {
            ($W, $H) = @$ev{qw(data1 data2)};
            $surf->configure($W, $H);
            $depth = $gpu->depth_buffer($W, $H);     # must match␣
↪the frame
        }
    }

    my $steer = input();
    my $now = SDL_GetTicks();
    $acc += $now - $last;
    $last = $now;
    my $steps = 0;
    while ($acc >= STEP) {
        update($steer);
        $acc -= STEP;
        if (++$steps >= MAX_STEPS) { $acc = 0; last }
    }

    show_score();
```

(continues on next page)

(continued from previous page)

```
    my $frame = $surf->next_frame or next;
    draw($frame);
    $frame->present;
    $running = 0 if $max_frames && ++$frames >= $max_frames;
}
printf "%d frames, score %d%s\n", $frames, $game{score}, $game{over}
 ↪? ', crashed' : '';
```

This is the *device chapter's* loop with two lines added, `input` and `show_score`. The event loop handles quitting and resizing; the keyboard state handles steering. The fixed step keeps the game's speed independent of the display: a 144 Hz display draws more frames of the same run, not a faster run. **The step cap and the dropped backlog matter more in a game than anywhere else.** Without them, a window dragged for a second would come back and run 60 steps at once, and the player would find the craft already wrecked on a block they never saw.

Running the game

Save the blocks above in order as `run.pl`, with `foundations.pl` beside it, and start it:

```
pperl run.pl
```

The left and right arrow keys steer, Space flies again after a crash, and Escape or the close button quits. An optional number runs that many frames and stops, which is how a script can run it unattended.

In a window, the program draws and presents a new frame on every pass of the loop. Captures of the window taken a quarter of a second apart show the blocks coming closer, and when the craft meets one the title bar changes from `pperl run - score 0` to `pperl run - score 0 - crashed, Space to fly again` and the craft turns red.

Checking it without a keyboard

A player cannot be scripted through the keyboard, as the input section found, and a picture is easiest to check as bytes in a texture, as the GPU arc checked its own. Both point the same way: keep the game and change its two ends. The five window lines are replaced by a device opened without one:

```
my $gpu    = Peta::WebGPU::Device->new;      # no window
my $FORMAT = 'rgba8unorm-srgb';              # what the window
 ↪shows, in R, G, B, A order
my ($W, $H) = (640, 480);
```

and the last two blocks, the title and the loop, are left out. What remains is the whole game, driven by calls instead of keys:

```
srand 7;                                     # the same field every
 ↪time
new_game();
update(0) for 1 .. 20;                       # straight on
```

(continues on next page)

(continued from previous page)

```
update(1) for 1 .. 12;                          # then hard right
my $img = $gpu->texture(width => $W, height => $H, format =>
 ↪$FORMAT);
draw($img);
save_png('turn.png', $img->read, $W, $H);
```

save_png is the *device chapter's*. Run this way, the behaviour described in this chapter was checked against the game's own subs:

```
straight on, seed 7     crash on step 91: a block at x = 1.27, width␣
 ↪1.01,
                        its left face at 0.765 against the box's 0.
 ↪80
60 more updates         state unchanged, keys or not
right arrow held        vx = 0.036 0.065 0.088 0.106 0.121 on steps␣
 ↪1 to 5
held at the edge        x = 3.200, vx = 0; the first step back moves␣
 ↪at once
gem straight ahead      inside the box for 14 steps, scored 1
gem at x = 1.1, 1.2     scored 1, 0: the box edge 0.8 plus the␣
 ↪radius 0.35 is 1.15
chase camera            craft in view space at [0.4 x, -0.986, -6.
 ↪247] throughout
5000 steps, lane clear  speed 0.3 from step 4500, 574 blocks and␣
 ↪gems reused,
                        never more than 88 instances
same seed, same inputs  a recorded run of 472 steering values,␣
 ↪replayed,
                        ends in the identical state
```

The camera line is the chase camera's invariant, in the form the *scene-graph chapter* tested its orbit camera: the craft's position relative to the camera never changes in height or depth, and sideways it moves by `0.4 x`, the part of its movement the camera does not follow. That is what "the camera follows" means as a number.

The rendered frames say the rest. The pixel at the point where foundations' screen mapping puts the top of the hull is the hull's colour, lit, in every frame; after the crash the same pixel is red; and the sky reads back as `128 160 205`, within two of the `0.5, 0.62, 0.8` picked by eye, because the decode before upload and the surface's encode on the way out undo each other:

```
frame     hull top       cockpit        sky
start     200 200 212    43 118 223     128 160 205
turn      179 179 190    37 105 200     128 160 205
crashed   212  16  16    212  16  16    128 160 205
```

The turning frame's hull is darker because the bank tips its top away from the light. The frame with two gems collected shows two small gems above the lane, at the pixels where the score row's positions project, 195 157 24 and 194 156 22.

What this demonstrates

A game is the smallest program that needs every part of this guide at once. The mathematics places the objects and the camera; the meshes chapter's volumes decide what touches what; the scene graph lets a part follow its parent; the GPU arc draws every object in one call and lights every pixel; the fixed step keeps the simulation honest about time.

And the shape that made the 2D game easy to reason about carried over unchanged. The world is a hash, `update` is arithmetic on it, and the draw step reads it. Because `update` takes its input as an argument, the whole game can be played by a list of numbers, and that is how it was checked.

4.2.12 pack and unpack - a tutorial

`pack` and `unpack` are Perl's low-level serialisers. They convert between Perl values and the raw byte strings that binary formats - network protocols, file headers, C structures, fixed-width records - actually carry on the wire.

Most Perl programmers meet them rarely and forget them quickly. This tutorial is here so that when you do meet them, you already know the shape of the problem.

When you need pack and unpack

Reach for them when you are:

- **Speaking a wire protocol** - DNS queries, TCP/IP headers, MQTT frames, anything with "big-endian 16-bit length" in its spec.
- **Parsing a binary file format** - PNG, GIF, WAV, ELF, GZip, Intel HEX, all the formats whose first four bytes are a magic number followed by well-known integer fields.
- **Slicing fixed-width text records** - log files, accounting ledgers, reports from mainframes. `unpack` is cleaner than chained `substr` calls and handles blank-field ambiguity that `split` collapses.
- **Passing data to a system call** - `syscall`, `ioctl`, and the rare `sockaddr_*` constructor expect bytes laid out as a C struct.

For text serialisation (JSON, YAML, CSV, XML), reach for the appropriate module instead. `pack` is for binary.

How this tutorial is organised

Each chapter teaches one topic and stands on its own. You can read them in order, or jump to the one that answers your question.

Bytes and widths

By the end of this chapter you will be able to pick the right integer directive for a given field, count the bytes a template produces, and tell when you need a fixed width versus a native width.

Binary formats are specified by *byte counts*. "A 16-bit unsigned length field followed by 4-byte packets" is an instruction about bytes, not about numbers. The first job of a `pack` template is to match those counts exactly.

The fixed-width integer directives

The spine of every template:

Bytes	Unsigned	Signed
1	`C`	`c`
2	`S`	`s`
4	`L`	`l`
8	`Q`	`q`

Each directive consumes one value from the list and produces exactly that many bytes:

```
length pack "C",  65                  # 1
length pack "S",  12345               # 2
length pack "L",  1_000_000_000       # 4
length pack "Q",  1_000_000_000_000   # 8 (if 64-bit Perl)
```

A repeat count packs that many of the same directive in a row:

```
length pack "C4", 1, 2, 3, 4          # 4
length pack "L3", 10, 20, 30          # 12
```

Compose them freely; the total length is the sum:

```
length pack "C S L", 65, 1000, 2_000_000   # 1 + 2 + 4 = 7
```

`C` is your default one-byte directive

For protocol bytes - flag fields, version numbers, opcodes, type tags - `C` (unsigned 8-bit, 0 to 255) is almost always the right choice. `c` (signed, −128 to 127) turns up only when the spec says "signed". `W` is a variant of `C` that allows values above 255 when your template is in `U0` (UTF-8) mode - ignore it until you hit that case.

IP addresses are the textbook example:

```
my $raw = pack "C4", 192, 168, 1, 42;
#         ^-- four unsigned bytes, in the order the list gives them
```

Native vs. fixed width

The directives `s` / `S` / `l` / `L` are specified to be exactly 2 and 4 bytes regardless of the `C` compiler. Their native-width counterparts add `!`:

```
length pack "l",  0          # 4  (always)
length pack "l!", 0          # 4 on most 32-bit and 64-bit systems,
```

(continues on next page)

(continued from previous page)

```
                                # 8 on 64-bit Alpha, some legacy␣
↪Unixes.
```

`i` and `I` are *always* native (whatever `sizeof(int)` returns on this machine, with a 32-bit minimum). `i!` is an alias for `i`.

Rule of thumb:

- For interoperable binary (wire formats, on-disk files, data crossing machines) use **fixed width** (`l`, `L`, ...) plus an explicit endianness - or the *portable shortcuts*.
- For in-process work that matches some local `C` struct you are about to hand to `ioctl` or `syscall`, use **native width** (`l!`, `L!`, `i`, `I`).

Mixing the two is fine, but stop and ask yourself why each time.

Counting the length of a template

Before you send a record, check the byte count is what the spec says. `length pack(TEMPLATE, ...)` with placeholder values is the simplest way:

```
my $sz = length pack "n N L C", 0, 0, 0, 0;
# 2 + 4 + 4 + 1 = 11
```

For templates whose length depends on native sizes, `length pack TEMPLATE, 0, 0, ...` gives you the answer on the current machine.

Worked example: a minimal packet header

A hypothetical protocol specifies:

- 8-bit version
- 8-bit flags
- 16-bit big-endian length (payload follows)

Three fields, four bytes total. The template writes itself:

```
my $hdr = pack "C C n", $version, $flags, length $payload;
my $pkt = $hdr . $payload;

length $hdr    # 4
```

Two things to note:

1. We did not spell out `C1 C1 n1` - a bare directive letter has an implicit repeat count of 1.
2. The length in the third field is computed from `$payload` in the same expression. A later chapter shows the */ form* that ties the two together so you cannot forget.

Edge cases worth remembering

- **Values out of range**: `pack "C", 256` silently truncates (you get `\x00`). There is no warning. Check ranges yourself if the data is untrusted.
- **Signed vs. unsigned round-trip**: a value packed as `C` and unpacked as `c` (or vice versa) flips sign for values ⩾ 128:

```
unpack "c", pack "C", 200        # -56
```

- **Too few values**: missing values pack as `0` for numeric directives, `""` for strings. No warning.

The next chapter handles the question every multi-byte integer raises: which end of the number goes first?

Endianness

By the end of this chapter you will be able to choose between native, big-endian, and little-endian byte order; use the portable `n` / `N` / `v` / `V` shortcuts; and apply the `<` / `>` modifiers to any integer or float.

A multi-byte integer has to be written somehow. `0x12345678` is four bytes - but in which order? Two conventions dominate:

- **Big-endian** - highest-order byte first: `12 34 56 78`. Used by most network protocols, by PowerPC and SPARC CPUs, and by file formats that care about portability (PNG, Java `.class`, most TCP/IP headers).
- **Little-endian** - lowest-order byte first: `78 56 34 12`. Used by x86, x86-64, ARM (in its common modes), and by file formats from the Intel lineage (BMP, WAV, TIFF in its common variant, all of FAT).

Mix these up and you will read nonsense. `pack` gives you three ways to pin down the byte order.

The portable shortcuts: n / N / v / V

Four letters cover the most common wire-format cases:

Directive	Bytes	Endian	Meaning
`n`	2	big-endian	Network (port numbers)
`N`	4	big-endian	Network (IPv4 addrs)
`v`	2	little-endian	“VAX”
`V`	4	little-endian	“VAX”

“Network byte order” is big-endian. If you are reading an RFC, the answer is almost always `n` or `N`.

```
my $port = pack "n", 443;          # "\x01\xbb"
my $ip   = pack "N", 0x7F000001;   # "\x7f\x00\x00\x01" - 127.0.0.1
```

These four are unsigned only. With the `!` modifier they become signed:

```
my $neg = pack "n!", -1;            # "\xff\xff"
```

If your field fits one of these four shapes, use them. They are portable, they are readable, and their meaning does not depend on the CPU running the code.

The general modifiers: < and >

Any integer directive (`s`, `S`, `l`, `L`, `i`, `I`, `q`, `Q`, `j`, `J`) and every float (`f`, `d`, `F`, `D`) accepts an endianness modifier:

Modifier	Meaning	Mnemonic
>	Big-endian	The “big end” touches the directive letter
<	Little-endian	The “little end” touches the directive

Read `l>` as “long, big end” and `l<` as “long, little end”.

```
my $be32 = pack "l>", -1_000_000;    # big-endian 32-bit signed
my $le64 = pack "q<", 2 ** 40;       # little-endian 64-bit signed
my $bed  = pack "d>", 3.14159;       # big-endian IEEE 754 double
```

For the common 16- and 32-bit unsigned cases, `n` / `N` / `v` / `V` are shorter and should be preferred. Use < / > when:

- the field is signed - `n` and `N` are unsigned by default
- the field is 64-bit - there are no `n`/`N` equivalents for `q`/`Q`
- the field is a float - `n`/`N` cover integers only
- you want to apply endianness to a whole group (see below)

Applying endianness to a group

If every integer in a sub-structure shares the same byte order, write the endianness modifier once on a group:

```
my $rec = pack "(s l l)<", $flags, $x, $y;
# same as "s<l<l<"
```

A group-level modifier cascades into every byte-ordered directive inside, including nested groups. Directives that do not accept a byte-order modifier (such as `C`) are silently ignored.

Native order: when it *is* what you want

The bare directives `s` / `S` / `l` / `L` / `i` / `I` / `q` / `Q` produce the host CPU’s native byte order. That is useful when:

- You are building a C struct in memory for an `ioctl` call on the *same* machine.
- You are writing a scratch file that only this program will read back.
- The protocol spec explicitly says “host byte order” (rare).

It is **wrong** when data leaves the process: a file written on a laptop, read on a server, must not use native order unless both machines happen to agree.

Detecting the host's byte order

Useful once, as a sanity check. Pack 1 as a 16-bit integer and look at the first byte:

```
my $little_endian = unpack("c", pack("s", 1)) == 1;
my $big_endian    = unpack("c", pack("s", 1)) == 0;
```

On x86 the first is true; on a big-endian host the second is true.

Floats have endianness too

IEEE 754 does not pin down byte order. A double written on a little-endian machine comes out reversed on a big-endian one:

```
# send a float in big-endian order - safe across machines that use␣
↪IEEE 754
my $buf = pack "d>", 2.718281828;
```

Mismatched hardware (non-IEEE float formats, exotic long double layouts) cannot be repaired by < / > alone. For portability across odd platforms, send floats as text or use a byte-exact format like the output of `sprintf "%a"`.

Worked example: parsing a BMP header

The BMP bitmap format begins with the ASCII magic `BM` followed by three little-endian 32-bit fields (file size, reserved, pixel data offset):

```
my ($magic, $filesize, $reserved, $pixoff) =
    unpack "A2 V V V", $bmp_header;

die "not a BMP" unless $magic eq "BM";
```

`V` is little-endian 32-bit unsigned - exactly what the spec demands. `A2` reads two bytes as a text string without stripping. Every field is the right shape; no byte-swapping required, no platform-dependent behaviour.

What to remember

- **Wire format? Use `n` / `N` / `v` / `V`** - or < / > if you need signed, 64-bit, or float.
- **Stay inside one process? Native is fine** - but flag the choice deliberately.
- **IEEE 754 is not enough.** If a float crosses a machine, pin the byte order.

With byte counts and byte order under control, the remaining directives - strings, positioning, groups - fall out naturally.

Strings, bits, and nybbles

By the end of this chapter you will be able to pack and unpack fixed-width strings, C-style NUL-terminated strings, bit strings, and hex strings - the three text-ish families of directives.

Binary formats carry string data in several distinct flavours: raw bytes, space-padded text, NUL-terminated C strings, bit fields, hex-encoded numbers. `pack` has a directive for each.

The three string directives: a / A / Z

All three pack exactly one value into a fixed width. They differ in what they pad with, and - crucially - in what `unpack` strips back off:

Letter	Pad byte	Unpack returns	Typical use
a	`"\0"` (NUL)	All bytes unchanged	Arbitrary binary
A	`" "` (space)	Trailing whitespace and NUL stripped	ASCII fixed-width text
Z	`"\0"` (NUL)	Bytes up to the first NUL	C-style NUL-terminated

The width is the **repeat count**, not a count of values:

```
pack "a4", "hi"           # "hi\0\0"
pack "A4", "hi"           # "hi  "
pack "Z4", "hi"           # "hi\0\0"

pack "a4", "abcdef"       # "abcd"  - truncated
pack "A*", "hello"        # "hello" - whatever the value's length is
```

Z guarantees a trailing NUL - with a caveat

`Z` always reserves space for at least one terminating NUL:

```
pack "Z*", "hello"        # "hello\0"   - a free NUL appended
pack "Z5", "hello"        # "hell\0"    - truncated to make room!
```

If a C program on the other side expects a zero-terminated `char[32]`, use `Z32`. It will pack up to 31 bytes of data plus the terminator.

Picking between a and A for text

- The data is ASCII, padded with spaces on disk → `A`.
- The data is arbitrary bytes (might contain spaces or NULs as valid content) → `a`.

The classic trap: you pack a name with `A20` and the wire format uses NUL padding. `unpack "A20"` strips both spaces *and* NULs, so it looks fine - but `pack "A20", "bob"` will round-trip to `"bob  "` padded with spaces. If the spec says NUL padding, use `a` on the pack side.

Worked example: fixed-width text records

A ledger file lays data out in columns:

```
0         1         2         3         4
01234567890123456789012345678901234567890123456789012345678
2026-04-22 coffee at the station           3.50
2026-04-23 train to Brussels              42.00
```

Column 1-10 is the date, 12-38 the description, 40-47 the amount. The gaps are single spaces (byte 11 and byte 39). unpack with `A10 x A27 x A*` peels each record apart:

```
while (<$ledger>) {
    chomp;
    my ($date, $desc, $amount) = unpack "A10 x A27 x A*", $_;
    print "  $date | $desc | $amount\n";
}
```

`x` skips one byte; `A27` eats the description column and strips trailing spaces; `A*` greedy-eats the remaining bytes as the amount. The output side uses wider fields so spaces survive the round-trip:

```
my $line = pack "A11 A28 A8 A*", $date, $desc,
                                 sprintf("%.2f", $amt_left),
                                 sprintf("%12.2f", $amt_right);
```

Notice the extra byte on each `A` width - that single extra byte is the column gap. A consistent byte budget per column is what makes fixed-width records tractable at all.

Bit strings: b and B

A bit string is a string of `"0"` and `"1"` characters that packs into actual bits. Two directives, differing only in which direction you read each byte:

Directive	Bit order within each byte
b	LSB first - bit 0, 1, 2, 3, 4, 5, 6, 7
B	MSB first - bit 7, 6, 5, 4, 3, 2, 1, 0

The repeat count is the number of **bits**, not bytes:

```
pack "B8", "10001100"      # "\x8c" - MSB first, bit 7 set, bit 3 &␣
↪2 set
pack "b8", "00110001"      # "\x8c" - LSB first, same byte
```

`B` matches the usual "left-to-right is high-to-low" convention you see in binary dumps; `b` matches the convention used by some hardware registers and the `vec` built-in. Use `B` when you are writing out a "normal" binary number, `b` when you are mirroring a data sheet that numbers bit 0 on the right.

Counting set bits

One of the more surprising uses of unpack: count set bits in a buffer with a single call. %32b* unpacks the bits and asks for a 32-bit sum, which is the count:

```
my $n_bits = unpack "%32b*", $mask;
```

The %N prefix is unpack-only - see the *positioning chapter* for the other uses.

Hex strings: h and H

Hex strings are the text representation most developers read - two hex digits per byte. Two directives, differing in nybble order:

Directive	Nybble order
h	Low nybble first
H	High nybble first

H is the "normal" hex dump - read left to right, high digit then low:

```
pack "H*", "deadbeef"      # "\xde\xad\xbe\xef"
unpack "H*", "\xde\xad\xbe\xef"   # "deadbeef"
```

h reverses each byte's nybbles - rarely what you want unless a spec explicitly says so. If the round-trip you expect is *not* what you get, try the other letter.

```
pack "h*", "ef"            # "\xfe"  (nybbles swapped)
pack "H*", "ef"            # "\xef"
```

Choosing between them

You have	Reach for
ASCII in a fixed-width column on disk	A
Arbitrary bytes in a fixed-width slot	a
A C-style NUL-terminated string	Z
A binary number as a string of 0 / 1	B
Same, with the register-convention bit order	b
A hex string in the normal reading order	H
A hex string with nybbles swapped	h

Bits, nybbles, and string directives all share one rule: the repeat count specifies the width of the *single* value being packed, not the number of values. That's the one fact to carry forward.

Next up: ( ) groups and repeat counts, the tools for applying a pattern of directives to a list of values of unknown length.

Grouping and counts

By the end of this chapter you will be able to repeat a pattern of directives across a list of values, use the `*` and `[…]` forms of repeat count, and write self-describing length-prefixed records with the `/` directive.

A bare repeat count applies to *one* directive letter: `C4` packs four bytes. The moment the repeating unit is more than one directive - "pack a short and two bytes, many times" - you need a group.

`()` - a group is a sub-template

Parentheses gather a sequence of directives so that a repeat count or endianness modifier applies to the whole. Compare:

```
pack "C S C S C S", @a, @b, @c, @d, @e, @f      # repeat by hand
pack "(CS)3",       @a, @b, @c, @d, @e, @f      # same thing, grouped
pack "(CS)*",       @pairs                      # repeat as often as␣
↪values last
```

A group has no byte cost of its own - it is a syntactic device. The total packed length is what the directives inside it would produce without the parentheses.

Grouping with endianness

The single most practical use of a group - a sub-structure whose every integer shares one byte order:

```
my $rec = pack "(l s s)<", $id, $x, $y;
# same as "l<s<s<"
```

The `<` cascades into every byte-ordered directive inside, including nested groups. Directives that do not accept a byte-order modifier (like `C`, `a`, `Z`) are silently unaffected.

Repeat counts in full

After any directive or group, you may write:

Form	Meaning
`N`	Apply the directive/group `N` times
`*`	Apply as often as values last. For `x`, `X`, `@`: equivalent to 0. For `u`: 45.
`[N]`	Same as `N`
`[templ]`	Repeat count is the packed-byte length of the bracketed template

The `[template]` form is the tool for expressing "as many bytes as a *foo* takes":

```
pack "x[L]"      # skip 4 bytes (sizeof a packed long)
pack "x[d]"      # skip 8 bytes (sizeof a packed double)
pack "a[Q]"      # one string 8 bytes wide
```

It is especially useful for alignment (see the *positioning chapter*) and for templates whose widths must track the platform-dependent size of a native integer:

```
pack "a[l!]", $native_long_buf      # room for one native long
```

* is per-value-group, not "swallow everything"

A single * counts "remaining values" *for this directive*. Two A*s in a row do not race:

```
pack "A*A*", "hello", "world"     # "helloworld"
```

The first A* packs all of "hello"; the second packs all of "world". Each * consumes *one* value from the list, at that value's full length. This is the general rule: **every directive corresponds to one piece of data**, whatever its repeat count.

The / directive - length and data together

Wire formats frequently store a count immediately before the thing being counted: "a 2-byte length, then that many bytes of payload". The / directive ties the two into one step.

In pack: *length-item*/*sequence-item*

Write *two* directives separated by a slash. The first packs the length; the second packs the payload. pack computes the length for you:

```
my $msg = pack "n/a*", "hello, world";
# "\x00\x0chello, world"
#  ^^^^^^^^ big-endian 16-bit length = 12
#           ^^^^^^^^^^^^ the payload itself
```

The *length-item* may be any numeric directive - C, n, N, w, S<, and so on - or a string directive such as A4 when the protocol writes the length as ASCII:

```
my $buf = pack "A4/A*", "Humpty-Dumpty";
# "13  Humpty-Dumpty"  - 4-char ASCII length, then the string
```

In unpack: /*item*

The unpack form is simpler: a bare / before the item. The count is taken from the most recent integer directive:

```
my ($payload) = unpack "n/a*", $msg;     # "hello, world"
```

Reading that template: "read a n integer, call it L; then read L bytes as an a*-style string." The length itself does not appear in the output list.

The common mistake: A* after a /

You cannot put another A* or a* after a /-introduced field in unpack and expect it to behave - the * is greedy:

```
# Wrong - $prio will be undef, $sm gets everything left
my ($src, $dst, $sm, $prio) = unpack "Z* Z* C A* C", $buf;

# Right - use /A* so $sm knows where to stop
my ($src, $dst, $sm, $prio) = unpack "Z* Z* C/A* C", $buf;
```

In the second template, C/A* reads a byte-count then that many bytes. Everything after the slash respects it, and the trailing C picks up the next byte as expected.

Worked example: key-value pairs

A protocol stores a dictionary as count followed by count pairs of (length, key, length, value):

```
my %env = ( HOST => "example.com",
            PORT => "443",
            USER => "alice" );

my $blob = pack "S (S/A* S/A*)*",
                scalar keys %env,
                %env;
```

Reading it back:

```
my %parsed = unpack "S/(S/A* S/A*)", $blob;
```

The pack template says: “one 16-bit count (of pairs), then repeat the sub-template S/A* S/A* for each pair.” The unpack template reads the count and applies it to the group directly - the count is no longer in the output list.

Edge cases and constraints

- **/ makes no sense with a fixed-length item.** The second directive must be variable-width: a*, A*, Z*, /A$n, or the like. Perl will reject a fixed-length second item.
- **()* with pack cannot be matched by ()* on unpack.** Pack has the values, so it can say “repeat until done”. Unpack does not know how many repetitions are encoded in the buffer unless a count directive precedes the group.
- **Nested groups** are legitimate and common:

  ```
  pack "((CC)(S))<", @records
  ```

  Endianness modifiers cascade through every nesting level.
- **Repeat count on a group** applies to the whole group and takes that many repetitions’ worth of values. (CS)3 consumes *six* list values, not three.

Next chapter: the directives that move around inside a template without producing a value - x, X, @, ..

Positioning and padding

By the end of this chapter you will be able to skip bytes, insert padding, jump to absolute offsets, and align fields to arbitrary boundaries - the four directives x, X, @, and ..

Some directives produce a byte without consuming a value from the list (pad bytes). Some consume a value without producing a byte (positioning commands). Taken together they let you model the gaps and alignments that real binary formats contain.

x - skip forward / insert NUL

x is the simplest of the four. In pack it inserts one NUL byte without consuming a value; in unpack it skips one byte without producing a value.

```
pack   "C x C", 65, 66       # "A\0B"
unpack "C x C", "A\0B"       # (65, 66)
```

With a repeat count, it inserts or skips that many bytes:

```
pack "C x3 C", 65, 66        # "A\0\0\0B"
```

x* means “skip to the end of the string” in unpack, and is equivalent to x0 (no-op) in pack.

x does not mean “a space”

The confusing thing about x is that it pads with NUL, not with space. If you want space padding for text, use A with a wider repeat count instead:

```
# Wrong - Perl inserts a NUL, not a space
pack "A10 x A10", "date", "label"
# "date      \0label     "

# Right - widen the field
pack "A11 A10", "date", "label"
# "date       label     "
```

X - back up

X moves the write/read head backward by one byte. With a repeat count, backward by that many bytes. Unlike x, X has no effect on the output bytes of pack - it just rewinds so a subsequent directive overwrites them.

```
pack "CC X C", 65, 66, 67    # "AC" - the second C overwrites 66␣
↪with 67
```

X is mostly useful with unpack: peek forward, then rewind to re-read:

```
# Read a 16-bit length, then re-read the same 2 bytes as two␣
 ↪separate bytes
my ($len, $hi, $lo) = unpack "n XX CC", $frame;
```

X before the start of the string is a fatal error.

@ - jump to absolute position

@N sets the current position to byte N **within the innermost ( ) group** (or from the start of the string if there is no enclosing group). In pack, this zero-fills from the current position forward, or truncates if the current position is past N:

```
pack "C @4 C", 65, 66        # "A\0\0\0B"  - byte 0 is 'A', byte 4␣
 ↪is 'B'
```

This is the template-level equivalent of `offsetof(struct, field)`: if a C header file tells you field 2 lives at offset 12, @12 parks you there regardless of what the preceding directives did.

@ inside a group

Within each repetition of a group, @ restarts at 0. That can surprise you:

```
pack '@1A((@2A)@3A)', qw(X Y Z)
# "\0X\0\0YZ"
```

Walking through: the outer @1A puts "X" at byte 1 (with a NUL at byte 0). The inner group starts a new coordinate system at "X"'s position; inside it, @2A puts "Y" at the group's byte 2 (two further NULs in between), then @3A puts "Z" at byte 3 of the group. Read the spec carefully before using @ inside groups.

. - absolute position from the data

. is the value-driven cousin of @. Instead of a fixed repeat count, the position comes from the next value in the list:

```
pack "C . C", 65, 4, 66     # "A\0\0\0B" - position 4 from the next␣
 ↪value
```

Useful when the offset is computed at runtime - say, from a header field you just read:

```
my $buf = pack "C .N C", $type, $offset, $value;
# $offset bytes of NUL fill before $value
```

In unpack, . returns the current position rather than zero-filling:

```
my ($a, $pos) = unpack "C .", "hello";
# $a = 104 ('h'), $pos = 1
```

With .*, the offset is measured from the start of the whole string; with .N for integer N ≥ 1, from the start of the enclosing N-th group; with .0, relative to the current position.

The ! modifier: alignment

`x!N` and `X!N` turn `x` and `X` into alignment commands: advance (or rewind) to the nearest multiple of `N`. This is how you respect C struct padding rules.

```
# struct { char c; double d; char cc[2]; }
my $s = pack "c x![d] d c2", $c, $d, $c1, $c2;
```

Reading `x![d]`: "align forward to a multiple of the width a packed `d` would take" - i.e. 8 bytes. That is exactly the rule most C compilers apply to a `double` field after a `char`. Use `[l!]`, `[d]`, `[Q]` to name the alignment requirement without having to know the numeric width.

A bare `x!0` or `x!1` is a no-op.

Checksums - %N on unpack only

`%N` is a prefix on a single directive, not a standalone directive. It says "instead of returning the values the next directive would produce, return their `N`-bit sum". Common uses:

```
my $bytesum = unpack "%32C*", $buf;    # sum every byte
my $setbits = unpack "%32b*", $mask;   # count set bits
```

Exists only in `unpack`. There is no `%` in `pack` - you cannot pack a checksum directly; compute it yourself and pack the result with whatever integer directive you need.

Worked example: reading a variable-offset payload

Some formats start with a header of fixed size but put the actual data at an offset the header contains. The classic recipe:

```
# Header: magic (4 bytes), version (1 byte), data-offset (4 bytes
 ↪BE),
# then data starts at the byte-offset the header pointed at.
my ($magic, $ver, $off) = unpack "A4 C N", $file;

die "bad magic" unless $magic eq "MAGI";

# Jump to the data section
my $data = unpack "x$off a*", $file;
```

Two separate `unpack` calls - the first reads the offset, the second uses it in the template. Template strings are not regular expressions; they have no way to look ahead and use a computed value within the same call. Unpack in stages when the shape of the data depends on what you have just read.

With positioning and alignment in the toolbox, the two worked examples in the next chapters should read as straightforward applications: parsing a real network header and a real file format.

Network protocols - a DNS query

By the end of this chapter you will be able to build and parse a DNS query header and question section using `pack` and `unpack` - and take the same approach to any RFC-defined protocol.

A "network protocol" is, in the end, a sequence of byte-exact fields described in English prose. If you can read the field table, you can write the template. This chapter walks one concrete example - the DNS query packet - from the RFC wire diagram to a fully working pair of encode / decode subroutines.

The problem

We want to ask a DNS server for the A record of `example.com`. The UDP payload we send is a **DNS message** (RFC 1035), consisting of:

1. A 12-byte **header**.
2. A **question section** - the domain we are asking about, and the record type we want.

(A real client also parses the **answer section** in the response. We stop at encoding and showing a sketch of the decode.)

The header

From RFC 1035 section 4.1.1, the header is six consecutive 16-bit big-endian fields:

Offset	Field	Meaning
0	ID	Arbitrary 16-bit identifier we choose
2	Flags	Opcode, RD bit, response flags, rcode
4	QDCOUNT	Number of entries in question section
6	ANCOUNT	Number of resource records in answer
8	NSCOUNT	Authority records
10	ARCOUNT	Additional records

Six unsigned big-endian shorts means six `n` directives:

```
sub dns_header {
    my (%opts) = @_;
    pack "n6",
        $opts{id}      // 0,
        $opts{flags}   // 0,
        $opts{qd}      // 0,
        $opts{an}      // 0,
        $opts{ns}      // 0,
        $opts{ar}      // 0;
}
```

That is 12 bytes, exactly as the spec requires. `n6` is shorthand for `n n n n n n`.

The question section

A question is:

1. A **name**, encoded as a sequence of length-prefixed labels terminated by a zero-length label.
2. A 16-bit **QTYPE** (1 = A record).
3. A 16-bit **QCLASS** (1 = IN, Internet).

For `example.com` the name encodes as:

```
\x07 e x a m p l e   \x03 c o m   \x00
```

Each label begins with a length byte; the whole name ends with a zero-length label. Two things to notice: the length is one byte (`C`), not two; and there is a trailing `NUL` but it is *not* exactly what `Z` produces - it terminates the list of labels, not a single string.

We can build this by hand from a domain name:

```
sub encode_name {
    my ($name) = @_;
    my $out = "";
    for my $label (split /\./, $name) {
        die "label too long" if length($label) > 63;
        $out .= pack "C/a*", $label;
    }
    $out .= "\0";                       # zero-length terminator
    return $out;
}
```

Two directives in the loop body: `C/a*` - a single-byte length followed by the label bytes, computed automatically (see the *grouping-and-counts chapter* for the `/` form). After the loop, a literal `"\0"` terminates the list.

With `encode_name` in place, the whole question section is two concatenations and one `pack`:

```
sub dns_question {
    my ($name, $qtype, $qclass) = @_;
    return encode_name($name) . pack "n n", $qtype, $qclass;
}
```

Assembling the packet

```
use constant {
    QR_QUERY    => 0,
    OPCODE_QUERY=> 0,
    RD          => 1 << 8,          # recursion desired
    TYPE_A      => 1,
    CLASS_IN    => 1,
};
```

(continues on next page)

(continued from previous page)

```
sub dns_query_for_A {
    my ($name) = @_;
    my $id    = int rand 65536;
    my $flags = RD;                  # standard query, recursion␣
 ↪desired

    my $pkt = dns_header(
        id    => $id,
        flags => $flags,
        qd    => 1,                  # one question
    );
    $pkt .= dns_question($name, TYPE_A, CLASS_IN);

    return ($id, $pkt);
}

my ($id, $pkt) = dns_query_for_A("example.com");
# send $pkt over a UDP socket to port 53
```

29 bytes total: 12 header + 13 for the name (`\x07example\x03com\x00` is 13) + 2 + 2 for QTYPE and QCLASS. Run `length $pkt` to confirm.

Parsing the response header

The response has the same header shape - only the flag bits change. Decoding is the reverse template:

```
sub parse_dns_header {
    my ($buf) = @_;
    my ($id, $flags, $qd, $an, $ns, $ar) = unpack "n6", $buf;

    my %hdr = (
        id    => $id,
        qr    => ($flags >> 15) & 1,
        op    => ($flags >> 11) & 0x0f,
        aa    => ($flags >> 10) & 1,
        tc    => ($flags >> 9)  & 1,
        rd    => ($flags >> 8)  & 1,
        ra    => ($flags >> 7)  & 1,
        rcode =>  $flags        & 0x0f,
        qd    => $qd,
        an    => $an,
        ns    => $ns,
        ar    => $ar,
    );
    return \%hdr;
}
```

n6 pulls the six shorts out at one go; the individual flag bits come out of shifts and masks

on `$flags`. This pattern is universal: use `pack` / `unpack` for the byte-level layout, plain Perl for bit-level decoding of the flag fields.

Parsing a name

Names in the answer section use the same length-prefix encoding, plus a **pointer compression** mechanism we will skip. For a fresh name (no compression) the decoder mirrors the encoder:

```
sub decode_name {
    my ($buf, $offset) = @_;
    my @labels;
    while (1) {
        my $len = unpack "x$offset C", $buf;
        last if $len == 0;
        die "compression pointer" if $len >= 0xc0;
        my $label = unpack "x${\ ($offset + 1)} a$len", $buf;
        push @labels, $label;
        $offset += 1 + $len;
    }
    return (join(".", @labels), $offset + 1);
}
```

Three `unpack` calls, each with an `x$offset` prefix to skip to the right byte. The name alternates between one-byte lengths and variable-width labels, and the loop keeps reading until it hits a zero-length label. The `x$offset a$len` trick - compose a template from Perl values, call `unpack` - is the standard pattern when the next field's width depends on the previous field's value.

What to carry forward

- **An RFC byte-layout table maps directly to a template string.** Big-endian shorts → `n`, big-endian longs → `N`, bytes → `C`.
- **Length-prefixed fields use `C/a*` / `n/a*` / …** - one expression, no off-by-one counting by hand.
- **Bit-level fields inside a byte get decoded in plain Perl** after `unpack` has handed you the byte or short that contains them.
- **Variable-width data requires staged unpacking.** Read a length, then build the template for the data from that length, then call `unpack` again.

The next chapter applies the same approach to a real file format - a GIF image header - and introduces the "magic number check" idiom.

File formats - a GIF header

By the end of this chapter you will be able to identify a binary file by its magic number, parse its fixed-layout header, and decode bit-packed flag bytes. The example is the GIF image format, but the approach transfers directly to PNG, WAV, BMP, ELF, ZIP, and every other format built around a fixed header.

The GIF header and logical screen descriptor

A GIF file starts with two fixed-size blocks, thirteen bytes in total. From the GIF89a specification:

Header (6 bytes)

Offset	Bytes	Field
0	3	Signature - `"GIF"`
3	3	Version - `"87a"` or `"89a"`

Logical Screen Descriptor (7 bytes)

Offset	Bytes	Field	Byte order
6	2	Logical screen width	little-endian
8	2	Logical screen height	little-endian
10	1	Packed flags	-
11	1	Background colour index	-
12	1	Pixel aspect ratio	-

The packed flags byte has five sub-fields:

```
Bit:  7  6 5 4  3  2 1 0
     [GCT][CR  ][Sort][GCT size]
```

- bit 7 - global colour table flag
- bits 6-4 - colour resolution (minus 1)
- bit 3 - sort flag
- bits 2-0 - global colour table size (actual size = `2^(value+1)`)

Every format specification looks roughly like this: fixed-width fields in a known order, with the occasional packed-bits byte.

Reading it in

The template writes itself field by field:

```
sub parse_gif_header {
    my ($fh) = @_;
    binmode $fh;

    my $header;
    read $fh, $header, 13
        or die "short read: $!";

```

(continues on next page)

(continued from previous page)

```
    my ($sig, $ver, $w, $h, $flags, $bg, $aspect) =
        unpack "A3 A3 v v C C C", $header;

    die "not a GIF" unless $sig eq "GIF";
    die "unknown version: $ver" unless $ver eq "87a" or $ver eq "89a
↪";

    return {
        version => $ver,
        width   => $w,
        height  => $h,
        flags   => $flags,
        bg      => $bg,
        aspect  => $aspect,
    };
}
```

Directive by directive:

- `A3` `A3` - six bytes of ASCII, split into signature and version. `A` (not `a` or `Z`) because the spec guarantees exactly three characters with no padding - `A` returns them unchanged and has the safer strip-on-unpack behaviour.
- `v` `v` - two little-endian 16-bit unsigned integers. The GIF format is little-endian throughout; `v` is the exact match.
- `C` `C` `C` - three unsigned bytes.

Total width: 3 + 3 + 2 + 2 + 1 + 1 + 1 = 13 bytes. We read exactly that many before unpacking; any less is an error.

The magic-number check

Two lines above illustrate a pattern you will repeat for every file format:

```
die "not a GIF" unless $sig eq "GIF";
die "unknown version: $ver" unless $ver eq "87a" or $ver eq "89a";
```

Check the magic first, unpack the rest second. Reversing the order lets nonsense data through into later stages where the error messages are less clear. For formats with a longer magic (PNG is 8 bytes, ELF is 4, WAV's "RIFF...WAVE" sandwich is 12), the same idea applies: `unpack` enough to identify the file, abort on mismatch, then `unpack` the rest.

Decoding the packed flags byte

`pack` and `unpack` cannot decode sub-byte bit fields directly (with the exception of the string-shaped `b` / `B` directives, see the *strings chapter*). The pragmatic approach is shifts and masks in plain Perl on the byte `unpack` returned:

```
sub decode_gif_flags {
    my ($flags) = @_;
    return {
        gct_present => ($flags >> 7) & 0x01,
        color_res   => (($flags >> 4) & 0x07) + 1,
        sort_flag   => ($flags >> 3) & 0x01,
        gct_size    => 2 ** (($flags) & 0x07) + 1),
    };
}
```

The pattern - "unpack the byte, pick the bits apart with shifts" - is so common it deserves to be named. Anything smaller than a byte belongs outside the template.

Worked run

```
open my $fh, "<:raw", "sample.gif" or die $!;
my $hdr = parse_gif_header($fh);
my $gct = decode_gif_flags($hdr->{flags});

printf "GIF%s %dx%d\n", $hdr->{version}, $hdr->{width}, $hdr->
 ↪{height};
printf "global colour table: %s (%d entries)\n",
       $gct->{gct_present} ? "yes" : "no",
       $gct->{gct_size};
```

On a real GIF89a file of dimensions 640×480, this prints:

```
GIF89a 640x480
global colour table: yes (256 entries)
```

Writing a header

The inverse is shorter still - every directive we unpacked, we can pack. One detail: the flags byte has to be re-assembled from its sub-fields first.

```
sub encode_gif_flags {
    my (%f) = @_;
    return (($f{gct_present} & 0x01) << 7)
         | (((log2($f{color_res})) & 0x07) << 4)
         | (($f{sort_flag}   & 0x01) << 3)
         | (log2($f{gct_size}) - 1);
}

sub write_gif_header {
    my ($fh, $hdr, $gct) = @_;
    my $flags = encode_gif_flags(%$gct);
    my $bytes = pack "A3 A3 v v C C C",
        "GIF", $hdr->{version},
        $hdr->{width}, $hdr->{height},
```

(continues on next page)

(continued from previous page)

```
        $flags, $hdr->{bg}, $hdr->{aspect};
    print {$fh} $bytes;
}
```

Same template on the pack side, same 13 bytes out, matches the GIF reader on the other end. Round-tripping with a test vector is the quickest way to gain confidence you have the template right: unpack an existing file, re-pack the result, compare byte-for-byte.

Scaling up to larger formats

The approach is compositional. A format like WAV has multiple chunks, each with its own header (`id` + `size` + `data`). You write:

- A low-level "read one chunk header" helper - two `unpack` calls on 8 bytes: 4-byte FourCC and 4-byte little-endian size.
- A dispatcher that looks at the FourCC and calls a per-chunk parser.
- Per-chunk parsers that `unpack` the chunk body according to that chunk's own byte layout.

Every layer is a small `unpack` against a tiny template. The moment a field's width depends on a value you just read, you stop, compose the new template from that value, and call `unpack` again - the same staged pattern we used for DNS names.

What you have learned

- **Magic number first, body second.** Fail fast on the wrong format.
- **Little-endian file formats use `v` / `V`**, big-endian use `n` / `N`; `<` / `>` handle the signed or 64-bit cases.
- **Sub-byte fields are plain-Perl shifts**, not template directives.
- **Larger formats decompose into many small templates.** The work of the programmer is to keep each template small and obvious.

With the reference pages as your lookup and these seven chapters under your belt, the next binary format that crosses your desk should read as a solved problem. The `pack` and `unpack` pages hold the full directive table when you need to confirm a width or a modifier.

- **Bytes and widths** - counting bytes, fixed-width integer directives, the native-vs-fixed distinction.
- **Endianness** - byte order, the `<` / `>` modifiers, the portable `n` / `N` / `v` / `V` shortcuts.
- **Strings** - `a` / `A` / `Z` (binary, text, C string), bit strings, hex strings.
- **Grouping and counts** - `()` groups, repeat counts, `[...]` forms, the `/` item-count-prefix directive.
- **Positioning** - `x` / `X` / `@` / `.` for moving around inside a template without producing a value.
- **Network protocols** - build and parse a DNS query header.

- **File formats** - parse a GIF header, the canonical “magic number plus fixed fields” file.

Two chapters (*network-protocols*, *file-formats*) are full worked examples. Walk through one of them once you have read enough of the other chapters to recognise the directives.

A first round-trip

The shortest useful example of pack and unpack working together - packing a dotted-quad IP address into four raw bytes and reading it back:

```
my $raw    = pack   "C4", 192, 168, 1, 42;   # "\xc0\xa8\x01\x2a"
my @octets = unpack "C4", $raw;              # (192, 168, 1, 42)
```

`C` packs one unsigned byte; `C4` packs four of them in a row. On the way out, the same template pulls them back into a list of four integers. Every more complicated binary format is layered on top of that idea.

Conventions in the examples

- Expected output appears as an inline `# ...` comment on the same line as the expression that produced it, or directly below the code block.
- Examples run under a byte-oriented interpreter - no `use utf8`, no `use open` pragma changing layers. If a filehandle appears, assume `binmode $fh` has already been called.
- Multi-byte constants such as `"\xc0\xa8\x01\x2a"` are written with hex escapes rather than raw byte values so the printed text matches what you see in the code.

4.2.13 I/O streams - a tutorial

Everything a Perl program does with a file, a pipe, or a buffer goes through a **filehandle**. This chapter is a recipe collection for the `open` built-in: how to read, write, append, pipe, encode, and recover from errors - each recipe small enough to copy, each warning pinned to the line that bites.

The pre-opened handles - `STDIN`, `STDOUT`, `STDERR`, `ARGV` - are available from the first line of a program:

```
print STDERR "debug: entered phase 2\n";
print STDOUT "name? ";
my $name = <STDIN> // die "no input";
while (<ARGV>) { ... }
```

Every other handle you open yourself. The modern form is always three arguments - handle, mode, target - and the handle is a lexical scalar:

```
open my $fh, "<", $path    or die "open $path: $!";
```

Read that form until it is muscle memory. Everything else on the following pages is a variation on it.

What this chapter does not cover

- **Locking.** Opening does not lock. See `flock` when you need advisory locking across processes.
- **Low-level descriptor control.** For `O_EXCL`, `O_NONBLOCK`, and explicit permission bits, see `sysopen`.
- **Directory iteration.** open is for files, pipes, and in-memory buffers. Directory reading uses `opendir` / `readdir`.

Where to go next

Reading, writing, appending

The three everyday tasks - read an existing file, create one from scratch, add to the end of one - are three different modes of the same `open` call. This page shows each in the form that actually belongs in new code.

The three-argument form

```
open my $fh, MODE, EXPR  or die "open: $!";
```

- `my $fh` is a fresh lexical. open fills it in with a handle reference when it succeeds; on failure it leaves `$fh` undefined and sets `$!`.
- `MODE` is a short string - `"<"`, `">"`, `">>"`, and a handful of others - that says what you want to do.
- `EXPR` is the pathname. It is treated as a plain string: leading or trailing whitespace is preserved, and the string is never reinterpreted as a shell command.

The older two-argument form (`open $fh, "<$path"`) still works, but it re-parses the string looking for mode characters and for the "magic" pipe syntax. Do not feed it a filename that came from a user.

Reading

```
my $path = "/etc/hostname";
open my $fh, "<", $path   or die "open $path: $!";

while (my $line = <$fh>) {
    chomp $line;
    # ... work with $line ...
}

close $fh                 or die "close $path: $!";
```

Two observations that trip up beginners:

- `<$fh>` calls `readline` under the hood. Inside a `while` head the result is tested for being defined, not truthy. That matters because a file containing the line `"0\n"` would otherwise stop the loop early. `while (<$fh>)` is a special-case spelling that Perl rewrites to `while (defined($_ = <$fh>))` for you.

- `close` can fail. On a read handle that is rare, but on a write handle it is the moment buffered data is finally flushed, so a closing error is a real error. Check it.

If you want the whole file as one string, slurp it:

```
open my $fh, "<", $path or die "open $path: $!";
local $/;                               # undef = slurp mode
my $all = <$fh>;
close $fh               or die "close $path: $!";
```

The `local $/` is the idiomatic "for the rest of this scope, turn off line splitting". When control leaves the enclosing block, `$/` returns to its previous value.

Writing (clobber)

```
open my $fh, ">", $path   or die "open $path: $!";
print $fh "first line\n";
print $fh "second line\n";
close $fh                 or die "close $path: $!";
```

- `">"` **creates** the file if it does not exist, and **truncates** it to zero bytes if it does. There is no confirmation step.
- The filehandle does not need a comma after it in a `print` invocation. `print $fh "text"` is correct; `print $fh, "text"` is a different expression that treats `$fh` as a list element.
- Buffered writes are held in memory until `close`, an explicit flush, or the buffer fills. If your program crashes between `print` and `close`, a writer with pending buffered data loses it. Always pair a writing open with a checked `close`.

Appending

```
open my $fh, ">>", $path  or die "open $path: $!";
print $fh scalar(localtime), " event\n";
close $fh                 or die "close $path: $!";
```

- `">>"` creates the file if needed and positions the write cursor at the end.
- On a POSIX kernel, each `write(2)` on an append-mode descriptor starts at the current end of file, so two processes appending to the same log never overwrite each other. A line only stays whole if it goes out in one `write(2)`: a Perl filehandle is buffered and writes whole buffers, which can end in the middle of a line. Turn on autoflush (`$fh->autoflush(1)`) so that each short `print` is one write. `PIPE_BUF` is the size limit for atomic writes to pipes; POSIX gives regular files no such number.

Closing as a real step, not a formality

A failed `close` on a write handle typically means the kernel refused to flush - out of space, I/O error, network filesystem trouble. Treat it exactly like a failed `print`:

```
close $fh or die "close $path: $!";
```

Ignoring it is how “the file looked fine during testing” becomes “production wrote half a log entry and moved on.”

What about read-write?

`"+<"` opens a file for both reading and writing, without truncating. It is a real mode and it works, but it is also the source of most “my file came out corrupted” bug reports. If you need random-access reads with in-place overwrites, reach for `sysopen`, `sysread`, and `syswrite` so the buffering layer does not guess at seek positions on your behalf.

For almost every job labelled “edit the file”, the right answer is write a new file, fsync, rename over the old one. That is the pattern atomic editors use.

Next

With reading, writing, and appending covered, the next question is what the bytes *mean*. See *Encoding and layers*.

Encoding and layers

A filehandle in Perl is not a raw pipe to bytes - it is a stack of PerlIO layers. Each layer transforms data on the way in or out: translating between characters and octets, converting newlines, applying compression, buffering. This page is about the two or three layers you will touch in everyday code.

Why this matters

When you read a text file, the bytes on disk are not the characters your program manipulates. UTF-8 encodes the character ä as two bytes (`0xC3 0xA4`). If Perl reads those bytes as if they were already characters, you get two garbage characters - classic “mojibake”. If you write characters without saying how they should be encoded, the reverse happens.

The layer stack is where you tell Perl how to translate. A correct stack for a UTF-8 text file is one line:

```
open my $fh, "<:encoding(UTF-8)", $path   or die "open $path: $!";
```

The two places layers can live

Layers attach to a handle in two ways:

1. As part of the mode string at open time, combined with the read/write direction:

   ```
   open my $fh, "<:encoding(UTF-8)",  $path or die;
   open my $fh, ">:encoding(UTF-8)",  $path or die;
   open my $fh, ">>:encoding(UTF-8)", $path or die;
   ```

2. After opening, via `binmode`:

```
open my $fh, "<", $path             or die;
binmode $fh, ":encoding(UTF-8)"     or die "binmode: $!";
```

The two forms are equivalent for freshly opened handles. `binmode` is what you reach for on a handle you did not open - `STDIN`, `STDOUT`, `STDERR`, and handles passed in from elsewhere.

The layers you actually need

Four cover ninety-nine percent of cases:

- `:encoding(UTF-8)` - characters in Perl-land, UTF-8 bytes on disk. The name of the encoding is anything Perl's `Encode` module understands: `"UTF-8"`, `"ISO-8859-15"`, `"Windows-1252"`, `"Shift_JIS"`, and so on.
- `:utf8` - a looser sibling of `:encoding(UTF-8)` that skips validation on input. It is faster and more dangerous: malformed bytes become malformed strings that misbehave later. Prefer `:encoding(UTF-8)` unless you have measured and proven the validation cost matters.
- `:raw` - bytes through, no translation. Equivalent to plain `binmode` with no second argument. Use it for binary formats, for anything you intend to hash or compare byte-exactly, and for data you are about to hand to a format-specific parser.
- `:crlf` - converts `\r\n` to `\n` on read and back on write. On Linux this is a no-op by default; you add it explicitly when reading files written on Windows and you want `\n`-terminated lines in memory.

Layers compose in writing order:

```
open my $fh, "<:raw:encoding(UTF-8)", $path   or die;
```

Reads as: strip to raw bytes first, then decode them as UTF-8. The `:raw` prefix is common when you want a predictable baseline to layer on top of, independent of whatever the open pragma set as the default.

A sensible default for text programs

If a program deals with UTF-8 text throughout, the cleanest move is to declare that at the top:

```
use open qw< :std :encoding(UTF-8) >;
```

- `:std` applies the given layer to `STDIN`, `STDOUT`, `STDERR`.
- `:encoding(UTF-8)` becomes the default layer for every subsequent `open` call inside the file.

After that pragma, bare three-argument opens decode UTF-8 automatically:

```
use open qw< :std :encoding(UTF-8) >;

open my $fh, "<", $path   or die "open $path: $!";
while (my $line = <$fh>) {
```

(continues on next page)

(continued from previous page)

```
    # $line is a character string, already decoded
}
```

Do not use a bare "<" without having either set a default encoding or explicitly planned for raw bytes. Reading without an announced encoding gives you the "Latin-1 superset" that looks right for ASCII and breaks on the first non-ASCII byte.

Binary files

For binary data, declare raw mode up front:

```
open my $in,  "<:raw", $in_path   or die "open $in_path: $!";
open my $out, ">:raw", $out_path  or die "open $out_path: $!";

my $buf;
while (my $n = read $in, $buf, 65536) {
    print $out $buf or die "write: $!";
}
close $in  or die;
close $out or die "close $out_path: $!";
```

`read` is the buffered fixed-size reader; `readline` is the line-oriented one. For binary data you almost always want `read`.

The explicit `:raw` on the output is the belt-and-braces move: if someone added `use open ":encoding(UTF-8)"` to the module three years from now, your binary writer would start mangling bytes. The layer on the handle overrides the pragma.

Changing a handle's layers after open

`binmode` without a layer argument strips all encoding layers, reducing the handle to raw bytes. With a layer argument, it pushes the given layer onto the stack:

```
binmode STDOUT;                         # raw bytes
binmode STDOUT, ":encoding(UTF-8)";     # decode strings as UTF-8
binmode STDIN,  ":encoding(MacRoman)";
```

Call `binmode` before any data moves through the handle. Pushing a layer mid-stream is technically possible but the transitional byte handling is brittle.

Next

The file-on-disk case is covered. Next: the same open call, but with a running program on the other end. See *Pipes*.

Pipes

The same open that attaches a handle to a file can attach it to a child process. Two modes select the direction:

- "-|" - you *read* the child's standard output.

- `"|-"` - you *write* to the child's standard input.

In both cases the child runs concurrently with your Perl program, and its exit status becomes available in `$?` when the handle is closed.

Reading from a command

```
open my $fh, "-|", "sort", "-u", glob("unsorted/*.txt")
    or die "pipe: $!";

while (my $line = <$fh>) {
    # ... process one sorted unique line at a time ...
}

close $fh or die pipe_close_diag($?, $!);
```

What happens:

- Perl `forks`. In the child, `stdout` is connected to the write end of a pipe, and the command is `execed`.
- In the parent, `$fh` reads from the other end. As far as your code is concerned, it is a normal read handle - `readline`, `read`, `<$fh>`, and so on all work.
- When you finish reading, `close` waits for the child to exit and records its status in `$?`.

Writing to a command

```
open my $fh, "|-", "gzip", "-9", "-c"
    or die "pipe: $!";

for my $line (@records) {
    print $fh $line, "\n";
}

close $fh or die pipe_close_diag($?, $!);
```

Symmetry: the child's `stdin` is connected to the read end of the pipe; your parent writes to the other end; `close` waits for the child to exit.

One behaviour that often surprises people: `close` on a write-pipe flushes everything buffered, shuts the pipe's write end, and *only then* waits for the child. If the child is slow, `close` blocks until the child is done. That is almost always what you want.

List form versus string form

Both open calls above used the **list form**:

```
open my $fh, "-|", "sort", "-u", glob("unsorted/*.txt")  or die;
```

Perl recognises that the third and subsequent arguments form a command and its argv. It does **not** involve a shell:

- No word splitting, no glob expansion, no redirection, no variable substitution.
- A filename containing a space or a `;` is just a filename.
- There is no shell to mis-quote against, so the class of bugs known as "shell injection" is structurally impossible.

The alternative is the **string form**:

```
open my $fh, "-|", "sort -u unsorted/*.txt"  or die;
```

Perl passes the single string to `/bin/sh -c`. The shell does word splitting, glob expansion, redirection, everything. That is what makes this form useful - you can write `"cat -n > numbered.txt"` on the command side and the shell sets up the `>` for you - and what makes it dangerous when any piece of the string came from outside your program.

Rule of thumb: **list form by default, string form only when a shell feature is the point**. When a filename glob is the reason to use a shell, consider pre-expanding the glob with Perl's own `glob` built-in and staying in list form, as the example at the top of this page did.

Decoding $?

After a pipe is closed, `$?` is set the same way `wait(2)` returns: a 16-bit value with the low byte carrying signal information and the high byte carrying the exit code. In Perl terms:

- `$? == 0` - the child exited cleanly with status 0. Success.
- `$? == -1` - `close` could not even wait for the child, because something went wrong before `fork`/`exec`. `$!` has detail.
- `$? & 0x7F` - signal number if the child died on a signal. `$? & 0x80` (the high bit of the low byte) is set if the child dumped core.
- `$? >> 8` - exit status if the child exited normally.

A diagnostic helper pulls these apart:

```
sub pipe_close_diag {
    my ($status, $err) = @_;
    return "pipe close: $err"               if $status == -1;
    my $signal = $status & 0x7F;
    return "child died on signal $signal"   if $signal;
    my $exit   = $status >> 8;
    return "child exited with status $exit" if $exit;
    return "child exited cleanly";
}
```

Pair that with every pipe `close`:

```
close $fh or die pipe_close_diag($?, $!);
```

Not checking the close on a pipe is the same mistake as not checking it on a write handle - except louder, because a child process failed silently in the background.

One command, one pipe

open with - | or | - gives you a pipe in *one* direction. There is no standard way to get both directions from a single open call, because two-way pipes to a co-process require careful buffering and deadlock avoidance that the open shape cannot express. When you need that, reach for `IPC::Open2` or `IPC::Open3` - built for the job, with the handshake caveats documented in their own POD.

Next

The third kind of thing open can attach a handle to is neither a file nor a process - it is a string in memory. See *In-memory handles*.

In-memory handles

The third thing open can attach a handle to is a plain scalar variable. The scalar is used as the "file": reads pull bytes out of it, writes push bytes into it. Nothing touches the filesystem, nothing forks.

This is the cleanest way to make a function that normally writes to a filehandle also produce an in-memory string. Any code that accepts "a filehandle" continues to work.

The signature

```
open my $fh, MODE, \$scalar  or die;
```

The third argument is a **reference** to a scalar. That reference is what distinguishes an in-memory handle from a pathname - a plain string `"$scalar"` would be a filename.

Modes work as they do for files:

- `"<"` - read from the contents of the scalar.
- `">"` - write into the scalar, clobbering any existing content.
- `">>"` - append to the scalar.

Writing into a buffer

```
my $buf = "";
open my $fh, ">", \$buf   or die "open buffer: $!";

print $fh "hello, ";
print $fh "world\n";
printf $fh "%d items\n", 42;

close $fh                 or die "close buffer: $!";

print $buf;               # hello, world\n42 items\n
```

After `close`, `$buf` holds everything that was printed. Before `close`, the data may still be sitting in PerlIO's buffer - treat it as not-yet-final, just as you would with a real file.

The typical use is capturing output from a function that expects to `print` to a handle:

```
sub render {
    my ($fh, @rows) = @_;
    for my $row (@rows) {
        print $fh join("\t", @$row), "\n";
    }
}

my $buf = "";
open my $fh, ">", \$buf or die;
render($fh, @rows);
close $fh               or die;
```

No temporary file, no `tmpfile`, no cleanup.

Reading from a buffer

```
my $source = "one\ntwo\nthree\n";
open my $fh, "<", \$source  or die "open buffer: $!";

while (my $line = <$fh>) {
    chomp $line;
    # ... work with $line ...
}

close $fh                   or die;
```

This turns any string-processing task that "would be nicer with a filehandle" into one. Test data in particular reads cleanly:

```
my $fixture = <<'END';
name,count
apples,3
pears,7
END

open my $fh, "<", \$fixture or die;
parse_csv($fh);
close $fh                   or die;
```

`parse_csv` sees a filehandle, does not know it is a string in disguise, and the test is entirely self-contained.

Layers work the same way

In-memory handles accept PerlIO layers:

```
my $buf = "";
open my $fh, ">:encoding(UTF-8)", \$buf   or die;
print $fh "café\n";
```

(continues on next page)

(continued from previous page)

```
close $fh                                       or die;

# $buf now contains the UTF-8 bytes: c a f 0xC3 0xA9 0x0A
```

Without the layer, `$buf` would contain the character string `"café\n"`. With the layer, the write step encodes to bytes on the way into the buffer. Which one you want depends on whether the buffer is destined for something that expects characters (another Perl function) or bytes (a socket, a hash function, a file on disk).

When not to use this

- **When you want capture with `fork` and `exec`.** In-memory handles are pure Perl; a child process cannot see them. If you want to capture the output of an external command, use a read-pipe as described in *Pipes*.
- **For bulk data that would not fit in RAM.** The scalar grows as you write. A one-gigabyte log is better served by a real tempfile.
- **For seekable random-access editing.** `seek` works on in-memory handles, but the semantics around growing the buffer past its current end are subtle. For random access, use a real file.

Next

Every example on every page in this chapter has been shouting `or die "...: $!"`. It is time to earn that idiom. See *Handling errors*.

Handling errors

Every I/O operation can fail. The filesystem can be full, the permissions can be wrong, the file can be missing, the network mount can vanish mid-read. Perl does not raise exceptions for these by default - it returns a false value and sets `$!`. The programmer's job is to notice.

The `or die` idiom

```
open my $fh, "<", $path   or die "open $path: $!";
```

The form has three loadbearing parts:

- **`or`**, not `||`. Precedence matters: written without parentheses, `open my $fh, '<', $path || die ...` binds the `||` to `$path`, so `die` never runs when the open fails. `or` binds lowest of all the operators involved and always does what it reads as.
- **The filename in the message.** `"$!"` alone says "Permission denied" with no context. `"open $path: $!"` says what file, in what step. When logs are all you have, context is everything.
- **`$!`, not `$@`.** `$!` is the system error - `errno` with a human-readable face. `$@` is the exception from the most recent `eval`. Mixing them up at 3am has happened to everyone.

Every open, every `close`, every `print`, every `read`, every `binmode` in production code has a check. There is no "it obviously cannot fail here". The one time it does, you will want the diagnostic.

Read `$!` immediately or not at all

`$!` reflects the **most recent** system call. A naive sequence:

```
open my $fh, "<", $path;
my $line = <$fh>;                          # might also set $!
warn "something failed: $!"  unless $line;
```

If the open failed, `$!` is already overwritten by the time `<$fh>` runs on the undefined handle. Check the return value of each call right where you call it.

Pattern: check the `close` as carefully as the open

```
open  my $fh, ">", $path   or die "open $path: $!";
print $fh @records         or die "write $path: $!";
close $fh                  or die "close $path: $!";
```

On a write handle, `close` is where the kernel actually commits the buffered data. If the disk is full, `print` may have succeeded into memory and `close` is where you find out. The close check is not a stylistic flourish; it is the one that catches the half-written log entry.

For pipes, `close` additionally waits for the child and sets `$?`. See *Pipes* for the diagnostic wrapper that decomposes `$?` into signal and exit-status components.

`autodie` - errors become exceptions

For code where every line is a `... or die ...` chain, the `autodie` pragma is worth the import:

```
use autodie qw(:all);

open my $fh, "<", $path;
while (my $line = <$fh>) {
    # ...
}
close $fh;
```

With `autodie`, built-ins like open, `close`, `print`, `read`, `chdir`, `unlink`, and a long list of siblings throw a structured exception on failure instead of returning false. The caller catches them with `eval { ... }` or a block-form `try`, and the message names the call and the path automatically.

Two practical notes:

- `autodie` is **lexical**. It applies only inside the block where it is `use`-d. A module you call from an `autodie` block does not pick up the pragma.

- `autodie` does not change the return convention for the **success** case. `open` still returns a true value; `$fh` is still the handle. You can delete your `or die` suffixes and everything else keeps working.

What the Perl exception looks like

Whether you `die` manually or let `autodie` do it, the exception travels up the stack until something catches it. Catching is `eval` plus a check of `$@`:

```
my $fh = eval {
    open my $h, "<", $path or die "open $path: $!\n";
    $h;
};
if ($@) {
    warn "could not read $path: $@";
    # ... fall back to a default, or return an error, or re-die ...
}
```

Two habits that save time:

- **Terminate die strings with \n when you want the raw message.** Without a trailing newline, Perl appends `" at FILE line N"` to the `die` string. Sometimes that is what you want; often it is noise.
- **Prefer structured objects for non-trivial errors.** A `bless { path => ..., errno => $!, step => "open" }, 'IOError'` costs three lines and gives the caller something to introspect instead of a string to regex.

A minimal checklist

Every open-and-process block should look, structurally, like this:

```
open  my $fh, MODE, $path   or die "open $path: $!";
binmode $fh, $layer         or die "binmode $path: $!"
                               if $layer_not_already_in_mode;

# ... do I/O; each op checked in-place ...

close $fh                   or die "close $path: $!";
```

If the code deviates - a missing check, a `||` where an `or` belongs, a `$@` where `$!` belongs - that is where the bug is hiding.

Where to go next

This chapter covered the everyday open. When you need flag-level control over the system call itself - for example, `O_EXCL` to refuse to clobber, or `O_NONBLOCK` for a pipe you intend to `select` on - reach for `sysopen` and its unbuffered companions `sysread` and `syswrite`.

Reference cross-links

Every recipe here links into the per-built-in reference when a function's full behaviour matters:

- `open` - full argument shape, all modes
- `close` - closing and checking the result
- `binmode` - layers after the fact
- `readline` - the function behind `<$fh>`
- `sysopen` - low-level open with flag bits
- `sysread` - unbuffered read
- `syswrite` - unbuffered write

4.2.14 References - a tutorial

A **reference** in Perl is a scalar that points at another value: an array, a hash, a subroutine, another scalar, even another reference. References are the single mechanism that turns Perl's three container types - scalars, arrays, hashes - into arbitrary nested data structures.

Without references you cannot build a hash of arrays, an array of hashes, a tree, a graph, a record with mixed fields, a callback table, or a closure. With them, you can. This tutorial walks you from the two reference-making operators, through the syntax for getting the underlying value back, up to anonymous constructors, code references, and the weak-reference trick that keeps cyclic structures from leaking.

Who this is for

You already write working Perl. You use arrays and hashes comfortably, but the moment the problem calls for a hash whose values are arrays - or an array of records, or a callback you can store in a table - the syntax stops being obvious. That gap is what references fill, and what this tutorial closes.

How this tutorial is organised

Each chapter introduces one concept and stands on its own. Read them in order the first time; jump to the relevant chapter later.

Basics - taking references and using them

There are only two ways to **make** a reference to something that already has a name, and only two ways to **use** a reference once you have it. This chapter covers both. Anonymous constructors (`[...]`, `{...}`) are covered in *Anonymous references*.

Making a reference with backslash

Put a backslash in front of any variable, and you get a scalar that refers to it:

```
my @array = (1, 2, 3);
my %hash  = (APR => 4, AUG => 8);
my $n     = 42;

my $aref  = \@array;   # reference to the array
my $href  = \%hash;    # reference to the hash
my $sref  = \$n;       # reference to the scalar
```

A reference is itself a scalar. You can copy it, pass it, store it in another data structure, without touching the thing it points at:

```
my $copy   = $aref;    # two scalars, one underlying array
my @slots;
$slots[3]  = $href;    # hash reference stored as an array element
my $pulled = $slots[3];
```

Both `$aref` and `$copy` point at the **same** array. Changing the array through one is visible through the other. To actually copy the contents, see *Anonymous references*.

What a reference prints as

Stringify a reference and you get a type tag plus an address:

```
print "$aref\n";       # ARRAY(0x55d3c1a02b40)
print "$href\n";       # HASH(0x55d3c1a02be0)
```

If you ever see output like that by accident, you printed the reference where you meant to print the contents.

Using a reference - the curly-brace form

Wherever you could have written a variable name, you can put the reference in curly braces instead. The rule is mechanical:

Operation on `@array`	Same operation via `$aref`
`@array`	`@{$aref}`
`scalar @array`	`scalar @{$aref}`
`$array[3]`	`${$aref}[3]`
`$array[3] = 17`	`${$aref}[3] = 17`
`for my $x (@array) { ... }`	`for my $x (@{$aref}) { ... }`

Hashes follow the same pattern:

Operation on `%hash`	Same operation via `$href`
`%hash`	`%{$href}`
`keys %hash`	`keys %{$href}`
`$hash{red}`	`${$href}{red}`
`$hash{red} = 17`	`${$href}{red} = 17`

And scalars:

```
my $x   = 10;
my $ref = \$x;
print ${$ref};          # 10
${$ref} = 99;
print $x;               # 99
```

The curly braces can usually be dropped when what's inside is a plain scalar variable: `@$aref` means the same as `@{$aref}`, and `$$sref` means the same as `${$sref}`. Start with the curly form until the short form stops looking ambiguous; `$$hash{key}` and `${$hash}{key}` both work, but `$$table{$country}[0]` is easier to misread than `${$table}{$country}[0]`.

Using a reference - the arrow form

The most common thing you do with a reference is fetch one element out of the array or hash it points at. The curly form works (`${$aref}[3]`) but is hard to read. The arrow form is the abbreviation:

```
$aref->[3]              # same as ${$aref}[3]
$href->{red}            # same as ${$href}{red}
```

The arrow is not cosmetic. It is the thing that distinguishes two different meanings:

```
$aref->[3]              # fourth element of the array $aref points at
$aref[3]                # fourth element of the unrelated array @aref
```

`$aref` and `@aref` are separate variables, as unrelated as `$x` and `@x`. Forgetting the arrow silently fetches from a different array that probably does not exist. Running under `use strict` turns the mistake into a compile-time error instead of a silent wrong answer at runtime.

Asking what a reference points at

The `ref` built-in returns a string describing the referent's kind:

```
ref $aref               # ARRAY
ref $href               # HASH
ref $sref               # SCALAR
ref \&printer           # CODE
ref $n                  # empty string - $n isn't a reference
```

Combine with boolean context to test "is this a reference?":

```
if (ref $maybe) {
    # it's a reference of some kind
}
if (ref $maybe eq 'ARRAY') {
    # it's specifically an array reference
}
```

See `ref` for the full list of return strings (including `CODE`, `GLOB`, `REF`, `Regexp`, and the package name when the reference has been blessed into a class).

Two common beginner mistakes

- **Treating a reference like the underlying container.** `push $aref, 1` does not push into the array; it's a type error. You must dereference: `push @{$aref}, 1` or (equivalently) `push @$aref, 1`.
- **Dropping the arrow between a variable and its subscript.** `$aref[3]` is not the same as `$aref->[3]`. The first reads from `@aref`, the second reads through the reference.

Where to go next

- *Arrays of arrays* - the two-dimensional case, and the arrow-between-subscripts rule.
- *Anonymous references* - `[...]` and `{...}` for building structures without named intermediates.

Arrays of arrays

Perl arrays hold scalars, and scalars cannot directly contain other arrays. To build a two-dimensional array - a matrix, a grid, a table of rows - you store **references** to inner arrays as the outer array's elements. This chapter shows how.

Building a matrix

The most direct form uses anonymous array constructors (`[...]`, covered in detail in *Anonymous references*) so you don't have to name every inner array:

```
my @m = (
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9],
);
```

`@m` has three elements. Each element is a scalar holding an array reference. There are four arrays in play: the outer `@m` and three anonymous inner arrays.

Reading and writing elements

`$m[1]` is the reference to the second row. To reach the third element of that row, apply the arrow:

```
$m[1]->[2]              # 6
$m[0]->[0]              # 1
$m[2]->[2] = 99;        # overwrite the bottom-right cell
```

Between **two subscripts**, the arrow is optional. This rule is what makes multidimensional access readable:

```
$m[1][2]                # same as $m[1]->[2]
$m[2][2] = 99;          # same as $m[2]->[2] = 99
```

The arrow remains **required** between a scalar variable and its first subscript: `$aref->[0]` cannot be shortened to `$aref[0]` because that would mean "first element of `@aref`". The shortcut applies only **inside** a chain of subscripts.

For three dimensions it's the same idea:

```
$cube[2][3][5]          # readable
${${$cube[2]}[3]}[5]    # the same expression fully spelled out
```

The second form is what you'd have to write without the arrow shortcut. Nobody does that.

Iterating over a matrix

To walk rows, loop over the outer array; each iteration gives you an array reference. To walk cells, dereference that reference inside the loop:

```
for my $row (@m) {
    for my $cell (@{$row}) {
        print "$cell ";
    }
    print "\n";
}
```

If you also need the indexes, use a counted loop on the outer array and on the dereferenced inner:

```
for my $i (0 .. $#m) {
    for my $j (0 .. $#{$m[$i]}) {
        print "m[$i][$j] = $m[$i][$j]\n";
    }
}
```

`$#{$m[$i]}` is the last-index operator applied to the dereferenced inner array. It's the `$#array` form with `{$m[$i]}` standing in for the array name.

Growing rows and columns

Append to a row by pushing into the inner array via a reference:

```
push @{$m[0]}, 10;      # row 0 is now (1, 2, 3, 10)
```

Append a whole row to the matrix by pushing an array reference onto the outer array:

```
push @m, [10, 11, 12]; # @m now has four rows
```

Both operations use `push` - once on a dereferenced inner array, once on the outer array directly.

Autovivification

Assigning to `$m[$i][$j]` works even if `$m[$i]` doesn't exist yet:

```
my @grid;
$grid[5][7] = 'X';
```

This creates `@grid` with six empty slots and one reference in slot 5. That reference points at a fresh anonymous array whose eighth slot holds `'X'`. Perl generates both the outer slot and the inner array automatically. This is called **autovivification** and is the reason you almost never have to write `$m[$i] = []` before using it.

Autovivification fires on *assignment* and on *dereferencing in lvalue context*. Plain reads do **not** autovivify:

```
my @grid;
my $x = $grid[5][7];   # $x is undef; no inner array created
exists $grid[5];       # false
```

Use `defined` to tell the "never existed" case from the "exists and holds `undef`" case, and `exists` to check the slot itself without autovivifying.

Common mistake: sharing a single inner array

This looks like three rows but is actually one row repeated:

```
my @inner = (0, 0, 0);
my @bad   = (\@inner, \@inner, \@inner);

$bad[0][1] = 9;
print "$bad[2][1]\n";  # 9 - they're all the same array
```

To get three independent rows, construct each one separately. The anonymous-constructor form handles this by always allocating fresh:

```
my @good = ([0, 0, 0], [0, 0, 0], [0, 0, 0]);
$good[0][1] = 9;
print "$good[2][1]\n"; # 0
```

A loop over `map` gives the same result for variable row lengths:

```
my @grid = map { [ (0) x 10 ] } 1 .. 10;  # 10×10 of zeros
```

Common mistake: storing the count instead of the elements

Assigning an array to a single scalar slot evaluates the array in **scalar context**, which yields its element *count*, not its contents:

```
for my $i (1 .. 10) {
    my @array = somefunc($i);
    $AoA[$i] = @array;          # WRONG - stores the count
}
```

After this loop every `$AoA[$i]` is a number. To store the elements, wrap them in an anonymous-array constructor so you get a reference to a fresh copy:

```
$AoA[$i] = [ @array ];          # right - a new array, copied
```

If you genuinely wanted the count, say so with `scalar` and a name that reflects it:

```
$counts[$i] = scalar @array;    # explicit - no surprise later
```

Common mistake: one reused array behind every reference

The repeated-array trap from the previous section also appears when the shared array is reused across loop iterations:

```
our @array;
for my $i (1 .. 10) {
    @array = somefunc($i);
    $AoA[$i] = \@array;         # WRONG - same array every time
}
```

Every `\@array` is a reference to the *same* array, so all ten slots end up pointing at whatever `somefunc` returned last. Again, `[ @array ]` fixes it by copying.

Declaring the array with `my` **inside** the loop is safe, however:

```
for my $i (1 .. 10) {
    my @array = somefunc($i);
    $AoA[$i] = \@array;         # fine - fresh array each iteration
}
```

`my @array` builds a new array on every pass, so each `\@array` refers to a distinct one. The reference outlives the loop body because the array it points at stays alive as long as the reference does. The constructor form `[ @array ]` is still the clearer choice; reach for the `\@array` form only when you understand why it works here.

Where to go next

- *Hashes and mixes* - hashes of arrays and arrays of hashes, the two shapes that cover most real-world records.
- *Anonymous references* - the `[...]` / `{...}` constructors used above, in detail.

Hashes and mixed structures

A hash's values are scalars, so a value cannot literally **be** an array or another hash. It can, however, be a **reference** to one - and that is enough to build every record shape real programs need: hashes of arrays, arrays of hashes, hashes of hashes, and nested combinations.

This chapter walks the three most common shapes. The patterns compose: once you have a hash of arrays and an array of hashes, you have "rows of records with multi-valued fields" for free.

Hash of arrays - grouping values under a key

The motivating example: you read lines of the form `city, country` and want to collect, for each country, the list of its cities.

```
Chicago, USA
Frankfurt, Germany
Berlin, Germany
Washington, USA
Helsinki, Finland
New York, USA
```

The natural shape is a hash whose keys are country names and whose values are references to arrays of cities:

```
my %cities_by_country;
while (<>) {
    chomp;
    my ($city, $country) = split /, /;
    push @{$cities_by_country{$country}}, $city;
}

for my $country (sort keys %cities_by_country) {
    my @cities = sort @{$cities_by_country{$country}};
    print "$country: ", join(', ', @cities), ".\n";
}
```

Two things to notice:

- `push @{$cities_by_country{$country}}, $city` works on the very first iteration for every new country. The hash entry does not exist yet, so Perl creates an anonymous array for it, stores a reference, and pushes the city in. That's autovivification again - and why you almost never need `$h{$k} = [] unless exists $h{$k}` as a separate line.
- The dereference `@{$cities_by_country{$country}}` is mechanical: "the array that this hash value points at." Wherever you'd have written `@cities` if it were a named array, you write the curly form instead.

The output is:

```
Finland: Helsinki.
Germany: Berlin, Frankfurt.
USA: Chicago, New York, Washington.
```

Array of hashes - a table of records

When each item in a list carries several named fields, store each record as a hash reference and put the references in an array:

```
my @people = (
    { name => 'Ada',    born => 1815, field => 'computing' },
```

(continues on next page)

(continued from previous page)

```
    { name => 'Hedy',  born => 1914, field => 'signals'   },
    { name => 'Grace', born => 1906, field => 'compilers' },
);
```

`$people[0]` is a hash reference; `$people[0]->{name}` or (with the arrow-between-subscripts shortcut) `$people[0]{name}` is `'Ada'`.

Typical operations:

```
# Sort by a field
my @by_year = sort { $a->{born} <=> $b->{born} } @people;

# Select rows
my @early = grep { $_->{born} < 1900 } @people;

# Project one field out
my @names = map { $_->{name} } @people;

# Mutate a record in place
$people[1]{field} = 'radio';
```

Inside `sort`, `grep`, and `map` the record is a hash reference, so you reach its fields through the arrow. `$a` and `$b` inside a `sort` comparator are hash references here, not hashes.

Hash of hashes - lookup tables keyed by structured values

When each record is keyed by a unique identifier, a hash of hashes is usually nicer than an array of hashes:

```
my %people = (
    ada   => { born => 1815, field => 'computing' },
    hedy  => { born => 1914, field => 'signals'   },
    grace => { born => 1906, field => 'compilers' },
);

print $people{ada}{born};              # 1815
$people{grace}{field} = 'COBOL';       # mutate one record
```

Iterate in a definite order by sorting the keys:

```
for my $key (sort keys %people) {
    my $rec = $people{$key};
    print "$key: born $rec->{born}, $rec->{field}\n";
}
```

Binding the inner reference to a named lexical (`$rec`) is a habit worth forming - it makes the code read top-to-bottom instead of repeating `$people{$key}{field}` five times.

Mixed - records with multi-valued fields

Put it all together: each record is a hash, and one of its fields is a reference to an array.

```
my %library = (
    'The Road'           => {
        author  => 'Cormac McCarthy',
        year    => 2006,
        genres  => ['fiction', 'post-apocalyptic'],
    },
    'Thinking in Systems' => {
        author  => 'Donella Meadows',
        year    => 2008,
        genres  => ['non-fiction', 'systems'],
    },
);

for my $title (sort keys %library) {
    my $rec = $library{$title};
    my $genres = join ', ', @{$rec->{genres}};
    print "$title ($rec->{year}, $rec->{author}): $genres\n";
}
```

`@{$rec->{genres}}` is the same pattern as before: reach into the hash, get the array reference, dereference it with `@{...}`.

A field can just as easily hold a **code reference** or a nested **lookup hash**. Nothing about the access changes; you follow the same arrow into whatever each field points at:

```
my $rec = {
    name     => 'lookup',
    CALLBACK => sub { uc $_[0] },
    LOOKUP   => { red => '#f00', green => '#0f0' },
};

$rec->{CALLBACK}->('hi');     # call the code ref - 'HI'
$rec->{LOOKUP}{red};          # nested hash lookup - '#f00'
```

`$rec->{CALLBACK}` is a code reference, so the trailing `->($arg)` calls it. `$rec->{LOOKUP}` is a hash reference, so `{red}` subscripts through it (the arrow between the two subscripts is elided).

Copying an array of refs copies the array, not the referents

When a record holds an array of references, copying that array with `[ @list ]` makes a *new* array whose elements are the *same* references. The copy and the original share their referents, so a mutation through one path is visible through every path that holds those same element refs:

```
my %TV = (
    simpsons => {
        kids => [ { name => 'Bart', age => 10 },
```

(continues on next page)

(continued from previous page)

```
                    { name => 'Lisa', age => 8 } ],
    },
);

my @also = @{ $TV{simpsons}{kids} };   # array copied; hashes shared

$TV{simpsons}{kids}[0]{age}++;         # mutate through one path
print $also[0]{age};                   # 11 - the other path sees it
 ↪too
```

`@also` is a separate array, so pushing onto it does not grow the original. But its elements are the very same hash references, so bumping age through `$TV{simpsons}{kids}[0]` changes the hash that `$also[0]` also points at. To get independent records you must copy one level deeper, building fresh hashes rather than reusing their references.

Checking what you have

Before acting on a nested slot it's often useful to know whether the slot is actually there. Use `exists` rather than `defined` when you want to avoid autovivifying empty intermediate structures:

```
if (exists $library{$title} and exists $library{$title}{genres}) {
    push @{$library{$title}{genres}}, 'essential';
}
```

Reading `$library{$title}{genres}` in lvalue context - the left side of an assignment, or as the argument of `push` - autovivifies missing levels. Reading it in rvalue context does not. The `exists` checks above keep the scan read-only.

To test the type of a reference without pulling values out of it, use `ref`:

```
ref $library{'The Road'}              # HASH
ref $library{'The Road'}{genres}      # ARRAY
```

Where to go next

- *Anonymous references* - the `[...]` / `{...}` constructors used throughout this chapter, and when a named variable is the better choice.
- *Subroutine references* - the remaining reference flavour, needed for callback tables and dispatch.

Anonymous references - `[...]` and `{...}`

You don't always want to give an inner array or hash a name before you reference it. Perl provides two constructors that build a new container and hand you a reference to it in a single expression:

- `[...]` builds a fresh anonymous **array** and returns a reference to it.
- `{...}` builds a fresh anonymous **hash** and returns a reference to it.

These are the primary way nested data structures are assembled.

Array: `[...]`

```
my $aref = [1, 'foo', undef, 13];
```

The right-hand side evaluates the list `(1, 'foo', undef, 13)`, allocates a new array, copies those four elements in, and yields a scalar holding a reference to that array. This is **exactly** equivalent to the two-step form with a named intermediate:

```
my @array = (1, 'foo', undef, 13);
my $aref  = \@array;
```

...except that no `@array` variable leaks into the surrounding scope. The anonymous array has no name; only the reference exists, and the array stays alive as long as the reference does.

Empty forms are useful as initial values:

```
my $aref = [];          # reference to a new empty array
push @{$aref}, 1, 2;    # now [1, 2]
```

Hash: `{...}`

```
my $href = { APR => 4, AUG => 8 };
```

Same idea for hashes. The expression is a list of key/value pairs (the => is just a comma that auto-quotes its left side), a new hash is allocated, the pairs are inserted, and you get a reference back:

```
my %hash = (APR => 4, AUG => 8);
my $href = \%hash;
```

is the two-step equivalent. And as with arrays, the empty form is a common starting point:

```
my $href = {};
$href->{red} = 17;
```

`{...}` vs. block - a parser corner

A bare `{...}` at statement position is parsed as a **block**, not as a hash constructor:

```
{ foo => 1 };           # block, with `foo => 1` as its expression
```

In most positions - on the right of =, inside a list, as an argument to a function - Perl figures out the hash-constructor reading automatically. If you ever need to force the hash-constructor interpretation, put + in front:

```
my $x = +{ foo => 1 }; # unambiguous: always a hash reference
```

You'll rarely need this. Inside `return +{...}` from a subroutine and in a few other "statement-like" positions, it's the fix.

Nesting - constructors inside constructors

Because each constructor yields a scalar, and scalars go anywhere, you compose structures by nesting:

```
my $tree = {
    name     => 'root',
    children => [
        { name => 'left',  children => [] },
        { name => 'right', children => [
            { name => 'r1', children => [] },
        ] },
    ],
};

print $tree->{children}[1]{children}[0]{name};   # r1
```

Every `[...]` and `{...}` allocates fresh. Unlike the "shared-inner-array" footgun from *Arrays of arrays*, nested constructors never share storage by accident.

Named intermediates vs. anonymous constructors

Both forms work. Pick based on readability:

- **Anonymous constructor** when the container is single-purpose and its only role is "be the value of this field". Nested data literals, one-off lookup tables, callback-table entries.
- **Named intermediate** when you're building the container incrementally, or when you want to use array/hash operators on it directly without repeating `@{...}` / `%{...}`:

```
my @row;
for my $x (@source) {
    push @row, transform($x);
}
$grid[$i] = \@row;
```

Copying vs. sharing

Anonymous constructors always allocate; they never share. But assigning one reference to another scalar does not duplicate anything:

```
my $a = [1, 2, 3];
my $b = $a;             # $a and $b refer to the SAME array
push @{$b}, 4;
print "@{$a}";          # 1 2 3 4
```

To get a shallow copy of the underlying container, dereference inside a fresh constructor:

```
my $copy_aref = [ @{$a} ];      # new array, same elements
my $copy_href = { %{$h} };      # new hash, same keys/values
```

The new reference is independent of the original. The **values** inside are still shared - if those values are themselves references, the two structures still point at common subtrees. That's shallow copying. For deep copying, reach for a module like `Storable` (`dclone`).

Where to go next

- *Subroutine references* - the third flavour of anonymous thing, using `sub { ... }` instead of `[...]` or `{...}`.
- *Weak references* - needed when a nested structure points back at itself.

Subroutine references

A subroutine reference is a scalar that points at a piece of code. Once you have one, you can store it in a variable, pass it to another subroutine as a callback, keep a table of them for dispatch, or call it. Code references are what make dispatch tables, callbacks, iterators, and closures possible in Perl.

Two ways to get a code reference

`\&name` - take a reference to a named subroutine:

```
sub greet { print "hello $_[0]\n" }

my $cref = \&greet;
$cref->('world');      # hello world
```

The `&` sigil is what marks `greet` as a subroutine. The backslash in front takes its reference. Without the `&`, you'd be calling the subroutine and taking a reference to the result.

`sub { ... }` - an **anonymous** subroutine. It evaluates to a reference to a fresh, nameless piece of code:

```
my $cref = sub { print "hello $_[0]\n" };
$cref->('world');
```

Anonymous subs are the direct analogue of `[...]` and `{...}` from the previous chapter: a constructor that gives you back a reference to something new and nameless.

Calling through a reference

`$cref->(@args)` is the one you reach for:

```
$cref->('world');
$cref->($x, $y);
$cref->();              # call with no arguments
```

There is also an older syntax, `&{$cref}(...)`, parallel to the `@{$aref}` / `%{$href}` form. It's still valid but the arrow form is standard everywhere modern.

Inside the called subroutine, arguments arrive in `@_` as usual:

```
my $max = sub {
    my ($a, $b) = @_;
    $a > $b ? $a : $b;
};
print $max->(3, 7);    # 7
```

Callbacks - passing code to another routine

Any subroutine that takes a code reference can accept either a named-subroutine reference or an anonymous one:

```
sub each_line {
    my ($filename, $cb) = @_;
    open my $fh, '<', $filename or die "$filename: $!";
    while (my $line = <$fh>) {
        $cb->($line);
    }
}

# Pass a named sub:
sub print_upper { print uc $_[0] }
each_line('notes.txt', \&print_upper);

# Or an anonymous one, inline:
each_line('notes.txt', sub { print uc $_[0] });
```

Inline anonymous subs are the idiomatic form when the callback body is short and specific to the call site.

Dispatch tables

A hash of code references replaces long `if` / `elsif` chains:

```
my %dispatch = (
    help => sub { print "usage: tool [cmd]\n" },
    list => sub { print "$_\n" for @items },
    quit => sub { exit 0 },
);

my $cmd = shift @ARGV;
if (my $action = $dispatch{$cmd}) {
    $action->();
} else {
    die "unknown command: $cmd\n";
}
```

Each value in `%dispatch` is a code reference. Looking up by command name and calling through the reference replaces branching with a single lookup. Adding a new command is a single hash entry.

Closures - code that captures lexicals

An anonymous subroutine that references a lexical (`my`) variable from its surrounding scope **captures** that variable. The variable survives for as long as the subroutine reference does, and each call sees the current value:

```
sub make_counter {
    my $n = 0;
    return sub { ++$n };
}

my $a = make_counter();
my $b = make_counter();
print $a->();           # 1
print $a->();           # 2
print $b->();           # 1   - independent from $a
print $a->();           # 3
```

Each call to `make_counter` creates a **new** lexical `$n` and a **new** anonymous subroutine that captures that `$n`. The subroutine returned to `$a` and the one returned to `$b` close over distinct variables, so their counters don't interfere.

This is how Perl expresses most of what other languages call "objects with private state" without any class machinery: the closed-over lexicals **are** the private state.

A closure captures by reference, not by copy. A loop like this captures the **same** loop variable:

```
my @subs;
for my $i (1 .. 3) {
    push @subs, sub { print "i=$i\n" };
}
$_->() for @subs;       # i=1 / i=2 / i=3
```

...because `my $i` in `for my $i` is re-my'd on every iteration, so each closure captures a *different* `$i`. The common mistake is using a `for`-loop variable that isn't `my`'d freshly:

```
my $i;                  # outer
my @subs;
for $i (1 .. 3) {
    push @subs, sub { print "i=$i\n" };
}
$_->() for @subs;       # i=3 / i=3 / i=3  - all share the outer $i
```

The fix is `for my $i (...)` - always prefer the inline `my`.

Introspection

`ref` on a code reference returns the string `CODE`:

```
ref $cref               # CODE
ref \&greet             # CODE
```

`defined` works as usual to tell "no value" from "any value, including a code reference":

```
if (defined $dispatch{$cmd}) { ... }
```

Where to go next

- *Weak references* - relevant when a structure of code references holds references back to objects that own it (event emitters, observer chains).

Weak references

Perl tracks object lifetime with **reference counting**. Every value knows how many references point at it; when the count drops to zero, the value is freed. This works well for tree-shaped data, where references flow one way, parent to child. It fails for **cycles**, where a chain of references eventually points back at something earlier in the chain. The cycle's members keep each other alive even when nothing outside the cycle refers to them any more.

A **weak reference** is a reference that does not increment the referent's reference count. It lets you point at something without keeping it alive. Weak references are Perl's standard answer to reference cycles.

How a cycle leaks

Consider a doubly-linked list node that holds both `next` and `prev` references:

```
sub new_node {
    my ($value) = @_;
    return { value => $value, next => undef, prev => undef };
}

my $a = new_node('A');
my $b = new_node('B');
$a->{next} = $b;
$b->{prev} = $a;
```

`$a` and `$b` each have a reference count of 2:

- `$a` is referenced by the lexical `$a` **and** by `$b->{prev}`.
- `$b` is referenced by the lexical `$b` **and** by `$a->{next}`.

When the lexicals go out of scope, their contribution drops:

- `$a` drops to count 1 (only `$b->{prev}` left).
- `$b` drops to count 1 (only `$a->{next}` left).

Neither reaches zero. Both nodes are unreachable from anywhere in your program, yet they keep each other alive for the life of the interpreter. That's a leak.

Fixing the leak with weaken

`Scalar::Util::weaken` turns an existing reference into a weak one. The reference still works for dereferencing; it just doesn't count toward keeping the target alive.

The standard pattern: whichever direction you consider "back-pointer" or "parent pointer", make **that** one weak.

```
use Scalar::Util qw(weaken);

my $a = new_node('A');
my $b = new_node('B');
$a->{next} = $b;
$b->{prev} = $a;
weaken $b->{prev};              # the back-pointer
```

Now `$a`'s reference count stays at 1 (only the lexical `$a` contributes; `$b->{prev}` is weak and does not count). When `$a` goes out of scope, `$a` is freed - and because freeing `$a` drops `$b->{next}`, `$b` is freed too. The whole structure collapses.

When a weak reference becomes undef

If the referent is freed while a weak reference still exists, the weak reference **automatically becomes undef**. You never see a dangling pointer:

```
use Scalar::Util qw(weaken);

my $target = { name => 'alive' };
my $weak   = $target;
weaken $weak;

print defined $weak ? "yes\n" : "no\n";   # yes
undef $target;                             # last strong ref gone
print defined $weak ? "yes\n" : "no\n";   # no
```

`defined $weak` is the standard way to check whether the target is still there. See `defined`.

Where weak references belong

The rule: use a weak reference for the "non-owning" pointer in any pair of references that would otherwise cycle.

- **Doubly-linked list.** `next` owns; `prev` is weak.
- **Parent/child tree with parent pointers.** `children` owns; `parent` on each child is weak.
- **Observer pattern.** The subject holds weak references to its observers; observers hold strong references to the subject. When an observer drops its reference, it goes

away; the subject's entry for it turns to `undef` and can be swept.

- **Caches that shouldn't keep entries alive.** A cache whose values are weak references does not extend the lifetime of the cached objects.

`isweak` - inspecting a reference

`Scalar::Util::isweak` tells you whether a given scalar holds a weak reference:

```
use Scalar::Util qw(weaken isweak);

my $x = { k => 1 };
my $y = $x;
print isweak($y) ? "weak\n" : "strong\n";   # strong
weaken $y;
print isweak($y) ? "weak\n" : "strong\n";   # weak
```

Useful mostly when debugging a lifetime problem. Production code rarely branches on `isweak`.

Common mistakes

- **Weakening the wrong end.** Weakening the "forward" pointer (the one that owns the successor) frees the successor immediately, before you ever use it. Weak goes on the **back-pointer**.
- **Weakening a lexical.** `weaken $ref` where `$ref` is a fresh lexical referencing `$target` makes the lexical weak. If nothing else holds a strong reference to `$target`, it vanishes at the next statement and `$ref` becomes `undef`. Weak references belong in data-structure slots, not typically in local lexicals.
- **Assuming weak references are automatic.** Perl does **not** detect cycles and weaken them for you. You have to opt in.

Beyond Scalar::Util

`Scalar::Util` ships with Perl; it is always available. For more elaborate needs - `WeakRef`, `Hash::Util::FieldHash` - reach for CPAN. For ordinary code, `weaken` is all you need.

Where to go next

You've finished the references tutorial. Good next reads:

- The `ref` reference page - the full list of strings it returns, including blessed-object cases.
- `bless` - how a plain hash reference becomes an object of a specific class.
- The object-orientation tutorial, once you're ready for classes built on top of the references you now understand.
- **Basics** - the `\` operator, scalar/array/hash dereference with `$$x`, `@$x`, `%$x`, curly-brace disambiguation, and the `->` arrow for element access.

- **Arrays of arrays** - two-dimensional data, iteration patterns, the arrow-between-subscripts rule that makes `$m[i][j]` read as a matrix.
- **Hashes and mixes** - hashes of arrays, arrays of hashes, and the common record-table shape.
- **Anonymous references** - `[...]` and `{...}` as inline constructors, when to choose them over named variables.
- **Subroutine references** - `\&name`, anonymous `sub { ... }`, calling with `->(...)`, and how closures capture lexicals.
- **Weak references** - `Scalar::Util::weaken` and why cyclic structures need it.

Related reference pages

- `ref` - ask a value what kind of reference it is.
- `defined` - the standard way to check "does this reference exist yet?" after autovivification.
- `bless` - the step that turns a plain reference into an object.
- *References (data-types reference)* - the terse one-page reference that complements this tutorial.

4.2.15 Unicode - a tutorial

Modern programs carry accented letters, currency symbols, CJK ideographs, and emoji through their input and output. Perl handles Unicode well, but only if you are explicit about where text is text and where bytes are bytes. This tutorial is the shortest path to that discipline.

These pages do not teach Unicode itself. You are assumed to know that a character set numbers characters, that an encoding maps those numbers to bytes, and that UTF-8 is one such encoding among several. What follows is the Perl-specific habit you need on top.

The one idea

A Perl string is either a **text string** - a sequence of Unicode code points, where `length` counts characters - or a **binary string** - a sequence of bytes, where `length` counts bytes. The same variable type holds both, but the meaning is different, and the boundary between them is where encoding and decoding happen.

Every bug in this area comes from losing track of which kind of string you have. Every fix comes from making the boundary explicit.

```
my $text  = "caf\x{e9}";          # 4 characters: c a f é
my $bytes = "caf\xc3\xa9";        # 5 bytes:      c a f 0xc3 0xa9
length $text;                     # 4
length $bytes;                    # 5
```

Both strings "contain" the word *café*. Only one of them can go through a regex that expects characters; only the other can go down a byte-oriented socket.

The pipeline

A program that handles text correctly follows one shape:

1. **Decode** everything coming in. Bytes on the outside become characters on the inside.
2. **Process** in characters. Regex, length, case conversion, the lot.
3. **Encode** everything going out. Characters on the inside become bytes on the outside.

Step 2 is where Perl is already comfortable. Steps 1 and 3 are where the discipline lives. This tutorial walks through both.

Chapters

Strings and encodings

A Perl string holds a sequence of integers. What those integers *mean* is up to you. In a text string they are Unicode code points; in a binary string they are bytes. Perl does not stamp "this is text" on a scalar - you carry that knowledge with the data, and the boundary between the two worlds is drawn by `encode` and `decode`.

Text in source code: `use utf8`

By default, Perl reads your source file as a sequence of bytes. A string literal containing é is a two-byte string `"\xc3\xa9"`, not the one-character string `"\x{e9}"`. That is almost never what you want when you typed é in an editor set to UTF-8.

`use utf8` tells the parser that the source file is UTF-8 and that literals should be decoded as they are read:

```
use utf8;
my $greeting = "café";
length $greeting;                # 4  - characters
```

Without `use utf8`, the same line produces:

```
my $greeting = "café";
length $greeting;                # 5  - bytes
```

Rule of thumb: if your source file contains any non-ASCII character, write `use utf8` at the top and save the file as UTF-8. The pragma only governs the source file; it has nothing to do with I/O.

The internal representation

Perl keeps text strings in a representation of its own choosing. You do not need to know what it is, and you must not depend on it. The only guarantee is:

- A text string is a sequence of code points. `length` counts them, `substr` indexes them, `chr` and `ord` map between characters and their code points.
- A binary string is a sequence of bytes. The same functions now count and index bytes.

The `utf8` pragma controls the *source*. There is also a `utf8::` namespace of functions for probing and forcing the internal flag - they exist, they are occasionally useful, and in normal application code you should not need them. Stick to `encode` and `decode`.

The Encode module

The standard way to convert between text and bytes is the `Encode` module:

```
use Encode qw(encode decode);

my $bytes = encode("UTF-8", $text);      # text  → bytes
my $text  = decode("UTF-8", $bytes);     # bytes → text
```

The first argument names the encoding. UTF-8 is the obvious default for new code, but `Encode` supports the `ISO-8859-*` family, every common Windows code page, Shift-JIS, EUC-JP, EUC-KR, GB2312, GBK, Big5, and dozens more.

Decoding is only reliable if you know which encoding the bytes are in. Nothing in the bytes themselves tells you - that information travels in a `Content-Type` header, a file's BOM, a protocol's metadata field, or a convention you have agreed with the producer. If you truly do not know, guess UTF-8 first, then fall back on `ISO-8859-1` which at least never fails (every byte is a valid code point in that encoding).

Lossy encodings

Not every Unicode character survives every encoding. Converting `"caf\x{e9}"` to `ISO-8859-1` works - é has code point `0xe9` which that encoding can hold. Converting an em-dash `\x{2014}` to `ISO-8859-1` does not - the character has no place in the target, and `encode` replaces it with a substitution marker by default.

```
use Encode qw(encode);
my $s = "em\x{2014}dash";
my $b = encode("ISO-8859-1", $s);          # "em?dash"
```

Pass a third argument to `encode` to change that behaviour - see the `Encode` module documentation for the full list of `Encode::FB_*` constants. For most application code, the default substitution is the right compromise.

A complete round-trip

Putting decoding, processing, and encoding together:

```
use utf8;
use Encode qw(encode decode);

# bytes arrive from somewhere - HTTP, a file, a socket
my $input_bytes = "caf\xc3\xa9\n";

# step 1: decode to text
my $text = decode("UTF-8", $input_bytes);  # "café\n"
```

(continues on next page)

(continued from previous page)

```
# step 2: process as text
chomp $text;
my $shouty = uc $text;                      # "CAFÉ"

# step 3: encode for output
my $output_bytes = encode("UTF-8", $shouty . "\n");
```

The text layer is short - one `uc` call - but that is the layer where `length`, regex, case conversion, and `chomp` all work in the units the program means.

When you want I/O to do this automatically

Doing `decode` on every read and `encode` on every write quickly becomes tedious. The next chapter - *I/O* - shows how to push the conversion into the filehandle itself with the `:encoding` layer, so ordinary `print` and `readline` already speak text.

I/O

The filehandle is the boundary between your program's text world and the outside world of bytes. Getting Unicode right means setting that boundary up correctly at `open` time (or with `binmode`), so that ordinary `print` and `readline` already speak the right encoding.

PerlIO layers

Every filehandle carries a stack of *layers* - small pieces of machinery that transform bytes as they cross the boundary. The relevant ones for Unicode are:

- `:raw` - no transformation. Bytes in, bytes out.
- `:utf8` - check that bytes read form valid UTF-8 and mark the string as text. Trust the user not to produce invalid UTF-8 on writes.
- `:encoding(ENCODING)` - decode on read, encode on write, for any named encoding. Rejects malformed input by default.
- `:crlf` - translate `"\r\n"` to `"\n"` on read, `"\n"` to `"\r\n"` on write. On Linux this is inactive; on Windows it is the default for text handles.

`:encoding(UTF-8)` is what you almost always want. Prefer it over the bare `:utf8` layer - the `:encoding(...)` form validates, `:utf8` does not.

Opening a file with an encoding

The three-argument form of `open` takes a mode that can include layers:

```
open my $in,  "<:encoding(UTF-8)", "input.txt"  or die "open: $!";
open my $out, ">:encoding(UTF-8)", "output.txt" or die "open: $!";

while (my $line = <$in>) {
    chomp $line;                        # characters, not bytes
    print $out uc $line, "\n";          # encoded on the way out
}
```

Every `readline` from `$in` returns a text string. Every `print` to `$out` is encoded to UTF-8 on the wire. The body of the loop sees only characters - no `decode`, no `encode`, no counting bytes by mistake.

Retrofitting with `binmode`

If the filehandle is already open - for example the standard streams, or a handle handed to you by another module - attach the layer with `binmode`:

```
binmode STDIN,  ":encoding(UTF-8)";
binmode STDOUT, ":encoding(UTF-8)";
binmode STDERR, ":encoding(UTF-8)";
```

Do this once, at the top of the program, before any I/O on those handles. After that the standard streams behave the same as a file opened with an encoding layer.

The `use open` pragma

Writing the same `:encoding(UTF-8)` layer on every `open` is noisy. The open pragma sets the default layers for all subsequent opens in the enclosing scope:

```
use open ":std", ":encoding(UTF-8)";
```

The `:std` tag also re-layers `STDIN`, `STDOUT`, and `STDERR` on the spot, so you get the effect of three `binmode` calls and a default for every later `open` in one line.

This is the one-liner at the top of a modern Unicode-aware script:

```
use utf8;
use open ":std", ":encoding(UTF-8)";
```

- `use utf8` - literals in this source file are UTF-8.
- `use open ":std", ":encoding(UTF-8)"` - stdin/stdout/stderr and every later open default to UTF-8 on the wire.

You can also push I/O to UTF-8 from outside the program with the `PERL_UNICODE` environment variable. The letters `I`, `O`, `E` select stdin, stdout and stderr individually, `S` selects all three at once the way the `use open ":std"` pragma does, and `A` decodes `@ARGV`, so `PERL_UNICODE=SA` is roughly the command-line equivalent of the two-line preamble above. To decode arguments inside the program instead, write `@ARGV = map { decode('UTF-8', $_, 1) } @ARGV`.

The matching `-C` command-line switch (`-CSA`) is not yet recognised by pperl, which is a discrepancy from stock perl; use the `PERL_UNICODE` variable or the `use open` pragma until it is implemented.

Binary I/O alongside text I/O

Some handles carry bytes, not text - image files, compressed streams, network protocols that frame their own encoding. Open those `:raw` and never attach an encoding:

```
open my $img, "<:raw", "photo.jpg" or die "open: $!";
```

Within one program, some handles can be text (`:encoding(UTF-8)`) and others binary (`:raw`). Keep the labels straight: a binary handle gives you byte strings, a text handle gives you text strings, and `$!` is the same either way.

Interactive input

A terminal does not tell you what encoding it is typing in. On modern Linux it is almost always UTF-8; on Windows the default console code page is still commonly CP1252 or CP437. The usual assumption for cross-platform scripts is UTF-8, with `binmode` applied at startup:

```
binmode STDIN, ":encoding(UTF-8)";
```

If your program must support legacy terminals, decode based on the `LANG` or `LC_CTYPE` environment variable, or accept a `--encoding` flag. There is no portable auto-detection.

Checking layers

To see which layers are on a handle, use `PerlIO::get_layers`:

```
use PerlIO;
my @layers = PerlIO::get_layers(\*STDOUT);
print "@layers\n";                          # e.g. unix perlio␣
 ↪encoding(utf-8-strict)
```

Useful when a program is silently producing double-encoded output and you want to find out which handle has the surprise layer on it.

Regex and properties

Once your strings are decoded, regex works on characters - and Perl's regex engine knows about Unicode. `\w` matches every Unicode word character, case-insensitive matching folds across scripts, and `\p{…}` gives you access to the full Unicode character database. This chapter covers the habits you need to make that work reliably.

Characters, not bytes

On a text string, every regex metacharacter speaks in characters:

```
use utf8;
my $s = "café";                             # 4 characters
$s =~ /^.{4}$/;                             # matches
$s =~ /^.{5}$/;                             # does not
```

On a binary string, the same metacharacters count bytes. That is usually not what you want - if you find yourself writing a regex against bytes that hold text, decode first.

`\w`, `\d`, `\s` and the friends

In default (Unicode) mode, these shorthands match their Unicode equivalents:

- `\w` - letters, digits, and underscore from every script.
- `\d` - decimal digits from every script (not just `0`–`9`).
- `\s` - whitespace of every kind, including U+00A0 no-break space.

```
use utf8;
"café"  =~ /^\w+$/;                      # matches
"café"  =~ /^[a-z]+$/;                   # does not - é is outside
"\x{0660}\x{0661}\x{0662}" =~ /^\d+$/;   # matches Arabic-Indic␣
 ↪digits
```

When you want the old ASCII-only meaning, add the `/a` modifier - see *Modifiers* below.

The `\p{…}` property classes

`\p{PROPERTY}` matches any character with a named Unicode property. The property names are drawn from the Unicode character database; the commonly useful ones are:

- `\p{Letter}`, shorthand `\p{L}` - any letter.
- `\p{Lu}`, `\p{Ll}`, `\p{Lt}` - upper, lower, title case letters.
- `\p{Number}`, `\p{N}`, `\p{Nd}` - all numbers, decimal digits.
- `\p{Punct}`, `\p{P}` - punctuation.
- `\p{Space}`, `\p{Zs}` - whitespace, space separators.
- `\p{ASCII}` - the 128 ASCII code points.
- `\p{Script=Greek}` - every character that belongs to the Greek script. Any script name from the Unicode data works here.
- `\p{Block=Cyrillic}` - every character in the Cyrillic block. (Script and block are different: *script* is the writing system a character belongs to; *block* is the code-point range it lives in.)

`\P{…}` is the negation - any character without that property.

```
use utf8;
my $s = "Γειά σου, κόσμε";
my @greek_letters = $s =~ /(\p{Script=Greek})/g;
scalar @greek_letters;                   # 12
```

The full catalogue of properties ships with Perl; `perldoc perluniprops` enumerates every name the engine accepts.

Modifiers: `/a`, `/u`, `/l`, `/aa`

Four modifiers change how character classes interpret themselves:

- `/u` - **Unicode mode.** `\w` matches Unicode word characters, `\d` matches Unicode decimal digits, case folding uses Unicode rules. This is the default on text strings.

- `/a` - **ASCII mode.** `\w` becomes `[A-Za-z0-9_]`, `\d` becomes `[0-9]`, `\s` becomes `[ \t\n\r\f]`. Case folding is still Unicode - `/s/ai` still matches ſ (U+017F, the long s), because it folds to `s` - which is almost never what you want when you reached for `/a`.
- `/aa` - **strict ASCII.** `/a` plus case folding is restricted to ASCII. `/s/aai` matches S but not ſ. Use this when you want an identifier match against a known-ASCII specification (HTTP header names, command keywords).
- `/l` - **locale mode.** Defers to the current POSIX locale. Almost never what you want in a Unicode-aware program; mentioned only so you can recognise it when another codebase uses it.

```
use utf8;
"café" =~ /\w+/;                          # matches whole word␣
↪(Unicode)
"café" =~ /\w+/a;                         # matches "caf" (ASCII␣
↪only)
"FOO"  =~ /foo/ai;                        # matches (ASCII case fold)
"ſ"    =~ /s/ai;                          # matches - U+017F folds to␣
↪s
"ſ"    =~ /s/aai;                         # does not - strict ASCII
```

Rule of thumb: default mode is right for text; switch to `/aa` when you are parsing an ASCII-specified protocol and want to reject smuggled non-ASCII characters.

Case-insensitive matching

`/i` folds both sides of the match using Unicode case-folding tables. This handles the obvious European pairs (é/É, ß/SS), and the less obvious ones (İ/i, fi/fi).

```
use utf8;
"Ångström" =~ /ångström/i;                # matches
"STRASSE"  =~ /straße/i;                  # matches - ß folds to SS
```

For literal-byte case folding (the old ASCII meaning of `/i`), use `/iaa`.

Named code points

`\N{...}` names a Unicode character, either by its official name or by its code-point number:

```
use charnames ();
"\N{LATIN SMALL LETTER E WITH ACUTE}" eq "\x{e9}";   # true
"\N{U+00E9}" eq "\x{e9}";                             # the same␣
↪character
```

`\N{U+HEX}` works without any pragma; the long names go through the `charnames` module, loaded automatically inside a `\N{...}` escape (or explicitly with `use charnames qw(:full)`). In a pattern, `\N{...}` is useful when you want the source to document which character you mean without relying on the reader to recognise a hex escape:

```
"café" =~ /caf\N{LATIN SMALL LETTER E WITH ACUTE}/;  # matches
```

Splitting and joining

`split` and `substr` both count in the units of the string they operate on. On a text string, the count is characters:

```
use utf8;
my $s = "a,é,b,Ω";
my @parts = split /,/, $s;                # ("a", "é", "b", "Ω")
substr($s, 2, 1);                         # "é"
```

On a binary string, the same operations count bytes. This is the most common subtle bug - an old codebase that decoded input in one function but indexed the result in another, built before the decode step was introduced.

Pitfalls

Most Unicode bugs in Perl programs show up as one of a handful of familiar symptoms. This chapter names each one, shows what produces it, and points to the fix.

"Wide character in print"

The warning:

```
Wide character in print at script.pl line N.
```

means you sent a text string to a filehandle that is not configured for text. The output gets written anyway - Perl encodes it as UTF-8 on the fly - but the warning is your notice that the configuration is wrong.

The fix is to attach an encoding layer to the handle:

```
binmode STDOUT, ":encoding(UTF-8)";
```

or, globally for the standard streams:

```
use open ":std", ":encoding(UTF-8)";
```

Do not silence the warning with `no warnings 'utf8'`. The warning is correct; it is telling you about a real missing configuration.

Double encoding: *café* becomes *cafÃ©*

Symptom: text looks almost right, but every non-ASCII character has been replaced by a pair of funny-looking bytes. `café` becomes `cafÃ©`, `Straße` becomes `StraÃŸe`.

Cause: you encoded to UTF-8, then encoded again. The second stage read the already-encoded bytes as if they were text, treated each byte as a Latin-1 code point, and encoded those to UTF-8. Two passes through UTF-8 encoding is the classic double-encode.

Common producers:

- A filehandle with `:encoding(UTF-8)` that receives an already encoded byte string. Fix: send text to the handle, or drop the layer.
- An HTTP response whose body is encoded in the application, and encoded again by a middleware that assumed the body was text.
- A database client that decodes on read, in a codebase that was written before the client learned to decode.

Diagnosis: look at the bytes on the wire. If a single non-ASCII character takes four or more bytes when it should take two, you have been through UTF-8 twice.

Missing decode: *café* stays *café* but indexes are wrong

Symptom: prose looks correct on the terminal, but `length` gives too many, `substr` cuts in the middle of a character, a regex like `/^.{5}$/` unexpectedly matches a four-character word.

Cause: you never decoded. The scalar holds UTF-8 bytes, the terminal happens to also render them as UTF-8, and the visual output hides the fact that Perl sees them as bytes.

Fix: decode on the way in, either with `binmode` on the handle or `Encode::decode` on the string:

```
use Encode qw(decode);
my $text = decode("UTF-8", $raw_bytes);
```

Heuristic: if `length $s` is unexpectedly larger than the number of visible characters, you are looking at bytes.

Mojibake: unknown encoding

Symptom: non-ASCII looks like random line-noise - `café` as `cafÃ©`, as `caf?`, as a box character. Different visual, different cause each time.

Cause: you decoded with the wrong encoding, or did not decode at all and the terminal guessed wrong.

There is no universal fix - you must know what encoding the source actually used. A few reliable anchors:

- Web: the `Content-Type` header's `charset=` parameter, or the HTML `<meta charset>` element for files served without one.
- Files from legacy Windows applications: usually `CP1252` (Western European) or a regional `CP1250`/`CP1251`/`CP1253`.
- Files from classic Mac: `MacRoman`. Rare but survives in old archives.
- Email bodies: the `Content-Type` charset of the MIME part.

If nothing tells you, try UTF-8 first; if that fails with a decode error, try `ISO-8859-1` (which cannot fail but may produce nonsense).

Mixing text and bytes in concatenation

Symptom: a string that concatenates a decoded text fragment with a raw byte fragment produces garbage after the boundary.

```
use utf8;
my $label = "café";                          # text
my $raw   = "\xc3\xa9_ok";                   # bytes that also look like␣
↪text
print $label, $raw;                          # "Wide character" or␣
↪mojibake
```

Cause: the text half goes through whatever encoding layer is on the handle; the byte half is reinterpreted as code points by that same layer, with predictable confusion.

Fix: pick a side. Either decode the byte fragment first, or encode the text fragment to match. Never concatenate the two kinds of string without a conscious choice about which side you are on.

`sprintf "%s"` is not a conversion

`sprintf` does not decode or encode; it formats. Passing a byte string through `%s` yields a byte string; passing a text string yields a text string. If you need to change encoding, call `encode` or `decode` explicitly.

Locale-dependent case folding

Symptom: `lc` or `uc` produces different results on different machines, or fails to round-trip through a regex.

Cause: the program is running under the `/l` locale-mode regex modifier, or `uc` / `lc` have been redirected through a POSIX locale with `use locale`.

Fix: do not use `use locale` for text processing. Unicode case folding is the right answer for text; POSIX locales are a remnant of byte-oriented C I/O. The only modern use of `use locale` is interacting with C libraries that consult `LC_CTYPE`.

Identifier vs data

Source code that contains non-ASCII characters needs `use utf8`. A program that processes non-ASCII data needs `binmode` or `:encoding(UTF-8)` layers on its handles. These are separate configurations, and both may be needed.

A short checklist for a new Unicode-aware script:

```
use strict;
use warnings;
use utf8;                                    # source is UTF-8
use open ":std", ":encoding(UTF-8)";         # stdio + default I/O
```

With those four lines at the top, string literals, standard I/O, and any later `open` call already speak text. Remaining Unicode work is narrow and local - decoding a byte buffer from a socket, or attaching `:raw` to a handle that must carry binary data.

Recipes

The conceptual chapters cover the discipline; this page is the quick-reference appendix. Each recipe is a short, self-contained answer to a common Unicode task - the kind of thing the classic Perl Unicode cookbook reaches for a CPAN module to do, but which pperl's builtins and regex engine handle directly.

Every example here assumes your strings are already text - decoded on the way in, as the *Strings and encodings* chapter describes. The operations below all speak characters, so a binary string fed to them will count bytes and surprise you.

Case-insensitive sort and comparison with `fc`

The `fc` builtin returns the *foldcase* of a string - the same Unicode case folding the `/i` pattern modifier uses. Two strings are equal under case folding exactly when `fc` returns the same result for both, which makes `fc` the right tool for case-insensitive sorting and comparison.

```
use v5.36;     # or: use feature 'fc';
use utf8;

# sort case-insensitively
my @sorted = sort { fc($a) cmp fc($b) } @list;

# both are true:
fc("tschüß")  eq fc("TSCHÜSS");
fc("Σίσυφος") eq fc("ΣΊΣΥΦΟΣ");
```

Folding handles the cases that simple `lc` cannot: the German sharp s folds to `ss`, and Greek final sigma ς folds together with medial σ. Reach for `fc` rather than `lc` whenever the comparison is meant to ignore case in the full Unicode sense.

This is case folding only. It does not ignore accents, and it does not impose a locale's alphabetic order - for that you would need a collation library such as `Unicode::Collate`, which is not yet usable in pperl (see the end of this page).

Normalising line breaks with `\R`

`\R` matches a single Unicode linebreak: the two-character CRLF grapheme, or any one of the vertical whitespace characters (LF, CR, form feed, vertical tab, next line, line separator, paragraph separator). It is the portable way to deal with text files that arrive from different operating systems.

```
my $text = "line one\r\nline two\rline three\n";
$text =~ s/\R/\n/g;          # normalise every linebreak to \n
```

After the substitution, `$text` uses a single `\n` everywhere regardless of how the input encoded its line endings. Use `\R` in place of an explicit `\r?\n` alternation: it covers the cases that the hand-written pattern forgets.

Matching a grapheme cluster with \X

A user-visible character - a base letter together with any combining marks - is a *grapheme cluster*. The . metacharacter matches one code point, which may be only part of such a cluster; \X matches one whole grapheme cluster.

```
use utf8;

# "é" written as e + COMBINING ACUTE ACCENT is two code points,
# but one grapheme.
my $str = "cafe\x{301}";     # c a f e + combining acute
$str =~ /^.{5}$/;            # matches - five code points
$str =~ /^\X{4}$/;           # matches - four graphemes

# find a vowel plus any combining diacritics
$str =~ / (?=[aeiou]) \X /xi;
```

\X is the right unit whenever “character” means “what the reader sees” rather than “code point”. The recipes below build on it.

First N graphemes, reverse, and length by grapheme

These three tasks all reduce to applying \X instead of indexing by code point. They work correctly no matter which normalization form the string is in.

Grab the first five graphemes:

```
my ($first_five) = $str =~ /^ ( \X{5} ) /x;
```

Reverse a string by grapheme. Reversing by code point would detach a combining mark from its base letter and corrupt the text; collecting graphemes first keeps each cluster intact:

```
$str = join "", reverse $str =~ /\X/g;
```

Count the length of a string in graphemes. The word brûlée is six graphemes but may be up to eight code points, so length and a grapheme count can differ:

```
my $count = 0;
$count++ while $str =~ /\X/g;
```

Fine-tuning UTF-8 warnings

Perl splits UTF-8 warnings into three subclasses, so you can silence one kind of malformed-Unicode warning without silencing the others. pperl recognises all three category names:

```
no warnings "nonchar";      # the forbidden non-characters
no warnings "surrogate";    # UTF-16 surrogate code points
no warnings "non_unicode";  # code points above 0x10_FFFF
```

Disable only the subclass you have a deliberate reason to allow - for example `no warnings "non_unicode"` in code that manipulates code points beyond the Unicode range on purpose - and leave the rest in force so that genuine encoding faults still surface.

Decoding @ARGV

Command-line arguments arrive as bytes. If a user passes a non-ASCII argument, you must decode it before treating it as text, exactly as you would decode any other input. The portable form maps `decode` over `@ARGV`:

```
use Encode qw(decode);
@ARGV = map { decode('UTF-8', $_, 1) } @ARGV;
```

After this line, each element of `@ARGV` is a text string and behaves under `length`, regex, and case operations in characters rather than bytes. The third argument `1` (the `Encode::FB_CROAK` check) makes a malformed argument a fatal error rather than a silent substitution, which is usually what you want for input you cannot re-request.

Custom property by code-point range

A `\p{...}` property can name a subroutine you define. When the regex engine meets an unknown property `\p{In_Foo}`, it calls `In_Foo`, which returns a list of code-point ranges (one `START\tEND` pair per line, in hexadecimal); the property then matches any character in those ranges.

```
sub In_Greek { return "0370\t03FF\n0780\t07BF\n" }

"\x{3b1}" =~ /\p{In_Greek}/;     # matches - alpha is in 0370..03FF
```

This is the portable way to define an ad-hoc character class once and reuse it by name across patterns, without spelling out the range in every regex.

Unicode modules, and the two still pending

Some recipes in the classic Perl Unicode cookbook reach for the Unicode helper modules that Perl provides as compiled extensions. Most of them are built into pperl and work as written: `Unicode::Normalize` (NFC, NFD, NFKC, NFKD normalization), `Unicode::GCString` (grapheme-aware `substr` and print-column width), `Unicode::UCD` (programmatic character category and numeric-value lookup), and `Unicode::LineBreak` (line-breaking by Unicode rules).

`Unicode::Collate` and `Unicode::Collate::Locale` (alphabetic and locale-sensitive sorting, accent-insensitive comparison) are not available yet: loading either fails with `Can't locate loadable object for module Unicode::Collate in @INC`. Until they are, there is no substitute here for locale collation; `fc` above ignores case, but not accents or a locale's alphabetic order.

- **Strings and encodings** - `use utf8`, the Encode module, what the internal representation is and is not.
- **I/O** - opening files with an encoding, standard streams, `binmode` and the `:encoding` layer.

- **Regex and properties** - matching Unicode in patterns, `\p{...}` property classes, the `/a` `/u` `/l` `/aa` modifiers.
- **Pitfalls** - "Wide character in print", double encoding, locale interactions, and how to recognise each one from its symptom.
- **Recipes** - quick-reference one-liners for case-insensitive sorting with `fc`, normalising line breaks with `\R`, and working by grapheme cluster with `\X`.

Conventions

- Expected output appears as an inline `# ...` comment or directly below the code block.
- Code points are written with `\x{...}` hex escapes so the printed text matches what you see.
- Examples assume a UTF-8-capable terminal. If yours is not, output containing non-ASCII characters will look different, but the program is still correct.

4.2.16 Boolean Logic for Perl Programmers - a tutorial

Almost every line of working Perl is shaped by boolean logic. `open ... or die`, `$h{$k} //= []`, `unless ($a == $b && $c eq $d)`, `/^(yes|no)$/`, `$flags & 0x4` - they are all the same handful of operations applied in different domains. This tutorial walks through those operations from the bottom up: what truth means in Perl, what the operators actually return, all sixteen binary truth functions and how to spell each one in Perl, the two laws that turn convoluted conditions into simple ones, and the three places that logic shows up most: bitwise arithmetic, regular expressions, and control flow.

Who this is for

You already write working Perl. You use `&&`, `||`, `if`, `unless` without thinking - but when a condition grows past three or four clauses you stop trusting it, and you reach for parentheses you suspect are not needed. This tutorial closes that gap. After reading it, you will know exactly which logical operation a given expression performs, why two seemingly different forms are the same expression, and how to reduce a tangled `unless` to a clean `if` with confidence rather than by trial and error.

Scope

Classical two-valued boolean logic - true and false, full stop. Perl's `undef` introduces a third value (the *unknown*) and a whole family of three-valued idioms (`defined`, `//`, `//=`). Those are mentioned where they intersect with two-valued logic, but a proper treatment of three-valued logic would double the size of this tutorial without finishing it; that is a separate guide.

How this tutorial is organised

Each chapter is one concept. Read them in order on the first pass; jump to the relevant chapter later when you are looking up a fact.

Truthiness

Before logic, the values logic operates on. Perl does not have a dedicated boolean type. Every scalar is either *true* or *false* when used in boolean context, and the rule is short:

> A scalar is **false** if it is one of the four values `undef`, the integer `0`, the string `""`, or the string `"0"`. Everything else is **true**.

Four false values, every other scalar - including the strings `"00"`, `"0.0"`, `"0E0"`, and `"false"` - is true.

```
if (0)         { ... }   # false  -- the integer zero
if (0.0)       { ... }   # false  -- numerically zero
if ("")        { ... }   # false  -- empty string
if ("0")       { ... }   # false  -- the string "0"
if (undef)     { ... }   # false  -- undefined value

if ("00")      { ... }   # TRUE   -- non-empty, not the string "0"
if ("0.0")     { ... }   # TRUE   -- non-empty, not the string "0"
if ("0 but true") { ... } # TRUE -- non-empty, not the string "0"
if (" ")       { ... }   # TRUE   -- single space is non-empty
if ("false")   { ... }   # TRUE   -- it's just a 5-character string
```

The `"0 but true"` idiom is not a joke. Some Perl APIs return a value that must compare equal to zero numerically (so an arithmetic test does the right thing) but be true in a boolean context. The classical case is `ioctl()` - success returning literally zero would test false. The string `"0 but true"` is numerically zero (with a warning suppressed by historical agreement) and boolean-true because it is not in the four-element false set.

```
my $rc = ioctl($fh, $cmd, $buf);
$rc == 0       and warn "numeric zero";   # would fire on success
$rc            or  warn "boolean false";  # FIRES if API returned 0
                                          # but NOT if "0 but true"
```

Two flavours of comparison

Perl has parallel families of comparison operators because it has parallel notions of equality. Numbers compare numerically; strings compare lexically. The wrong family on the wrong type is one of the most common bugs in beginner Perl.

Test	Numeric	String
equal	`==`	`eq`
not equal	`!=`	`ne`
less than	`<`	`lt`
greater than	`>`	`gt`
less or equal	`<=`	`le`
greater or equal	`>=`	`ge`
three-way (sort)	`<=>`	`cmp`

`==` on strings always succeeds for any two non-numeric strings - both convert to `0`, and `0 == 0` is true:

```
"foo" == "bar";    # TRUE  -- both stringify to numeric 0
"foo" eq "bar";    # FALSE -- the actual question

"10" eq 10;        # TRUE  -- string "10" equals numeric 10␣
↪stringified
"10" == 10;        # TRUE  -- 10 == 10 numerically
"10" lt "9";       # TRUE  -- lexical: "1" < "9"
"10" <  "9";       # FALSE -- numeric: 10 > 9
```

`<=>` and `cmp` return `-1`, `0`, or `+1` - the standard sort comparator shape. They are not boolean operators in the strict sense, but they appear inside boolean contexts often enough that you should know them.

Why this matters for logic

Every boolean operator in Perl - `&&`, `||`, `//`, `!`, `xor`, `and`, `or`, `not` - uses the four-value false set above as its input rule. Whenever a chapter below says "if `$x` is false", that is exactly what is meant: `$x` is `undef`, `0`, `""`, or `"0"`. Whenever it says "if `$x` is true", `$x` is anything else.

Two consequences worth keeping in mind:

1. **A scalar can be true without being a boolean true value.** Strings, references, blessed objects, large numbers - all true. When you ask "is this true?" you are not asking "does this equal 1?" - you are asking "is this not in the false set?".
2. **The result of a boolean operator is not always 1 or 0.** The next chapter covers this in detail. Briefly: `&&` and `||` return one of their *operands*, not a normalised boolean. This is a feature, not a quirk, and many Perl idioms depend on it.

See also

- *perlop - Numeric comparison* and *String comparison* - the reference pages for ==/eq and the rest of the comparison families introduced above.

Operators

Perl spells the boolean operators twice, in two different precedence classes. The C-style symbolic forms (`&&`, `||`, `!`) bind tightly; the word forms (`and`, `or`, `not`) bind very loosely. There is also `//` for the *defined-or* idiom, `xor` for exclusive-or, and the ternary `? :` which is logical selection in disguise.

The full inventory

Operator	Reads as	Returns	Precedence
`!`	not	normalised 1 or empty string	high
`&&`	and	left if false, else right (the *operand*)	high
`||`	or	left if true, else right (the *operand*)	high
`//`	defined-or	left if defined, else right (the *operand*)	high
`xor`	exclusive or	normalised 1 or empty string	very low
`not`	not	normalised 1 or empty string	very low
`and`	and	left if false, else right (the *operand*)	very low
`or`	or	left if true, else right (the *operand*)	very low
`?:`	ternary	middle if condition true, else right (operand)	low

Two things in this table do most of the work:

- **`&&` and `||` return an operand, not a boolean.** This is the property that makes Perl's idioms work the way they do. See below.
- **The symbolic and word forms differ in precedence, not meaning.** `&&` and `and` compute the same logical function. They differ in how tightly they bind to other operators around them. That is why you see `or die` after function calls but `||` inside expressions - the chapter on the `or die` idiom explains.

Short-circuit and the operand-return rule

`&&` and `||` evaluate left-to-right and stop as soon as the result is determined.

- `A && B` - if `A` is false, return `A` without evaluating `B`. Otherwise return `B`.
- `A || B` - if `A` is true, return `A` without evaluating `B`. Otherwise return `B`.

Crucially, the **return value is the operand itself**, not a normalised true/false. This is what makes patterns like `$config{port} || 8080` work: if `$config{port}` is set (truthy), the expression is `$config{port}`'s value; if not, it is `8080`.

```
my $port = $config{port} || 8080;
my $name = $user_input  || "anonymous";
my $list = shift        || [];
```

The same property powers `or die`:

```
open my $fh, '<', $path  or die "open $path: $!";
#  ^^^^^^^^^^^^^^^^^^^   ^^^^^^^^^^^^^^^^^^^^
#  if open returns true,  the right-hand operand
#  the whole expression   is never evaluated
#  is true and we move on
```

open returns true on success, undef on failure. On success the short-circuit stops on the left and `die` never runs. On failure the right-hand operand evaluates, which is the `die` call.

`//` - defined-or

`||` treats `0`, `""`, and `"0"` the same as `undef`. Sometimes that is wrong: a configured `0` is a real value, not a missing one. `//` short-circuits on **definedness** instead of truthiness:

- `A // B` - if A is defined, return A. Otherwise return B.

```
my $port    = $config{port} || 8080;   # 0 means "use default" -␣
↪wrong
my $port    = $config{port} // 8080;   # 0 means "0", undef means
↪"use default"
my $verbose = $opt{verbose} // 0;       # 0 is a real choice
```

The compound assignment forms `||=` and `//=` save you from repeating the variable on the left:

```
$config{port} //= 8080;   # set only if currently undef
$cache{$key}  ||= compute($key);   # set if currently false
```

`xor` and the absence of a high-precedence form

There is no `^^` in Perl. `xor` exists only as the very-low-precedence word form, and unlike `and`/`or` it does *not* short-circuit (it cannot - both sides have to be evaluated to determine the result). It returns a normalised `1` or `""`, not an operand.

```
if ($admin xor $guest) { ... }   # exactly one of the two
```

For `xor` on values rather than on truthiness, the bitwise `^` applies to integers - covered in *the applications chapter*.

`!` and `not`

Both negate. `!` is the symbolic high-precedence form; `not` is the word low-precedence form. Both return a normalised value: `1` for "the operand was false", empty string `""` for "the operand was true".

```
my $missing = ! $config{port};       # 1 if port is unset/0/""/"0"
return if not @items;                # cleaner reading than `if !
↪@items`
```

The ternary `?:`

`COND ? X : Y` is logical selection: if `COND` is true the value is `X`, otherwise `Y`. It is not just sugar for an `if`/`else` because it is an *expression* - it has a value, can be assigned, returned, passed as an argument.

```
my $label = $count == 1 ? "1 item" : "$count items";
return $ok ? \%result : undef;
```

Beware nesting: `$a ? $b : $c ? $d : $e` parses as `$a ? $b : ($c ? $d : $e)` - chained `?:` is right-associative, which is usually what you want for an `if/elsif/else` cascade but is worth knowing rather than guessing.

Why the word forms exist: precedence, not style

`||` binds more tightly than `=`. `or` binds less tightly than `=` (or `,`, or anything else of conventional weight). That is the entire reason the word forms exist.

```
my $fh = open $h, '<', $path || die "no $path: $!";
#                         ^^^^^^^^^^^^^^^^^^^^^^^^^^
# parses as: open $h, '<', ($path || die "no $path: $!")
# which is: open the file named "either $path or the die-message"
# which is: catastrophe.

my $fh = open $h, '<', $path or die "no $path: $!";
# parses as: ($fh = open $h, '<', $path) or die "no $path: $!"
# which does what you meant.
```

Rule of thumb:

- Inside an expression, where you want the value to flow into a variable or another operator: use `&&`, `||`, `!`, `//`.
- After a statement, where you want a side-effect (`die`, `warn`, `return`) to fire conditionally: use `or`, `and`, `not`.

What you should remember from this chapter

- `&&` and `||` are not predicates that return 1 or 0 - they return one of their operands. This is the foundation of `||`, `||=`, `or die`, and most short defaulting idioms.
- `//` is `||` for definedness, not truthiness. Use it when `0` is a legitimate value.
- The word forms exist to bind looser than assignment, not for style.
- `xor` exists, has no symbolic form, does not short-circuit.

See also

- *perlop - Logical* - the reference companion, with the precedence-row context and PetaPerl-specific notes.
- *perlop - Precedence* - the full table; rows 5, 15, 16, 22, 23, 24 are the operators on this page.
- *perlop - Ternary* - `?:` as logical selection.

Truth tables

A binary boolean function takes two boolean inputs and returns one boolean output. There are exactly **sixteen** such functions - that is `2^(2^2)`: each of the four input combinations `(0,0) (0,1) (1,0) (1,1)` is independently mapped to one of two outputs, so `2 × 2 × 2 × 2 = 16`. No more, no fewer.

The five with their own Perl operator (`&&`, `||`, `xor`, and `!` for each of the two negations) are the ones everyone knows, and four more are trivial: always-true, always-false, and

the two projections. The other seven are quietly there too, expressed as small compositions. This chapter lists all sixteen, names them, and shows how to write each one in Perl.

The convention

Each function is fully described by its output for the four input rows, written top-to-bottom in this order:

```
a  b   →  output
0  0   →  bit 1
0  1   →  bit 2
1  0   →  bit 3
1  1   →  bit 4
```

Reading those four output bits as a 4-bit number gives the function its index 0..15. So `AND` is `0001` (only the last row is true), `OR` is `0111`, `XOR` is `0110`. Three quarters of the table is just looking up the right row.

All sixteen

#	Truth 00 01 10 11	Name	Symbol	Perl direct	Perl idiomatic
0	0 0 0 0	false (contradiction)	⊥	`0` / `""`	-
1	0 0 0 1	AND	∧	`$a && $b`	`$a and $b`
2	0 0 1 0	a AND NOT b	a ∧ ¬b	-	`$a && !$b`
3	0 0 1 1	a (left projection)	a	`$a`	-
4	0 1 0 0	NOT a AND b	¬a ∧ b	-	`!$a && $b`
5	0 1 0 1	b (right projection)	b	`$b`	-
6	0 1 1 0	XOR	⊕	`$a xor $b`	`!$a != !$b` / `($a?1:0) != ($b?1:0)`
7	0 1 1 1	OR	∨	`$a || $b`	`$a or $b`
8	1 0 0 0	NOR	↓ (Peirce)	-	`!($a || $b)` / `!$a && !$b`
9	1 0 0 1	XNOR / equivalence	↔	-	`!$a == !$b` / `($a?1:0) == ($b?1:0)`
10	1 0 1 0	NOT b	¬b	`!$b`	`not $b`
11	1 0 1 1	b implies a	b → a	-	`!$b || $a`
12	1 1 0 0	NOT a	¬a	`!$a`	`not $a`
13	1 1 0 1	a implies b	a → b	-	`!$a || $b`
14	1 1 1 0	NAND	↑ (Sheffer)	-	`!($a && $b)` / `!$a || !$b`
15	1 1 1 1	true (tautology)	⊤	`1`	-

Five rows have a direct Perl operator (`&&`, `||`, `xor`, and `!` for each of the two negations). The two constants and the two projections need no operator at all. The other seven are short compositions of those. There is no fundamental reason Perl spells some directly and not others - it is historical, inherited from C. Every column on the right is something you can type today.

Two things are worth pulling out of the table.

Implication: →

Implication is the operator nobody thinks they use, and everyone uses constantly. `a → b` is read "a implies b" and is true in every case **except** when `a` is true and `b` is false. That is it.

a	b	a → b
0	0	1
0	1	1
1	0	0
1	1	1

The Perl spelling is `!$a || $b`. This identity -

a → b ≡ ¬a ∨ b

- is so common that it deserves a name; and the name is *material implication*. Once you see it once, you spot it everywhere:

```
# "if the user is admin, the request must be authenticated"
# admin → authenticated
die unless !$user->is_admin || $req->is_authenticated;

# read aloud: "either not admin, or authenticated, or both"
```

The `unless ... !X || Y` shape is awkward. The next chapter on De Morgan's laws shows how to flip it into something readable.

Worth keeping at hand is the lookup table for flipping between common shapes:

Goal	One way	Equivalent way
a → b	¬a ∨ b	¬(a ∧ ¬b)
a ↔ b (XNOR)	(a ∧ b) ∨ (¬a ∧ ¬b)	¬(a ⊕ b)
a ⊕ b (XOR)	(a ∧ ¬b) ∨ (¬a ∧ b)	¬(a ↔ b)
¬a	a ↑ a (NAND self)	a ⊕ 1

The last column is a teaser for the chapter on functional completeness.

Equivalence and XOR are duals

XOR is true exactly when a and b differ. Equivalence (XNOR) is true exactly when they agree. They are negations of each other, and that is exactly what the truth-table rows say (row 6 is `0 1 1 0`, row 9 is `1 0 0 1` - bitwise negation).

In Perl the cleanest spelling for both is to canonicalise each operand to `0` or `1` first and then compare with `==` or `!=`:

```
my $a01 = $a ? 1 : 0;
my $b01 = $b ? 1 : 0;

my $xor  = $a01 != $b01;   # true if they differ
my $xnor = $a01 == $b01;   # true if they agree
```

Why the canonicalisation? Because `$a` and `$b` may be any truthy values - the strings `"yes"` and `5` are both true but not equal. Comparing them directly with `==` or `eq` answers

a different question. Boolean XOR asks "do they have the same *truthiness*?", not "are they equal?".

A short alternative without the temporaries is `!$a != !$b` - each `!` returns the canonical `1` or `""`, and then `!=` compares those. It works but reads sideways; the explicit `? 1 : 0` form is more readable.

Reading the table backwards

You will sometimes know the truth pattern you want and need to find the operator. Example: "I want a function that is true unless both `a` and `b` are false." Read the rows: `a=0,b=0 → 0`, `a=0,b=1 → 1`, `a=1,b=0 → 1`, `a=1,b=1 → 1`. That is `0 1 1 1` - row 7 - OR.

This sounds trivial here but is exactly how you debug a tangled condition: write down its truth table, read off the four bits, find the row, and you have the canonical name and the simplest spelling.

What you should remember from this chapter

- There are exactly sixteen binary truth functions; every boolean expression in any language computes one of them.
- Five of the sixteen have a dedicated Perl operator, and the two constants and two projections need none. The other seven are short compositions and the table above is the lookup.
- Implication `a → b` is `¬a ∨ b`. It shows up constantly under the surface of `unless` and `if` clauses.
- XOR and XNOR are negations of each other. The cleanest Perl spelling for both starts by canonicalising operands to `0` or `1`.

De Morgan's laws

Two identities, due to Augustus De Morgan, written for over a century in every textbook of formal logic, and the single most useful pair of equations a working programmer can know.

$$\neg(a \wedge b) \equiv \neg a \vee \neg b$$

$$\neg(a \vee b) \equiv \neg a \wedge \neg b$$

In words: pushing a `not` through a parenthesised expression flips every operator inside ($\wedge$ becomes $\vee$, $\vee$ becomes $\wedge$) and negates each operand. Equivalently: pulling a `not` out flips every operator and removes the negations on the operands.

Why this matters in Perl

`unless` is the canonical case. `unless (X)` is `if (!X)` - and the moment `X` is itself a compound, you have a leading `not` over a parenthesised expression: exactly the De Morgan setup.

```
unless ($a == $x && $b == $y) { ... }
```

The condition under the `unless` is `!($a == $x && $b == $y)`. Apply De Morgan: this is `!($a == $x) || !($b == $y)`. Each `!=` is the negation of the corresponding `==`, so the clean form is:

```
if ($a != $x || $b != $y) { ... }
```

Three transformations happened, each mechanical:

1. `unless` flipped to `if` (the outer `not`).
2. `&&` flipped to `||` (De Morgan, inside).
3. `==` flipped to `!=` on each operand (the negations De Morgan placed on the operands cancel into the opposite comparison).

This is *not* a stylistic preference. Every programmer who has debugged a wrong condition has stared at `unless (! ... && ...)` for a minute too long; the flat `if` reads in one pass.

A worked example

A real shape, slightly tangled:

```
unless ($x !~ /^\d+$/ && $y !~ /^[a-z]+$/) {
    print "at least one of \$x and \$y looks valid\n";
}
```

This says: *unless* `$x` is non-numeric and `$y` is non-alphabetic, print. Five negations stacked across two lines. Walk through it:

The condition under the `unless` is

```
!( $x !~ /^\d+$/  &&  $y !~ /^[a-z]+$/ )
```

De Morgan flips the `&&` to `||` and negates each side:

```
!($x !~ /^\d+$/)  ||  !($y !~ /^[a-z]+$/)
```

`!($x !~ ...)` is just `$x =~ ...` (double negation of a regex match). Same for the right side:

```
$x =~ /^\d+$/  ||  $y =~ /^[a-z]+$/
```

So the cleaned-up version is:

```
if ($x =~ /^\d+$/ || $y =~ /^[a-z]+$/) {
    print "at least one of \$x and \$y looks valid\n";
}
```

Five negations down to zero. The condition now reads exactly the way the comment said it: *if* `$x` *is numeric or* `$y` *is alphabetic.*

The same trick for OR

The second form of the law is the symmetric case:

```
unless ($a == 0 || $b == 0) { divide($a, $b) }
```

becomes

```
if ($a != 0 && $b != 0) { divide($a, $b) }
```

- same mechanical steps, only this time the inner operator is `||` flipping to `&&`.

NAND, NOR - same identity, read the other way

The NAND row in the truth table from the previous chapter has two entries in the "Perl idiomatic" column:

```
14   NAND    !($a && $b)    /    !$a || !$b
```

These are equal *because of De Morgan's first law*. NAND is exactly the negation of AND, and pushing the negation in flips the operator and negates the operands. The NOR row works the same way for the second law:

```
8    NOR     !($a || $b)    /    !$a && !$b
```

So De Morgan is not a separate fact you have to remember alongside NAND and NOR - it *is* the identity that turns one spelling into the other.

Mechanics, in one paragraph

Negation distributes over ∧ and ∨ by flipping the connective and negating each operand. Negation also self-cancels: ¬¬x ≡ x. The two rules together let you push a `not` arbitrarily deep, or pull it arbitrarily out, of any expression built from ∧, ∨, and ¬. That is the whole content of De Morgan's laws - every "convoluted unless" you ever rewrite is one application of those two rules plus double-negation cancellation.

Practical procedure

When you stare at a condition you don't like:

1. If the outer keyword is `unless`, prepend a `not` mentally and make it `if`.
2. If the leading `not` is over a parenthesised `&&` or `||`, distribute it: flip the connective, negate each operand.
3. If a subexpression is `!(x == y)`, replace with `x != y`. Same for `!~` ↔ `=~`, `!defined` ↔ `defined`, etc.
4. Stop when there are no more leading negations to push or no more double-negations to cancel.

In practice you will do all four steps in your head in a single pass. The point of the procedure is that there is one - you are not searching for the right form, you are deriving it.

What you should remember from this chapter

- Two identities: pushing ¬ through ∧ flips it to ∨ (and vice versa) and negates each operand.
- Most "ugly `unless`" is one mechanical De Morgan flip away from a clean `if`.
- NAND and NOR are not extra facts; they are De Morgan's laws read as a pair of equivalent spellings.

See also

- *perlop - Logical* - the operator-reference companion. The De Morgan transforms here apply to every Perl expression that uses `&&`, `||`, `!`, and the `unless` keyword.

Functional completeness

A set of boolean operators is *functionally complete* when every one of the sixteen binary truth functions can be built from operators in that set, by composition. The classical complete sets you already know are {∧, ∨, ¬} and {→, ⊥}. The surprising - and useful - result is that **a single operator suffices**, but only two of the sixteen binary functions have this property: NAND and NOR.

NAND is named after Henry M. Sheffer (1913) and written ↑; NOR is named after Charles Sanders Peirce (who described it earlier) and written ↓. They are the two *Sheffer strokes*.

Building everything from NAND

NAND alone gives you all sixteen functions. The constructions below show the four most useful - NOT, AND, OR, XOR - in terms of ↑. Think of `a ↑ b` as `!($a && $b)`.

Target	In terms of NAND	Why
¬a	`a ↑ a`	`!(a ∧ a)` is `!a`
a ∧ b	`(a ↑ b) ↑ (a ↑ b)`	NAND then negate the result, i.e. `!!(a ∧ b)` ≡ `a ∧ b`
a ∨ b	`(a ↑ a) ↑ (b ↑ b)`	De Morgan: `a ∨ b` ≡ `¬(¬a ∧ ¬b)` ≡ `¬a ↑ ¬b`
a ⊕ b	`(a ↑ (a ↑ b)) ↑ (b ↑ (a ↑ b))`	XOR is the AND of (a ∨ b) and ¬(a ∧ b) - left as exercise

The same constructions exist with NOR; they are the De Morgan duals.

In Perl

You will not write your code this way in production - `(a ↑ a) ↑ (b ↑ b)` is a worse spelling of OR than `||` is. But it is worth seeing the constructions actually run, because the abstract claim ("everything reduces to one operator") becomes concrete when you type it.

```
sub nand { !($_[0] && $_[1]) }       # the only primitive

sub not_a { nand($_[0], $_[0]) }                        # ¬a
sub and_  { my $n = nand($_[0], $_[1]); nand($n, $n) } # a ∧ b
sub or_   { nand(nand($_[0], $_[0]), nand($_[1], $_[1])) }   # a ∨ b
sub xor_  {
    my ($a, $b) = @_;
    my $n = nand($a, $b);
    nand(nand($a, $n), nand($b, $n));
}

# Verify across all four input combinations:
for my $a (0, 1) {
    for my $b (0, 1) {
        printf "a=%d b=%d  not=%d and=%d or=%d xor=%d\n",
            $a, $b,
            not_a($a)         ? 1 : 0,
            and_($a, $b)      ? 1 : 0,
            or_($a, $b)       ? 1 : 0,
            xor_($a, $b)      ? 1 : 0;
    }
}
```

Output:

```
a=0 b=0  not=1 and=0 or=0 xor=0
a=0 b=1  not=1 and=0 or=1 xor=1
a=1 b=0  not=0 and=0 or=1 xor=1
a=1 b=1  not=0 and=1 or=1 xor=0
```

Each row matches the truth table from the previous chapter.

Why this matters

Three things, in order of practical-to-deep:

1. **It explains the bestiary of identities.** NAND, NOR, and the eleven non-direct rows of the truth-table chapter are not a zoo of separate facts. They are all expressible from a single operator; the eleven non-direct spellings in Perl are just readable shortcuts for short NAND or NOR compositions.
2. **It is how silicon actually works.** Static-CMOS logic produces NAND and NOR more cheaply than AND and OR - a two-input AND gate is built as a NAND followed by an inverter. Every other gate on a chip is a composition of NANDs (or NORs). The pattern repeats in solver and constraint-engine internals: SAT solvers normalise to CNF precisely because OR-of-(literals) and AND-of-(clauses) is the minimal vocabulary they need.
3. **It teaches the discipline of canonical form.** Every boolean expression has many equivalent shapes, but only a few are *canonical*. CNF (conjunctive normal form: an AND of ORs of possibly-negated literals) and DNF (disjunctive normal form: an OR of ANDs) are the two most common. Reducing a tangle to CNF or DNF is a pure me-

chanical procedure based on De Morgan plus distributivity, and once your expression is in normal form you can compare it with another normal-form expression to test equivalence - without truth tables, without testing.

You will not implement a SAT solver next week. But the next time you find yourself writing `if ((!$a && $b) || (!$c && $b) || ($a && !$c))` and wondering whether the previous version was equivalent, you have a tool: distribute, factor, normalise. Two expressions that reduce to the same DNF compute the same function.

What you should remember from this chapter

- NAND and NOR are each individually sufficient to express every boolean function. No other two-input operator has this property.
- The constructions are short and you can verify them in Perl in ten lines.
- The point is not to write code in NAND, but to know that all sixteen binary functions live on one foundation. The apparently disparate rows of the truth-table chapter are one family.

Applications

The previous chapters were about boolean logic in the abstract. This one is about where it shows up most often in working Perl: bitwise arithmetic, regular expressions, and the short-circuit idioms that take the place of explicit conditionals.

Bitwise: logic on integer bits

The bitwise operators apply boolean operations to each pair of bits in two integers in parallel. A 32-bit integer is, from the operators' point of view, thirty-two parallel one-bit values.

Op	Reads as	Per-bit rule
&	bitwise AND	each output bit = a∧b
|	bitwise OR	each output bit = a∨b
^	bitwise XOR	each output bit = a ⊕ b
~	bitwise NOT	each output bit = ¬a
<<	left shift	shift bits left, zero-fill from right
>>	right shift	shift bits right

The same boolean operators - ∧, ∨, ⊕, ¬ - appear here in purely numeric form. That is not a coincidence; it is the definition.

Setting, clearing, toggling, testing a flag

The four basic flag operations on a single bit:

```
use constant FLAG_VERBOSE  => 0x01;
use constant FLAG_DRY_RUN  => 0x02;
use constant FLAG_RECURSE  => 0x04;
```

(continues on next page)

(continued from previous page)

```
use constant FLAG_FORCE    => 0x08;

my $flags = 0;

$flags |=  FLAG_VERBOSE;       # SET    -- OR with the bit
$flags |=  FLAG_DRY_RUN;       # SET another
$flags &= ~FLAG_DRY_RUN;       # CLEAR  -- AND with the inverted bit
$flags ^=  FLAG_VERBOSE;       # TOGGLE -- XOR with the bit
my $on = $flags & FLAG_RECURSE;# TEST   -- AND, then test truthiness
```

Each of these is a one-bit application of a boolean operator. Set is `bit ∨ flag`, clear is `bit ∧ ¬flag`, toggle is `bit ⊕ flag`, test is `bit ∧ flag`.

XOR-swap: swapping without a temporary

XOR has two properties that combine into a memorable trick: `a ⊕ a = 0`, and `a ⊕ b ⊕ b = a`. Apply them in sequence and you can swap two integers without a third variable:

```
my ($a, $b) = (0xFEED, 0xBEEF);

$a ^= $b;      # a := a ⊕ b
$b ^= $a;      # b := b ⊕ (a ⊕ b) = a
$a ^= $b;      # a := (a ⊕ b) ⊕ a  = b

print "a=$a b=$b\n";   # a=48879 b=65261  (0xBEEF, 0xFEED)
```

The trick is not actually useful in Perl - `($a, $b) = ($b, $a)` is faster, clearer, and works on any scalar including strings and references. But it shows up in two places that matter:

- **Embedded code without a free register.** A microcontroller with three values to juggle in two registers reaches for this.
- **Interview folklore.** Knowing it exists and *why* it works (XOR is its own inverse) is worth the thirty seconds it takes to read.

The reason it works is exactly the boolean identity from the truth-table chapter: `x ⊕ y ⊕ y = x`. Each of the three lines above is one application of that identity.

Common bit tricks

A handful of patterns you will see in performance-sensitive code:

```
$x &  ($x - 1)          # $x with its lowest set bit cleared
($x & ($x - 1)) == 0    # true when $x is a power of two (and non-
↪zero)
$x | -$x                # signed: zero iff $x was zero, non-zero␣
↪otherwise
($x >> 31) & 1          # the sign bit, on a 32-bit signed integer
```

(continues on next page)

(continued from previous page)

```
1 << $n                 # the integer with only bit $n set
$x & (1 << $n)          # is bit $n set in $x?
```

Each of these is an exercise in tracking what the bits do - pure boolean reasoning applied 32 (or 64) times in parallel.

Regular expressions: logic on sets of strings

A regex matches a *set* of strings. The empty set, the singleton `{"foo"}`, the infinite set "anything that begins with a digit" - all are sets, and the regex denotes one of them.

Once you see a regex as a set, the boolean operations have geometric meaning:

Boolean operation	On sets	In regex syntax
OR (∨)	union	alternation: `foo|bar`
AND (∧)	intersection	lookahead pair: `(?=.{4})(?=\d+$)`
NOT (¬)	complement	negative lookahead: `(?!foo)`
AND-NOT (a ∧ ¬b)	difference	`(?=A)(?!B)A`

Two specifics worth pulling out.

Alternation is OR over sets

```
$s =~ /yes|no|maybe/;
```

Matches the union of three singleton sets. There is nothing more to it; `|` in regex syntax is precisely the boolean ∨.

Inside a character class the meaning is the same:

```
$s =~ /[abc]/;          # union of {"a"}, {"b"}, {"c"}
$s =~ /[^abc]/;         # complement: anything NOT in {"a","b","c"}
```

`[^...]` is the regex syntax for ¬ applied to a character set.

Lookarounds are AND and NOT

A regex without lookaround consumes characters as it matches; the two ends of an alternation `A|B` cannot both match the same span (only one branch wins). To express *both* `A` *and* `B` at the same position, you need lookaround: a zero-width assertion that demands a property without consuming.

```
# matches only if the rest of the string is BOTH digits AND ≤ 4␣
↪chars
$s =~ /^(?=\d+$)(?=.{1,4}$)/;
#       \_____/\_______/
#         A         B          intersection: A ∧ B
```

`(?=...)` is positive lookahead; `(?!...)` is negative lookahead (the boolean NOT). Combining them gives you the full set algebra on regex predicates:

```
# "starts with a digit but is not the literal '0'":
# (digit) ∧ ¬(literal "0")
$s =~ /^(?=\d)(?!0$)/;
```

Why the framing helps

Once you read regex this way, the readability of complicated patterns improves dramatically. A regex with two lookaheads is not "two regexes glued together" - it is the intersection of two sets. A negative lookahead followed by a positive match is not "first reject, then match" - it is set difference. The boolean algebra you already know is the algebra of regexes; only the notation changes.

Control flow: short-circuit logic as conditional execution

The chapter on operators introduced the operand-return rule for `&&`, `||`, and `//`. That rule, plus precedence, generates a small family of idioms that replace explicit `if`/`else` for short conditional logic:

```
# defaulting
my $port = $cfg{port} // 8080;

# guard with side-effect
open my $fh, '<', $path  or die "open $path: $!";

# lazy initialisation (set if currently false)
$cache{$key} ||= compute($key);

# lazy initialisation (set if currently undef - a real `0` is kept)
$cfg{retries} //= 3;

# guard chain: do step 2 only if step 1 succeeded
my $rc = step_one() && step_two();

# logical selection inside an expression
my $label = $count == 1 ? "1 item" : "$count items";
```

Each of these is a boolean expression chosen for its *side-effects* - the short-circuit decides whether the right operand even runs. `open ... or die` works exactly because `or` does not evaluate its right side when the left is true.

Two pieces of advice.

Use `//` and `//=` when `0` or `""` is a valid value. This single change has prevented more bugs than any other modern Perl idiom; `||` defaulting on a configured zero is one of the classic ways to lose data.

Reach for `?:` when you would otherwise build an `if`/`else`/scalar-assign triplet. The ternary keeps the expression as an expression and the assignment in one line:

```
# verbose
my $kind;
if ($n == 1) { $kind = 'singular' }
else         { $kind = 'plural'   }

# idiomatic
my $kind = $n == 1 ? 'singular' : 'plural';
```

Don't nest `?:` more than two levels deep. Past two, the chained form is harder to read than the `if`/`elsif`/`else`. The ternary is for *choosing*; sequential decisions belong in a block.

What you should remember from this chapter

- Bitwise `&`, `|`, `^`, `~` are the boolean operators applied bit-by-bit in parallel; flag set/clear/toggle/test are the four one-bit applications.
- A regex denotes a set of strings; `|` is union, `[^...]` is complement, `(?=)` and `(?!)` make intersection and difference.
- `&&`, `||`, `//`, `?:` give you most of conditional control flow without writing `if`. Use `//` when `0` is real; use `?:` for short selection inside an expression.

See also

- *perlop - Bitwise* - the reference companion to the bitwise section, with the full shift semantics and the suffixed `&.` `|.` `^.` `~.` byte-string forms.
- *perlop - Binding* - `=~` and `!~`, the operators that connect strings to regex matching.
- *Regular expressions guide* - the full regex language reference.
- *perlop - Logical* - the short-circuit operators behind the control-flow idioms.

What you will be able to do after reading

- Look at any boolean expression in Perl source and name the truth function it computes.
- Convert between the four equivalent shapes of an implication (`a → b`, `!a || b`, `!(a && !b)`, the contrapositive) without having to think.
- Reduce an `unless` with two negated subclauses to a flat `if` in one mechanical step.
- Recognise when a regex alternation is a logical OR over sets and when a lookaround pair is a logical AND, and use that framing to read complicated regexes.
- Use the bitwise operators to set, clear, toggle and test individual flags without consulting a reference.
- Pick the right short-circuit idiom (`||`, `//`, `||=`, `//=`, `or die`) for the situation, knowing why each one works.

4.2.17 Interprocess communication - a tutorial

Two programs that need to talk have a handful of channels to choose from: a pipe, a named FIFO on disk, a socket over the network, a socket pair in memory, or a slab of System V shared memory. This tutorial is a cookbook. Each page hands you a complete, runnable program - a producer and a consumer, or a client and a server - that you can copy, run, and adapt.

The mechanics of each channel - what `socket` returns, how `fork` splits a process, what `%SIG` does - are already documented in the per-built-in reference and the signals chapter. This tutorial does not restate them. It assembles them into working IPC programs and links each call back to its reference page.

PetaPerl runs only on Linux, so everything here is POSIX. There is no Win32 IPC, no named-pipe-on-Windows alternative, no `Win32::Process`. The recipes use the system calls a Linux kernel actually provides.

Choosing a channel

Pick the lightest channel that carries what you need:

- **A child running a command** - you want the output of `sort` or the input of `gzip`. Use a *pipe open*, or `IPC::Open2` / `IPC::Open3` when you need both directions. See *Subprocesses and co-processes*.
- **Two related processes, one machine** - a parent and a child it just forked, sharing a byte stream. Use `pipe` or `socketpair`. See *Subprocesses* and *UNIX-domain and UDP sockets*.
- **Unrelated processes, one machine** - two programs started independently that rendezvous on a known path. Use a *FIFO* or a UNIX-domain socket. See *FIFOs* and *UNIX-domain and UDP sockets*.
- **Processes on different machines** - a network client and server. Use a TCP socket. See *TCP sockets*.
- **A datagram, fire-and-forget** - no connection, message boundaries preserved. Use a UDP socket. See *UNIX-domain and UDP sockets*.
- **A shared counter or buffer with no copying** - System V semaphores and shared memory. Use `semget` and `shmget`. See *System V IPC*.

How this tutorial is organised

Each page stands on its own. Read the one that answers your question.

Subprocesses and co-processes

The most common IPC task is the simplest: run an external command and talk to it. This page covers the spectrum, from a one-way pipe open to a two-way co-process - a child you both write to and read from.

The mechanics of a one-direction pipe open (`"-|"` and `"|-"`) are covered in full on the *Pipes* page. This page picks up where that one stops: when one direction is not enough, and when you need a raw `pipe` and `fork` of your own.

One direction: the pipe open

To read a command's output, or to feed a command its input, a pipe open is all you need:

```
open my $fh, "-|", "sort", "-u", "names.txt" or die "pipe: $!";
while (my $line = <$fh>) { ... }
close $fh;
```

That is the whole story for one-way work, and it is told on the *Pipes* page - including the list-versus-string form, why list form sidesteps the shell, and how to decode the child's exit status. Start there if one direction is enough.

Both directions: `IPC::Open2`

When you need to write to a command *and* read its reply - a filter you drive interactively, a co-process - one pipe is not enough. `IPC::Open2` opens two: one into the child's `stdin`, one out of its `stdout`.

```
use IPC::Open2;

my $pid = open2(my $from_child, my $to_child, "tr", "a-z", "A-Z");

print $to_child "hello\n";
close $to_child;                    # signal end-of-input to the
 ↪child

my $reply = <$from_child>;
print "got: $reply";                # got: HELLO

waitpid($pid, 0);
```

open2 returns the child's process id. The argument order is the trap worth memorising: **reader handle first, writer handle second**, then the command. `open2(OUT, IN, ...)` - the handle you read *from* comes before the handle you write *to*.

The deadlock you must design around

A co-process can deadlock, and `IPC::Open2` does nothing to prevent it. The shape of the trap:

- You write a large block to the child.
- The child reads some, produces output, and that output fills the pipe back to you.
- The child blocks writing to a full pipe; you block writing to a child that has stopped reading. Neither moves.

The recipe above sidesteps it by closing `$to_child` before reading - sending the whole input, then signalling end-of-file, then draining the reply. That works when the input fits comfortably in the pipe buffer. For unbounded streaming in both directions at once, you need non-blocking I/O or a `select` loop; the two-pipe shape alone is not enough.

Capturing standard error too: IPC::Open3

`IPC::Open2` joins you to `stdin` and `stdout`. When you also need the child's `stderr` on its own handle, `IPC::Open3` adds a third:

```
use IPC::Open3;
use Symbol qw(gensym);

my $err = gensym;                    # a fresh anonymous handle for␣
↪stderr
my $pid = open3(my $to_child, my $from_child, $err, "sort");

print $to_child "banana\napple\ncherry\n";
close $to_child;

my @sorted = <$from_child>;          # ("apple\n", "banana\n",
↪"cherry\n")
print @sorted;                       # apple / banana / cherry, one␣
↪per line

waitpid($pid, 0);
```

Two differences from open2 to note:

- **The argument order flips.** open3 takes the writer handle first, then the reader, then the error handle: `open3(IN, OUT, ERR, ...)`. This is the opposite of open2's reader-first order - a long-standing inconsistency that bites everyone once.
- **The error handle needs gensym.** Pass a fresh anonymous handle from `Symbol::gensym` for `stderr`; a bareword or an undefined lexical will not collect it cleanly.

Rolling your own: pipe and fork

When the command is not an external program but Perl code you want to run in a separate process, build the channel yourself. A raw `pipe` gives a connected reader/writer pair before the fork; after `fork` each side keeps the end it needs and closes the other:

```
pipe(my $reader, my $writer) or die "pipe: $!";

my $kid = fork() // die "fork: $!";

if ($kid == 0) {
    # Child writes, then exits.
    close $reader;                   # child has no use for the read␣
↪end
    $writer->autoflush(1);
    print $writer "down-the-pipe\n";
    close $writer;
    exit 0;
}
```

(continues on next page)

(continued from previous page)

```
# Parent reads.
close $writer;                          # parent has no use for the␣
 ↪write end
my $line = <$reader>;
print "parent read: $line";             # parent read: down-the-pipe
close $reader;

waitpid($kid, 0);
```

The discipline is the same as the forking server on the *TCP sockets* page: **each side closes the end it does not use.** If the parent keeps the write end open, the child's `close` never produces an end-of-file on the read end, and the parent's read blocks forever.

Explicit `fork()` creates a separate operating-system process; PetaPerl's auto-parallelism is an in-process thread pool. A forked child runs its own auto-parallel numeric reductions normally within itself, and the parent's and child's parallelism do not interfere - they are independent processes with independent thread pools. For running work across CPU cores *inside* one process rather than across separate processes, see the *Concurrent Execution* guide.

Reference cross-links

- `open` - the one-way pipe open (" - | ", " | - ")
- `pipe` - a raw connected handle pair
- `fork` - split into two processes
- `waitpid` - reap the child and recover its status
- `exec` - replace the child image with a command

Next

For two unrelated programs that meet on a filesystem path rather than through a shared parent, see *FIFOs*.

FIFOs

A FIFO - a *named pipe* - is a pipe with a name on the filesystem. Two programs that share no parent, started independently from different shells, can still talk: one opens the path for writing, the other opens it for reading, and bytes flow between them exactly as through an anonymous `pipe`. The path is a rendezvous point; no data is ever stored in the file itself.

This is the channel to reach for when the processes are *unrelated* - when a plain `pipe` is impossible because there was never a common `fork` to inherit the descriptor from.

Creating the FIFO

A FIFO is created once, as a special file, with `POSIX::mkfifo`. After that it is opened like any other path:

```
use POSIX qw(mkfifo);

my $path = "/tmp/demo.fifo";
mkfifo($path, 0600) or die "mkfifo $path: $!";
```

The second argument is the permission mode, the same `0600` / `0644` you would pass to `chmod` - it decides who may open the FIFO. The call fails if the path already exists, so a long-running setup typically `unlink`s a stale FIFO first.

Producer and consumer

The defining behaviour of a FIFO is that **open blocks until both ends are present**. A reader opening the path waits until some writer opens it too, and vice versa. That rendezvous is the feature: it synchronises two programs that never agreed on timing.

This single program demonstrates both ends by forking, but the two halves would work just as well as two separate scripts run from two terminals:

```
use POSIX qw(mkfifo);

my $path = "/tmp/demo.fifo";
unlink $path;                          # clear any stale FIFO
mkfifo($path, 0600) or die "mkfifo $path: $!";

my $kid = fork() // die "fork: $!";

if ($kid == 0) {
    # Producer: open for writing (blocks until a reader appears).
    open my $w, ">", $path or die "producer open: $!";
    $w->autoflush(1);
    print $w "from-producer\n";
    close $w;
    exit 0;
}

# Consumer: open for reading (blocks until a writer appears).
open my $r, "<", $path or die "consumer open: $!";
my $line = <$r>;
print "consumer read: $line";          # consumer read: from-producer
close $r;

waitpid($kid, 0);
unlink $path;                          # remove the FIFO when finished
```

The two open calls are the synchronisation. Whichever process reaches its open first parks there until the other arrives; then both proceed together. Once open, the read and write handles behave like the two ends of any pipe - line reads, `readline`, buffered

`print`, all unchanged.

Two practical notes:

- **Autoflush on the writer.** As with every pipe and socket, a buffered write may never reach the reader until the buffer fills or the handle closes. Set `autoflush(1)` if the reader expects each line as it is produced.
- **A FIFO persists on disk.** `mkfifo` leaves a special file behind that outlives both processes. Remove it with `unlink` when the conversation is over, the same way you would clean up a lock file.

Reference cross-links

- `open` - open the FIFO path for reading or writing
- `pipe` - the anonymous-pipe relative for related processes
- `unlink` - remove the FIFO when finished
- `readline` - the function behind `<$fh>`

Next

When two processes need a connection rather than a one-way path - request and reply, both directions at once - reach for a socket. See *UNIX-domain and UDP sockets* and *TCP sockets*.

TCP sockets

A TCP socket carries a reliable, ordered byte stream between two processes that may be on different machines. This page builds the two halves of a TCP conversation from the raw built-ins: a **client** that connects and exchanges a line, and a **forking server** that accepts connections and handles each in its own child.

Every recipe here uses the built-in `socket` layer directly, with address helpers from the `Socket` module, rather than the high-level convenience module `IO::Socket::INET`. That is no loss: the built-in form is what `IO::Socket::INET` wraps, and seeing it in the open makes the handshake legible.

The address helpers

A socket address is a packed C `struct sockaddr_in`. The `Socket` module builds and takes apart that structure for you:

```
use Socket;
my $packed = sockaddr_in($port, $ip);     # build
my ($port, $ip) = sockaddr_in($packed);   # take apart
```

`$ip` is itself a packed four-byte address. `inet_aton` turns a dotted-quad string into one; `inet_ntoa` turns it back. The constant `INADDR_LOOPBACK` is the packed form of `127.0.0.1`, and `INADDR_ANY` means “every local interface”.

```
use Socket;
my $ip = inet_aton("127.0.0.1");
print inet_ntoa($ip), "\n";          # 127.0.0.1
```

A TCP client

The client creates a socket, connects it to a server address, and then reads and writes the handle like any other stream.

```
use Socket;

my $host = "127.0.0.1";
my $port = 8080;

socket(my $sock, PF_INET, SOCK_STREAM, getprotobyname("tcp"))
    or die "socket: $!";

my $addr = sockaddr_in($port, inet_aton($host));
connect($sock, $addr) or die "connect: $!";

$sock->autoflush(1);                 # send each line as it is
 ↪printed

print $sock "ping\n";
my $reply = <$sock>;
print "server said: $reply";         # server said: pong: ping

close $sock;
```

The one line that catches everyone is `$sock->autoflush(1)`. Without it, `print $sock "ping\n"` sits in Perl's output buffer, the server never sees it, and both sides block forever waiting on each other. On any socket where you write a request and then read a reply, set autoflush before the first write.

A forking server

A server binds to a port, listens, and then loops accepting connections. The classic structure handles each client in a forked child so one slow client cannot stall the next:

```
use Socket;
use POSIX qw(WNOHANG);               # for the non-blocking reaper

my $port    = 8080;
my $backlog = 128;                   # pending-connection queue
 ↪depth

socket(my $server, PF_INET, SOCK_STREAM, getprotobyname("tcp"))
    or die "socket: $!";

# Let the address be reused immediately after a restart.
```

(continues on next page)

(continued from previous page)

```
setsockopt($server, SOL_SOCKET, SO_REUSEADDR, 1)
    or die "setsockopt: $!";

bind($server, sockaddr_in($port, INADDR_ANY)) or die "bind: $!";
listen($server, $backlog)                     or die "listen: $!";

# Reap finished children so they do not become zombies.
$SIG{CHLD} = sub { 1 while waitpid(-1, WNOHANG) > 0 };

print "listening on port $port\n";

while (accept(my $client, $server)) {
    my $pid = fork();
    die "fork: $!" unless defined $pid;

    if ($pid == 0) {
        # Child: serve this one connection, then exit.
        close $server;              # child does not need the
↪listener
        $client->autoflush(1);

        while (my $line = <$client>) {
            print $client "pong: $line";
        }
        close $client;
        exit 0;
    }

    # Parent: hand the connection to the child and move on.
    close $client;                  # parent does not need this
↪client
}
```

Three details make this server correct rather than merely working:

- **`SO_REUSEADDR` before `bind`.** After a server exits, the kernel keeps the port in `TIME_WAIT` for a while. Without this option a restart fails with "Address already in use". Set it and the restart binds immediately.
- **The `$SIG{CHLD}` reaper.** Every forked child becomes a zombie when it exits until the parent calls `waitpid`. The handler reaps all finished children non-blockingly with `WNOHANG`, so the accept loop never pauses to wait.
- **Both sides close the descriptor they do not use.** The child closes the listening socket; the parent closes the accepted client socket. If either skips its `close`, the underlying file descriptor stays open in the wrong process and the connection never sees EOF.

Running both halves together

For a self-contained demonstration, fork the server into the background and run the client in the parent. The server binds to an ephemeral port (port `0`) and reports the port the kernel assigned:

```
use Socket;

# --- server child on an ephemeral port ---
socket(my $server, PF_INET, SOCK_STREAM, getprotobyname("tcp"))
    or die "socket: $!";
setsockopt($server, SOL_SOCKET, SO_REUSEADDR, 1) or die "setsockopt:
 ↪$!";
bind($server, sockaddr_in(0, INADDR_LOOPBACK))   or die "bind: $!";
listen($server, 128)                             or die "listen: $!
 ↪";
my ($port) = sockaddr_in(getsockname($server));

my $kid = fork() // die "fork: $!";
if ($kid == 0) {
    accept(my $client, $server) or die "accept: $!";
    $client->autoflush(1);
    my $line = <$client>;
    print $client "pong: $line";
    close $client;
    exit 0;
}

# --- client in the parent ---
socket(my $sock, PF_INET, SOCK_STREAM, getprotobyname("tcp")) or
 ↪die;
connect($sock, sockaddr_in($port, INADDR_LOOPBACK)) or die "connect:
 ↪$!";
$sock->autoflush(1);
print $sock "ping\n";
print "client got: ", scalar <$sock>;    # client got: pong: ping
close $sock;

waitpid($kid, 0);
```

`getsockname($server)` reports the local address the socket is bound to. Unpacking it with `sockaddr_in` recovers the port the kernel chose, which the client then connects to - the standard way to write a test that does not hard-code a port.

Reference cross-links

Each call has a per-built-in reference page with the full argument shape and return value:

- `socket` - create the endpoint
- `bind` - attach it to a local address

- `listen` - mark it as accepting connections
- `accept` - take the next pending connection
- `connect` - reach out to a server
- `setsockopt` - set `SO_REUSEADDR` and friends
- `getsockname` - discover the bound port
- `fork` and `waitpid` - spawn and reap the per-client child
- `%SIG` - the `CHLD` reaper handler

Next

For local-only stream sockets without the network stack, and for connectionless datagrams, see *UNIX-domain and UDP sockets*.

UNIX-domain and UDP sockets

Not every socket reaches across a network. This page covers two lighter-weight relatives of the TCP socket from the *previous page*: a **UNIX-domain socket pair** for two related processes on one machine, and a **UDP socket** for connectionless datagrams.

Both use the same `socket` family of built-ins and the same `Socket` address helpers. The recipes stay at that built-in layer, which is what the high-level `IO::Socket::*` wrappers are built on.

A socket pair for parent and child

`socketpair` creates two connected sockets in one call, with no addresses, no `bind`, and no `connect`. It is the ideal channel between a parent and a child it is about to fork: full-duplex (both ends can read and write), local, and already connected the moment it returns.

```
use Socket;

socketpair(my $child_end, my $parent_end, AF_UNIX, SOCK_STREAM, PF_
↪UNSPEC)
    or die "socketpair: $!";

my $kid = fork() // die "fork: $!";

if ($kid == 0) {
    # Child keeps its end, closes the parent's.
    close $parent_end;
    $child_end->autoflush(1);

    print $child_end "child-says-hi\n";
    my $back = <$child_end>;
    print "child got: $back";        # child got: ack
    exit 0;
}
```

(continues on next page)

(continued from previous page)

```
# Parent keeps its end, closes the child's.
close $child_end;
$parent_end->autoflush(1);

my $msg = <$parent_end>;
print "parent got: $msg";            # parent got: child-says-hi
print $parent_end "ack\n";

waitpid($kid, 0);
```

Why reach for `socketpair` over a plain `pipe`? A pipe is one-directional - the reader reads, the writer writes, and that is fixed at creation. A socket pair is bidirectional: each end both reads and writes, so a request-and-reply exchange needs one `socketpair` call instead of two pipes wired in opposite directions.

The autoflush rule from TCP applies unchanged: set it on each end before the first write, or a buffered request never reaches the other side and both processes block.

A UDP datagram exchange

UDP is connectionless. There is no `connect`, no `accept`, no stream - just discrete messages, each addressed to a destination and each preserving its own boundaries. A receiver binds to a port; a sender fires a datagram at it with `send`; the receiver collects it with `recv`, which also reports who sent it.

```
use Socket;

# --- receiver on an ephemeral port ---
socket(my $server, PF_INET, SOCK_DGRAM, getprotobyname("udp"))
    or die "socket: $!";
bind($server, sockaddr_in(0, INADDR_LOOPBACK)) or die "bind: $!";
my ($port) = sockaddr_in(getsockname($server));

my $kid = fork() // die "fork: $!";
if ($kid == 0) {
    # Sender fires one datagram at the receiver's port.
    socket(my $client, PF_INET, SOCK_DGRAM, getprotobyname("udp"))␣
 ↪or die;
    my $to = sockaddr_in($port, INADDR_LOOPBACK);
    send($client, "hello-udp", 0, $to) or die "send: $!";
    exit 0;
}

# recv fills $buf and returns the sender's packed address.
my $from = recv($server, my $buf, 65535, 0);
my ($sender_port, $sender_ip) = sockaddr_in($from);
print "got '$buf' from ", inet_ntoa($sender_ip), "\n";
                                    # got 'hello-udp' from 127.0.0.
 ↪1
```

(continues on next page)

(continued from previous page)

```
waitpid($kid, 0);
```

What UDP gives you and what it does not:

- **Message boundaries are preserved.** One `send` is one `recv`. A `recv` returns exactly one datagram, never two concatenated and never half of one. The `65535` is the maximum length you are willing to accept; a shorter datagram returns its real length.
- **There is no `connect` and no handshake.** The sender does not open a connection; it addresses each datagram with `send`'s fourth argument. `recv` reports the sender's address so a reply can go back the same way.
- **Delivery is not guaranteed.** UDP datagrams can be dropped, duplicated, or reordered by the network. Over the loopback interface, as in this recipe, that effectively never happens; over a real network you must handle loss yourself if it matters.

Reference cross-links

- `socketpair` - two connected local sockets
- `socket` - create a datagram or stream endpoint
- `bind` - attach the receiver to a local port
- `send` - fire a datagram at an address
- `recv` - collect a datagram and its sender
- `getsockname` - discover the bound port

Next

For shared memory and semaphores - channels the kernel owns rather than streams between processes - see *System V IPC*.

System V IPC

System V IPC gives two unrelated processes a shared resource the kernel owns: a **semaphore set** for coordination, or a **shared memory segment** for passing bytes with no copying through a pipe. Each resource is identified by an integer key and survives until a process explicitly removes it.

The recipes here use the raw `semget` / `shmget` built-in family with the flag values written as plain numbers, rather than the convenience modules `IPC::SysV`, `IPC::Semaphore` and `IPC::SharedMem`. Those built-ins are the layer the modules wrap; using them directly costs only a few named constants you supply yourself.

The flag constants you need

A handful of small integers control how these calls behave. They are the standard Linux values:

```
use constant {
    IPC_PRIVATE => 0,        # "give me a fresh, private key"
    IPC_CREAT   => 0o1000,   # create the resource if absent
    IPC_EXCL    => 0o2000,   # fail if it already exists
    IPC_RMID    => 0,        # control command: remove the resource
};
```

The low nine bits of the flag word are ordinary permission bits, the same `0600` / `0644` you would pass to `chmod`. `IPC_CREAT | 0600` means "create it, owner read/write".

A shared-memory hand-off

Shared memory is the fastest channel: one process writes bytes into a segment, another reads them straight out, with no kernel copy in between. This program creates a segment, writes a payload, forks, and the child reads the same bytes back:

```
use constant {
    IPC_PRIVATE => 0,
    IPC_CREAT   => 0o1000,
    IPC_RMID    => 0,
};

my $size = 1024;

my $id = shmget(IPC_PRIVATE, $size, IPC_CREAT | 0600);
defined $id or die "shmget: $!";

my $payload = "shared-payload";
shmwrite($id, $payload, 0, length $payload) or die "shmwrite: $!";

my $kid = fork() // die "fork: $!";
if ($kid == 0) {
    # Child reads the same segment the parent wrote.
    shmread($id, my $buf, 0, length $payload) or die "shmread: $!";
    print "child read: $buf\n";          # child read: shared-
↪payload
    exit 0;
}

waitpid($kid, 0);

# The segment outlives the processes - remove it explicitly.
shmctl($id, IPC_RMID, 0) or warn "shmctl: $!";
```

Two points decide whether this code leaks:

- **`shmread` / `shmwrite` take an explicit length.** They do not stop at a NUL or a new-

line; you tell them exactly how many bytes to move, starting at an offset. Pass `length $payload` so reader and writer agree on the size.

- **`IPC_RMID` is not automatic.** A shared segment persists in the kernel after every process attached to it has exited. You can see orphaned segments with `ipcs -m` at the shell. Always pair a `shmget` with a matching `shmctl IPC_RMID`.

A semaphore for coordination

A semaphore set is a small array of counters the kernel guards. The classic use is a lock: a process decrements the counter to enter a critical section and increments it to leave. This recipe creates a one-element set, initialises it, and removes it:

```
use constant {
    IPC_PRIVATE => 0,
    IPC_CREAT   => 0o1000,
    IPC_RMID    => 0,
    SETVAL      => 16,       # semctl command: set one value
    GETVAL      => 12,       # semctl command: read one value
};

# A set with one semaphore.
my $sem = semget(IPC_PRIVATE, 1, IPC_CREAT | 0600);
defined $sem or die "semget: $!";

# Initialise semaphore 0 to the value 1 (one token available).
semctl($sem, 0, SETVAL, 1) or die "semctl SETVAL: $!";

my $value = semctl($sem, 0, GETVAL, 0);
print "semaphore value: $value\n";       # semaphore value: 1

# Remove the set when finished.
semctl($sem, 0, IPC_RMID, 0) or warn "semctl IPC_RMID: $!";
```

The wait/signal operations themselves go through `semop`, which takes a packed string of `(sem_num, sem_op, sem_flg)` triples built with `pack`. Decrementing by one waits until a token is free; incrementing by one releases it. As with shared memory, a semaphore set the kernel owns must be removed with `IPC_RMID` or it lingers - visible under `ipcs -s`.

Reference cross-links

- `shmget` - create or open a shared segment
- `shmread` / `shmwrite` - move bytes in and out
- `shmctl` - control, including `IPC_RMID`
- `semget` - create or open a semaphore set
- `semop` - the wait/signal operations
- `semctl` - set, read, and remove
- `pack` - build the `semop` operation string

Next

Shared memory and semaphores are the heavyweight channels. For the lighter, stream-oriented ones, see *Subprocesses and co-processes* and *TCP sockets*.

- **Subprocesses and co-processes** - running a command and talking to it, in one or both directions, with the pipe open and with `IPC::Open2` / `IPC::Open3`.
- **FIFOs** - a named pipe on the filesystem that lets two unrelated programs meet on a path.
- **TCP sockets** - a complete client and a forking echo server over the loopback interface.
- **UNIX-domain and UDP sockets** - local stream sockets via `socketpair`, and connectionless UDP datagrams.
- **System V IPC** - semaphores and shared memory with the raw `semget` / `shmget` built-ins.

What this tutorial does not cover

- **Signals as a coordination mechanism beyond the basics.** Sending a signal with `kill` and catching it through `%SIG` is covered where each recipe needs it; the full signal-disposition table lives in the `%SIG` reference.
- **In-process parallelism.** Running work across CPU cores inside a single program is auto-parallelization, a different subject. See the *Concurrent Execution* guide.
- **High-level networking modules.** This tutorial stays at the built-in `socket` layer so the mechanics are visible. The recipes are the foundation a higher-level wrapper would build on.

4.2.18 Installing CPAN modules - a tutorial

Sooner or later a pperl program needs something from CPAN. This tutorial covers how to get it, where it lands, how pperl finds it afterwards, and what to do when a distribution refuses to install.

You are assumed to have installed Perl modules before, with `cpan`, `cpanm`, or your distribution's package manager. What follows is what changes when the interpreter is pperl rather than perl5.

The one idea

pperl runs **no XS**. Not "XS is discouraged", not "XS is slower" - a compiled extension cannot be loaded at all, because pperl is a Rust reimplementation of Perl 5 and does not have perl5's C-level ABI to bind against.

Two consequences follow, and everything else in this tutorial is a detail of one of them.

Pure Perl installs; XS does not. A distribution written in Perl works exactly as it does anywhere else. A distribution that needs to compile C does not, and no amount of configuration will change that.

pperl has to introduce itself to the installer. Several hundred modules are built into the pperl binary and have no `.pm` file on disk. An installer decides "do I already have

this?" by looking for that file, so without help it reads some *other* perl's inventory and draws the wrong conclusion in both directions - rebuilding modules pperl already has, and believing it has ones it does not.

`pperl --cpan` exists to arrange both. It is not a new installer: it runs the CPAN client that ships with Perl, the one behind the `cpan` command, with the environment set up so the answers come out right.

```
pperl --cpan Text::CSV
```

What you will need

Nothing besides pperl for the installer itself. The CPAN client is part of Perl's core library, so it is built into the pperl binary and pperl runs it directly - inside that run, `$^X` is pperl and not a host perl. No perl5 has to be installed on the machine.

What a distribution's own build needs is still needed: `make` for nearly all of them, and `git` if you install from a repository.

The client keeps its downloads, its build directories and its configuration under `~/.pperl/cpan`. A `~/.cpan` that belongs to a host perl is neither read nor changed.

Chapters

Installing, and finding what you installed

The command

Everything after `--cpan` goes to the CPAN client, so any option of the `cpan` command you already know still works: `-T` to skip the tests, `-f` to force an install.

```
pperl --cpan Text::CSV
pperl --cpan Plack Starman
pperl --cpan -T Try::Tiny
```

Besides module names, `--cpan` takes a git repository: a URL that starts with `git:` or ends in `.git`. It is cloned and installed from the clone. Append `@` and a branch, tag or commit to install that revision.

```
pperl --cpan https://github.com/example/Some-Dist.git
pperl --cpan https://github.com/example/Some-Dist.git@v1.2
```

With nothing to install, it says so rather than guessing:

```
pperl --cpan
# pperl: --cpan: nothing to install: pperl --cpan Module::Name [...]
```

Where it goes

Modules land in a pperl-owned tree, never in a perl5 one. The base is `$PPERL_HOME`, or `~/.pperl` if that is unset:

```
~/.pperl/
├── lib/perl5/          <- the modules
└── cpan/               <- the installer's downloads and builds
```

`--cpan --system` installs for every account on the machine instead, under `/usr/local/share/pperl`:

```
/usr/local/share/pperl/
├── site_perl/          <- the modules
├── bin/                <- their scripts
├── man/                <- their manuals
└── cpan/               <- the installer's downloads and builds
```

Every pperl searches `site_perl` there without being told. Writing there takes root's rights, or an account that may write `/usr/local/share/pperl`, so run it as `sudo pperl --cpan --system Some::Module`. Without those rights it stops before it downloads anything and says so.

Keeping the tree separate is deliberate. A machine commonly has both pperl and a real perl5, and a module built for one has no business being found by the other.

Making pperl see it

pperl searches its own tree without being told: `lib/perl5` under `$PPERL_HOME`, or under `~/.pperl` if that is unset, comes ahead of every system tree. A module installed with `pperl --cpan` is found by the next `pperl` run, and it outranks a same-named copy belonging to the host perl:

```
pperl --cpan Text::CSV
# pperl: installed into /home/you/.pperl/lib/perl5

pperl -e 'print "$_\n" for @INC'
# ...
# /home/you/.pperl/lib/perl5
# /usr/share/perl5/site_perl
# ...
```

The same holds for the `--system` tree, which comes after the per-user one. Other trees you add yourself, with `-I`, `PERL5LIB`, or `PPERL5LIB`. `PPERL5LIB` is the pperl-private twin of `PERL5LIB`: same syntax, colon-separated, but read **only** by pperl, and honoured ahead of `PERL5LIB`:

```
PPERL5LIB=/opt/mytree PERL5LIB=/home/you/perl5/lib/perl5 \
    pperl -e 'print "$_\n" for @INC'
# ...
# /opt/mytree                       <- PPERL5LIB
# /home/you/perl5/lib/perl5         <- PERL5LIB
# /home/you/.pperl/lib/perl5
# /usr/share/perl5/site_perl
# ...
```

Why a separate variable rather than reusing `PERL5LIB`: on a machine running both interpreters, `PERL5LIB` is usually already pointing at the host perl's trees. Overloading it would force you to choose between the two. Both are read, `PPERL5LIB` first.

Why pperl reads the system trees at all

pperl searches the host perl's directories because a pure-Perl module found there runs perfectly well under pperl. That is free inventory and there is no reason to refuse it.

pperl also searches the architecture directory of each tree you add with `PPERL5LIB`, `PERL5LIB` or `-I`, and of `~/.pperl/lib/perl5`, as perl does: `<tree>/x86_64-linux` and `<tree>/<version>/x86_64-linux`, ahead of the tree, when they exist. A distribution that declares XS installs its modules there, even its pure-Perl ones. Their compiled parts pperl cannot load, and *the no-XS rule* explains where that bites.

The tools it uses, and downloads over https

Like perl's own `cpan` on its first run, `pperl --cpan` looks on your `PATH` for the programs the CPAN client works with: `make`, `tar`, `gzip`, `bzip2`, `unzip`, `gpg`, `patch` and `apply-patch`, and a downloader, the first it finds of `wget`, `curl`, `lynx`, `ncftpget`, `ncftp` and `ftp`.

pperl speaks https by itself only in a build with the `native-max` module set, which contains `IO::Socket::SSL`. Any other build downloads from CPAN over https through `wget` or `curl`, and fetches the *XS substitutes* the same way. To check which you have, run:

```
pperl -MIO::Socket::SSL -e 1
```

A build with TLS support prints nothing. One without it says `Can't locate IO/Socket/SSL.pm in @INC`, adding that it was built without the `native-max` capability.

A host with neither TLS support in pperl nor `wget` or `curl` falls back to plain http, and the CPAN client warns about it; see *troubleshooting*. Installing `curl` or `wget` is the remedy.

Installing without a network

A distribution archive or an unpacked directory on disk installs without a download of its own. Name a directory with a leading `./` or `/`, so that it cannot be taken for a module name:

```
pperl --cpan ./Some-Dist-1.23.tar.gz
pperl --cpan ~/src/Some-Dist
```

The client still fetches the CPAN index on its first run, and any dependency that is not installed yet. It also tries to refresh the *XS substitutes*; without a network it says so and uses the ones it already has.

Tests run non-interactively: `PERL_MM_USE_DEFAULT` and `NONINTERACTIVE_TESTING` are both set, so a distribution that would otherwise stop to ask a question takes its own default instead of hanging.

Distribution preferences

`pperl --cpan` applies CPAN.pm's distribution preferences, the "distroprefs" described in *CPAN* under "Configuration for individual distributions". A preference is a YAML file that matches distributions by name and changes how they are built: environment variables, build arguments, extra prerequisites, patches applied before the build.

Preferences go into `prefs/` under the installer's directory: `~/.pperl/cpan/prefs`, or `/usr/local/share/pperl/cpan/prefs` for `--system`. A patch that a preference names is looked up in the same directory, so a preference and its patch sit side by side:

```
---
match:
  distribution: '^AUTHOR/Some-Dist-1\.23'
patches:
  - Some-Dist-1.23.patch
```

pperl reads `.yml` preferences with its built-in *YAML::XS*, so nothing else needs installing.

XS substitutes

Before it installs anything, each `pperl --cpan` run fetches pperl's pure-Perl substitutes for distributions that have a compiled part, from `https://perl.petamem.com/downloads/cppan`, into `prefs/cppan` beside your own preferences. Each substitute is a preference with a patch: it matches one XS distribution and replaces its compiled part with Perl. Because the set is fetched on every run, a substitute published after your pperl was released applies to it too.

`Text::CSV_XS`, for example, installs and runs on the Perl engine of `Text::CSV`; other substitutes rest on features and functions perl and pperl have built in. Where a distribution's own tests show a difference from its compiled version, those tests are marked TODO, so you can see exactly what differs.

An unchanged set is not downloaded again. When the server cannot be reached, the run warns and uses the substitutes it fetched last. `PPERL_CPPAN_URL` names another address to fetch from; set to an empty string, it turns the fetch off, and the substitutes already on disk still apply.

What pperl already has

Several hundred modules are built into the pperl binary, in two kinds. About a hundred are compiled in: `Storable`, `Socket`, `Digest::MD5`, `Data::Dumper`, `Time::HiRes`, `POSIX`, `Encode` and friends are not files pperl loads - they are Rust code, and `require` is routed to it. The rest are the Perl modules of the core library, `Test::More`, `File::Spec`, `Getopt::Long` and so on, whose source the binary carries. For both kinds `%INC` records `./pperl-native/<name>` (for example `./pperl-native/Storable.pm`) - it reads like a path, but there is no file behind it.

That is invisible and harmless when you are writing Perl. It is a problem when an *installer* is trying to work out what you have.

The probe

Every installer answers "is this module present, and at what version?" the same way: look for the `.pm` file on the search path, read `$VERSION` out of it. A built-in module has no file. So the probe walks past pperl entirely and finds the answer in the host perl's tree - a different interpreter, with different versions.

On the reference machine, five of six modules checked disagreed:

module	pperl provides	host perl says
`Digest::MD5`	2.58	2.59
`Storable`	3.41	3.37
`Socket`	2.038	2.040
`Time::HiRes`	1.9777	1.9778
`Data::Dumper`	2.183	2.192

Every consequence of that is wrong, and two of the three are silent.

- A dependency on `Storable >= 3.40` reads as unsatisfied, because the host says 3.37. The installer fetches Storable - an XS distribution - to replace a module pperl already has at 3.41. The build then fails, and it fails for a module you never needed.
- On a host with no perl at all - a minimal container, or the static build whose entire point is not needing one - every built-in module reads as missing and the same false build is attempted for each.
- In the other direction, and this one is silent: a dependency on `Data::Dumper >= 2.190` reads as *satisfied* by the host's 2.192, while pperl actually runs 2.183.

Asking pperl instead

pperl answers the question itself. `Config::lib_file` returns the path under which pperl sees a library file, a built-in one first:

```
use Config;
print Config::lib_file("Storable.pm"), "\n";    # ./pperl-native/
↪Storable.pm
print Config::lib_file("Foo/Bar.pm"), "\n";    # /home/you/.pperl/
↪lib/perl5/Foo/Bar.pm
```

The path can be opened, so the version can be read from it as from any installed file. A built-in Perl module reads as its source. A compiled-in module that has no Perl source reads as three lines, its package, its `$VERSION` and a true value:

```
use Config;
use ExtUtils::MakeMaker ();
print MM->parse_version(Config::lib_file("Storable.pm")), "\n";   #␣
↪3.41
```

The installer that is part of pperl asks exactly this. So do the `ExtUtils::MakeMaker` and `Module::Metadata` pperl carries, which is what a `Makefile.PL` and a `Build.PL` use

to check their prerequisites. A built-in module is then simply skipped: no download, no build, no diagnostic to explain.

Asked for a module that is built in, `--cpan` says so and stops:

```
pperl: Storable 3.41 is built into pperl; nothing to install
```

> **Note**
>
> The built-in path opens, but it answers no file test: `-e`, `-f` and `-r` say no. Code that looks for a module by testing `"$dir/Storable.pm"` for each directory of `@INC` does not find a built-in one. Use `Config::lib_file` there.

When the built-in version is too old

pperl reports the version it actually has. That does not make the version acceptable. If a distribution asks for `Storable >= 3.50` and pperl carries 3.41, the dependency is genuinely unsatisfied, the installer goes after the real distribution, and - Storable being XS - the build fails.

That is the honest outcome rather than a bug: the module pperl carries is not the version being asked for. It does mean a built-in module falling behind CPAN can turn into an install failure some way downstream, so a version mismatch is worth checking early when an install fails for no visible reason:

```
pperl -MStorable -e 'print Storable->VERSION, "\n"'
```

The no-XS rule, and the shapes you will meet

pperl cannot load a compiled extension. This is structural, not a setting: XS binds to perl5's C-level ABI, and pperl is a Rust reimplementation of Perl 5 that does not have one to bind to. An `XSLoader::load` under pperl fails with perl's own message for a compiled part that is not there:

```
Can't locate loadable object for module Foo::Bar in @INC (@INC␣
↪contains: ...)
```

That is the message modules with a Perl fallback test for, so they switch to their Perl implementation on it, as they would under perl.

What that means for an install depends entirely on how the distribution is put together. There are five shapes in the wild and they behave very differently, so it is worth knowing which one you are looking at.

1. Pure Perl

Nothing to say. It installs and runs exactly as it does anywhere else. This is the large majority of CPAN.

2. A Perl module with an optional XS accelerator

The module is Perl; a *separate* distribution speeds it up if present. `Readonly` and `ReadonlyXS` are the classic pair.

Install the Perl one and ignore the accelerator. The module probes for its companion, does not find it, and uses its own implementation - which is the documented, supported path, not a degraded mode.

```
pperl --cpan Readonly       # yes
pperl --cpan ReadonlyXS     # no, and not needed
```

This is the shape to prefer when you have a choice of dependency. The Perl module is the module, and XS is an optimisation.

3. One distribution, both halves, XS optional at build time

The distribution ships C *and* Perl, and its `Makefile.PL` treats the compile as optional: if the compiler step fails or is declined, it installs the Perl half and carries on. At runtime the module notices the XS is absent and uses its own fallback.

`Cookie::Baker` and `WWW::Form::UrlEncoded` are of this shape. They install cleanly under pperl.

`pperl --cpan` asks every build for that Perl-only form by the toolchain's own switches: `PUREPERL_ONLY=1` for `ExtUtils::MakeMaker` and `--pureperl-only` for `Module::Build`. A distribution that honours them, such as `Params::Validate`, builds its Perl half instead of stopping for want of a C compiler.

4. One distribution, both halves, XS mandatory at build time

Same idea, but the build compiles C unconditionally. The compile fails, the installer aborts, and with an ordinary installer the Perl half - which would have worked - never reaches disk.

`HTTP::Parser::XS` is the worked example. pperl implements it natively now, so you will not hit it in practice - an installer sees 0.17 already present and never fetches the distribution at all. It is still the clearest illustration of the shape, because the failure is not where you would guess. Its runtime logic is impeccable:

```
eval {
    require XSLoader;
    XSLoader::load(__PACKAGE__, $VERSION);
    $BACKEND = 'xs';
};
die $@ if $@ && $BACKEND =~ /\bxs\b/;    # only if you FORCED xs
if (not __PACKAGE__->can('parse_http_response')) {
    require HTTP::Parser::XS::PP;        # automatic fallback
    $BACKEND = 'pp';
}
```

Given the files on disk, that falls back by itself, with no intervention. The problem is purely that `Makefile.PL` builds picohttpparser unconditionally, so the files never get

there. Its `PERL_ONLY=1` switch selects the runtime backend and drives a test target; it does not make the build skip the C.

> **Note**
>
> A native implementation is the escape hatch, and it is the one that was used here. But it only holds while pperl's version satisfies what dependents ask for: pperl reports the native version, so a dependency on `HTTP::Parser::XS >= 0.18` would send the installer after the distribution again and the abort would come back.

A shape-4 distribution pperl does *not* implement natively installs all the same, because pperl's `ExtUtils::MakeMaker` never compiles XS. Under pperl a `Makefile.PL` produces a Makefile for the Perl half only: no perl header files and no `xsubpp` are looked for, so the build cannot fail at the C. Two things then decide whether the distribution is installed:

- every module in it that names an XS loader is loaded once. One that falls back to Perl, as above, loads. One that does not is named, and stays: a module can switch to its Perl half by an environment variable or a setting, as `Params::Util` does under `PERL_PARAMS_UTIL_PP`, which its own tests set;
- the distribution's own tests run, as for any other module, and they decide.

```
pperl: does not load without the compiled part, which pperl cannot␣
↪build: Some/Dist/Fast.pm
```

A distribution built with `Module::Build` that does not honour `--pureperl-only` still attempts its compile and fails. For those, `--cpan` installs the Perl files the build had already copied into its `blib` directory. Each file that names an XS loader is loaded once first; one that does not load is installed all the same, for the same reason as above, and named. It says so:

```
pperl: Some-Dist-1.23-0: XS build failed; installed its 4 pure-Perl␣
↪file(s)
pperl: Some-Dist-1.23-0: installed, but without the compiled part it␣
↪does not load: Some/Dist/Fast.pm
```

That run ends with a failure status, because the distribution's tests never ran: test what was installed before you rely on it.

5. XS only

No Perl implementation exists anywhere in the distribution. There is nothing to fall back to and nothing pperl can do with it. The load check names the module:

```
pperl: does not load without the compiled part, which pperl cannot␣
↪build: String/CRC32.pm
```

Tests that load the module then fail with the missing-object message above, and a distribution whose tests fail is not installed.

The exception is a distribution pperl publishes a substitute for: a patch that replaces the compiled part with Perl, which `--cpan` fetches on every run, so it reaches a pperl

released before the substitute existed. `Text::CSV_XS` installs that way, running on the Perl engine of `Text::CSV`. "XS substitutes" in *installing and finding modules* has the details.

When you actually need the speed

The reason XS exists is performance, so it is fair to ask what replaces it. pperl's answer has three parts, and none of them is "write it in Perl and hope".

- **Native modules.** About a hundred of the modules people reach for XS to get are already Rust inside the binary, at Rust speed, with no build step. See *what pperl already has*.
- **The JIT.** Hot loops are compiled, so the pure-Perl version of an algorithm is not automatically the slow version. See the *performance guide*.
- **Calling C directly.** When you genuinely need a C library, call it rather than wrapping it in XS. `Peta::FFI` is pperl's own interface; see the *FFI primer*. *FFI::Platypus*, the interface CPAN's FFI modules are written against, is built in as well, and adds structs passed by value and callbacks with typed signatures. A distribution that bundles its own C for it (an `ffi/` directory, built by *FFI::Build*) builds and installs with pperl when a C compiler is on `PATH`: that C becomes a library called through FFI, not an XS extension.

If you hit a module in shape 4 or 5 that you cannot work around, that is worth reporting, since a substitute can follow without a new pperl release: https://github.com/petamem/pperl-bugtracker/issues

When an install goes wrong

Symptoms first, since that is what you have when you arrive here.

"Can't locate Foo/Bar.pm" after a successful install

The install said it worked, and the module is not there. pperl searches `lib/perl5` under `$PPERL_HOME`, or under `~/.pperl` when that is unset, so the usual cause is that the install and the run see different trees: `PPERL_HOME` was set for one and not the other, or the install went into a tree pperl does not search unasked. Compare the "installed into" line of the install with pperl's search path:

```
pperl -e 'print "$_\n" for @INC'
```

A tree that is missing there can be added with `PPERL5LIB`.

A distribution that declares XS installs its modules into the tree's architecture directory, `lib/perl5/x86_64-linux/` on this platform, even when they are pure Perl. That is not a cause: as perl does, pperl searches the architecture directory of every tree on `PPERL5LIB`, `PERL5LIB` and `-I`, and of `~/.pperl/lib/perl5`.

The installer wants to build a module pperl already has

Confirm what pperl thinks it provides, and at which version:

```
pperl -MConfig -MExtUtils::MakeMaker -e \
  'my $f = Config::lib_file("Storable.pm"); print "$f ", MM->parse_
↪version($f), "\n"'
# ./pperl-native/Storable.pm 3.41
```

If the path starts with `./pperl-native/` the module is built in. If the installer still wants it, compare the versions: the dependency may genuinely be newer than the version pperl carries, in which case no arrangement will satisfy it.

An installer that is not pperl's own, run under pperl, looks for modules with file tests and does not see the built-in ones. Use `pperl --cpan`.

"Can't locate loadable object for module Foo::Bar in @INC"

XS: the module wants its compiled part, and pperl does not build one. Read *the no-XS rule* to work out which shape the distribution is, because shapes 2 and 3 there have a way forward and shapes 4 and 5 do not.

A module that has a Perl implementation of its own catches this message and uses that implementation, so you do not see it. When you do, the module either has no Perl implementation, or has one that is switched on another way, such as by an environment variable its documentation names.

"Need to fall back to http"

```
Missing or unusable modules 'Net::SSLeay', 'IO::Socket::SSL', and␣
↪found neither curl nor wget installed.
Need to fall back to http.
```

This pperl has no TLS support of its own, and the host has neither `curl` nor `wget`, so the CPAN client downloads over plain http. Install `curl` or `wget`; the next `pperl --cpan` run finds it and downloads over https. The XS substitutes are not fetched at all in this state, and that warning (below) ends in `neither curl nor wget is configured`.

"XS substitutes not refreshed from ..."

```
pperl: --cpan: XS substitutes not refreshed from https://perl.
↪petamem.com/downloads/cppan: ...
```

The run could not fetch pperl's substitutes for XS distributions, for the reason named at the end: no network, or a server error. The install carries on with the substitutes fetched last, so only one published since then is missing. On a machine that is never online, set `PPERL_CPPAN_URL` to an empty string to stop the attempt.

```
pperl: --cpan: cannot unpack .../cppan.tar.gz; the XS substitutes␣
↪stay as they were
```

The download arrived but is not an archive pperl can unpack. The install carries on with the substitutes fetched before it, and the next run downloads the file afresh.

The install hangs waiting for input

It should not - under `--cpan`, `PERL_MM_USE_DEFAULT` and `NONINTERACTIVE_TESTING` are both set, so distributions take their own defaults. If you are driving an installer yourself, set both.

Tests fail but the module is fine

`-T` goes straight through to the CPAN client and skips the tests:

```
pperl --cpan -T Some::Module
```

Worth doing once to find out whether the module or its test suite is the problem. A test suite that itself depends on XS will fail against a perfectly good pure-Perl module.

Starting over

The tree is disposable. Nothing else refers to it:

```
rm -rf ~/.pperl
pperl --cpan Your::Module
```

Reporting

If a pure-Perl distribution fails to install, or installs and then cannot be loaded, that is a pperl bug and worth filing:

https://github.com/petamem/pperl-bugtracker/issues

Include the distribution name and version, the full `--cpan` output, and `pperl -e 'print "$_\n" for @INC'`.

4.2.19 ipps, the interactive mode - a tutorial

`ipps`, the interactive PetaPerl shell, is a shell for Perl: a prompt whose language is Perl. You type a line, pperl runs it and shows you the value, and everything the line declared is still there for the next one. It is the quickest way to try an expression, look inside a data structure, or find out whether a loop gets compiled.

You are assumed to know Perl. Nothing here needs installing: the prompt, its line editor and its history are built into the pperl binary.

Starting it

```
pperl --shell
```

or, if you would rather type less, give pperl a second name once:

```
ln -s "$(command -v pperl)" ~/bin/ipps
ipps
```

pperl started under the name `ipps` is the interactive mode. A bare `pperl` without arguments is unchanged: it reads a program from standard input, as perl does.

> **Note**
>
> `--shell` needs a pperl built with the `shell` feature. A binary without it says so and stops. Check with `pperl -V`, which lists the features under `PPERL_FEATURES`.

The one idea

A line at the prompt is Perl, compiled and run as it would be in a file, with one difference: **what a line leaves behind stays.** A `my` variable, a `sub`, a loaded module, a `use strict`, the current `package` - the next line starts where this one stopped.

```
ipps> my @primes = (2, 3, 5, 7)
(2, 3, 5, 7)
ipps> sub square { $_[0] ** 2 }
ipps> map { square($_) } @primes
(4, 9, 25, 49)
```

Everything else in this tutorial is a detail of that: how values are shown, what the commands that start with a colon do, how to make the prompt yours, and the few places where a line is not quite a file.

Chapters

A first session

Values

Type an expression and its value is shown, written the way you would write it in Perl:

```
ipps> 2 ** 10
1024
ipps> "total: " . 4
'total: 4'
ipps> 1 == 1
true
ipps> undef
undef
ipps> { name => "ada", langs => ["perl", "rust"] }
{ langs => ['perl', 'rust'], name => 'ada' }
```

A number is bare and a string is in quotes, so you can tell 12 from `'12'`. A string gets the quotes that need the fewest escapes:

```
ipps> "it's"
q{it's}
ipps> "a\tb"
"a\tb"
```

A line is evaluated in list context, and a list of several values is shown in parentheses. A hash is a list of its keys and values:

```
ipps> my %age = (ada => 36, alan => 41)
('ada', 36, 'alan', 41)
ipps> localtime
(59, 47, 10, 13, 9, 126, 2, 285, 1)
```

Large values are cut, with a note of what was left out. The `:dump` command of the next chapter shows them whole.

```
ipps> [1 .. 40]
[
  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,␣
↪20,
  21, 22, 23, 24, ... (16 more)
]
```

Staying quiet

Every Perl statement has a value, and you do not always want to see it. End the line with a semicolon and nothing is shown:

```
ipps> my $n = 12;
ipps> $n * 2
24
```

Three kinds of line show nothing even without the semicolon: a definition (`sub`, `use`, `package`), a loop or conditional typed by itself, and a line whose last statement is `print`, `printf` or `say`, whose value would only be whether the output worked.

```
ipps> for my $p (2, 3) { print "$p squared is ", $p ** 2, "\n" }
2 squared is 4
3 squared is 9
```

When your code prints something that does not end in a newline, the prompt still starts on a line of its own, and a % in reverse video marks the place where the output stopped:

```
ipps> print "no newline"
no newline%
ipps>
```

The % is not part of the output. It tells you that the newline you see was not printed by your code.

The last value

The value last shown is in `$ans`, and when there were several, all of them are in `@ans`:

```
ipps> $n * 2
24
ipps> $ans + 1
25
ipps> map { $_ * $_ } 2, 3, 5, 7
(4, 9, 25, 49)
ipps> scalar @ans
4
```

More than one line

Input that is not finished gets a second prompt until it is. An open bracket, an open quote, a here-document without its end and an expression that expects more all count:

```
ipps> for my $p (2, 3, 5, 7) {
 ...>     print "$p squared is ", $p ** 2, "\n";
 ...> }
2 squared is 4
3 squared is 9
5 squared is 25
7 squared is 49
ipps> 1 +
 ...> 2
3
```

Blank lines inside are kept, so pasted code works. To give up on unfinished input, press Ctrl-C.

Errors

An error is printed and the session goes on with everything it had. The place is given as the line's number at the prompt and the line within what you typed:

```
ipps> die "something went wrong\n"
something went wrong
ipps> $n
12
ipps> $undeclared_typo + 1
Use of uninitialized value $undeclared_typo in addition (+) at␣
 ↪(ipps:14) line 1.
1
```

The session starts with warnings on and with `strict` for references and barewords, but not for variables: an undeclared `$x` works at a prompt. If you want all of `strict`, say so, and it holds from then on:

```
ipps> use strict;
ipps> $another_typo = 1
Global symbol "$another_typo" requires explicit package name (did␣
 ↪you forget to declare "my $another_typo"?) at (ipps:16) line 1.
```

Keys

Tab
: Completes the word in front of the cursor: a variable, a key of a hash, a method after `->`, a sub, a module after `use`, a file name inside a string, a command. With several candidates it completes as far as they agree; a second Tab lists them. When nothing of Perl fits the word, Tab completes a file name.

Up and Down
: Walk through the lines entered before, including those of earlier sessions. Ctrl-R searches them.

Ctrl-C
: At the prompt, drops what you have typed. While a line runs, stops that line. The session goes on either way.

Ctrl-D
: On an empty prompt, ends the session. So do `exit` and `:quit`.

Tab knows the session. With a `%colour` and an object in `$dog`:

```
ipps> $col<Tab>
ipps> $colour{<Tab><Tab>
green}  red}
ipps> $dog-><Tab><Tab>
DOES      can       import    name      speak
VERSION   fetch     isa       new       unimport
```

It reads what exists and runs nothing, so it cannot complete through the result of a call.

Commands

A line that starts with a colon and a word is a command of `ipps`, not Perl. `:help` lists them all. A command may be shortened while it stays unique: `:q` is `:quit`, `:v` is `:vars`.

Looking at what you have

`:vars` lists the variables of the session with their kind and a short form of their value:

```
ipps> my @primes = (2, 3, 5, 7); my %age = (ada => 36, alan => 41);␣
 ↪my $name = "ada";
ipps> :vars
%age     hash of 2   (ada => 36, alan => 41)
$ans     undef       undef
@ans     array of 0  ()
$name    string      'ada'
@primes  array of 4  (2, 3, 5, 7)
```

`:scalar` evaluates in scalar context, where a line by itself is in list context:

```
ipps> @primes
(2, 3, 5, 7)
ipps> :scalar @primes
4
ipps> :scalar localtime 0
'Thu Jan  1 01:00:00 1970'
```

`:dump` shows a value whole, however large:

```
ipps> my @big = (1 .. 30);
ipps> \@big
[
  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,␣
↪20,
  21, 22, 23, 24, ... (6 more)
]
ipps> :dump \@big
[
  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,␣
↪20,
  21, 22, 23, 24, 25, 26, 27, 28, 29, 30
]
```

`:source` shows the code of a sub, by its name or by an expression that gives a code reference. The pragmas in force where the sub was written are part of what it shows:

```
ipps> sub area { my ($w, $h) = @_; return $w * $h }
ipps> :source area
sub area {
    use warnings;
    use strict 'refs', 'subs';
    my($w, $h) = @_;
    return $w * $h;
}
```

`:which` says where pperl gets a module from, and whether it is loaded:

```
ipps> :which List::Util
built into pperl (loaded)
ipps> :which JSON::PP
built into pperl
ipps> :which No::Such
ipps: No::Such: not on the search path
```

A module that is a file on disk is answered with its path.

Timing a line, and asking the JIT

`:time` runs a line and then shows how long it took, compiling included. The first number is wall-clock time, the second the processor time, which the system counts in steps of ten milliseconds:

```
ipps> :time my $s = 0; $s += $_ for 1 .. 1_000_000; $s
500000500000
wall 17.377 ms, cpu 20.000 ms
```

`:jit` runs a line and then shows what pperl's JIT did while it ran:

```
ipps> :jit my $t = 0; for my $i (1 .. 1_000_000) { $t += $i * 2 } $t
1000001000000
jit: 1 loop compiled, 1 new loop seen (2 of 71 since the start)
```

The line had one loop, and it was compiled to machine code. The numbers in parentheses are compiled and seen loops since `ipps` started, its own code included.

A loop is compiled once it has run often enough, and that can be in a later line than the one that wrote it. The loop of a sub is seen where the sub is defined and compiled where a call makes it hot:

```
ipps> sub total { my $t = 0; for my $i (1 .. $_[0]) { $t += $i } $t␣
↪}
ipps> :jit total(300_000)
45000150000
jit: 1 loop compiled, 0 new loops seen (1 of 68 since the start)
```

A line whose loop was seen and not compiled looks like this:

```
ipps> :jit my @w = map { "w$_" } 1 .. 3; scalar @w
3
jit: 0 loops compiled, 1 new loop seen (2 of 72 since the start)
```

The counts say that a loop was compiled. They do not say why another was not.

Files and modules

`:load` runs a file in the session. What it declares is there afterwards; the value of its last statement is not shown.

```
ipps> :load ~/lib/helpers.pl
```

`:save` writes every line that ran without an error as a script, in the order you entered them. Commands are not written, the lines they ran are.

```
ipps> :save /tmp/session.pl
12 entries saved to /tmp/session.pl
```

Lines you typed only to look at a value are statements of that script too, and perl warns that such a statement is useless there. Delete them from the file.

`:install` installs modules from CPAN with pperl's own client, as `pperl --cpan` does. They can be used at once.

```
ipps> :install Text::CSV
```

Commands of your shell

A line that starts with `!` and a space is not a command of `ipps` and not Perl: it is handed to your shell and runs on the terminal.

```
ipps> ! ls Cargo.* | wc -l
2
```

The section on running programs in *where a line is not a file* has the details: the exit status, aliases, and how to get a command's output into a variable.

`:programs` goes the other way and makes programs Perl: after `:programs ls`, `ls` is a sub that runs the program and returns its output, and `:programs` alone lists such names. *Programs as functions* in the next chapter explains it, together with the `autorun` setting.

The session itself

`:history` shows the last 20 lines entered, `:history 5` the last five. On a terminal these include earlier sessions.

`:reset` starts afresh: no `my` variables, the pragmas of a new session, the package `main`. Subs and package variables belong to the interpreter and stay.

`:quit` leaves, as Ctrl-D does.

`:set` belongs to the next chapter.

> **Note**
>
> A name with two colons in front, such as `::helper`, is Perl - a sub in `main` - and stays Perl.

Making it yours

Settings

`:set` shows the settings, `:set` with a name shows one, and with a name and a value changes it:

```
ipps> :set
  autorun    off        an unknown sub of main runs a program: on or
↪off
  max_depth  6          levels shown of a nested structure; 0: all
  max_items  24         elements shown of an array or hash; 0: all
  max_string 240        characters shown of a string; 0: all
  prompt     'ipps> '   the prompt
```

(continues on next page)

(continued from previous page)

```
  prompt2    ' ...> '   the prompt for unfinished input
  shell      '/bin/bash -c' what "! command" runs the command with
  time       off        show the times of every line: on or off
  width      0          columns for a printed value; 0: the terminal
↪'s
ipps> :set max_items 5
ipps> [1 .. 30]
[1, 2, 3, 4, 5, ... (25 more)]
```

A prompt in quotes keeps its trailing space:

```
ipps> :set prompt "perl> "
perl>
```

A setting made this way lasts for the session. To keep it, put it in the start file.

The start file

`ipps` keeps two files in `~/.pperl/ipps`, or in `$PPERL_HOME/ipps` when that variable is set:

`rc.pl`
: Perl that runs in the session before the first prompt. What it declares is there when you start typing.

`history`
: The lines you typed, read at the start and written at the end.

`rc.pl` is ordinary Perl, so it can load the modules you always want and define the subs you always use:

```
use List::Util qw(sum max min first);
use JSON::PP ();

sub slurp { my ($file) = @_; open my $fh, '<', $file or die "$file:
↪$!"; local $/; <$fh> }
```

Start `ipps` with `--norc` to leave it out for once.

Settings, macros and commands in the start file

Three calls reach the prompt itself from `rc.pl`.

A setting:

```
Peta::Shell->set(max_items => 100);
Peta::Shell->set(time => 'on');
```

A macro is a word that is replaced when it is the first word of a line and a space or the end of the line follows it. Given a string, the word is replaced and the rest of the line stays:

```
Peta::Shell->macro(p => 'print');
```

```
ipps> p "printed by a macro\n"
printed by a macro
```

Given a sub, the macro gets the rest of the line and returns the Perl to run in its place:

```
Peta::Shell->macro(sq => sub { my ($rest) = @_; "($rest) ** 2" });
```

```
ipps> sq 3 + 4
49
```

A command of your own is called with a colon, like the built-in ones, and gets the rest of the line:

```
Peta::Shell->command(greet => sub { my ($shell, $rest) = @_; print
↪"hello, $rest\n" });
```

```
ipps> :greet world
hello, world
```

`:help` lists your commands and macros after its own. The names of the built-in commands are taken.

Programs as functions

With the `autorun` setting on, a call of a sub that does not exist runs the program of that name instead and returns what the program wrote:

```
ipps> :set autorun on
ipps> whoami()
"rj\n"
ipps> my @manifests = ls('Cargo.*')
("Cargo.lock\n", "Cargo.toml\n")
```

In list context the output is a list of lines, otherwise one string. The arguments are joined with spaces into a command line for `/bin/sh`, so the shell still expands the `*` above. What the program writes to its error output appears on the terminal, and its exit status is in `$?`:

```
ipps> false(); $? >> 8
1
```

Perl's rules for calling a sub still decide what you can type:

- Without parentheses a name must be declared first, and `:programs` declares it. After `:programs ls whoami`, `ls 'Cargo.*'` and a bare `whoami` work. Before it, the first is a syntax error and the second is refused under `use strict`.
- Every argument is Perl. In `ls -l` the `-l` is Perl's file test, not an option; write `ls '-l'`.

- A name of one of Perl's own functions, such as `kill` or `sleep`, stays Perl's, and `:programs` refuses it. A program whose name Perl cannot use for a sub, such as `apt-get`, cannot be called this way.
- It works in the package `main` only.

A mistyped sub name now runs a program too: `pritn("x")` reports `Can't exec "pritn": No such file or directory`, not an undefined sub.

For options, pipes and redirections the shell's own syntax is simpler, and `! command` gives you exactly that.

To have `autorun` in every session, put it in `rc.pl`:

```
Peta::Shell->set(autorun => 'on');
```

`autorun` is `main`'s `AUTOLOAD`, and `ipps` does not switch it on while you have one of your own. Writing your own is the way to make programs behave differently, for example to give lines without their newlines:

```
use subs qw(ls cat);

sub AUTOLOAD {
    my $program = our $AUTOLOAD =~ s/.*::/r;
    return if $program eq 'DESTROY';
    chomp(my @lines = qx{$program @_});
    return wantarray ? @lines : join "\n", @lines;
}
```

`use subs` declares names for it, as `:programs` does for `autorun`.

Without a terminal

When standard input is not a terminal, `ipps` reads it line by line without a prompt and prints values and errors the same way. That makes it usable in a pipe:

```
printf 'my @w = split " ", "the quick brown fox";\nscalar @w\njoin
↪"-", reverse @w\n' | pperl --shell
```

```
4
'fox-brown-quick-the'
```

The commands work there too.

Where a line is not a file

Each line you enter is compiled and run by itself, and `ipps` carries over what the line declared. A few things in Perl last exactly as long as the code that surrounds them, and at the prompt that is one line.

The match variables

`$1`, `$2` and their relatives belong to the line that matched:

```
ipps> "2026-10-13" =~ /(\d+)-(\d+)/; "$1 / $2"
'2026 / 10'
ipps> $1
undef
```

Take what you need into variables in the same line:

```
ipps> my ($year, $month) = "2026-10-13" =~ /(\d+)-(\d+)/;
ipps> "$year / $month"
'2026 / 10'
```

`local`

`local` at the top of a line lasts to the end of that line:

```
ipps> our $level = 1; local $level = 2; $level
2
ipps> $level
1
```

Inside a sub or a block it behaves as everywhere.

Lexical subs

A sub declared with `my sub` exists for its line only:

```
ipps> my sub helper { 42 } helper()
42
ipps> helper()
Undefined subroutine &main::helper called at (ipps:2) line 1.
```

An ordinary `sub` stays.

Packages

A `package` statement holds for the following lines, as it does in a file, and `my` variables stay visible across it:

```
ipps> package Counter; my $count = 0; sub bump { ++$count }
ipps> bump(); bump()
2
ipps> __PACKAGE__
'Counter'
ipps> package main;
ipps> Counter::bump()
3
```

Unfinished input is compiled twice

While you are typing a statement over several lines, `ipps` finds out that it is unfinished by compiling what there is. A `use` or a `BEGIN` block in the first lines has run by then, and runs again when the whole text is compiled. For nearly every module that makes no difference. Where it would, put the `use` on a line of its own.

Completion runs nothing

Tab looks at the variables, packages and files that exist. It does not call your code, so it cannot know what a call returns. After

```
make_thing()->
```

there is nothing to offer. Put the result in a variable and complete on that.

Running programs

`ipps` is a shell for Perl: a line at its prompt is Perl, and a bare `ls` is a Perl word, not a command. To hand a line to your own shell, start it with `!` and a space:

```
ipps> ! ls -l Cargo.toml
-rw-r--r-- 1 rj rj 8046 Oct 12 09:18 Cargo.toml
ipps> ! ls Cargo.* | wc -l
2
```

The line goes to the shell as it stands, with its globs and pipes, and runs on your terminal. Nothing of it is Perl and its output is not kept. An exit status other than 0 is reported:

```
ipps> ! ll Cargo.toml
/bin/bash: line 1: ll: command not found
ipps: exit status 127
```

That `ll` is an alias, and the command runs under `$SHELL -c`, which does not read the start file of an interactive shell. The `shell` setting says what to run instead. For the aliases of bash:

```
ipps> :set shell bash -ic
ipps> ! ll Cargo.toml
-rw-r--r-- 1 rj rj 8046 Oct 12 09:18 Cargo.toml
```

The space after `!` matters. Without it the line is Perl's negation:

```
ipps> my $done = 0;
ipps> !$done
true
```

When you want a command's output in a variable, that is Perl, and Perl starts programs as it always has:

```
ipps> my @files = `ls Cargo.*`
("Cargo.lock\n", "Cargo.toml\n")
```

(continues on next page)

(continued from previous page)

```
ipps> system "echo", "run by system"
run by system
0
ipps> open my $pipe, "-|", "echo", "read from a pipe"; scalar␣
 ↪readline $pipe
"read from a pipe\n"
```

The `0` after the first line is what `system` returned. Listing files needs no program at all:

```
ipps> my @manifests = glob "Cargo.*"
('Cargo.lock', 'Cargo.toml')
```

CHAPTER

FIVE

REFERENCE

Reference for pperl, a faithful Rust reimplementation of perl5 5.44. Two parts: the **Perl language** reference (variables, functions, operators, regex, data types, subroutines) and the **Module** reference for every built-in native module.

5.1 Perl language

- *Variables*
- *Functions*
- *Operators*
- *Data types*
- *Subroutines*
- *Locale handling*
- *Diagnostics*
- *Security and taint mode*

5.1.1 Special variables

Perl ships with a fixed set of built-in variables that are populated, read, or honoured by the language itself. Some hold state from the last operation (the regex match variables, `$!`, `$@`); some configure how built-ins behave (`$/`, `$\`, `$,`); some expose the running interpreter and its environment (`$^V`, `$^O`, `%ENV`, `@INC`); a few are not really variables at all but hooks into runtime services (`%SIG`).

PetaPerl implements every special variable that Perl 5.44 documents, with identical semantics. Where the runtime does something noteworthy of its own - a JIT fast path that reads a cached cell, an Arc-backed COW string for `$_` - that is an implementation detail; user-visible behaviour matches perl5.

This reference is split into one page per family. Variables that share context (you almost never set `$,` without thinking about `$\` and `$|`) live on the same page; variables that don't (process IDs and regex captures) don't.

Choose a family

Default variables

Two variables hold the implicit operands of Perl. `$_` is the default scalar that countless built-ins read when called without an argument. `@_` is the argument array of every subroutine. Together they account for most of the "where did that value come from?" surprises in Perl code.

`$_` - the default scalar

`$_` is the *topic*: the value being talked about right now. A huge swath of Perl's built-ins consult it when called with no explicit argument:

```
for my $line (<$fh>) {
    chomp;                      # chomp $line  - but $line was␣
↪aliased to $_
    print if /\bERROR\b/;       # print $_ if $_ =~ /\bERROR\b/
}
```

The full list of built-ins that default to `$_` includes `chomp`, `chop`, `lc`, `lcfirst`, `length`, `pos`, `print`, `printf`, `quotemeta`, `say`, `split` (default pattern), `study`, `uc`, `ucfirst`, `unlink`, the `m//` and `s///` operators, the `tr///` operator, and the `<>`/`-X` filetest operators. Their detail pages name `$_` as the default; this page is the canonical description.

Aliasing inside `for` and `while`

The `for` (and `foreach`) loop **aliases** `$_` to each element of the list, in turn:

```
my @items = (1, 2, 3);
for (@items) {
    $_ *= 2;                    # mutates @items in place
}
# @items is now (2, 4, 6)
```

Modifying `$_` inside the loop modifies the list element itself. This is identical behaviour to `map`, `grep`, `any`, `all` - and `first` and `reduce` from `List::Util`.

`while (<$fh>)` does **not** alias - it reads each line into `$_` as a fresh copy. You cannot push back into the file by assigning to `$_` in a `while (<$fh>)` loop.

Localising `$_`

A subroutine that wants its own `$_` without disturbing the caller's must `local`-ise it:

```
sub categorise {
    local $_ = shift;           # decouple from caller's $_
    return 'small'  if /^\d{1,3}$/;
    return 'medium' if /^\d{4,6}$/;
    return 'large';
}
```

(continues on next page)

(continued from previous page)

```
my @sizes;
for ('5', '12345', '999999999') {
    push @sizes, categorise($_); # caller's $_ stays the loop alias
}
```

Without the `local`, calling `categorise()` with `shift` would work - but any `=~`, `/.../`, or `chomp` inside it would silently read or write the loop alias, breaking the caller's iteration.

$_ as an lvalue

`$_` is a real scalar variable; you can assign to it, take a reference to it, and pass it as an lvalue:

```
$_ = 'hello';
chomp;                              # nothing to chomp; $_ unchanged
s/l/L/g;                             # $_ is now 'heLLo'

my $ref = \$_;                       # \$_ is the alias's address
$$ref = 'goodbye';                  # mutates whatever $_ is currently␣
↪bound to
```

Beware that the second `$$ref` will write to **whatever $_ is right now** - which inside a `for` loop is the current list element. References to `$_` taken outside a loop and used inside one rarely do what the author meant.

Lexical $_

`our $_` and `my $_` were experimental features that have been removed. `$_` is always the package-global; use `local $_` for isolation.

@_ - subroutine arguments

Inside a subroutine, `@_` is the array of arguments passed to that call. Each element is **aliased** to the caller's expression - `@_` is not a copy:

```
sub double {
    $_[0] *= 2;                     # mutates the caller's variable
}

my $x = 5;
double($x);
# $x is now 10
```

This is why most subs start with `my (...) = @_;` - the unpacking creates copies, after which the routine cannot accidentally mutate caller state:

```
sub greet {
    my ($name, $greeting) = @_;  # decoupled from caller
```

(continues on next page)

(continued from previous page)

```
    $greeting //= 'Hello';
    return "$greeting, $name!";
}
```

The aliasing is occasionally useful - it is how `chomp(@lines)` mutates every element of the caller's array - but in everyday code the unpacked-copy idiom is the rule.

wantarray and the calling context

A sub that wants to behave differently depending on whether it was called in scalar, list, or void context inspects `wantarray`:

```
sub items {
    return wantarray ? (1, 2, 3) : 3;
}

my @list = items();              # (1, 2, 3)
my $cnt  = items();              # 3
items();                          # void - return value discarded
```

`@_` and `wantarray` are the two pieces of the calling-convention picture. `@_` says *what* was passed; `wantarray` says *what is expected back*.

@_ after shift, pop, unshift, push

Modifying `@_` itself (not its elements) has no aliasing effect on the caller - it's a normal local array variable. `shift @_` removes the first argument, leaving the rest:

```
sub method_call {
    my $self = shift;            # standard OO pattern
    my %args = @_;               # rest of args as a hash
    ...
}
```

`shift` with no arguments inside a subroutine defaults to `@_`, which is why you see `my $self = shift;` everywhere.

$_ = shift - wait, what?

A common deceptive idiom:

```
sub trim {
    $_ = shift;                  # - set caller's $_ - DO NOT DO␣
↪THIS
    s/^\s+//;
    s/\s+$//;
    return $_;
}
```

That `$_ = shift` writes to the package-global `$_`, which is *the caller's loop topic* if the caller is inside a `for`. The fix is `local`-isation, as above:

```
sub trim {
    local $_ = shift;
    s/^\s+//;
    s/\s+$//;
    return $_;
}
```

See also

- `map`, `grep` - the most common consumers of `$_` aliasing.
- `shift`, `pop` - default to `@_` inside a subroutine.
- `wantarray` - the companion of `@_` in the calling-convention picture.
- `local` - the safe way to set `$_` inside a sub that the caller might be iterating with.
- *Regex binding* - =~ is what you use when you don't want the implicit `$_` target.

Process identity

Six variables expose what the operating system thinks of this process: its name, its PID, and its real-and-effective user and group IDs. They are most useful in scripts that need to act based on *who is running them*, drop privileges after a privileged operation, or distinguish parent from child after a `fork`.

Variable	What it holds	Mutable
`$0`	Program name (and `ps` display)	yes
`$$`	Current process PID	no (read-only)
`$<`	Real UID	yes (privileged)
`$>`	Effective UID	yes (privileged)
`$(`	Real GID (plus supplementaries)	yes (privileged)
`$)`	Effective GID (plus supplementaries)	yes (privileged)

`$0` - program name

The name of the program being executed, as passed in `argv[0]`:

```
print "Started by: $0\n";        # e.g. "/usr/local/bin/myscript.pl"
```

Assigning to `$0` changes how the process appears in `ps` output and similar listings - handy for long-running scripts that fork and want each child to be identifiable:

```
$0 = "myserver: worker pid $$";   # appears in `ps` as that string
```

The change writes into the area of process memory that `ps` reads on Linux. Older Unix variants and Windows have different rules; PetaPerl follows perl5's behaviour exactly.

Reading `$0` always returns the current value, including any modifications you have made.

`$0` is **not** the program name as the *script* sees it - for that, see `@ARGV` (which has `$0` as the companion at index `[-1]` of the original `argv`, not as `@ARGV[0]`).

$$ - process ID

Read-only. The PID of the current process.

```
print "Running as PID $$\n";

open my $log, '>>', "/var/log/myapp.log" or die $!;
print $log "[$$] starting...\n";
```

After `fork`, the child sees a different `$$` from the parent - this is the standard way to tell which side of the fork you are on:

```
my $pid = fork;
defined $pid or die "fork: $!";
if ($pid == 0) {
    # child
    print "child: my pid is $$\n";
    exit 0;
}
# parent
print "parent: my pid is $$, child is $pid\n";
waitpid $pid, 0;
```

The English alias `$PROCESS_ID` (and the redundant `$PID`) work the same as `$$`.

To get the *parent's* PID, use `getppid`.

Real vs effective IDs - what they mean

Every Unix process has *two* user IDs and *two* group IDs:

- The **real** UID/GID is who you are - usually the user who invoked the program.
- The **effective** UID/GID is what the kernel checks for permission decisions - typically the same as real, but different for *setuid* and *setgid* programs.

When you run a setuid-root binary, the kernel sets the effective UID to root (the file's owner) but leaves the real UID as your own. The program can then act as root for the privileged operation, and either *drop privileges* by setting effective back to real, or *check who really invoked it* by reading the real UID.

$< - real UID

```
print "Invoked by user with UID $<\n";
```

`$<` is the real UID. It almost never changes during a process's lifetime; the cases where it can are very limited (only `root` can change it, and only via specific syscalls).

Mnemonic: *the UID you came **from***.

$> - effective UID

```
print "Privileges currently effective: UID $>\n";
```

$> is the effective UID - the one that matters for permission checks. In a setuid program, this is initially the file owner's UID; in an ordinary program, it equals $<.

Mnemonic: *the UID you went **to***.

The classic setuid pattern

A setuid-root helper that needs to do *one* privileged operation and then drop privileges so the rest of the script runs as the real user:

```
#!/usr/bin/perl
use strict;
use warnings;

# Privileged work first - we still have effective root.
open my $secret, '<', '/etc/some-restricted-file'
    or die "open: $!";
my $config = do { local $/; <$secret> };
close $secret;

# Drop privileges. From now on, both real and effective UID
# are the invoking user's, so any subsequent open/system call
# is checked against the real user.
$> = $<;                          # set effective UID  to real UID
$) = $(;                          # set effective GID  to real GID

# Now run the rest of the script as the real user.
do_user_work($config);
```

Permanent privilege drop **must** set effective before real (or both at once via `POSIX::setuid()`); on most systems setting real first will lock you out of restoring effective. Check `$!` after each assignment - the kernel rejects unauthorised changes:

```
$> = $<;
die "could not drop privileges: $!" if $> != $<;
```

Swapping real and effective

When you want to *temporarily* switch the effective ID for a single operation and then restore it:

```
my $saved_uid = $>;
$> = $<;                          # become real user
my $ok = open(my $fh, '<', $user_path);
```

(continues on next page)

(continued from previous page)

```
$> = $saved_uid;                    # back to effective
$ok or die "open $user_path: $!";
```

This works only on systems supporting `setreuid()` - most modern Unixes, including Linux. `POSIX::seteuid()` from the `POSIX` module is the explicit and portable spelling.

`$(` - real GID (with supplementary groups)

`$(` returns the real GID followed by every *supplementary group* the process belongs to, all space-separated:

```
print "Real GID list: $(\n";      # e.g. "1000 1000 4 27 100 999"
                                  #       primary + supplementaries
my @groups = split ' ', $(;
my $primary = $groups[0];
```

The leading number is the primary GID; the rest are supplementary groups (from `/etc/group`). Assigning to `$(` sets the GID; the syntax accepts a single number:

```
$( = 100;                           # become primary GID 100
```

`$)` - effective GID (with supplementary groups)

Same shape as `$(`, but for the effective GID:

```
print "Effective GID list: $)\n";

# Drop both real and effective GID to a specific gid:
$( = $) = "$gid $gid";
```

Permission checks use the effective GID and supplementaries; the real GID list is informational.

When you actually use these

Day-to-day Perl code does not touch UIDs and GIDs. The realistic use cases are:

1. **Setuid scripts** that need to drop privileges. Standard idiom shown above.
2. **Daemon startup** that runs as root to bind a privileged port, then drops privileges before serving requests:

```
socket(my $srv, ...) or die $!;
bind($srv, sockaddr_in(80, INADDR_ANY)) or die $!;
listen($srv, SOMAXCONN) or die $!;

# Now drop to nobody:nogroup before accepting connections.
my (undef, undef, $u, $g) = getpwnam('nobody');
$) = "$g $g";
$( = $g;
```

(continues on next page)

(continued from previous page)

```
$> = $u;
$< = $u;
```

3. **Privilege check** in a script that should refuse to run as root: `die "do not run me as root\n" if $< == 0`.

Tainted-mode interaction

Under `-T`, Perl notices when the real and effective IDs differ (i.e. the script is setuid) and turns on extra checks: certain operations refuse tainted arguments, and the search path for `system()` and `exec()` is constrained. The relevance here is that whether `-T` is on is decided based on `$<` vs `$>` at startup.

See also

- `getppid` - the *parent's* PID, paired with `$$`.
- `fork` - after which `$$` differs in parent and child.
- `getpwuid`, `getgrgid` - turn the numeric IDs into user and group names.
- `POSIX::setuid` - the portable, explicit way to change UIDs together.
- *Error variables* - `$!` after an `$<` / `$>` assignment tells you why the kernel refused.

Error state

Four variables report errors from four different layers. They are not interchangeable and they are populated by entirely different mechanisms; misusing one for another is the most common cause of "why doesn't my error handling work" bugs in Perl code.

Variable	Source	Lifetime
`$!`	C library `errno` from the last syscall	Set on syscall failure; meaningless otherwise
`$@`	The exception caught by the last `eval`	Set on `eval` exit (empty string on success)
`$?`	Exit status of the last child process	Set by every `wait`/pipe-close/`system`
`$^E`	OS-extended error info	Same as `$!` on Unix; richer on VMS/Win32

The single best thing you can do for your error handling is to internalise *which variable answers which question*:

- "Did my open/read/connect/syscall fail?" → check the return value first, then `$!`.
- "Did my `eval` block catch an exception?" → check `$@`.
- "What was the exit code of the program I just ran?" → decode `$?`.
- "Am I on Windows or VMS and need OS-specific error text?" → also look at `$^E`.

$! - the system error

`$!` is the Perl interface to the C library's `errno`. It has a *dual nature*: read it in numeric context to get the integer `errno`, in string context to get the corresponding error string:

```
open(my $fh, '<', '/no/such/file') or do {
    print  "errno = ", $! + 0, "\n";        # numeric: 2
    print  "errstr = $!\n";                 # string: "No such file
 ↪or directory"
};
```

Crucially, `$!` is **only meaningful immediately after a failure**. The C `errno` is not cleared on success, so a later read might see a stale value from any earlier system call. The reliable shape:

```
if (open my $fh, '<', $path) {
    # success - $! is meaningless here
    process($fh);
} else {
    # ONLY here is $! meaningful
    die "open $path: $!";
}
```

Writing the failure on the same line as the operation that caused it is the safest spelling - it leaves no opportunity for an intervening Perl operation to clobber `errno`.

%! - symbolic errno checks

`%!` is a hash whose keys are errno symbol names like `ENOENT`, `EACCES`, `EAGAIN`. A key tests true exactly when `$!` currently equals that errno:

```
unless (open my $fh, '<', $path) {
    if ($!{ENOENT}) {
        return [];                  # missing file → empty list
    }
    if ($!{EACCES}) {
        die "permission denied: $path";
    }
    die "open $path: $!";           # anything else: re-raise
}
```

This is portable - the numeric values of errno symbols differ across operating systems, but `%!` keys are always the symbol names. Behind the scenes, `%!` is provided by the `Errno` module, which is loaded on demand the first time you use the hash.

Assigning to $!

Writing to `$!` sets `errno` in the C library - useful for faking up an error message that downstream code will see:

```
$! = 13;                            # EACCES
print "$!\n";                       # "Permission denied"

# Slightly more idiomatic - set errno by symbol:
use Errno qw(:POSIX);
$! = EACCES;
die "synthetic permission failure: $!";
```

$@ - the exception variable

Set by `eval` when the block (or string) it ran threw an exception. It contains either the value passed to `die` (which is most often a string, but can be any reference) or the parsing/runtime error message from the interpreter.

```
eval {
    die "no port configured\n" unless defined $port;
    open(my $sock, '<', "/dev/tcp/$host/$port") or die "connect: $!
 ↪";
    1;
};
if ($@) {
    warn "could not open $host:$port - $@";
    return;
}
```

Successful evaluation sets `$@` to the empty string `''`. The `if ($@)` check is therefore reliable: empty string is false, any non-empty value is true.

The classic $@ clobbering bug

There is a hazard here. Anything that runs *between* the `eval` exit and your `if ($@)` check can clobber `$@`. The most common offender is an object's `DESTROY` method - when objects go out of scope at the close of the `eval` block, their `DESTROY` runs, and if `DESTROY` itself does any `eval`, your exception is gone:

```
eval {
    my $obj = MyClass->new;
    $obj->might_die;
    1;
};
# If MyClass::DESTROY does an eval, $@ is now empty.
warn "got: $@";                     # may print nothing!
```

The historically-recommended fix is to capture `$@` immediately, or to use the `Try::Tiny` module, which handles this and several related pitfalls:

```
my $err;
{
    local $@;                       # save and isolate
    eval {
```

(continues on next page)

(continued from previous page)

```
        risky_op();
        1;
    } or $err = $@ || 'unknown error';
}
if ($err) {
    warn "got: $err";
}
```

The Perl 5.34+ native `try/catch` feature is the cleaner modern spelling - it does not use `$@` at all and is not affected by the destructor problem:

```
use feature 'try';
try {
    risky_op();
} catch ($err) {
    warn "got: $err";
}
```

PetaPerl supports `try/catch` natively. New code should prefer it.

`eval` and `$!`

`eval` does **not** preserve `$!`. If your `eval` block did a syscall that failed and then `die`d, `$!` is whatever the *last* operation set it to - which inside a destructor or `local $SIG{__DIE__}` handler can be anything. If your error message needs `$!`, capture it inside the `eval`:

```
eval {
    open(my $fh, '<', $path) or die "open $path: $!\n";
    ...
};
```

The `\n` at the end is conventional - it suppresses the "at line N" suffix that Perl appends to die messages, which is right for user-facing errors.

`$?` - the child error

After a `system`, `waitpid`, `wait`, backtick command, or pipe close, `$?` contains the 16-bit `wait()` status of the child process. It is **not** simply the exit code:

```
system($cmd, @args);
my $status = $?;

my $exit_code = $status >> 8;        # exit value passed to exit() in the child
my $signal    = $status & 0x7F;      # signal that killed the child, or 0
my $coredump  = $status & 0x80;      # set if the child dumped core

if ($status == -1) {
```

(continues on next page)

(continued from previous page)

```
    die "system: $!";                # could not even fork/exec
} elsif ($signal) {
    die "$cmd died with signal $signal";
} elsif ($exit_code) {
    die "$cmd failed (exit $exit_code)";
}
```

The `>> 8` shift is the most-cited Perl gotcha there is. Only the upper byte is the exit code; the lower byte holds the signal-and-coredump flags. Forgetting the shift produces values multiplied by 256.

Modern Perl provides the `POSIX` module's `WEXITSTATUS`, `WIFSIGNALED`, `WTERMSIG`, `WIFEXITED` macros, which are the portable named accessors:

```
use POSIX ':sys_wait_h';
if (WIFEXITED($?))   { my $code = WEXITSTATUS($?); ... }
if (WIFSIGNALED($?)) { my $sig  = WTERMSIG($?);    ... }
```

`$?` is **not** preserved across other operations. If you need the value, capture it into a local variable on the next line - otherwise the next backtick or `system` will overwrite it.

`$?` in `END` blocks

Inside an `END` block, `$?` contains the intended exit code. Assigning to it overrides the script's exit status:

```
END {
    $? = 0 if $? == 255 && $shutdown_was_clean;
}
```

`$^E` - the extended OS error

On Unix, `$^E` is exactly the same as `$!`. On Windows, it contains the `GetLastError()` value, which is sometimes more informative than `errno` (which Win32 maps a coarse subset onto). On VMS, it is the native VMS status code.

PetaPerl is Linux-only. `$^E` and `$!` are interchangeable here; the variable exists for portable code that may also run on non-Unix perls.

Putting them together - a worked example

```
sub copy_file {
    my ($src, $dst) = @_;

    open my $in,  '<', $src or die "open $src for reading: $!";
    open my $out, '>', $dst or die "open $dst for writing: $!";

    eval {
        local $/ = \65536;       # block reads, scoped to this eval
```

(continues on next page)

(continued from previous page)

```
        while (defined(my $chunk = <$in>)) {
            print {$out} $chunk or die "write $dst: $!";
        }
        close $out                or die "close $dst: $!";
        1;
    } or do {
        my $err = $@ || 'unknown error';
        unlink $dst;              # clean up the partial file
        close $in;                # may set $! itself
        die "copy_file($src → $dst): $err";
    };

    close $in or warn "close $src: $!";
    return 1;
}
```

Three error variables in five paragraphs. `$!` reports each syscall failure on the line that triggered it. `$@` collects any exception that crossed the `eval` boundary. `$?` is not in this example because there is no child process - but if the function shelled out to `cp`, you would decode `$?` in the same shape as the `system` page.

See also

- `die` - populates `$@`. The usual way to raise an exception from your own code.
- `eval` - catches exceptions; sets `$@`.
- `system`, `waitpid`, `wait` - populate `$?`.
- `try/catch` - the modern alternative to `eval`/`$@` that side-steps the clobbering hazard.
- `Errno` - the source of `%!` keys; provides named errno constants.
- `Try::Tiny` - robust `eval` wrapper that preserves `$@` correctly.
- `%SIG` - `$SIG{__DIE__}` and `$SIG{__WARN__}` are the hooks that fire when `die` and `warn` run; they can rewrite `$@`.
- *Logical operators* - the `or die` idiom that drives almost all `$!` reporting.

Filehandle and printing state

Seven variables that drive Perl's I/O. They are *globals* - setting `$/` in one routine changes how every other routine in the same process reads from a filehandle. The discipline that keeps that from being a disaster is `local`-isation: every change to a variable on this page should be inside a `local` block unless the program genuinely wants the new value to be permanent.

Vari-able	Reads as	Default	Used by
`$/`	input record separator	`"\n"`	`<$fh>`, `readline`
`$\`	output record separa-tor	`undef`	`print`, `printf`, `say` (forced `\n`)
`$,`	output field separator	`undef`	`print`, `printf` (between args)
`$"`	list separator	`" "`	array interpolation `"@arr"`
`` `$ ``	`` ` ``	output aut-oflush	`0`
`$.`	current line number	`0`	last filehandle read
`$;`	subscript separator	`"\034"`	`$h{$x, $y}` multidim emula-tion

`$/` - the input record separator

Controls what `<$fh>` and `readline` treat as the end of one record. The default `"\n"` means *line-by-line* reading; setting it to other values produces other reading modes.

Line mode (the default)

```
while (my $line = <$fh>) {
    chomp $line;                # strips the terminator $/
    process($line);
}
```

`chomp` removes whatever `$/` currently holds - when you change `$/`, `chomp` follows it.

Slurp mode

`undef $/` makes <> read the entire file in one go:

```
my $whole_file = do {
    local $/;
    <$fh>;
};
```

The `local` is essential. Without it, every other piece of code that subsequently reads from a filehandle in the same process will silently slurp.

Paragraph mode

`local $/ = "";` reads paragraph-at-a-time, splitting on runs of two or more newlines:

```
{
    local $/ = "";
    while (my $para = <$fh>) {
        # $para ends with exactly two newlines (or one at EOF)
```

(continues on next page)

(continued from previous page)

```
        process_paragraph($para);
    }
}
```

This is awk's `RS=""` behaviour. Leading blank lines at the start of the file are silently skipped.

Fixed-record mode

`local $/ = \N` reads `N` bytes per call:

```
local $/ = \4096;                # binary 4-KiB chunks
while (defined(my $chunk = <$fh>)) {
    process($chunk);
}
```

The reference is to a number, not the number itself. This is the right shape for reading binary data, network packets, or doing bulk copy without scanning for `\n`.

Multi-character terminators

```
local $/ = "\r\n";               # CRLF mode
local $/ = "\nEND-OF-RECORD\n";  # custom marker
```

The terminator is a literal string, not a regex.

`$\` - the output record separator

Appended to every `print` call. Default is `undef` (nothing appended). Setting it changes every subsequent `print`:

```
{
    local $\ = "\n";             # all prints get a trailing newline
    print "first";
    print "second";
}
# Output: "first\nsecond\n"
```

This is the natural fit for `-l` on the command line - `-l` sets `$\` to `\n`. In script code, `local $\ = "\n"` inside a tight reporting loop is occasionally worth it; otherwise `say` (which forces `\n` regardless of `$\`) is the cleaner spelling.

`$,` - the output field separator

Inserted between `print`'s arguments. Default is `undef`:

```
print "a", "b", "c";              # "abc"

{
```

(continues on next page)

(continued from previous page)

```
    local $, = ", ";
    print "a", "b", "c";          # "a, b, c"
}

{
    local $, = "\t";
    print @items;                 # tab-separated record
}
```

`$,` is per-print, not per-line - it does not separate *successive* `print` calls. Combine with `$\` for full TSV-style output:

```
local ($,, $\) = ("\t", "\n");
print 'col1', 'col2', 'col3';     # "col1\tcol2\tcol3\n"
```

$" - the list separator

Joins array elements when an array is interpolated into a double-quoted string. Default is `" "` (single space):

```
my @arr = (1, 2, 3);
print "items: @arr\n";            # "items: 1 2 3"

{
    local $" = ', ';
    print "items: @arr\n";        # "items: 1, 2, 3"
}

{
    local $" = "\n  ";
    print "items:\n  @arr\n";     # "items:\n  1\n  2\n  3"
}
```

`$"` is *only* consulted on interpolation. `print @arr` (no quotes) uses `$,`, not `$"`.

$| - output autoflush

Forces a flush after every write on the **currently selected** filehandle:

```
$| = 1;                           # autoflush STDOUT
print "Loading..."; sleep 1;      # appears immediately, not at exit
```

The "currently selected" qualification matters. By default, `select` returns `STDOUT`, so `$| = 1` autoflushes STDOUT. To autoflush a different handle:

```
my $old = select($log_fh);
$| = 1;
select($old);                     # restore selection
```

The slightly nicer modern spelling uses `IO::Handle`:

```
$log_fh->autoflush(1);
```

`$|` is read-as-boolean: `print $|` shows `1` or `0`.

`$.` - current line number

The line number of the last <>-style read. Each filehandle has its own counter; reading from a different handle aliases `$.` to that handle's count:

```
while (<$fh>) {
    print "$.: $_";              # numbered listing
}

# After the loop, $. is the total number of lines read.
```

`$.` is reset when the filehandle is *closed*. Reopening (without an explicit close in between) does **not** reset.

For multi-file iteration via <>, lines are numbered cumulatively across files unless you explicitly close `ARGV` at each file boundary - see `eof` for the standard `close ARGV if eof;` idiom.

`$;` - the subscript separator

Used to fake multi-dimensional hashes - a deeply legacy feature preserved for backward compatibility:

```
$h{$x, $y} = $value;             # equivalent to $h{"$x\034$y"}
```

The default value is `"\034"` (FS, "file separator", an obscure ASCII control byte). Real multi-dimensional structures should use nested hash refs (`$h{$x}{$y}`); `$;` exists because Perl 4 had no proper references.

When you actually localise these

The most common shape, by far:

```
sub slurp_text {
    my ($path) = @_;
    open my $fh, '<', $path or die "open $path: $!";
    local $/;                    # slurp inside this sub only
    return <$fh>;
}

sub paragraphs_of {
    my ($path) = @_;
    open my $fh, '<', $path or die "open $path: $!";
    local $/ = "";
    return <$fh>;                # list context: list of paragraphs
}
```

(continues on next page)

(continued from previous page)

```
sub csv_print {
    my (@row) = @_;
    local ($,, $\) = (",", "\n");
    print @row;
}
```

The cost of forgetting `local` is invisible at write time and catastrophic at use time - a subroutine that "just sets `$/`" can break unrelated code reading from completely different filehandles. Treat `$/`, `$\`, `$,`, `$"` as *global mode flags*: always set them with `local`.

See also

- `open`, `readline`, `<$fh>` - the consumers of `$/`.
- `print`, `printf`, `say` - the consumers of `$\`, `$,`.
- `chomp` - removes whatever `$/` currently holds; pairs with `$/` changes.
- `eof` - the `close ARGV if eof` idiom that makes `$.` count per-file across `<>`.
- `select` - selects the filehandle that `$|` and the unprefixed `$,`/`$\` apply to.
- `IO::Handle` - provides `autoflush`, `output_record_separator`, etc. as method calls.
- `local` - the safe way to change any variable on this page.
- *Interpolation* - how `$"` shapes array expansion inside double-quoted strings.

Regex match variables

After every successful pattern match, Perl populates a fixed set of variables with information about *what* matched and *where*. These are read-only - you observe them, you do not write to them. The "successful" qualification is critical: an unsuccessful match leaves the variables holding *whatever the previous successful match in the same dynamic scope set them to*. Always gate your access with the boolean result of the match itself.

Variable	Holds
`$1..$N`	Text captured by the *N*th capturing group
`$&`	The entire matched substring
`$``	The string preceding the match
`$'`	The string following the match
`$+`	Text captured by the highest-numbered group
`$^N`	Text captured by the *most recently closed* group
`@-`	Start offsets: `$-[0]` = match, `$-[N]` = $N start
`@+`	End offsets: `$+[0]` = match end, `$+[N]` = $N end
`%+`	Hash of named captures: `$+{name}` = `(?<name>...)`
`%-`	Hash of *all* captures by name (arrayref values)
`@{^CAPTURE}`	Array of captures: `${^CAPTURE}[0]` = $1, etc.
`${^MATCH}`	Same as $&, populated only with the `/p` flag
`${^PREMATCH}`	Same as $`, only with `/p`
`${^POSTMATCH}`	Same as $', only with `/p`

The basic pattern

```
if ("Mr. Smith, age 47" =~ /(\w+)\s+(\w+),\s+age\s+(\d+)/) {
    print "title: $1\n";          # Mr
    print "name:  $2\n";          # Smith
    print "age:   $3\n";          # 47
    print "match: $&\n";          # the full match
}
```

The `if` is what makes the access safe. Without it, on a *non-matching* string, `$1`/`$&` would still hold values from some earlier successful match - there is no "match failed → clear" rule.

Numbered captures - $1..$N

Each `(...)` capturing group fills one variable. The numbering is left-to-right by *opening* parenthesis:

```
"alpha-beta=42" =~ /^(\w+)-(\w+)=(\d+)$/;
print "$1 / $2 / $3\n";           # alpha / beta / 42
```

Non-capturing groups `(?:...)` and lookarounds `(?=...)`/`(?!...)` do **not** consume a number; they are invisible to the count.

For `s///` substitutions, `$1..$N` are visible inside the replacement (along with the substitution-only `$1`-style backreferences in single-quoted replacements):

```
my $s = "John Smith";
$s =~ s/(\w+)\s+(\w+)/$2, $1/;    # "Smith, John"
```

Named captures - (?<name>...) and %+

Named captures are clearer than counting parentheses, especially in patterns with many groups:

```
my $log = "2024-03-15 14:32:01 ERROR connection refused";
if ($log =~ /^(?<date>\d{4}-\d{2}-\d{2})
              \s+
              (?<time>\d{2}:\d{2}:\d{2})
              \s+
              (?<level>\w+)
              \s+
              (?<msg>.*)$/x) {
    print "[$+{level}] $+{date} $+{time}: $+{msg}\n";
}
```

`%+` is the named-capture hash. The numbered forms still work (named captures are also assigned numbers in the order they appear), so `$1` would be `$+{date}` here.

`%-` is similar but its values are array references, holding *every* capture under that name (relevant when using branch-reset groups `(?|...|...)` or when a name is reused across alternation branches):

```
"abc" =~ /(?|(?<x>a)|(?<x>b)|(?<x>c))/;
print "%-{x} has @{$-{x}}\n";    # captured value(s) under name 'x'
```

Most code only ever reads `%+`. `%-` is for the corner cases.

Match boundaries - `@-` and `@+`

The match's start and end offsets in the matched string:

```
"hello world" =~ /(\w+)\s+(\w+)/;
print "match started at $-[0], ended at $+[0]\n";   # 0 .. 11
print "group 1: $-[1] .. $+[1]\n";                  # 0 .. 5
print "group 2: $-[2] .. $+[2]\n";                  # 6 .. 11
```

`$-[N]` and `$+[N]` give the same information as `substr($var, $-[N], $+[N] - $-[N])` would extract - they are how you reconstruct *positions*, not values, after a match.

The classic use: replacing matched substrings while preserving the original (without the `s///` operator):

```
my $s = "the quick brown fox";
$s =~ /quick (brown)/;
my $before = substr($s, 0, $-[0]);
my $after  = substr($s, $+[0]);
my $g1     = substr($s, $-[1], $+[1] - $-[1]);
print "$before|FOUND $g1|$after\n";
```

`$&`, `$``, `$'` - match, prematch, postmatch

```
"hello world" =~ /wo\w+/;
print "before: '$`'\n";          # 'hello '
print "match:  '$&'\n";          # 'world'
print "after:  '$\''\n";         # ''
```

These three are Perl's oldest match variables and historically the most expensive - see *Performance: the* `$&` *story* below.

`$+` is the text captured by the *highest-numbered* group that participated in the match:

```
"abc" =~ /(a)(b)?(c)?/;          # $1='a', $2='b', $3='c', $+='c'
"a"   =~ /(a)(b)?(c)?/;          # $1='a', $2=undef, $3=undef, $+='a
 ↪'
```

`$^N` is similar but holds the most-recently-*closed* group's text - useful inside complex patterns that need the value of "the group that just finished":

```
my $s = "tag:value";
$s =~ /(\w+):(\w+) (?{ $cb = $^N })/;
# $cb is the text most recently captured (here: "value")
```

Performance: the $& story

Historical caveat that you will find in old code and in books:

> Don't mention `$&`, `$\``, or `$'` *anywhere* in your code, including in modules you require - they cause every successful match to copy the whole matched string, slowing the program down.

This was true through Perl 5.10. From Perl 5.18 onward, the runtime tracks *which* of the three variables your code actually mentions and only copies what is needed. From Perl 5.20 a copy-on-write scheme makes them effectively free.

PetaPerl follows the modern behaviour: `$&` and friends are safe to use anywhere. The `/p` flag and `${^MATCH}` / `${^PREMATCH}` / `${^POSTMATCH}` exist for the era between 5.10 and 5.20 when `/p`-gated copying was a useful optimisation. There is no reason to write new code with them.

Scoping - they are dynamically scoped

The match variables are not lexicals; they behave like dynamically-scoped variables. Every successful match localises a *global match state* to the current dynamic scope. Crucially:

- An *unsuccessful* match does **not** clear the variables.
- A successful match *inside an inner block* overrides them, but only within that block - when the inner block exits, the outer scope's match state is restored.

```
"alpha" =~ /(\w+)/;              # $1 = 'alpha'
{
    "1234" =~ /(\d+)/;           # $1 = '1234' (inside this block)
    print "inner: $1\n";          # 1234
}
print "outer: $1\n";              # alpha - restored
```

This is occasionally surprising: a function you call from inside a regex-handling block does not pollute your `$1` unless that function does its own successful match in the *same* dynamic scope (which is rare unless they are running in the same lexical block).

${^LAST_SUCCESSFUL_PATTERN}

A read-only reference to the regex that produced the current match state - useful for diagnostics:

```
"hello" =~ /(\w+)/;
print "last pattern was: ${^LAST_SUCCESSFUL_PATTERN}\n";
```

When matches fail - pos

The position on a string after a `/g`-anchored match is held *not* in one of the variables on this page, but in `pos`:

```
my $s = "1 2 3 4";
while ($s =~ /(\d+)/g) {
    print "matched $1 at ", pos($s) - length($1), "\n";
}
# After the loop, pos($s) is undef.
```

`pos` is per-string, settable, and is what `\G` anchors to.

`@{^CAPTURE}` - captures as an array

The numbered captures, also exposed as a zero-indexed array:

```
"alpha=42" =~ /(\w+)=(\d+)/;
print "name = ${^CAPTURE}[0]\n"; # 'alpha' (same as $1)
print "val  = ${^CAPTURE}[1]\n"; # '42'    (same as $2)
print "n    = scalar @{^CAPTURE}\n";  # 2
```

This is occasionally easier to iterate than `$1, $2, ...`, but most code uses the numbered or named forms directly.

See also

- `m//`, `s///` - the operators that populate every variable on this page.
- `qr//` - compiles a pattern; the resulting regex object can later be matched against and produce these same captures.
- `pos` - the per-string offset for `/g` and `\G`.
- *Regex binding* - =~, the operator that decides which string the match runs against.
- *Regular expressions guide* - the regex language itself.
- *Groups and captures* - the chapter on the capture variables.

Interpreter introspection

Variables that expose the running Perl interpreter and its configuration. Most code never touches them; the cases where they matter are version checks, OS-conditional logic, profiler hooks, and in-place editing.

Variable	Holds
`$]`	Perl version as a numeric string (`5.044000`)
`$^V`	Perl version as a `version` object (`v5.44.0`)
`$^O`	Operating system name (`linux`, `darwin`, ...)
`$^X`	Path to the running Perl/PetaPerl executable
`$^T`	Process basetime (Unix epoch seconds)
`$^W`	Warnings master switch (boolean)
`$^I`	In-place edit extension (set by `-i`)
`$^P`	Debugger flag bitmask
`$^R`	Last regex code-block result
`$^S`	Exception-handling state (in `eval`? `undef`/0/1)
`${^GLOBAL_PHASE}`	Current interpreter phase (`START`, `RUN`, `END`, ...)

`$]` - numeric version

A floating-point representation of the interpreter version. Format: `5.PPPSSS` where `PPP` is the major and `SSS` the patch:

```
print "$]\n";                    # e.g. "5.044000"
warn "old perl"  if "$]" < 5.038;
warn "new perl"  if "$]" >= 5.040;
```

The `"$]"` stringification before comparison is recommended - it sidesteps subtle floating-point representation issues that can make `$] == 5.044` mysteriously fail.

The English alias `$OLD_PERL_VERSION` is documented but discouraged; `$]` itself is fine.

`$^V` - version object

The interpreter version as a `version` object, suitable for reading and for richer comparison:

```
print "$^V\n";                   # "v5.44.0"
warn "need 5.40+" if $^V lt v5.40.0;
```

The literal `v5.40.0` is a v-string - Perl's native syntax for version constants. Use `$^V` when comparing to v-string literals; use `$]` when comparing to bare numerics.

In PetaPerl, `$^V` reports the underlying Perl-compatibility level (5.44), not the PetaPerl release version.

`$^O` - operating system name

```
print "$^O\n";                   # "linux"
if ($^O eq 'MSWin32') {
    use_windows_path_separator();
}
```

PetaPerl is Linux-only; `$^O` is always `"linux"`. The variable exists for source-compatibility with portable code that branches on it. The full list of values

upstream Perl can produce includes `linux`, `darwin`, `freebsd`, `openbsd`, `MSWin32`, `cygwin`, `solaris`, and several historical entries.

`$^X` - executable path

The path to the interpreter currently running, suitable for re-invoking:

```
print "$^X\n";                    # e.g. "/usr/local/bin/pperl"

# Rerun the script under the same interpreter, with extra args:
exec $^X, $0, @ARGV, '--debug';
```

`$^X` is what makes `exec $^X, $0, ...` portable - you do not need to know whether the script was started via `pperl`, `/usr/bin/env perl`, or an absolute path.

`$^T` - basetime

The Unix epoch second at which the program started running:

```
my $uptime = time() - $^T;
print "running for $uptime seconds\n";
```

The classic use is the `-M` and `-A` filetests, which by default report file ages *relative to* `$^T` rather than to `time()`. If you set `$^T = time()` periodically, those filetests recompute against the new reference point - useful in long-running daemons that need to ignore files older than "since I started this batch".

`$^W` - the warnings switch

The master warnings flag, set by the `-w` command-line switch:

```
$^W = 1;                          # all warnings on
$^W = 0;                          # all warnings off
```

`$^W` is the Perl-1-era global flag. The modern lexical `use warnings` pragma is the recommended way to control warnings, and operates independently - a `use warnings` block stays on regardless of `$^W`, and a `no warnings` block stays off regardless of `$^W`. The variable mostly matters in code that pre-dates the lexical pragma or that wants to toggle the global default at runtime.

`${^WARNING_BITS}` is the lexical bitmap that `use warnings` reads and writes. Its format is internal; do not rely on specific bit positions.

`$^I` - in-place edit extension

Set by the `-i` command-line switch. When `$^I` is defined, reading from `<>` (the `while (<>)` form) re-opens each file for both read and write and redirects subsequent `print` to the rewritten file:

```
# perl -i.bak -ne 's/\bold\b/new/g; print' *.txt
# Equivalent in script form:
```

(continues on next page)

(continued from previous page)

```
local $^I = '.bak';
while (<>) {
    s/\bold\b/new/g;
    print;
}
```

`$^I = ''` (empty string) means "rewrite in place without a backup". `$^I = '.bak'` saves the original as `<filename>.bak`. Setting `$^I = undef` turns in-place editing off.

$^P - debugger flag bitmask

Read by the Perl debugger; if non-zero, `DB::DB` is called before every statement. Manipulated by `perl -d` and the `Devel::*` family of debugger and profiler modules.

User code rarely sets `$^P`. The flag bits are documented in upstream perlvar; the most commonly seen value is `0x73f` (full debugging).

$^R - regex code-block result

The result of the last `(?{...})` or `(?(N)...)` regex code assertion. Lives only as long as the surrounding match:

```
"abc123" =~ /(?<word>\w+?)(\d+)(?{ length($1) + length($2) })/;
print "$^R\n";                  # 6 - the code block's last␣
↪expression
```

PetaPerl supports `(?{ })` regex code blocks; see the *performance chapter* of the regex guide for the language details.

$^S - exception-handling state

Reports whether the interpreter is currently inside an `eval`. It holds one of three values:

$^S	Meaning
undef	Parsing a module, an `eval`, or the main program
1 (true)	Executing inside an `eval` or `try` block
0 (false)	Anywhere else

The `undef` state is the one that matters: it can occur inside a `$SIG{__DIE__}` or `$SIG{__WARN__}` handler, where the interpreter is compiling rather than running. A handler that needs to behave differently during compilation tests for it:

```
local $SIG{__DIE__} = sub {
    return if $^S;              # real runtime exception - let it␣
↪propagate
    warn "compile-time failure: $_[0]";
};
```

The legacy long name (`use English`) is `$EXCEPTIONS_BEING_CAUGHT`, which is slightly misleading: the `undef` value does not tell you whether exceptions are being caught, because compiling the main program does not catch them.

`${^GLOBAL_PHASE}` - current interpreter phase

The current life-cycle phase of the Perl interpreter:

Phase	When
`CONSTRUCT`	Inside the C-level interpreter constructor
`START`	`BEGIN` blocks, `use`, `require`, top-level compile
`CHECK`	After compilation, before `INIT`
`INIT`	`INIT` blocks
`RUN`	Main program execution
`END`	`END` blocks
`DESTRUCT`	Global destruction

```
END {
    if (${^GLOBAL_PHASE} eq 'END') {
        log_shutdown();           # we are still in user END
    }
}

# Inside a destructor, distinguish "object cleanup during normal
# scope exit" from "object cleanup during interpreter shutdown":
sub DESTROY {
    my $self = shift;
    if (${^GLOBAL_PHASE} eq 'DESTRUCT') {
        # global destruction - order is undefined, file handles
        # may already be torn down - be conservative
        return;
    }
    $self->{handle}->flush;
}
```

The "don't run during global destruction" guard inside `DESTROY` is the variable's classic use case. During `DESTRUCT`, the order of object teardown is not defined; an object's references may already be invalid.

See also

- `use VERSION` - the conventional way to require a minimum Perl version, replacing `if "$]" < ...`.
- `require VERSION` - same as `use VERSION` but at runtime.
- `Config` - the full Perl build configuration, much richer than `$^O` and `$]`.
- `use warnings` - the lexical pragma that supersedes `$^W` for new code.
- *Command-line switches* - documents `-i` (`$^I`), `-w` (`$^W`), `-d` (`$^P`).

Program input - @ARGV, %ENV, @INC

Three globals carry input *into* the program: command-line arguments, environment variables, and the module search path. They share enough context that decisions about one usually inform the others - `@ARGV` interacts with `$0`, `%ENV`'s Perl-relevant keys influence `@INC`, and `@INC`'s ordering question is mirrored by similar concerns for `@ARGV` consumers and shell `PATH`.

Variable	Holds
`@ARGV`	Command-line arguments (excluding script name)
`$0`	Script name (paired with `@ARGV`)
`%ENV`	Process environment
`@INC`	Module search path (used by `use`/`require`)
`%INC`	Already-loaded modules
`$INC`	Current `@INC` index during a hook (5.37.7+)

@ARGV and its relationship to $0

After Perl starts, the *script name* is in `$0` and the *script's arguments* are in `@ARGV`:

```
$ pperl myscript.pl -v --output /tmp/x foo bar
```

```
print "$0\n";                    # myscript.pl
print "@ARGV\n";                 # -v --output /tmp/x foo bar
print scalar(@ARGV), " args\n";  # 5 args
```

`$ARGV[0]` is the *first script argument*, **not** the script name itself. This differs from C's `argv` (where `argv[0]` is the program name); Perl exposes the program name separately via `$0`.

Standard argument handling

Hand-written arg parsing is fine for trivial scripts:

```
my %opt;
while (@ARGV && $ARGV[0] =~ /^-/) {
    my $flag = shift @ARGV;
    last if $flag eq '--';
    if    ($flag eq '-v')      { $opt{verbose} = 1 }
    elsif ($flag eq '--output'){ $opt{output}  = shift @ARGV }
    else                       { die "unknown flag: $flag\n" }
}
my @files = @ARGV;
```

For anything beyond toy scripts, the `Getopt::Long` module handles every standard pattern (long options, abbreviation, mandatory arguments, repeated flags, `--` separator, automatic help):

```
use Getopt::Long;
my %opt;
GetOptions(
    'verbose|v'    => \$opt{verbose},
    'output|o=s'   => \$opt{output},
    'help|h'       => \$opt{help},
) or die "bad options; try --help\n";
my @files = @ARGV;              # what's left after option␣
↪processing
```

`GetOptions` mutates `@ARGV` in place - when it returns, `@ARGV` contains only the non-option arguments.

@ARGV and the diamond operator

The `<>` (diamond) operator iterates lines from the files named in `@ARGV`, falling back to STDIN if `@ARGV` is empty:

```
while (<>) {
    # one line from one of the @ARGV files (or STDIN)
    chomp;
    process($_);
}
```

This is the default contract of every awk-style filter program. `<>` opens each file in turn, sets `$ARGV` to its name during reading, and closes it at EOF.

`-i` and the other `-` switches (`-n`, `-p`, `-l`, `-a`) all build their behaviour on this `@ARGV` → `<>` pattern.

Modifying @ARGV before <>

You can pre-process `@ARGV` to alter what `<>` reads:

```
# Add an extra file to the front of the argv list:
unshift @ARGV, 'header.txt';

# Process compressed files transparently by rewriting argv:
@ARGV = map { /\.gz$/ ? "gunzip -c $_ |" : $_ } @ARGV;

while (<>) {
    process($_);
}
```

The `cmd |` form turns the argument into a pipe-from-command open; this is a long-standing Perl idiom for transparent decompression in awk-style scripts.

%ENV - the process environment

%ENV is the hash of environment variables inherited from the parent process (typically the shell). Reading is straightforward:

```
my $home     = $ENV{HOME};
my $term     = $ENV{TERM} // 'dumb';
my $path     = $ENV{PATH};
```

Writing changes the environment seen by *child processes* that this Perl script subsequently spawns:

```
$ENV{LC_ALL}      = 'C';           # POSIX locale for child commands
$ENV{TZ}          = 'UTC';
$ENV{LANG}        = 'C';
delete $ENV{LC_NUMERIC};           # remove a key entirely

system('date');                     # child sees the modified ENV
```

The change does *not* propagate back to the parent - the shell that started the script keeps its original environment. If you need the parent shell to pick up new values, write them to a file or have the shell `eval` your output (the standard `dotenv` pattern).

Stringification of values

As of Perl 5.18, %ENV values are always stringified at assignment time. Storing a reference no longer round-trips:

```
my $arr = [1, 2, 3];
$ENV{DATA} = $arr;
print $ENV{DATA};                  # "ARRAY(0x...)" - stringified
```

Environment variables are a string-typed interface to the operating system; preserving structure across a `fork`/`exec` boundary would not work anyway. Marshal anything non-trivial as JSON or similar before storing.

Perl-specific environment variables

A small set of environment variables are honoured by Perl itself, *not* by user scripts:

Variable	Effect
PERL5LIB	Colon-separated paths prepended to @INC (subject to taint rules)
PERLLIB	Same, older variable; used only if PERL5LIB is unset
PERL5OPT	Switches applied as if on the command line (subset only)
PERLDB_OPTS	Options for the debugger
PERL_UNICODE	Default UTF-8 layer settings (mirrors -C)
PERL_USE_UNSAFE_INC	Re-add . to @INC (removed by default since 5.26 - strongly discouraged)
PERL_HASH_SEED	Per-run hash randomisation seed
PERL_SIGNALS	unsafe to opt out of deferred signal handling
HOME	Read by ~ glob expansion; not Perl-specific but Perl-relevant
PATH	Searched by system/exec for unqualified commands

In tainted mode (-T), PERL5LIB, PERL5OPT, and PERLLIB are ignored to prevent privilege escalation. See the *command-line switches guide* for the full security model.

delete $ENV{KEY} versus $ENV{KEY} = undef

The two are *not* equivalent:

```
delete $ENV{KEY};                  # KEY is not in the environment
$ENV{KEY} = undef;                 # KEY is in the environment, value
 ↪""
```

A child process started after delete will not see KEY at all - getenv("KEY") returns NULL. After undef assignment, the child sees KEY="" (empty string). Code that distinguishes "unset" from "set to empty" cares about this difference.

@INC - the module search path

When use and require look up a module, they walk @INC left to right and stop at the first match:

```
print "$_\n" for @INC;
# /usr/local/lib/perl5/site_perl/5.44.0/x86_64-linux
# /usr/local/lib/perl5/site_perl/5.44.0
# /usr/local/lib/perl5/5.44.0/x86_64-linux
# /usr/local/lib/perl5/5.44.0
# (no trailing "." since Perl 5.26)
```

@INC is initialised at startup from a combination of:

1. Compiled-in defaults (the build-time installprivlib, installsitelib, ...).
2. -I /path switches on the command line, *prepended* in order.
3. The PERL5LIB (or PERLLIB) environment variable, also prepended.

After startup it is just an ordinary array. *Order matters*: a match in an early element wins.

`use lib` versus `unshift @INC` versus `push @INC`

Three ways to add a path. They are not equivalent:

```
use lib '/my/lib';                      # unshift at compile-time
unshift @INC, '/my/lib';                # unshift at runtime
push    @INC, '/my/lib';                # push    at runtime
```

The differences:

- **`use lib`** runs at *compile time*. It happens before any `use` statement that follows it on the page. This is what you almost always want when you're saying "this script needs a custom library directory":

  ```
  use lib '/opt/myapp/lib';
  use MyApp::Module;                    # found because lib was␣
   ↪unshifted first
  ```

  Internally `use lib '/path'` does `BEGIN { unshift @INC, '/path' }`, so the new path is searched *first* - a copy of a module in `/opt/myapp/lib` shadows one in the system `@INC`.

- **`unshift @INC, '/path'`** at the top level has *runtime* semantics. It works for `require` calls that happen after the `unshift`, but a `use` statement on a later line of the same file *will not see it* (because `use` is itself compile-time):

  ```
  unshift @INC, '/my/lib';
  use Some::Module;                     # NOT found in /my/lib -␣
   ↪too early
  ```

  This is a frequent gotcha. `BEGIN { unshift @INC, '/my/lib' }` fixes it; `use lib` is the cleaner spelling of exactly that.

- **`push @INC, '/path'`** appends. The new path is searched *last*. This is the right choice when you want your library to be a fallback - used only if no other location has the module - for example, providing a built-in version of a CPAN module that the user might have installed on the system.

The rule of thumb: *always* use `use lib` for "I want my code's module path to take effect at compile time", which is almost every case.

`@INC` hooks

`@INC` may contain not only paths but also code references and blessed objects. When `require` encounters one of these, it calls it as a hook - passing the requested module path - and the hook returns either an open filehandle or a list of source code, generators, etc. This is how mechanisms like `Module::Pluggable`, `PAR`, and `Test::MockModule` intercept module loading.

The full hook protocol is documented under `require`. Most users never write one; you read about them when debugging "why does this module not load from where I expect" problems.

%INC - the loaded-modules cache

After a module is successfully loaded, %INC records it. The key is the path Perl was asked to find (e.g. Foo/Bar.pm); the value is the absolute path that satisfied the request:

```
use Data::Dumper;
print "Data::Dumper loaded from $INC{'Data/Dumper.pm'}\n";
# /usr/local/lib/perl5/site_perl/5.44.0/Data/Dumper.pm

# Show every loaded module:
for my $key (sort keys %INC) {
    print "$key  →  $INC{$key}\n";
}
```

require checks %INC before walking @INC - a second require Foo; is a no-op because Foo.pm is already in the cache. To force a re-require:

```
delete $INC{'Foo.pm'};
require Foo;                    # actually re-runs Foo.pm
```

This is the standard idiom for testing reload behaviour. Be aware that re-running a module file does **not** un-define what the first run created - packages, subs, and globals stay around.

$INC - the index inside an @INC hook

Available since Perl 5.37.7. When an @INC hook is being called, $INC is set to the index of the hook in @INC. After the hook returns, the iterator advances based on $INC + 1 - so the hook can rewrite @INC and steer the search to a specific position afterwards. This is an advanced feature; everyday code never touches it.

See also

- use, require - the consumers of @INC/%INC.
- use lib - the canonical compile-time spelling for prepending to @INC.
- Getopt::Long - the standard parser for @ARGV.
- readline - the function form of the diamond <> operator that drives @ARGV → STDIN filtering.
- -i, -n, -p, -a - command-line switches built on @ARGV.
- *Process variables* - $0 is the script name; $$ is the PID; pair these with @ARGV for self-restart and logging.
- *Command-line one-liners* - the tutorial showing every @ARGV/<>/switch combination in context.

Signal handling - %SIG

%SIG is the hash where Perl exposes signal handlers. Setting $SIG{NAME} installs a handler for signal NAME; deleting it restores the default action. Two pseudo-signals - __WARN__ and __DIE__ - are not real OS signals at all but interpreter hooks into warn and die.

The signal table

Real OS signals come from the POSIX and signal.h lists. The common ones - keys you will actually use:

Key	Signal	Default action	When you handle it
INT	SIGINT	terminate	Ctrl-C: graceful shutdown
TERM	SIGTERM	terminate	kill <pid>: graceful shutdown
HUP	SIGHUP	terminate	terminal hangup; daemons reload config
QUIT	SIGQUIT	terminate + core	Ctrl-: like INT but with a core dump
PIPE	SIGPIPE	terminate	wrote to a closed pipe; usually ignored
CHLD	SIGCHLD	ignore	a child process died (waitpid here)
USR1	SIGUSR1	terminate	application-defined event
USR2	SIGUSR2	terminate	application-defined event
ALRM	SIGALRM	terminate	alarm() timer expired
WINCH	SIGWINCH	ignore	terminal window resized
IO	SIGIO	terminate	async I/O ready

The full list - including signals you cannot catch (KILL, STOP) and platform-specific signals - comes from the POSIX module, which exports the numeric constants (SIGINT, SIGTERM, ...) and the WIFEXITED-family wait macros.

Installing a handler

Three handler-value forms are accepted:

```
$SIG{INT}  = \&handler;          # code reference - preferred
$SIG{INT}  = sub { ... };         # anonymous sub  - also fine
$SIG{INT}  = 'IGNORE';            # ignore the signal
$SIG{INT}  = 'DEFAULT';           # restore default action
$SIG{INT}  = '';                  # same as DEFAULT
$SIG{INT}  = undef;               # same as DEFAULT
```

Don't use a *bareword* ($SIG{INT} = 'handler') - that references a *name* in the main:: package, which Perl looks up each time the signal arrives. If you rename handler, the bareword form silently keeps targeting the old name.

The handler receives one argument - the signal name as a string:

```
sub handler {
    my ($signame) = @_;
    print "received SIG$signame\n";
}
```

(continues on next page)

(continued from previous page)

```
$SIG{INT}  = \&handler;
$SIG{TERM} = \&handler;
```

Safe signals - the deferred-delivery rule

Since Perl 5.8, signal delivery is **deferred**. When a signal arrives, the OS sets a flag in the interpreter; the actual Perl handler runs only at the next *safe point* - typically between op-code executions. This means:

- A handler **cannot** interrupt a `read`, `open`, `connect`, or any other syscall mid-call. Slow syscalls return early with `EINTR` in `$!` and the handler runs after.
- A handler **cannot** interrupt a regex match in progress. A long-running regex on a malicious input will not be cut short by SIGINT.
- The handler runs in regular Perl context. It can call any Perl code, allocate memory, throw exceptions - none of the re-entrancy hazards of C signal handlers.

This is the right trade-off for almost every program. If you genuinely need pre-5.8 *unsafe* signal handling - handlers that run synchronously in the signal context, with all the re-entrancy hazards - set the `PERL_SIGNALS=unsafe` environment variable before starting Perl. Do not do this casually.

PetaPerl implements deferred-delivery signal handling.

The graceful-shutdown pattern

The standard shape: a flag set by the handler, checked by the main loop:

```
my $shutting_down = 0;
$SIG{INT}  = sub { $shutting_down = 1 };
$SIG{TERM} = sub { $shutting_down = 1 };

while (my $job = $queue->next) {
    last if $shutting_down;       # check at a safe point
    process($job);
}

if ($shutting_down) {
    log_and_close();
    exit 0;
}
```

The main loop polls the flag at a place where it is safe to unwind cleanly. The handler does *only* "set the flag" - no logging, no I/O - minimising the time it spends running.

For applications that block on a long syscall (`accept`, `read`, `select`), the loop has to be structured to retry on `EINTR`:

```
my $shutting_down = 0;
$SIG{INT}  = sub { $shutting_down = 1 };
```

(continues on next page)

(continued from previous page)

```
while (!$shutting_down) {
    my $client = accept($srv, $sock);
    unless (defined $client) {
        next if $!{EINTR};        # signal woke us up - re-check␣
↪flag
        die "accept: $!";
    }
    handle($client);
}
```

`$!{EINTR}` (see *error variables*) tells you the syscall exited because a signal arrived. The flag was set by the handler; the main loop sees the wake-up, re-checks `$shutting_down`, and exits cleanly.

Reaping child processes - SIGCHLD

When a child process exits, the parent gets SIGCHLD. The handler must call `waitpid` to *reap* the zombie:

```
$SIG{CHLD} = sub {
    while ((my $kid = waitpid(-1, WNOHANG)) > 0) {
        my $status = $?;
        log_child_death($kid, $status);
    }
};
```

The `while` loop with `WNOHANG` (from `POSIX :sys_wait_h`) handles the case where two children exit close together - only one SIGCHLD is delivered (it is not queued), so the handler must drain *all* pending exits each time it runs.

`$SIG{CHLD} = 'IGNORE'` is a special case: it does not just ignore the signal, it tells the kernel to auto-reap children. Use this when your script forks children whose exit status you do not care about.

SIGPIPE - the closed-pipe case

When you `print` to a pipe whose reader has gone away, the kernel sends SIGPIPE. The default action is to terminate the process - which is rarely what you want. Most filter programs should ignore SIGPIPE and check the `print` return value instead:

```
local $SIG{PIPE} = 'IGNORE';

print {$child_pipe} $data
    or die "write to pipe: $!";  # $! will be EPIPE if the reader␣
↪closed
```

With the handler ignored, the syscall returns failure with `EPIPE` rather than killing the program.

Timeouts via SIGALRM

Combined with `alarm` and an `eval` block, SIGALRM provides a portable timeout for any blocking operation:

```
sub with_timeout {
    my ($seconds, $code) = @_;
    my $result;
    eval {
        local $SIG{ALRM} = sub { die "TIMEOUT\n" };
        alarm $seconds;
        $result = $code->();
        alarm 0;                    # cancel the timer
        1;
    } or do {
        die $@ unless $@ eq "TIMEOUT\n";
        return undef;
    };
    return $result;
}

my $line = with_timeout(5, sub { <$slow_socket> });
```

The pattern:

1. `local $SIG{ALRM}` - handler is in scope only for this call.
2. `alarm $seconds` - schedule the signal.
3. Perform the blocking work.
4. `alarm 0` - cancel the timer if work succeeded.
5. Inside the handler, `die "TIMEOUT\n"` - the trailing `\n` suppresses the "at line N" suffix and makes the error value easy to test against.

__WARN__ and __DIE__ - interpreter hooks

These two keys in `%SIG` are *not* OS signals. They are called when the interpreter is about to emit a warning (`warn`) or throw a fatal exception (`die`).

$SIG{__WARN__} - warning interception

```
my @warnings;
{
    local $SIG{__WARN__} = sub {
        my ($msg) = @_;
        push @warnings, $msg;
    };
    do_thing_that_might_warn();
}
# Now @warnings holds every warning that fired inside the block,
# without anything having printed to STDERR.
```

A `__WARN__` handler that does nothing silences the warning entirely. A handler that calls `die` upgrades warnings into exceptions:

```
local $SIG{__WARN__} = sub { die $_[0] };
eval { something_warny() };       # warns become catchable
```

The lexical pragma `use warnings FATAL => 'all'` is a more targeted way to do this - it makes warnings fatal only in the lexical scope where the pragma is active. Use it instead of `__WARN__` for "treat warnings as errors" requirements.

$SIG{__DIE__} - die interception

A `__DIE__` handler runs when `die` is about to throw - both for uncaught exceptions and inside `eval`. Inside an `eval`, the handler runs *before* the exception is caught, which is the classic source of confusion: a global `__DIE__` handler that logs every "death" will fire on every `eval { ... }` that calls `die` for ordinary control flow.

```
$SIG{__DIE__} = sub { print STDERR "dying: $_[0]" };
eval { die "expected error" };    # the handler still runs!
```

If you want a global death logger, gate it on `$^S`:

```
$SIG{__DIE__} = sub {
    return if $^S;                # inside eval - let it be caught
    print STDERR "uncaught: $_[0]";
};
```

Realistically, **avoid $SIG{__DIE__} for new code**. The pragma `use warnings` and the *error variables* cover what you need without the surprise. The modern `try/catch` feature handles exceptions explicitly without the action-at-a-distance of `$SIG{__DIE__}`.

Listing available signals

```
use Config;
my @signals = split / /, $Config{sig_name};
print "available signals: @signals\n";
# HUP INT QUIT ILL TRAP ABRT BUS FPE KILL ...
```

Or, for the numeric→name mapping:

```
use Config;
my @names = split / /, $Config{sig_name};
my @nums  = split / /, $Config{sig_num};
my %sig_num  = map { $names[$_] => $nums[$_] }  0..$#names;
my %sig_name = map { $nums[$_]  => $names[$_] } 0..$#names;
```

This is what most signal-listing utilities do internally.

See also

- POSIX - the canonical source of SIG* numeric constants and the WIFEXITED/WEXITSTATUS macros for decoding child status in a SIGCHLD handler.
- kill - the partner of %SIG: send a signal to a process (typically to test your own handler).
- alarm - schedule a SIGALRM; the basis of timeout patterns.
- waitpid - reap a child after SIGCHLD.
- fork - produces the children whose deaths SIGCHLD reports.
- *Error variables* - $! after EINTR, $? after waitpid, $@ after die from inside a handler.
- *Interpreter introspection* - $^S for the __DIE__ "are we inside eval" check.
- try/catch - the modern alternative to $SIG{__DIE__}-based exception handling.
- *Default variables* - $_, @_. The implicit operand of most built-ins; the argument array of every sub.
- *Process identity* - $0, $$, $<, $>, $(, $). Program name, PID, real and effective UIDs/GIDs.
- *Error state* - $!, $@, $?, $^E. The four error variables, each reporting from a different layer (C library, eval/die, child process, OS-extended).
- *Filehandle and printing state* - $/, $\, $,, $", $|, $., $;. Input record separator, output separators, autoflush, line counter, subscript joiner.
- *Regex match variables* - $&, $`, $', $1..$N, %+, %-, @+, @-. The result of the last successful match; the performance story behind $& and friends.
- *Interpreter introspection* - $], $^V, $^O, $^X, $^T, $^W, $^I, $^P. Version, OS, executable path, basetime, warnings, in-place edit, debugger flags.
- *Program input* - @ARGV, %ENV, @INC, %INC, $0. How the script gets its arguments, environment, and module search path; the use lib vs unshift @INC story.
- *Signal handling* - %SIG, $SIG{__WARN__}, $SIG{__DIE__}. The signals you can catch, the safe-signals rule, and the standard graceful-shutdown shape.

Variable namespaces

Each special variable lives in **one** specific namespace. Most are in main::; some (@F, @ISA) are package-local. Every variable's documentation says which.

The naming convention is a punctuation choice and a sigil:

- $<digit> - regex capture variables.
- $<symbol> - single-character variables ($_, $!, $@, $/, ...).
- $^<letter> - control-letter variables ($^V, $^W, $^X). Spelled with a literal caret followed by an uppercase letter.
- ${^NAME} - extended-name variables (${^GLOBAL_PHASE}, ${^CAPTURE}). The braces and caret are required.

- `$NAME` after `use English` - the named alias for almost every punctuation variable. `$ERRNO` is `$!`, `$EVAL_ERROR` is `$@`, etc.

The English module

```
use English;            # all aliases, plus $PREMATCH/$MATCH/
 ↪$POSTMATCH
use English qw( -no_match_vars );    # skip $`, $&, $' aliases
```

`English` provides readable aliases for the punctuation variables: `$EVAL_ERROR` for `$@`, `$INPUT_RECORD_SEPARATOR` for `$/`, and so on. The aliases are real Perl variables that share storage with the originals - assigning to one writes the other.

The `-no_match_vars` import is recommended in code that doesn't explicitly need `$&`, `$`` , or `$'`. See *regex match variables* for the historical performance reason and why it no longer matters on modern perls.

This page lists punctuation forms throughout because most code in the wild is written that way. The English aliases are noted alongside on the family pages where they are useful.

Localising special variables

Almost every special variable that controls a built-in's behaviour (`$/`, `$\`, `$,`, `$"`, `$|`, `$_`) is global - set it in one place and every other piece of code in the same process sees the change. The standard idiom for "I need a different value just for this block" is `local`:

```
sub slurp {
    my ($path) = @_;
    open my $fh, '<', $path or die "open $path: $!";
    local $/;                           # slurp mode, only inside this
 ↪sub
    return <$fh>;
}
```

Without the `local`, the caller's `$/` would be permanently clobbered. `local` saves the current value, sets the new one, and restores the old value when the enclosing scope exits - even on an exception. This applies to every special variable on the family pages, except the read-only ones (regex match variables, `$0`'s formal argument, the version variables).

See also

- `perlop` - every operator that reads or writes a special variable as part of its definition (regex binding, printing, range, comparison).
- `perlfunc` - built-in functions, almost all of which read a default from one of these variables when called with no argument (`chomp`, `chop`, `print`, `lc`, `length`, `<>`, ...).
- *Regular expressions guide* - the regex language behind the match variables.
- *Boolean Logic for Perl Programmers* - the conceptual companion for the `||//` defaulting idioms that recur throughout this reference.

5.1.2 Built-in Functions

Alphabetical index of every Perl 5 built-in. The content mirrors `perldoc perlfunc` with pperl-specific divergences flagged inline on each detail page.

For an alternate browsing view grouped by problem domain see *Built-in Functions by Category*.

A

- **`abs`** - Return the absolute value of a number.
- **`accept`** - Accept an incoming connection on a listening socket.
- **`alarm`** - Schedule a `SIGALRM` to be delivered to the current process after a whole number of wallclock seconds.
- **`all`** - Test whether `BLOCK` returns true for **every** element of `LIST`.
- **`any`** - Return true if `BLOCK` yields true for at least one element of `LIST`.
- **`atan2`** - Arc tangent of `Y/X` in the range -π to π.

B

- **`bind`** - Attach a local address to a socket.
- **`binmode`** - Set the I/O layer stack on a filehandle - typically to make it deliver raw bytes, or to attach a character encoding.
- **`bless`** - Mark the thing a reference points at as an object in a package.
- **`break`** - Break out of a `given` block.

C

- **`caller`** - Return information about the subroutine, `eval`, or `require` that called the currently executing code.
- **`catch`** - Handle an exception thrown by a preceding `try` block.
- **`chdir`** - Change the process's current working directory.
- **`chmod`** - Change the permission bits of a list of files.
- **`chomp`** - Strip the trailing input record separator from a string in place.
- **`chop`** - Remove the last character from a string and return it.
- **`chown`** - Change the owner and group of a list of files.
- **`chr`** - Return the character whose codepoint is the given number.
- **`chroot`** - Change the root directory of the current process and every child it later spawns.
- **`class`** - Declare a namespace that behaves as a native object class.
- **`close`** - Close a filehandle, flush its buffers, and release the underlying file descriptor.

- **closedir** - Close a directory handle opened by opendir.
- **connect** - Initiate a connection from a socket to a remote address.
- **continue** - Attach a block to a loop that runs after every iteration, just before the condition is re-tested.
- **cos** - Return the cosine of a number given in radians.
- **crypt** - One-way passwd(5)-style hashing of a plaintext string with a salt.

D

- **dbmclose** - Break the binding between a DBM file and a hash previously attached by dbmopen.
- **dbmopen** - Bind a DBM file on disk to a hash so hash reads and writes become lookups and stores in the database.
- **defer** - Schedule a block to run when the enclosing scope exits, for any reason.
- **defined** - Test whether a value, variable, or subroutine is defined.
- **delete** - Remove the named key-value pair(s) from a hash, or element(s) from an array, and return what was removed.
- **die** - Raise an exception.
- **do** - Execute a block of code or run a Perl source file as if it were part of the current program.
- **dump** - Cause the running program to produce a core dump immediately.

E

- **each** - Step through a hash or array one entry at a time.
- **endgrent** - Close the group database after iterating it.
- **endhostent** - Close the hosts database after iteration.
- **endnetent** - Close the networks database after iteration.
- **endprotoent** - Close the protocols database after iteration.
- **endpwent** - Close the passwd-database iterator opened by getpwent or setpwent.
- **endservent** - Close the services database after a walk with getservent.
- **eof** - Test a filehandle for end-of-file.
- **eval** - Run a piece of Perl code with any fatal error trapped instead of killing the program.
- **evalbytes** - Compile and run a string of Perl source, forcing the source to be interpreted as **bytes** rather than characters.
- **exec** - Abandon this program and run another in the same process.
- **exists** - Test whether a hash or array element, or a named subroutine, is present - without creating it and without caring what it holds.
- **exit** - Terminate the program with a status value.

- **exp** - Raise *e* to a power.

F

- **fc** - Return the Unicode casefolded form of a string for case-insensitive comparison.
- **fcntl** - Perform a `fcntl(2)` file-control operation on a filehandle.
- **field** - Declare a per-instance variable inside a `class` block.
- **fileno** - Return the OS-level file descriptor number behind a filehandle.
- **finally** - Run cleanup code on the way out of a `try`/`catch`, whether the body succeeded, threw, or jumped away.
- **flock** - Place an advisory lock on an open file.
- **fork** - Create a new process running the same program at the same point.
- **format** - Declare a picture-based report template for use by `write`.
- **formline** - Format a list of values into a picture string and append the result to the format accumulator `$^A`.

G

- **getc** - Read the next single character from a filehandle.
- **getgrent** - Read the next entry from the system group database.
- **getgrgid** - Look up a group record by numeric group ID.
- **getgrnam** - Look up a Unix group by name and return its `/etc/group` record.
- **gethostbyaddr** - Look up a host record by its packed IP address.
- **gethostbyname** - Look up a host record by DNS name.
- **gethostent** - Fetch the next entry from the host database.
- **getlogin** - Return the login name of the user associated with the controlling terminal.
- **getnetbyaddr** - Look up a network record by its numeric network address.
- **getnetbyname** - Look up a network record by name.
- **getnetent** - Read the next entry from the networks database.
- **getpeername** - Return the address of the remote end of a connected socket.
- **getpgrp** - Return the POSIX process-group ID that a process belongs to.
- **getppid** - Return the process ID of the current process's parent.
- **getpriority** - Return the current scheduling nice value of a process, a process group, or a user.
- **getprotobyname** - Look up an IP protocol by its textual name and return its database entry.
- **getprotobynumber** - Look up a network protocol entry by its assigned protocol number.

- **getprotoent** - Fetch the next entry from the protocols database.
- **getpwent** - Return the next entry from the system password database.
- **getpwnam** - Look up a user's passwd record by login name.
- **getpwuid** - Look up a user's passwd record by numeric UID.
- **getservbyname** - Look up a network service by its textual name and protocol.
- **getservbyport** - Look up a network service by its numeric port and protocol.
- **getservent** - Read the next entry from the system services database.
- **getsockname** - Return the local address of a connected or bound socket.
- **getsockopt** - Read one socket option out of the kernel as an opaque packed string.
- **glob** - Expand a shell-style filename pattern into the list of matching paths.
- **gmtime** - Convert an epoch time to a broken-down UTC time, either as a 9-element list or as a `ctime(3)`-style string.
- **goto** - Transfer execution elsewhere in the program without returning.
- **grep** - Filter a list to the elements where the block or expression is true.

H

- **hex** - Interpret a string as a hexadecimal numeral and return its numeric value.

I

- **import** - Populate the caller's namespace with names a module chooses to export.
- **index** - Find the position of a substring inside a string.
- **int** - Return the integer portion of a number, truncating toward zero.
- **ioctl** - Perform a device-control `ioctl(2)` system call on a filehandle.
- **isa** - Test whether an object is an instance of a class or of any subclass derived from it.

J

- **join** - Concatenate a list of strings with a separator between each adjacent pair, and return the single resulting string.

K

- **keys** - List all keys of a hash, or all indices of an array.
- **kill** - Send a signal to a list of processes.

L

- **last** - Exit a loop immediately, skipping the rest of the body and the `continue` block.
- **lc** - Return a lowercased copy of a string.
- **lcfirst** - Return a copy of a string with its first character lowercased.
- **length** - Return the number of characters in a string.
- **link** - Create a hard link from `NEWFILE` to the existing `OLDFILE`.
- **listen** - Mark a socket as passive so it can accept incoming connections.
- **local** - Save the current value of a package variable and restore it when the enclosing scope exits.
- **localtime** - Convert an epoch time to a broken-down calendar time in the local time zone.
- **lock** - Place an advisory lock on a shared variable, array, hash, or subroutine until the lock goes out of scope.
- **log** - Return the natural logarithm (base *e*) of a number.
- **lstat** - Return the 13-element status list for a path **without** following a symbolic link.

M

- **m//** - Search a string for a pattern and report whether - and what - it matched.
- **map** - Apply a block or expression to each element of a list and return the flattened results.
- **method** - Declare a named instance method inside a `class` block.
- **mkdir** - Create a single directory on the filesystem.
- **msgctl** - Perform a control operation on a System V IPC message queue.
- **msgget** - Create or look up a System V IPC message queue and return its id.
- **msgrcv** - Receive a message from a System V IPC message queue.
- **msgsnd** - Send a message to a System V IPC message queue.
- **my** - Declare one or more lexically scoped variables.

N

- **next** - Start the next iteration of the enclosing loop immediately.
- **no** - The compile-time inverse of `use` - call a module's `unimport` method to turn off what `use` turned on.

O

- **oct** - Interpret a string as a number written in octal, hexadecimal, or binary, and return the resulting integer.
- **open** - Associate a filehandle with a file, a command, or an in-memory scalar.
- **opendir** - Open a directory for reading.
- **ord** - Return the Unicode code point of the first character of a string.
- **our** - Declare a lexically scoped alias to a package variable.

P

- **pack** - Convert a list of Perl values into a binary string according to a template.
- **package** - Declare the compile-time namespace for the declarations that follow.
- **pipe** - Open a pair of connected filehandles - one for reading, one for writing.
- **pop** - Remove and return the last element of an array.
- **pos** - Report or set where the next `/g` regex match will resume in a string.
- **print** - Write a list of values to a filehandle.
- **printf** - Write a formatted string to a filehandle.
- **prototype** - Return the prototype string of a subroutine, or `undef` if it has none.
- **push** - Append one or more values to the end of an array.

Q

- **q//** - Single-quoted string literal with a choice of delimiter.
- **qq//** - Build a double-quoted, interpolated string with a delimiter you pick.
- **qr//** - Compile a pattern once and hand back a reusable regex object.
- **quotemeta** - Return a copy of a string with every regex-significant character backslash-escaped, so the result can be interpolated into a pattern and match its own literal content.
- **qw//** - Build a list of barewords by splitting a delimited string on whitespace, without quoting or comma-separating each word.
- **qx//** - Run a shell command and capture its standard output.

R

- **rand** - Return a pseudo-random floating-point number in the half-open range `[0, EXPR)`.
- **read** - Read a fixed amount of buffered input from a filehandle into a scalar.
- **readdir** - Read the next entry, or all remaining entries, from a directory handle opened by `opendir`.
- **readline** - Read one or more records from a filehandle.

- **readlink** - Return the target path a symbolic link points to.
- **readpipe** - Run a shell command and return its standard output.
- **recv** - Read an incoming message from a socket into a scalar.
- **redo** - Restart the current iteration of a loop without re-testing the condition and without running the `continue` block.
- **ref** - Return a string describing what a reference points to.
- **rename** - Change the name of a file.
- **require** - Load a Perl source file at runtime, or demand a minimum Perl version.
- **reset** - Clear every package variable whose name begins with one of a set of letters, and re-arm one-shot `m?pattern?` matches.
- **return** - Leave the current subroutine, `eval`, do `FILE`, `sort` block, or regex eval block, yielding a value to the caller.
- **reverse** - Reverse a list - or, in scalar context, reverse the characters of a string.
- **rewinddir** - Reset a directory handle to the start of its listing.
- **rindex** - Find the position of the **last** occurrence of a substring inside a string.
- **rmdir** - Remove an empty directory.

S

- **s///** - Search a string for a pattern and replace every match with a replacement.
- **say** - Print a list of values followed by a newline.
- **scalar** - Force `EXPR` to be evaluated in scalar context and return its value.
- **seek** - Reposition a filehandle for random-access reads or writes.
- **seekdir** - Restore a directory handle to a position previously captured by `telldir`.
- **select** - Either set the default output filehandle, or call the `select(2)` syscall for I/O multiplexing. Same name, two unrelated jobs - disambiguated by argument count.
- **semctl** - Perform a control operation on a System V semaphore set.
- **semget** - Create or look up a System V semaphore set and return its identifier.
- **semop** - Perform one or more System V semaphore operations atomically.
- **send** - Send a message on a socket.
- **setgrent** - Rewind the group-database iterator back to the first entry.
- **sethostent** - Open or rewind the hosts database for iteration.
- **setnetent** - Open or rewind the networks database for iteration.
- **setpgrp** - Set the process group of a process.
- **setpriority** - Set the scheduling priority (nice value) of a process, process group, or user.
- **setprotoent** - Open the protocols database and prepare it for sequential reads.

- **setpwent** - Rewind the password-database iterator so the next `getpwent` call returns the first entry again.
- **setservent** - Rewind the services database and optionally keep it open across lookups.
- **setsockopt** - Set a kernel-level option on an open socket.
- **shift** - Remove and return the first element of an array.
- **shmctl** - Control or query a System V shared memory segment.
- **shmget** - Create or look up a System V shared memory segment and return its identifier.
- **shmread** - Copy bytes out of a System V shared memory segment into a Perl scalar.
- **shmwrite** - Copy bytes into a System V shared memory segment.
- **shutdown** - Shut down one direction of a socket connection, or both.
- **sin** - Return the sine of a number given in radians.
- **sleep** - Pause the process for a number of whole seconds.
- **socket** - Create a socket filehandle.
- **socketpair** - Create an unnamed, connected pair of sockets that talk to each other.
- **sort** - Sort a list, returning the sorted list.
- **splice** - Remove and/or replace a slice of an array in place, and return the removed elements.
- **split** - Cut a string into a list of fields using a regex separator.
- **sprintf** - Build a formatted string from a format template and a list of values.
- **sqrt** - Return the non-negative square root of `EXPR`.
- **srand** - Seed the pseudo-random number generator.
- **stat** - Get a file's status information.
- **state** - Declare a lexically scoped variable whose value persists across calls to its enclosing subroutine.
- **study** - A no-op retained for source compatibility with older Perl code.
- **sub** - Declare or define a subroutine.
- **substr** - Extract, replace, or alias a contiguous slice of a string.
- **symlink** - Create a symbolic link at `NEWFILE` that points at the string `OLDFILE`.
- **syscall** - Invoke a raw system call by its kernel number, passing the remaining arguments as `int` or as a pointer to a string buffer.
- **sysopen** - Open a file the low-level way, passing an integer `MODE` bitmask straight through to the underlying `open(2)` system call.
- **sysread** - Read raw bytes from a filehandle by calling the underlying `read(2)` system call.
- **sysseek** - Reposition a filehandle at the system level, bypassing PerlIO buffering.

- **system** - Run a separate program and wait for it to finish.
- **syswrite** - Write bytes to a filehandle with the raw `write(2)` system call, bypassing Perl's buffered I/O.

T

- **tell** - Return the current byte position of a filehandle.
- **telldir** - Return the current read position of a directory handle as an opaque token.
- **tie** - Bind a variable to a class so that every access to the variable dispatches through that class's methods.
- **tied** - Return the object backing a tied variable.
- **time** - Return the current wall-clock time as an integer number of seconds since the system epoch.
- **times** - Report CPU time consumed by this process and its terminated children.
- **tr///** - Character-by-character substitution. `tr` scans a string and replaces every occurrence of a character from `SEARCHLIST` with the positionally corresponding character from `REPLACEMENTLIST`, returning the count of characters it touched.
- **truncate** - Shorten (or extend) a file to an exact byte length.
- **try** - Run a block and divert any exception it throws into a `catch` block, with an optional `finally` block that always runs on the way out.

U

- **uc** - Return an uppercased copy of a string.
- **ucfirst** - Return a copy of a string with its first character titlecased.
- **umask** - Set or read the process file-creation mode mask.
- **undef** - The undefined value, and the operator that produces it.
- **unlink** - Remove one or more directory entries that name files.
- **unpack** - Extract typed values from a binary or fixed-width string according to a template.
- **unshift** - Prepend one or more values to the start of an array and return the array's new length.
- **untie** - Break the binding between a variable and its tied class.
- **use** - Load a module at compile time and import its symbols into the current package.
- **utime** - Set access and modification times on a list of files.

V

- **values** - List every value of a hash (or every element of an array).
- **vec** - Read or write a fixed-width slot inside a string treated as a packed bit vector.

W

- **wait** - Block until any child process exits and reap it.
- **waitpid** - Wait for a specific child process to terminate and reap it.
- **wantarray** - Report the calling context of the currently executing subroutine.
- **warn** - Emit a warning to STDERR.
- **write** - Render one record to a filehandle through its associated `format`.

Y

- **y///** - Transliterate characters in a string. `y///` is a synonym for `tr///` - the two are identical in every respect.

__ (double-underscored)

- **__CLASS__** - Return the class name of the instance currently being acted on.
- **__FILE__** - The name of the source file the token is compiled in.
- **__LINE__** - A compile-time token that evaluates to the line number at which it appears in the source.
- **__PACKAGE__** - Return the name of the package currently in effect at the point where the token appears.
- **__SUB__** - Return a reference to the currently executing subroutine.

Built-in Functions by Category

The same built-ins as in *the alphabetical index*, grouped here by problem domain. A function may appear under more than one category: categories are a tag set, not a tree. Every link points at the same detail page regardless of which category the reader arrived through.

SCALARs and strings

- **chomp** - Strip the trailing input record separator from a string in place.
- **chop** - Remove the last character from a string and return it.
- **chr** - Return the character whose codepoint is the given number.
- **crypt** - One-way `passwd(5)`-style hashing of a plaintext string with a salt.
- **fc** - Return the Unicode casefolded form of a string for case-insensitive comparison.
- **hex** - Interpret a string as a hexadecimal numeral and return its numeric value.
- **index** - Find the position of a substring inside a string.

- **lc** - Return a lowercased copy of a string.
- **lcfirst** - Return a copy of a string with its first character lowercased.
- **length** - Return the number of characters in a string.
- **oct** - Interpret a string as a number written in octal, hexadecimal, or binary, and return the resulting integer.
- **ord** - Return the Unicode code point of the first character of a string.
- **pack** - Convert a list of Perl values into a binary string according to a template.
- **q//** - Single-quoted string literal with a choice of delimiter.
- **qq//** - Build a double-quoted, interpolated string with a delimiter you pick.
- **reverse** - Reverse a list - or, in scalar context, reverse the characters of a string.
- **rindex** - Find the position of the **last** occurrence of a substring inside a string.
- **sprintf** - Build a formatted string from a format template and a list of values.
- **substr** - Extract, replace, or alias a contiguous slice of a string.
- **tr///** - Character-by-character substitution. `tr` scans a string and replaces every occurrence of a character from `SEARCHLIST` with the positionally corresponding character from `REPLACEMENTLIST`, returning the count of characters it touched.
- **uc** - Return an uppercased copy of a string.
- **ucfirst** - Return a copy of a string with its first character titlecased.
- **y///** - Transliterate characters in a string. `y///` is a synonym for `tr///` - the two are identical in every respect.

Regular expressions and pattern matching

- **m//** - Search a string for a pattern and report whether - and what - it matched.
- **pos** - Report or set where the next `/g` regex match will resume in a string.
- **qr//** - Compile a pattern once and hand back a reusable regex object.
- **quotemeta** - Return a copy of a string with every regex-significant character backslash-escaped, so the result can be interpolated into a pattern and match its own literal content.
- **s///** - Search a string for a pattern and replace every match with a replacement.
- **split** - Cut a string into a list of fields using a regex separator.
- **study** - A no-op retained for source compatibility with older Perl code.

Numeric functions

- **abs** - Return the absolute value of a number.
- **atan2** - Arc tangent of `Y/X` in the range -π to π.
- **cos** - Return the cosine of a number given in radians.
- **exp** - Raise *e* to a power.

- **hex** - Interpret a string as a hexadecimal numeral and return its numeric value.
- **int** - Return the integer portion of a number, truncating toward zero.
- **log** - Return the natural logarithm (base *e*) of a number.
- **oct** - Interpret a string as a number written in octal, hexadecimal, or binary, and return the resulting integer.
- **rand** - Return a pseudo-random floating-point number in the half-open range `[0, EXPR)`.
- **sin** - Return the sine of a number given in radians.
- **sqrt** - Return the non-negative square root of `EXPR`.
- **srand** - Seed the pseudo-random number generator.

Arrays

- **each** - Step through a hash or array one entry at a time.
- **keys** - List all keys of a hash, or all indices of an array.
- **pop** - Remove and return the last element of an array.
- **push** - Append one or more values to the end of an array.
- **shift** - Remove and return the first element of an array.
- **splice** - Remove and/or replace a slice of an array in place, and return the removed elements.
- **unshift** - Prepend one or more values to the start of an array and return the array's new length.
- **values** - List every value of a hash (or every element of an array).

Lists

- **all** - Test whether `BLOCK` returns true for **every** element of `LIST`.
- **any** - Return true if `BLOCK` yields true for at least one element of `LIST`.
- **grep** - Filter a list to the elements where the block or expression is true.
- **join** - Concatenate a list of strings with a separator between each adjacent pair, and return the single resulting string.
- **map** - Apply a block or expression to each element of a list and return the flattened results.
- **qw//** - Build a list of barewords by splitting a delimited string on whitespace, without quoting or comma-separating each word.
- **reverse** - Reverse a list - or, in scalar context, reverse the characters of a string.
- **sort** - Sort a list, returning the sorted list.
- **unpack** - Extract typed values from a binary or fixed-width string according to a template.

Hashes

- **delete** - Remove the named key-value pair(s) from a hash, or element(s) from an array, and return what was removed.
- **each** - Step through a hash or array one entry at a time.
- **exists** - Test whether a hash or array element, or a named subroutine, is present - without creating it and without caring what it holds.
- **keys** - List all keys of a hash, or all indices of an array.
- **values** - List every value of a hash (or every element of an array).

I/O

- **binmode** - Set the I/O layer stack on a filehandle - typically to make it deliver raw bytes, or to attach a character encoding.
- **close** - Close a filehandle, flush its buffers, and release the underlying file descriptor.
- **closedir** - Close a directory handle opened by `opendir`.
- **dbmclose** - Break the binding between a DBM file and a hash previously attached by `dbmopen`.
- **dbmopen** - Bind a DBM file on disk to a hash so hash reads and writes become lookups and stores in the database.
- **die** - Raise an exception.
- **eof** - Test a filehandle for end-of-file.
- **fileno** - Return the OS-level file descriptor number behind a filehandle.
- **flock** - Place an advisory lock on an open file.
- **format** - Declare a picture-based report template for use by `write`.
- **getc** - Read the next single character from a filehandle.
- **print** - Write a list of values to a filehandle.
- **printf** - Write a formatted string to a filehandle.
- **read** - Read a fixed amount of buffered input from a filehandle into a scalar.
- **readdir** - Read the next entry, or all remaining entries, from a directory handle opened by `opendir`.
- **readline** - Read one or more records from a filehandle.
- **rewinddir** - Reset a directory handle to the start of its listing.
- **say** - Print a list of values followed by a newline.
- **seek** - Reposition a filehandle for random-access reads or writes.
- **seekdir** - Restore a directory handle to a position previously captured by `telldir`.

- **select** - Either set the default output filehandle, or call the `select(2)` syscall for I/O multiplexing. Same name, two unrelated jobs - disambiguated by argument count.
- **syscall** - Invoke a raw system call by its kernel number, passing the remaining arguments as `int` or as a pointer to a string buffer.
- **sysread** - Read raw bytes from a filehandle by calling the underlying `read(2)` system call.
- **sysseek** - Reposition a filehandle at the system level, bypassing PerlIO buffering.
- **syswrite** - Write bytes to a filehandle with the raw `write(2)` system call, bypassing Perl's buffered I/O.
- **tell** - Return the current byte position of a filehandle.
- **telldir** - Return the current read position of a directory handle as an opaque token.
- **truncate** - Shorten (or extend) a file to an exact byte length.
- **warn** - Emit a warning to `STDERR`.
- **write** - Render one record to a filehandle through its associated `format`.

Fixed-length data

- **pack** - Convert a list of Perl values into a binary string according to a template.
- **read** - Read a fixed amount of buffered input from a filehandle into a scalar.
- **syscall** - Invoke a raw system call by its kernel number, passing the remaining arguments as `int` or as a pointer to a string buffer.
- **sysread** - Read raw bytes from a filehandle by calling the underlying `read(2)` system call.
- **sysseek** - Reposition a filehandle at the system level, bypassing PerlIO buffering.
- **syswrite** - Write bytes to a filehandle with the raw `write(2)` system call, bypassing Perl's buffered I/O.
- **unpack** - Extract typed values from a binary or fixed-width string according to a template.
- **vec** - Read or write a fixed-width slot inside a string treated as a packed bit vector.

Filehandles, files, directories

- **chdir** - Change the process's current working directory.
- **chmod** - Change the permission bits of a list of files.
- **chown** - Change the owner and group of a list of files.
- **chroot** - Change the root directory of the current process and every child it later spawns.
- **fcntl** - Perform a `fcntl(2)` file-control operation on a filehandle.

- **glob** - Expand a shell-style filename pattern into the list of matching paths.
- **ioctl** - Perform a device-control `ioctl(2)` system call on a filehandle.
- **link** - Create a hard link from `NEWFILE` to the existing `OLDFILE`.
- **lstat** - Return the 13-element status list for a path **without** following a symbolic link.
- **mkdir** - Create a single directory on the filesystem.
- **open** - Associate a filehandle with a file, a command, or an in-memory scalar.
- **opendir** - Open a directory for reading.
- **readlink** - Return the target path a symbolic link points to.
- **rename** - Change the name of a file.
- **rmdir** - Remove an empty directory.
- **select** - Either set the default output filehandle, or call the `select(2)` syscall for I/O multiplexing. Same name, two unrelated jobs - disambiguated by argument count.
- **stat** - Get a file's status information.
- **symlink** - Create a symbolic link at `NEWFILE` that points at the string `OLDFILE`.
- **sysopen** - Open a file the low-level way, passing an integer `MODE` bitmask straight through to the underlying `open(2)` system call.
- **umask** - Set or read the process file-creation mode mask.
- **unlink** - Remove one or more directory entries that name files.
- **utime** - Set access and modification times on a list of files.

Control flow

- **__FILE__** - The name of the source file the token is compiled in.
- **__LINE__** - A compile-time token that evaluates to the line number at which it appears in the source.
- **__PACKAGE__** - Return the name of the package currently in effect at the point where the token appears.
- **__SUB__** - Return a reference to the currently executing subroutine.
- **break** - Break out of a `given` block.
- **caller** - Return information about the subroutine, `eval`, or `require` that called the currently executing code.
- **catch** - Handle an exception thrown by a preceding `try` block.
- **continue** - Attach a block to a loop that runs after every iteration, just before the condition is re-tested.
- **defer** - Schedule a block to run when the enclosing scope exits, for any reason.
- **die** - Raise an exception.

- **do** - Execute a block of code or run a Perl source file as if it were part of the current program.
- **dump** - Cause the running program to produce a core dump immediately.
- **eval** - Run a piece of Perl code with any fatal error trapped instead of killing the program.
- **evalbytes** - Compile and run a string of Perl source, forcing the source to be interpreted as **bytes** rather than characters.
- **exit** - Terminate the program with a status value.
- **finally** - Run cleanup code on the way out of a `try`/`catch`, whether the body succeeded, threw, or jumped away.
- **goto** - Transfer execution elsewhere in the program without returning.
- **last** - Exit a loop immediately, skipping the rest of the body and the `continue` block.
- **method** - Declare a named instance method inside a `class` block.
- **next** - Start the next iteration of the enclosing loop immediately.
- **redo** - Restart the current iteration of a loop without re-testing the condition and without running the `continue` block.
- **return** - Leave the current subroutine, `eval`, `do FILE`, `sort` block, or regex eval block, yielding a value to the caller.
- **sub** - Declare or define a subroutine.
- **try** - Run a block and divert any exception it throws into a `catch` block, with an optional `finally` block that always runs on the way out.
- **wantarray** - Report the calling context of the currently executing subroutine.

Scoping

- **caller** - Return information about the subroutine, `eval`, or `require` that called the currently executing code.
- **class** - Declare a namespace that behaves as a native object class.
- **field** - Declare a per-instance variable inside a `class` block.
- **import** - Populate the caller's namespace with names a module chooses to export.
- **local** - Save the current value of a package variable and restore it when the enclosing scope exits.
- **my** - Declare one or more lexically scoped variables.
- **our** - Declare a lexically scoped alias to a package variable.
- **package** - Declare the compile-time namespace for the declarations that follow.
- **state** - Declare a lexically scoped variable whose value persists across calls to its enclosing subroutine.
- **use** - Load a module at compile time and import its symbols into the current package.

Misc

- **defined** - Test whether a value, variable, or subroutine is defined.
- **formline** - Format a list of values into a picture string and append the result to the format accumulator `$^A`.
- **lock** - Place an advisory lock on a shared variable, array, hash, or subroutine until the lock goes out of scope.
- **prototype** - Return the prototype string of a subroutine, or `undef` if it has none.
- **reset** - Clear every package variable whose name begins with one of a set of letters, and re-arm one-shot `m?pattern?` matches.
- **scalar** - Force `EXPR` to be evaluated in scalar context and return its value.
- **undef** - The undefined value, and the operator that produces it.

Processes

- **alarm** - Schedule a `SIGALRM` to be delivered to the current process after a whole number of wallclock seconds.
- **exec** - Abandon this program and run another in the same process.
- **fork** - Create a new process running the same program at the same point.
- **getpgrp** - Return the POSIX process-group ID that a process belongs to.
- **getppid** - Return the process ID of the current process's parent.
- **getpriority** - Return the current scheduling nice value of a process, a process group, or a user.
- **kill** - Send a signal to a list of processes.
- **pipe** - Open a pair of connected filehandles - one for reading, one for writing.
- **qx//** - Run a shell command and capture its standard output.
- **readpipe** - Run a shell command and return its standard output.
- **setpgrp** - Set the process group of a process.
- **setpriority** - Set the scheduling priority (nice value) of a process, process group, or user.
- **sleep** - Pause the process for a number of whole seconds.
- **system** - Run a separate program and wait for it to finish.
- **times** - Report CPU time consumed by this process and its terminated children.
- **wait** - Block until any child process exits and reap it.
- **waitpid** - Wait for a specific child process to terminate and reap it.

Modules

- **do** - Execute a block of code or run a Perl source file as if it were part of the current program.
- **import** - Populate the caller's namespace with names a module chooses to export.
- **no** - The compile-time inverse of `use` - call a module's `unimport` method to turn off what `use` turned on.
- **package** - Declare the compile-time namespace for the declarations that follow.
- **require** - Load a Perl source file at runtime, or demand a minimum Perl version.
- **use** - Load a module at compile time and import its symbols into the current package.

Classes and OO

- **__CLASS__** - Return the class name of the instance currently being acted on.
- **bless** - Mark the thing a reference points at as an object in a package.
- **class** - Declare a namespace that behaves as a native object class.
- **dbmclose** - Break the binding between a DBM file and a hash previously attached by `dbmopen`.
- **dbmopen** - Bind a DBM file on disk to a hash so hash reads and writes become lookups and stores in the database.
- **field** - Declare a per-instance variable inside a `class` block.
- **isa** - Test whether an object is an instance of a class or of any subclass derived from it.
- **method** - Declare a named instance method inside a `class` block.
- **package** - Declare the compile-time namespace for the declarations that follow.
- **ref** - Return a string describing what a reference points to.
- **tie** - Bind a variable to a class so that every access to the variable dispatches through that class's methods.
- **tied** - Return the object backing a tied variable.
- **untie** - Break the binding between a variable and its tied class.
- **use** - Load a module at compile time and import its symbols into the current package.

Sockets

- **accept** - Accept an incoming connection on a listening socket.
- **bind** - Attach a local address to a socket.
- **connect** - Initiate a connection from a socket to a remote address.
- **getpeername** - Return the address of the remote end of a connected socket.

- **getsockname** - Return the local address of a connected or bound socket.
- **getsockopt** - Read one socket option out of the kernel as an opaque packed string.
- **listen** - Mark a socket as passive so it can accept incoming connections.
- **recv** - Read an incoming message from a socket into a scalar.
- **send** - Send a message on a socket.
- **setsockopt** - Set a kernel-level option on an open socket.
- **shutdown** - Shut down one direction of a socket connection, or both.
- **socket** - Create a socket filehandle.
- **socketpair** - Create an unnamed, connected pair of sockets that talk to each other.

SysV IPC

- **msgctl** - Perform a control operation on a System V IPC message queue.
- **msgget** - Create or look up a System V IPC message queue and return its id.
- **msgrcv** - Receive a message from a System V IPC message queue.
- **msgsnd** - Send a message to a System V IPC message queue.
- **semctl** - Perform a control operation on a System V semaphore set.
- **semget** - Create or look up a System V semaphore set and return its identifier.
- **semop** - Perform one or more System V semaphore operations atomically.
- **shmctl** - Control or query a System V shared memory segment.
- **shmget** - Create or look up a System V shared memory segment and return its identifier.
- **shmread** - Copy bytes out of a System V shared memory segment into a Perl scalar.
- **shmwrite** - Copy bytes into a System V shared memory segment.

User and group info

- **endgrent** - Close the group database after iterating it.
- **endpwent** - Close the passwd-database iterator opened by `getpwent` or `setpwent`.
- **getgrent** - Read the next entry from the system group database.
- **getgrgid** - Look up a group record by numeric group ID.
- **getgrnam** - Look up a Unix group by name and return its `/etc/group` record.
- **getlogin** - Return the login name of the user associated with the controlling terminal.
- **getpwent** - Return the next entry from the system password database.
- **getpwnam** - Look up a user's passwd record by login name.
- **getpwuid** - Look up a user's passwd record by numeric UID.

- **setgrent** - Rewind the group-database iterator back to the first entry.
- **setpwent** - Rewind the password-database iterator so the next `getpwent` call returns the first entry again.

Network info

- **endhostent** - Close the hosts database after iteration.
- **endnetent** - Close the networks database after iteration.
- **endprotoent** - Close the protocols database after iteration.
- **endservent** - Close the services database after a walk with `getservent`.
- **gethostbyaddr** - Look up a host record by its packed IP address.
- **gethostbyname** - Look up a host record by DNS name.
- **gethostent** - Fetch the next entry from the host database.
- **getnetbyaddr** - Look up a network record by its numeric network address.
- **getnetbyname** - Look up a network record by name.
- **getnetent** - Read the next entry from the networks database.
- **getprotobyname** - Look up an IP protocol by its textual name and return its database entry.
- **getprotobynumber** - Look up a network protocol entry by its assigned protocol number.
- **getprotoent** - Fetch the next entry from the protocols database.
- **getservbyname** - Look up a network service by its textual name and protocol.
- **getservbyport** - Look up a network service by its numeric port and protocol.
- **getservent** - Read the next entry from the system services database.
- **sethostent** - Open or rewind the hosts database for iteration.
- **setnetent** - Open or rewind the networks database for iteration.
- **setprotoent** - Open the protocols database and prepare it for sequential reads.
- **setservent** - Rewind the services database and optionally keep it open across lookups.

Time

- **gmtime** - Convert an epoch time to a broken-down UTC time, either as a 9-element list or as a `ctime(3)`-style string.
- **localtime** - Convert an epoch time to a broken-down calendar time in the local time zone.
- **time** - Return the current wall-clock time as an integer number of seconds since the system epoch.
- **times** - Report CPU time consumed by this process and its terminated children.

Classes and OO

CLASS

Return the class name of the instance currently being acted on.

`__CLASS__` is a compile-time-recognised token that evaluates at runtime to the class name of the invoking instance. Inside a `method` body or a `field` initializer expression (introduced by `use feature 'class'`), it gives the **actual** class of the object under construction or in use - which may be a subclass of the package the code textually lives in. Outside of class-feature code it is essentially equivalent to `ref($self)`, but unlike `ref` it is usable in a field initializer, where `$self` is not yet bound.

Synopsis

```
use feature 'class';

field $x = __CLASS__->DEFAULT_X;        # in a field initializer
method m { return __CLASS__; }          # in a method body
```

What you get back

A plain string: the name of the class of the invoking instance. The value is the same kind of string `ref` returns for a blessed reference. It is **not** a reference; call methods on it as you would on any class name.

```
my $name = __CLASS__;                   # e.g. "DifferentCustomField
 ↪"
my $obj  = __CLASS__->new(...);         # class-method dispatch
```

When it differs from __PACKAGE__

`__PACKAGE__` is resolved at compile time and names the package the token textually appears in. `__CLASS__` is resolved at runtime and names the class of the *instance* - which, for a method or field initializer inherited by a subclass, is the subclass.

```
use feature 'class';

class Base {
    field $f = __CLASS__->default_f;    # evaluated per instance
    sub default_f { 10 }
    method class_name { __CLASS__ }
    method pkg_name  { __PACKAGE__ }
}

class Derived :isa(Base) {
    sub default_f { 20 }
}
```

(continues on next page)

(continued from previous page)

```
my $obj = Derived->new;
$obj->class_name;                       # "Derived"
$obj->pkg_name;                         # "Base"
# $f was initialised to 20, not 10, because __CLASS__ dispatched
# through the Derived package.
```

In a non-inherited context (no subclassing involved), `__CLASS__` and `__PACKAGE__` yield the same string.

Examples

Class method invoked from a field initializer - the canonical use case. Without `__CLASS__`, a subclass cannot override the default:

```
use feature 'class';

class WithCustomField {
    use constant DEFAULT_X => 10;
    field $x = __CLASS__->DEFAULT_X;
}

class DifferentCustomField :isa(WithCustomField) {
    sub DEFAULT_X { rand > 0.5 ? 20 : 30 }
}

my $obj = DifferentCustomField->new;    # $x is 20 or 30, not 10
```

Inside a method body, equivalent to `ref $self`:

```
class Logger {
    method log ($msg) {
        print STDERR "[", __CLASS__, "] $msg\n";
    }
}
```

Constructing a fresh instance of the same class as the invoking object - useful for clone-like factory methods that must respect subclasses:

```
class Node {
    field $label :param;
    method spawn ($child_label) {
        return __CLASS__->new(label => $child_label);
    }
}
```

Comparison with `__PACKAGE__` for diagnostics that want to pin the exact dispatch target:

```
class Thing {
    method describe {
```

(continues on next page)

(continued from previous page)

```
        return sprintf "defined in %s, dispatched as %s",
            __PACKAGE__, __CLASS__;
    }
}
```

Edge cases

- **Only valid under `use feature 'class'`** inside a `method` body, a `field` initializer, or another class-feature construct (for example `ADJUST` blocks). Using it outside such a context is a compile-time error.
- **No parentheses, no arguments.** `__CLASS__` is a bareword token, not a function call. `__CLASS__()` is a syntax error.
- **Not the same as `ref $self`** during field initialization - `$self` is not yet bound while fields are being set, so `ref $self` is unavailable. `__CLASS__` is the only way to reach the instance's class from inside a field initializer.
- **Not `ref` for a class method call.** When called as a class method (no instance in hand), `__CLASS__` yields the class the method was invoked on, not the package the method was defined in. This mirrors normal method dispatch.
- **String, not reference.** Method calls via `__CLASS__->...` are class-method dispatches, not instance-method dispatches. The method will not see a blessed reference as its invocant.
- **The `class` feature is marked experimental** in Perl 5.44; it emits a `experimental::class` warning unless silenced with `no warnings 'experimental::class'`.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `__PACKAGE__` - compile-time package name of the surrounding code; use this when you want the textual package, not the runtime class
- `ref` - runtime class name of a blessed reference; use this when you already hold an instance and are not inside a field initializer
- `bless` - associate a reference with a class name; the low-level primitive that `__CLASS__`-style dispatch ultimately resolves against
- `isa` - test whether the current class is a subclass of a given name; pairs naturally with `__CLASS__` for runtime checks
- `sub` - declares subroutines, including the `method` form that makes `__CLASS__` meaningful

Control flow

FILE

The name of the source file the token is compiled in.

`__FILE__` is a compile-time token, not a runtime function. At the point the compiler reads it, it is replaced with a string literal holding the filename the compiler is currently reading from. It takes no arguments, is never called with parens, and has no runtime cost - by the time your program runs, `__FILE__` has already become a plain string constant.

Its one job is to let code identify the file it lives in, for error messages, log lines, test diagnostics, and similar reporting.

Synopsis

```
__FILE__
```

What you get back

A string: the filename as the compiler knew it when it read this token. For a script invoked as `perl /tmp/run.pl` the value is `/tmp/run.pl`; for a module loaded through `require` or `use` it is the path `require` resolved against `@INC` (typically an absolute path). For code fed on standard input or through `-e` it is `-` or `-e` respectively.

Because substitution happens at compile time, the value is fixed for each textual occurrence of the token. Two `__FILE__` tokens in different files produce different strings; two in the same file produce the same string.

Global state it touches

None at runtime. At compile time the value is taken from the compiler's current filename, which is the same value `caller` reports in list context as element `[1]` and which `die` / `warn` use to build their default `"at FILE line LINE."` suffix.

That filename can be **rewritten** by a `#line` directive embedded in the source - see *Edge cases* below. Once rewritten, `__FILE__` tokens compiled after the directive report the new name.

Examples

Identify the running script in a log line:

```
warn "started from ", __FILE__, "\n";
# started from /tmp/run.pl
```

Attach source location to a custom error without `die`'s default suffix:

```
sub fail {
    my ($msg) = @_;
    die sprintf("%s at %s line %d\n", $msg, __FILE__, __LINE__);
}
```

Build a path relative to the current module, so resource files ship alongside the code that loads them:

```
use File::Basename qw(dirname);
use File::Spec;
my $data = File::Spec->catfile(dirname(__FILE__), "data.txt");
```

Use `__FILE__` inside a string passed to `eval` to keep error reports pointing at the original source rather than at the anonymous `(eval N)` buffer:

```
my $code = qq[\n#line ] . __LINE__ . qq[ "] . __FILE__ . qq["\n]
         . q[die "boom"];
eval $code;
# boom at <current file> line <current line>.
```

Distinguish “run as a script” from “loaded as a module” - the common test is whether the file was invoked directly:

```
__PACKAGE__->run(@ARGV) if __FILE__ eq $0;
```

Edge cases

- **Compile-time, not runtime.** `__FILE__` is a token the parser replaces with a string literal. It is not a function you can take a reference to; `\&__FILE__` is a syntax error. Stringify it, store it, or pass it - but you cannot defer its evaluation.
- **Stable per occurrence.** Calling the same subroutine from two different files does not change the `__FILE__` the subroutine body sees; that value was baked in when the subroutine was compiled. If you need the caller’s filename, use `caller`.
- **`#line` directives rewrite the value.** A comment of the form `# line N "NAME"` at column zero resets both the line number and (optionally) the filename the compiler reports. Every `__FILE__` token compiled after such a directive picks up the new name. This is the mechanism code generators and template engines use to make errors point at the original `.tt` / `.pod` / `.xs` source instead of the generated `.pl`.

  ```
  # line 1 "config.tt"
  my $x = __FILE__;       # "config.tt"
  ```

  Full syntax of the directive is specified in `perlsyn` under *Plain Old Comments (Not!)*.
- **Modules.** Inside a module, `__FILE__` is the path `require` used to load the file - usually absolute and under an `@INC` root. Do not parse it as `Some/Module.pm`; use `dirname(__FILE__)` if you need the directory, not string surgery.
- **Special command-line sources.** Code from `-e` reports `-e`; code from `-E` reports `-e` as well; code piped on stdin reports `-`. String `eval` without a `#line` directive reports `(eval N)` where `N` is the eval counter.
- **One-argument `open` and its `__FILE__` lookalike.** There is no `__FILE__` handle; the token has nothing to do with filehandles. `open(__FILE__)` does not open the current source file - it tries to open a file literally named after the token’s expanded string, which is rarely what you want.

- **Not modifiable at runtime.** Assigning to `__FILE__` is a syntax error. The mechanism for changing what `__FILE__` reports is the `#line` directive at compile time; there is no runtime equivalent.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `__LINE__` - companion compile-time token for the current line number; the two are almost always used together
- `__PACKAGE__` - compile-time token for the current package name, same family of special tokens
- `caller` - runtime lookup of the caller's filename, line, and package; use this when you need the caller's source location rather than your own
- `die` - appends `at FILE line LINE.` automatically; reach for `__FILE__` explicitly only when you suppress that suffix with a trailing newline
- `perlsyn` - the *Plain Old Comments (Not!)* section specifies the `#line` directive that rewrites `__FILE__`

Control flow

LINE

A compile-time token that evaluates to the line number at which it appears in the source.

`__LINE__` is not a function. It is a special token the parser replaces, at compile time, with an integer literal - the line number of the file position where the token sits. It is the standard way to stamp a diagnostic, a log line, or a debug print with the exact location that produced it, without paying for a `caller` lookup at runtime.

Synopsis

```
__LINE__
```

What you get back

An integer: the 1-based line number of the token within the current compilation unit. The value is a plain number, usable anywhere a number is - in arithmetic, in string interpolation, as a hash key, as an argument to `die` or `warn`.

```
print __LINE__, "\n";        # prints the line number of this line
my $where = "line " . __LINE__;
```

Because the substitution happens at parse time, every occurrence of `__LINE__` is effectively a different constant. Two copies on two different lines produce two different numbers; a single `__LINE__` inside a loop produces the same number on every iteration.

Compile-time, not run-time

The value is frozen when the source is compiled. A common confusion:

```
sub where { return __LINE__ }     # returns the line where `__LINE__
↪`
                                  # was written, i.e. the `sub` body
↪'s
                                  # line - NOT the caller's line
```

To capture the *caller's* line number, use `caller`:

```
sub where { my (undef, undef, $line) = caller; return $line }
```

Inside `eval` of a string, `__LINE__` numbers the lines of the string being compiled (starting at 1 for the first line of the string), not the lines of the surrounding file.

Altering the reported line number

The line number `__LINE__` reports is the line number the lexer believes it is on. A `#line` directive resets that belief:

```
#line 1000 "virtual.pl"
print __LINE__, "\n";        # prints 1001
```

`#line NNN` on its own sets only the line number; `#line NNN "NAME"` also changes what `__FILE__` reports. Code generators, template engines, and preprocessors use this to make diagnostics point back at the original source rather than the generated file. See *Plain Old Comments (Not!)* in the perlsyn section of the reference for the precise syntax and scope rules.

Examples

Stamp a diagnostic with its origin:

```
die "bad input at ", __FILE__, " line ", __LINE__, "\n"
    unless defined $x;
```

`die` with a message that does **not** end in a newline already appends `" at FILE line LINE.\n"`. Use the explicit form above only when you want to control the formatting, or when you've suppressed the automatic suffix with a trailing newline.

Log with file-and-line prefix:

```
sub logf {
    my ($file, $line) = (__FILE__, __LINE__);
    warn "[$file:$line] @_\n";
}
```

This logs the line where `__LINE__` was *written* - inside `logf` - which is rarely what you want. The correct pattern uses `caller`:

```
sub logf {
    my (undef, $file, $line) = caller;
    warn "[$file:$line] @_\n";
}
```

Use `__LINE__` directly when you are marking *this* piece of code, not the caller:

```
print STDERR "checkpoint at line ", __LINE__, "\n";
```

Relocate diagnostics with `#line`:

```
#line 42 "user-input.pl"
die "syntax error\n";        # dies reporting user-input.pl line 42
```

Edge cases

- **Not a function, not callable.** `__LINE__()` is a syntax error. The token takes no arguments and has no parentheses form.
- **Not interpolated inside double-quoted strings.** `"see line __LINE__"` prints the literal string `see line __LINE__`. Concatenate instead: `"see line " . __LINE__`.
- **Not a valid hash-key bareword in the usual short form.** Inside `{ ... }` hash-key context, `$h{__LINE__}` works because the token is parsed as an expression, but `$h{ __LINE__ }` also works and is clearer. `$h{__LINE__ => 1}` parses `__LINE__` as a fat-comma bareword (i.e. the string `"__LINE__"`), which is almost never what you want - write `(__LINE__, 1)` or `(__LINE__() => 1)` if the literal value is wanted.
- **Inside string `eval`**, `__LINE__` counts from line 1 of the evaluated string. Prepend your own `#line` directive to the string to make diagnostics point at the enclosing file:

  ```
  my $code = qq{#line @{[__LINE__+1]} "@{[__FILE__]}"\n} . $user_
  ↪code;
  eval $code;
  ```

- **Inside a here-doc body**, `__LINE__` evaluates to the line of the *opening* `<<` on the line where the here-doc is referenced, not lines inside the body - here-doc bodies are data, not code.
- **Heredoc and line directives.** `#line` inside a here-doc body has no effect; it is just text. `#line` only works as a beginning-of-line directive in code context.
- **Negative or zero line numbers.** `#line 0` is legal and makes the next physical line report as line 1 (the directive itself is line 0, the line after is line 1). Negative numbers are accepted but produce surprising diagnostics; don't.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `__FILE__` - the companion token for the current source file name; the two are almost always used together
- `__PACKAGE__` - compile-time token for the current package name, same family of special tokens
- `caller` - runtime equivalent when you need the *caller's* line number, not the line of the call-site itself
- `die` - auto-appends `at FILE line LINE` when the message has no trailing newline, removing the need for explicit `__LINE__`
- `warn` - same auto-location behavior as `die` for non-fatal diagnostics

Control flow

PACKAGE

Return the name of the package currently in effect at the point where the token appears.

`__PACKAGE__` is a compile-time literal: the parser replaces it with a string constant naming whichever package the most recent `package` statement put into effect for this lexical scope. It does not look anything up at runtime, does not follow inheritance, and does not care what class the calling code is using - it is simply the name of the package that owns the source text it sits in.

Synopsis

```
__PACKAGE__
__PACKAGE__->some_method(@args)
$__PACKAGE__::VERSION          # does NOT work - see Edge cases
```

What you get back

A plain string containing the fully qualified package name (e.g. `"My::Thing"`, `"main"`), interned at compile time. The value is a constant for the duration of the enclosing scope - it cannot change at runtime and does not depend on how the code is called.

Outside any explicit `package` declaration, the value is `"main"`.

Global state it touches

None. `__PACKAGE__` is resolved at compile time from the lexical package in effect at its source location; it reads no interpreter state when executed and has no side effects.

Why use a token instead of writing the name

Three reasons, in the order you hit them in practice:

1. **The file gets renamed or the package gets renamed and nothing else has to change.** Every `__PACKAGE__` updates automatically; every hand-typed `"My::Thing"` has to be found and edited.
2. **It is the only portable way to write "this class" inside a package block form**, where the package ends at the closing brace and an inner helper cannot assume the file name matches.
3. **It composes with class methods cleanly**: `__PACKAGE__->new(...)` in a constructor helper calls the right class whether the file is loaded directly or copied into a subclass that has not yet redefined the helper.

For the runtime equivalent - "what class is `$self` actually blessed into?" - use `ref` or, in `use feature 'class'` code, `__CLASS__`. They are not the same thing; see *Edge cases*.

Examples

Read the current package inside a module:

```
package My::Thing;
say __PACKAGE__;                # My::Thing
```

Use `__PACKAGE__` to reach a package variable without hard-coding the name - robust against renames:

```
package My::Thing;
our $VERSION = '1.23';

sub version_string {
    no strict 'refs';
    return ${ __PACKAGE__ . '::VERSION' };
}
```

Constructor helper that stays correct when copy-pasted into a subclass or when the package is switched with the block form:

```
package My::Widget {
    sub new {
        my ($class, %args) = @_;
        bless { %args, kind => __PACKAGE__ }, $class;
    }
}
# kind is always "My::Widget", even if someone calls
# My::Widget::SubClass->new and forgets to override `new`.
```

Debug trace that prints where a log line came from, without having to duplicate the package name in every message:

```
package My::Worker;
use v5.36;

sub log_debug {
    my ($msg) = @_;
    warn __PACKAGE__ . ": $msg\n";
}
```

Cross-package resolution with the block form - each `__PACKAGE__` sees whichever package wraps it:

```
package Outer;
say __PACKAGE__;                # Outer

package Inner {
    say __PACKAGE__;            # Inner
}

say __PACKAGE__;                # Outer again
```

Edge cases

- **Constant at compile time, not a variable.** `__PACKAGE__` is not a scalar; `$__PACKAGE__` and `${__PACKAGE__}` do not exist. It interpolates in double-quoted strings as the string it resolves to:

  ```
  package Foo;
  say "in @{[__PACKAGE__]}";     # in Foo
  say "in $__PACKAGE__";         # prints "in " - $__PACKAGE__ is
   ↪undef
  ```

  The `@{[ ... ]}` idiom is the shortest way to interpolate it.

- **Symbolic reference to a package variable** requires string concatenation, not interpolation of a sigil:

  ```
  no strict 'refs';
  my $v = ${ __PACKAGE__ . '::VERSION' };   # works
  my $v = ${"__PACKAGE__::VERSION"};        # WRONG - literal
   ↪package
  ```

- **Not the same as `ref $self`.** `__PACKAGE__` is where the source was written; `ref $self` is what the object was actually blessed into. A method inherited from a base class returns the base class from `__PACKAGE__` but the subclass from `ref $self`:

  ```
  package Base;
  sub who { (__PACKAGE__, ref $_[0]) }

  package Child;
  our @ISA = ('Base');
  ```

(continues on next page)

(continued from previous page)

```
say join " / ", Child->who;     # Base / Child
```

When you want the class the method was invoked on, use `ref` on the invocant, `__CLASS__` inside a `class`/`method` block, or take `$_[0]` / `shift` as the class name for a class method.

- **Block-form scoping.** Inside `package NAME { ... }`, `__PACKAGE__` is `NAME`; after the closing brace the previous value is restored. See the nested example above.
- **`eval STRING` captures its compile-time package.** The string is compiled in the caller's package, so `__PACKAGE__` inside `eval "..."` is the package that called `eval`:

```
package Foo;
eval 'say __PACKAGE__';          # Foo
```

- **Before any `package` statement** (top of a script, top of a one-liner), `__PACKAGE__` is `"main"`.
- **`__PACKAGE__` does not mean "current class".** Under `use feature 'class'`, the class name and the package name agree on the declaration line but can differ at a call site invoked on a subclass. Inside a `method`, prefer `__CLASS__` when you want the class of the receiver.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `package` - the declaration that sets what `__PACKAGE__` returns; every non-`main` value comes from one
- `__CLASS__` - the runtime counterpart for code inside a `class` block: returns the class of the invoking instance, which may be a subclass
- `ref` - ask an object what class it was blessed into; the right tool when you have `$self` and want its actual class
- `caller` - the calling package at runtime; use when you need the caller's package, not your own
- `bless` - the primitive that turns a reference into an object of a given package; typically paired with `__PACKAGE__->new(...)` style constructors
- `our` - declares a lexical alias to a package variable in the current compile-time package (the same package `__PACKAGE__` names)

Control flow

SUB

Return a reference to the currently executing subroutine.

__SUB__ is the self-reference token: inside a sub, it evaluates to a code reference pointing at that same sub, without the sub having to know its own name. Outside any sub, it is `undef`. The common reason to reach for it is anonymous recursion - a sub stored in a lexical or passed around as a callback needs a way to call itself that does not depend on a package-qualified name, and `__SUB__` is it.

Synopsis

```
__SUB__
__SUB__->(@args)
goto __SUB__
```

Takes no arguments and has no parentheses form - it is a compile-time token, not a function call.

What you get back

A code reference (the same thing `\&name` would give you for a named sub), or `undef` when evaluated outside any subroutine. The reference is to the **runtime** sub, including any closure over lexicals captured when the sub was created, so calling it recursively sees the same captured environment.

```
my $fact = sub {
    my $n = shift;
    $n < 2 ? 1 : $n * __SUB__->($n - 1);
};

print $fact->(5);               # 120
```

The call form is an ordinary coderef invocation: `__SUB__->(@args)` or `&{__SUB__}(@args)`. No arguments are passed implicitly; recursion must pass them on.

Why not just use the sub's name

Named subs can already recurse by name - `factorial(...)` inside `sub factorial` works. __SUB__ earns its place in three situations where a name is unavailable, unreliable, or expensive:

- **Anonymous subs**. An anonymous sub assigned to a lexical has no stable package-qualified name. A common pre-5.16 workaround was capturing a reference to `\&name` or using Y-combinator tricks; __SUB__ replaces both.
- **Subs passed as callbacks**. A sub handed to another API (sort comparator, event handler, iterator step) may be called back before it has a name visible at the recursion site.
- **Tail-call recursion via goto**. `goto __SUB__` reuses the current call frame and jumps to the top of the same sub with whatever `@_` you set up, which matters for

deep recursion that would otherwise blow the stack.

Global state it touches

__SUB__ reads the currently executing call frame - the same frame `caller` inspects. It does not read or modify any special variables.

Examples

Anonymous factorial, the textbook case:

```
my $fact = sub {
    my $n = shift;
    $n < 2 ? 1 : $n * __SUB__->($n - 1);
};
print $fact->(6);               # 720
```

Tail-recursive sum with `goto __SUB__` - no stack growth per iteration, because `goto` reuses the current frame:

```
my $sum = sub {
    my ($acc, @rest) = @_;
    return $acc unless @rest;
    @_ = ($acc + shift @rest, @rest);
    goto __SUB__;
};
print $sum->(0, 1..1_000_000);  # 500000500000
```

Walking a nested data structure without naming the walker:

```
my $count_leaves = sub {
    my $node = shift;
    return 1 unless ref $node eq 'ARRAY';
    my $n = 0;
    $n += __SUB__->($_) for @$node;
    return $n;
};
print $count_leaves->([1, [2, 3], [[4], 5]]);   # 5
```

Outside any sub, __SUB__ is `undef`:

```
print defined(__SUB__) ? "yes" : "no";          # no
```

Captured, not resolved-on-call. __SUB__ is evaluated each time it runs, in the frame it runs in - so in a nested sub it refers to the **inner** sub:

```
sub outer {
    my $inner = sub { __SUB__ };    # inner's own coderef
    return $inner->();
}
my $ref = outer();
print $ref == $ref;                 # 1 - stable coderef
```

Edge cases

- **Requires the `current_sub` feature**. `__SUB__` is gated behind `use feature 'current_sub'`, enabled by default under `use v5.16` or any later version bundle. Code targeting older syntax levels must opt in explicitly, or prefix the token with `CORE::` as `CORE::__SUB__`.
- **No parentheses form**. `__SUB__()` is a syntax error - the token is not a function, it's a keyword that evaluates to a coderef. Call the result with `__SUB__->()` or `&{__SUB__}()`.
- **Outside any sub**, including at file scope and inside `BEGIN`, `UNITCHECK`, `CHECK`, `INIT`, and `END` blocks, `__SUB__` returns `undef`. Calling it (`__SUB__->()`) in those positions dies with `Can't use an undefined value as a subroutine reference`.
- **Inside `eval` blocks**, `__SUB__` refers to the **enclosing sub**, not the `eval` - an `eval { ... }` does not create a new sub frame for this purpose. Inside `eval EXPR` the same rule applies: the token resolves to the outer sub.
- **Inside a regex code block** (`/(?{...})/` or `/(??{...})/`), the behaviour of `__SUB__` is subject to change - upstream documents this as unstable. Do not rely on what it returns there.
- **Identity across calls**. Two evaluations of `__SUB__` inside the same sub yield the same coderef (`==` compares true). Across different closures made from the same `sub { ... }`, each closure sees its own coderef - one per captured environment.
- **`goto __SUB__` vs `__SUB__->()`**. The `goto` form reuses the current call frame and replaces `@_` with whatever you assigned before the `goto`; it does not grow the call stack. The coderef call form makes a fresh frame every time, so it hits the recursion depth limit on deeply recursive inputs.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `sub` - defines the subroutine whose reference `__SUB__` hands back; the first place to look when anonymous recursion is the tool for the job
- `goto` - pairs with `__SUB__` for tail-call recursion via `goto __SUB__`, the idiom that avoids stack growth
- `caller` - inspect the same call frame from the outside (package, file, line) instead of grabbing a reference to it
- `ref` - classify what `__SUB__` returned; always `"CODE"` inside a sub, empty string outside
- `return` - the usual way to exit the sub whose reference `__SUB__` is; relevant when choosing between recursion and tail-call via `goto __SUB__`

Numeric functions

abs

Return the absolute value of a number.

`abs` takes a single scalar, forces it to numeric context, and returns its magnitude - the value with any sign stripped. Non-numeric strings are converted using the same rules as arithmetic (leading whitespace and an optional sign, followed by digits; the rest is discarded, with a warning under `use warnings`). If `VALUE` is omitted, `abs` operates on `$_`.

Synopsis

```
abs VALUE
abs
abs($x)
```

What you get back

A non-negative number. The result is an integer if the input was an integer within the platform's integer range, otherwise a floating-point number. `abs` never changes its argument - the result is a fresh scalar, so `abs $x` leaves `$x` alone.

Global state it touches

With no argument, `abs` reads `$_`. It neither writes nor reads any other special variable. Under `use warnings`, a non-numeric argument triggers the standard `Argument "…" isn't numeric` warning via the normal numeric-conversion path.

Examples

Strip the sign from an integer:

```
print abs(-7);              # 7
print abs(7);               # 7
```

Floating-point magnitude - the result keeps the full precision of the input:

```
print abs(-3.14159);        # 3.14159
print abs(-1e-10);          # 1e-10
```

Use the default argument form inside a loop over `$_`:

```
for (-3, -1, 0, 2, 4) {
    print abs, "\n";        # 3, 1, 0, 2, 4 on separate lines
}
```

Distance between two points on a number line:

```
sub distance { abs($_[0] - $_[1]) }
print distance(10, 3);      # 7
print distance(3, 10);      # 7
```

String that looks like a negative number - `abs` parses the leading numeric prefix and strips the sign:

```
print abs("-42");            # 42
print abs("  -3.5xyz");      # 3.5  (warning under use warnings)
```

Edge cases

- **`abs(undef)`** returns `0`. Under `use warnings` it emits `Use of uninitialized value in abs`.
- **Non-numeric string**: `abs("hello")` returns `0` and warns under `use warnings` that the argument isn't numeric.
- **Integer overflow boundary**: `abs` of the most-negative platform integer (`IV_MIN`, e.g. `-2**63` on 64-bit builds) promotes to a floating-point result, because the positive counterpart does not fit in a signed integer:

  ```
  use Config;
  my $min = -(2 ** ($Config{ivsize} * 8 - 1));
  print abs($min);            # 9.22337203685478e+18 on 64-bit
  ```

- **IEEE 754 special values**: `abs(-0.0)` returns `0` (a positive zero). `abs("-Inf")` returns `Inf`. `abs("NaN")` returns `NaN` - `NaN` has no sign that `abs` can meaningfully remove.
- **Complex numbers and big integers** are not handled by the built-in. `use Math::Complex` overloads `abs` to compute the modulus; and provide their own sign-stripping on blessed objects via operator overloading.
- **No in-place form**: `abs` always returns a new scalar. To replace a variable's value with its magnitude, assign back:

  ```
  $x = abs $x;
  ```

- **Parser note**: the unary form `abs $x` binds tighter than `,` but looser than most arithmetic operators, so `abs $x + 1` parses as `abs($x) + 1`, not `abs($x + 1)`. Parenthesise when in doubt:

  ```
  print abs $x + 1;           # abs($x) + 1
  print abs($x + 1);          # abs of the sum
  ```

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `sqrt` - square root; like `abs` it is a pure numeric unary with a default-to-`$_` form
- `int` - truncate toward zero; pairs with `abs` when you want the integer magnitude of a float

- `sprintf` - formatted numeric output; use %d or %g together with `abs` to render unsigned magnitudes
- `$_` - the default argument when `abs` is called without one

Sockets

accept

Accept an incoming connection on a listening socket.

`accept` is the server side of the connection handshake. It waits on `GENERICSOCKET` - a socket that has already been created with `socket`, bound with `bind`, and placed in the listening state with `listen` - until a client connects, then installs a brand-new connected socket in `NEWSOCKET` and returns the client's packed address. `GENERICSOCKET` stays in the listening state, ready for the next call. Behaviour mirrors the POSIX `accept(2)` system call.

Synopsis

```
accept NEWSOCKET, GENERICSOCKET
```

What you get back

The client's address in the packed form used by the address family of `GENERICSOCKET` - the same shape you'd pass to `connect` or unpack with `Socket::unpack_sockaddr_in` for IPv4, or with `Socket::unpack_sockaddr_in6` / `unpack_sockaddr_un` for IPv6 and Unix-domain sockets respectively. On failure returns a false value (the empty string) and sets `$!` to the underlying `errno`.

`NEWSOCKET` is populated as a side effect - it is a plain filehandle argument, not a return value. A bareword like `CLIENT` autovivifies a typeglob; a lexical filehandle like `my $client` is populated in place:

```
my $client;
accept($client, $server) or die "accept: $!";
# $client is now a readable/writable handle to the connected peer
```

Global state it touches

- `$!` - set on failure to the underlying `errno` (`EINTR`, `EAGAIN`/`EWOULDBLOCK` on non-blocking listeners, `ECONNABORTED`, `EMFILE`, `ENFILE`).
- `$^F` - the system maximum file descriptor. On systems that support the close-on-exec flag, `accept` sets `FD_CLOEXEC` on the new descriptor when its number is greater than `$^F` (default 2, covering the standard streams). Raise `$^F` before `accept` if you want the accepted socket to survive `exec`.

Examples

Minimal TCP echo-style accept loop:

```
use Socket;

socket(my $server, PF_INET, SOCK_STREAM, getprotobyname("tcp"))
    or die "socket: $!";
bind($server, sockaddr_in(8080, INADDR_ANY)) or die "bind: $!";
listen($server, SOMAXCONN)                   or die "listen: $!";

while (my $peer = accept(my $client, $server)) {
    my ($port, $iaddr) = sockaddr_in($peer);
    print $client "hello ", inet_ntoa($iaddr), ":$port\n";
    close $client;
}
die "accept: $!";   # loop exits only on error
```

Unpack the returned address for logging:

```
use Socket;

my $peer = accept(my $client, $server)
    or die "accept: $!";
my ($port, $iaddr) = sockaddr_in($peer);
printf "connection from %s port %d\n", inet_ntoa($iaddr), $port;
```

Non-blocking accept. Put `$server` in non-blocking mode first; a pending-connection-less call then returns false with `$!` set to `EAGAIN` or `EWOULDBLOCK`:

```
use Errno qw(EAGAIN EWOULDBLOCK);
use Fcntl qw(F_GETFL F_SETFL O_NONBLOCK);

my $flags = fcntl($server, F_GETFL, 0);
fcntl($server, F_SETFL, $flags | O_NONBLOCK);

my $peer = accept(my $client, $server);
if (!$peer) {
    if ($! == EAGAIN || $! == EWOULDBLOCK) {
        # no pending connection - try again later
    } else {
        die "accept: $!";
    }
}
```

Preserve the accepted socket across `exec` by raising `$^F` so the runtime does not set `FD_CLOEXEC`:

```
local $^F = 10_000;
accept(my $client, $server) or die "accept: $!";
exec "/usr/libexec/handler", fileno($client);
```

Edge cases

- **GENERICSOCKET not listening** - `accept` fails with `$!` set to `EINVAL`. Call `listen` on the server socket first.
- **NEWSOCKET already open** - the existing handle is silently closed before the new descriptor is installed, exactly as for `open` and `socket`. If the old handle was the only reference to a buffered stream, any unflushed data is lost. Close explicitly beforehand when that matters.
- **Signal during the call** - a signal delivered while `accept` is blocked makes the underlying syscall return `EINTR`. pperl does not auto-restart: `accept` returns false and leaves `$!` set to `EINTR`. Wrap in a retry loop if your signal handlers do not terminate the process.
- **Descriptor exhaustion** - `EMFILE` (process limit) or `ENFILE` (system limit) on a busy server is the common reason `accept` starts failing under load. Treat them as transient; close idle clients and retry.
- **Address-family mismatch of the packed return value** - the packed buffer follows the family of `GENERICSOCKET`. Unpacking a Unix-domain address with `Socket::unpack_sockaddr_in` produces garbage. Track the family yourself or use `Socket::sockaddr_family` to dispatch.
- **Parentheses are optional, commas are not** - `accept $c, $s` and `accept($c, $s)` both work; `accept $c $s` is a syntax error.
- **List vs scalar context** - the packed-address return is a byte string; list context does **not** expand it into `(port, addr)`. Call `sockaddr_in` / equivalent to unpack.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `socket` - create the server socket that `accept` waits on
- `bind` - attach the server socket to a local address before `listen` and `accept`
- `listen` - mandatory step between `bind` and `accept`; sets the backlog depth
- `connect` - the client-side counterpart; `accept` returns the address a `connect` call supplied
- `getpeername` - recover the same packed address later from `NEWSOCKET` itself
- `Socket` - the constants (`PF_INET`, `SOCK_STREAM`, `INADDR_ANY`) and packers (`sockaddr_in`, `unpack_sockaddr_in`) that turn the packed return value into something usable
- `$^F` - threshold that controls whether `accept` sets `FD_CLOEXEC` on the new descriptor

Processes

alarm

Schedule a SIGALRM to be delivered to the current process after a whole number of wall-clock seconds.

alarm arms a single per-process countdown timer. When the timer expires, the kernel delivers SIGALRM; what happens next is entirely up to the installed signal handler in %SIG. The timer runs in wallclock time, not CPU time, so a sleeping or blocked process still receives the signal on schedule (subject to scheduling jitter).

Synopsis

```
alarm SECONDS
alarm
```

With no argument, the value in $_ is used.

What you get back

The number of seconds remaining on the **previous** timer, or 0 if no timer was pending. This lets you save and restore a caller's alarm across a nested timed operation:

```
my $prev = alarm 10;
# ... do something that must finish in 10s ...
alarm $prev;                        # restore caller's timer
```

alarm 0 cancels any pending timer and returns the seconds that were left on it.

Global state it touches

- %SIG - the ALRM slot names the handler invoked when the timer fires. Without a handler, the default disposition of SIGALRM is to terminate the process.
- $_ - read when alarm is called with no argument.
- $@ - carries the die message out of the canonical eval { alarm ...; ...; alarm 0 } idiom.
- $! - a blocking syscall interrupted by SIGALRM sets $! to EINTR, **but** Perl restarts many syscalls automatically on some systems, so you cannot rely on seeing EINTR. Use eval / die instead (see below).

The canonical timeout idiom

Use alarm with eval and die to put a wall-clock cap on any block of code, including blocking I/O:

```
eval {
    local $SIG{ALRM} = sub { die "timeout\n" };   # \n matters
    alarm $timeout;
    my $nread = sysread $sock, $buf, 4096;
    alarm 0;                                      # cancel on␣
```

(continues on next page)

(continued from previous page)

```
 ↪success
};
if ($@) {
    die $@ unless $@ eq "timeout\n";              # rethrow others
    # ... handle timeout ...
}
```

Three details are load-bearing:

- `local` scopes the handler to the `eval` block so it does not leak to the caller.
- The `\n` in `"timeout\n"` suppresses the file/line suffix that `die` would otherwise append, making the exact-match test in `$@` reliable.
- `alarm 0` **inside** the `eval` cancels the timer on the success path. Without it, a timer left armed can fire during unrelated code after the `eval` returns.

Examples

Simple wallclock deadline:

```
$SIG{ALRM} = sub { die "timeout\n" };
eval {
    alarm 5;
    my $line = <$slow_pipe>;        # may block indefinitely
    alarm 0;
};
warn "gave up waiting\n" if $@ eq "timeout\n";
```

Nested timers - save and restore:

```
my $saved = alarm 30;               # remember caller's timer
do_something();
alarm $saved;                       # put it back
```

Cancel a pending timer:

```
my $left = alarm 0;                 # 0 if none was pending
```

Periodic tick without `Time::HiRes`:

```
$SIG{ALRM} = sub {
    print "tick\n";
    alarm 1;                        # re-arm from inside the handler
};
alarm 1;
sleep 10;                           # do not mix sleep+alarm in real
 ↪code
```

Sub-second delay - use `Time::HiRes`:

```
use Time::HiRes qw(alarm);
alarm 0.25;                          # 250 ms
```

Edge cases

- **Integer seconds only.** The core `alarm` truncates fractional arguments to whole seconds. For finer granularity, load `Time::HiRes`, which exports a replacement `alarm` that accepts floats.
- **Scheduling jitter.** The signal may arrive up to roughly one second early or late because of how the kernel counts seconds and schedules your process. Treat `alarm N` as a *not-before-N-minus-1* bound, not a precise deadline.
- **One timer per process.** Each `alarm` call replaces the previous timer; there is no queue. The return value is your only record of what was overwritten.
- **Default disposition is fatal.** With no handler installed for `ALRM`, timer expiry terminates the process with `Alarm clock`. Install a handler *before* arming the timer.
- **Do not mix with `sleep`.** On many systems `sleep` is implemented on top of `alarm`, so arming one clobbers the other. Pick one per block.
- **`EINTR` is unreliable.** Perl installs signal handlers with `SA_RESTART` on many platforms, so a `SIGALRM` that interrupts a blocking syscall transparently restarts the call instead of returning with `$!` set to `EINTR`. The portable way to enforce a deadline is `eval` / `die` from the handler, as shown above.
- **Signals are delivered between Perl ops.** Inside a tight pure-Perl loop the handler runs at the next safe point, not instantly; a CPU-bound XS call can defer delivery further.
- **`fork` clears the timer.** Child processes start with no pending alarm regardless of what the parent had armed.
- **exec clears the timer** on Linux; the replacement process image starts with no pending `SIGALRM`.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `sleep` - wallclock delay; often implemented on top of `alarm`, so the two do not compose
- `die` - raise the exception from the `ALRM` handler that `eval` catches to implement a timeout
- `eval` - catches the `die` from the handler and lets the program recover from a timeout
- `%SIG` - where the `ALRM` handler is installed; the handler decides what the timer *means*

- `select` - the four-argument form accepts a floating-point timeout and is an alternative to `alarm` for I/O deadlines
- `Time::HiRes` - drop-in `alarm` replacement that accepts sub-second floats via `setitimer(2)`

Lists

all

Test whether `BLOCK` returns true for **every** element of `LIST`.

`all` walks `LIST` left to right, aliasing `$_` to each element in turn and evaluating `BLOCK` with that element. It returns a true value if every evaluation yields truth, and a false value as soon as any evaluation yields falsity. As soon as one element makes the block false, `all` stops and returns - it does not evaluate the block for the remaining elements.

This is the language-level `all` keyword, enabled by the `keyword_all` feature. It is distinct from `List::Util::all` - see *Edge cases* below.

Synopsis

```
use feature 'keyword_all';
no warnings 'experimental::keyword_all';

all { /\w+/ } @words;
all { defined $_ } @list;        # canonical "all defined" check
all { $_ > 0 } @numbers;
```

What you get back

A boolean. True if every element passes, false otherwise. The exact true/false values are unspecified - treat the result as a condition, not a datum to propagate.

Empty list returns true. This is *vacuous truth*: the statement "every element satisfies `BLOCK`" is trivially true when there are no elements to check. It is the single most common source of surprise with `all`:

```
all { $_ > 0 } ();               # TRUE - no elements, nothing to
 ↪fail
```

Guard with an explicit size check when an empty list should be treated as failure:

```
if (@numbers && all { $_ > 0 } @numbers) { ... }
```

Global state it touches

- `$_` - aliased to each element of `LIST` in turn for the duration of the block. Because this is an alias and not a copy, assigning to `$_` inside the block mutates the underlying list element. The previous value of `$_` is restored when `all` returns.

Examples

Check that every value in a list is defined - the canonical use:

```
my @row = get_record();
die "missing field" unless all { defined $_ } @row;
```

Check that every number is positive:

```
my @prices = (1.99, 4.50, 0.99);
say "ok" if all { $_ > 0 } @prices;          # ok
```

Validate a record where every field must match its own predicate. Combine multiple tests inside the block rather than chaining `all` calls:

```
my @fields = ($name, $age, $email);
my $ok = all {
    defined $_ && length $_ > 0
} @fields;
```

Combine `all` with `any` for an exclusive check - at least one match, but not every element the same:

```
if (any { $_ eq 'admin' } @roles
    && !all { $_ eq 'admin' } @roles) {
    say "mixed roles, including admin";
}
```

Short-circuit behaviour: the block runs only until the first failure. Useful for predicates with side effects or cost:

```
all { expensive_check($_) } @candidates;    # stops at first failure
```

Edge cases

- **Empty list returns true.** Vacuous truth. Guard with `@list && all { ... } @list` when emptiness should fail.
- **Short-circuits on first false.** The block is not called for any element after the first failure. Do not rely on side effects in the block running for every element - use a `for` loop if you need that.
- **`$_` is aliased, not copied.** Modifying `$_` inside the block modifies the list element:

  ```
  my @xs = (1, 2, 3);
  all { $_++; $_ > 0 } @xs;
  # @xs is now (2, 3, 4)
  ```

- **Experimental under 5.44.** The feature emits a compile-time warning in the `experimental::keyword_all` category unless that category is silenced:

  ```
  no warnings 'experimental::keyword_all';
  ```

- **Not the same as `List::Util::all`.** The built-in is a language keyword enabled by the `keyword_all` feature and parses `BLOCK` as a true block (no leading `sub`, no trailing comma). `List::Util::all` is a regular subroutine imported from a module and takes a code reference / block prototype argument. For most practical purposes they behave the same on non-empty lists, but:
  - The keyword form is a parser-level construct and can be optimised more aggressively.
  - The two share neither implementation nor warning category.
  - Code that needs to run on pre-5.42 perls should use `List::Util::all`.
- **Equivalent `grep` form.** `all { COND } @xs` is semantically equivalent to `@xs == grep { COND } @xs` - but `grep` always evaluates the block for every element, so it lacks `all`'s short-circuit behaviour.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `any` - true if the block returns true for **at least one** element; the mirror operator to `all`
- `grep` - evaluates the block for every element and returns matches; `scalar @list == grep { … } @list` is `all` without the short-circuit
- `defined` - the predicate inside the canonical `all { defined $_ } @list` check
- `List::Util::all` - module-level equivalent available on perls older than 5.42; also provides none and `notall` which have no language-level counterpart
- `List::Util::any` - module-level counterpart to the `any` keyword, alongside none and `notall` for the opposite polarities

Lists

any

Return true if `BLOCK` yields true for at least one element of `LIST`.

any runs `BLOCK` once per element of `LIST`, aliasing `$_` to the current element, and returns a true value the moment the block yields true. If no element makes the block true - including the case where `LIST` is empty - any returns false. It never runs the block more times than it has to: the first true result decides the answer and the remaining elements are not examined.

any is not unconditionally available. It is a feature-gated core keyword introduced in Perl 5.42; enable it with:

```
use feature 'keyword_any';
```

The long-standing `List::Util::any`, available since at least Perl 5.20, implements the same semantics as a subroutine. The two are interchangeable for everyday code - pick

the core keyword when you want the slightly lower call overhead and an extra stack frame fewer in profiles, or `List::Util::any` when you also need other `List::Util` exports or have to support Perls older than 5.42.

Synopsis

```
use feature 'keyword_any';
any BLOCK LIST
```

What you get back

A boolean: a true value if any iteration of `BLOCK` yielded true, a false value otherwise. The exact shape of the true/false value is unspecified - treat the return as boolean, do not rely on getting 1 and `""` specifically.

An empty `LIST` returns false. This is the mathematically correct answer (the existential quantifier over an empty domain is false) and matches `List::Util::any`.

Global state it touches

- `$_` is aliased to each element of `LIST` in turn while `BLOCK` runs. The aliasing is the same kind `map` and `grep` use: modifying `$_` inside the block modifies the list element itself. Use a named lexical via `my $x = $_` in the first line of the block if you need a safe copy.

Examples

Test whether any line in a list matches a pattern:

```
use feature 'keyword_any';
my @lines = ("foo\n", "bar ERROR baz\n", "quux\n");
if (any { /ERROR/ } @lines) {
    warn "log contains an error line\n";
}
```

Numeric threshold check:

```
use feature 'keyword_any';
my @nums = (1, 4, 7, 12, 3);
print "over limit\n" if any { $_ > 10 } @nums;   # over limit
```

Command-line flag detection, without pulling in a module:

```
use feature 'keyword_any';
if (any { $_ eq '--verbose' } @ARGV) {
    $verbose = 1;
}
```

any paired with `all` as a sanity check - every record has an `id`, and at least one is flagged active:

```
use feature qw(keyword_any keyword_all);
my @records = (
    { id => 1, active => 0 },
    { id => 2, active => 1 },
);
die "malformed" unless all { defined $_->{id} } @records;
die "none active" unless any { $_->{active} } @records;
```

Using the `List::Util` version when you cannot depend on Perl 5.42:

```
use List::Util qw(any);
print "found\n" if any { $_ eq 'needle' } @haystack;
```

Edge cases

- **Short-circuits on the first true result.** Side effects in `BLOCK` run only for elements examined before (and including) the first true element. Do not use `any` to drive a loop that must touch every element - use `for` or `map` for that.
- **Empty `LIST` returns false.** `any { anything } ()` is false without ever running the block.
- **`$_` aliasing is identical to `map` and `grep`.** Writing to `$_` mutates the source list element. The idiom `any { my $x = $_; ... }` makes the copy explicit.
- **`return` inside `BLOCK` returns from the enclosing sub**, not from `any`. To terminate the scan early with a specific truth value, let the block evaluate to that value - `any` already short-circuits.
- **Core keyword vs `List::Util::any`.** Both implement the same semantics. The core keyword avoids one Perl-level call frame per invocation of `any` itself, which shows up in tight profiling; the block itself still runs per-element either way. Mixing both in the same file is legal but confusing; pick one per file.
- **`experimental::keyword_any` warning.** Because the keyword is still flagged experimental, enabling it emits a warning in the `experimental::keyword_any` category on first use. Silence it with `no warnings 'experimental::keyword_any';` once you have decided to depend on the keyword.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `all` - the universal-quantifier counterpart; true iff `BLOCK` yields true for **every** element
- `grep` - collects every matching element; `scalar grep { ... } @list` has the same truth value as `any { ... } @list` but runs the block for every element
- `List::Util::any` - pre-5.42 subroutine form with identical semantics

- `List::Util::first` - returns the first element for which the block is true, rather than a boolean; use this when you need the element itself
- `map` - same `$_`-aliasing contract, but always visits every element and collects results

Numeric functions

atan2

Arc tangent of `Y/X` in the range -π to π.

`atan2` is the two-argument arctangent - the inverse of tangent that keeps enough information about its inputs to return the correct quadrant. Unlike a single-argument `atan`, which collapses the two signs of `Y` and `X` into one ratio and cannot distinguish the upper half-plane from the lower, `atan2` looks at both arguments and returns an angle covering the full circle: `(-π, π]` radians. It is the canonical way in Perl to convert Cartesian coordinates `(X, Y)` into the polar angle θ.

Synopsis

```
atan2 Y, X
```

What you get back

A floating-point number, the angle in radians between the positive x-axis and the point `(X, Y)`, measured counter-clockwise. Range: `-π < result ≤ π`. Sign follows `Y`: positive `Y` gives a positive angle (upper half-plane), negative `Y` gives a negative angle (lower half-plane). To convert to degrees multiply by `180 / π`:

```
my $deg = atan2($y, $x) * 180 / 3.14159265358979;
```

Examples

Compute π to machine precision. The angle from the origin to `(0, 1)` is exactly π/2, so `atan2(1, 0) * 2` gives π:

```
my $pi = atan2(1, 0) * 2;
printf "%.15f\n", $pi;          # 3.141592653589793
```

Quadrant handling. The four cardinal directions land on exact multiples of π/2, with `(-1, 0)` at the boundary +π, not -π:

```
print atan2( 1,  0), "\n";      #  1.5707963267949   (+π/2, up)
print atan2( 0,  1), "\n";      #  0                 (right)
print atan2(-1,  0), "\n";      # -1.5707963267949   (-π/2, down)
print atan2( 0, -1), "\n";      #  3.14159265358979  (+π, left)
```

Polar conversion. Given a point `(X, Y)`, recover radius and angle:

```
my ($x, $y) = (3, 4);
my $r     = sqrt($x*$x + $y*$y);    # 5
my $theta = atan2($y, $x);          # 0.927295218001612 rad ≈ 53.13°
```

Tangent via the standard identity. Perl has no built-in `tan`, but `sin` and `cos` give it directly - and `atan2` inverts the result unambiguously:

```
sub tan { sin($_[0]) / cos($_[0]) }
my $angle = 0.7;
print atan2(tan($angle), 1), "\n";  # 0.7  (recovers the input)
```

Full-circle sweep. Iterating `Y` around the unit circle shows the range covers `(-π, π]`:

```
for my $deg (0, 45, 90, 135, 180, -135, -90, -45) {
    my $rad = $deg * 3.14159265358979 / 180;
    printf "%4d°  ->  %+.4f rad\n",
        $deg, atan2(sin($rad), cos($rad));
}
```

Edge cases

- **Both arguments zero**: `atan2(0, 0)` is mathematically undefined. The value Perl returns is whatever the platform's C `atan2(3)` returns for this case - on glibc and musl that is `0`, and POSIX permits any value in `[-π, π]`. Do not rely on a specific result; check for `($x == 0 && $y == 0)` before calling if zero is a real possibility in your inputs.
- **Signed zeros**: on IEEE-754 platforms `atan2(+0, -0)` returns `+π` and `atan2(-0, -0)` returns `-π`. This only matters in code that deliberately tracks zero signs; most Perl code never distinguishes them.
- **Infinities**: `atan2(Inf, Inf) == π/4`, `atan2(Inf, -Inf) == 3π/4`, `atan2(-Inf, Inf) == -π/4`, `atan2(-Inf, -Inf) == -3π/4`. The direction "toward `(±Inf, ±Inf)`" is still a well-defined angle bisecting the appropriate quadrant.
- **NaN inputs**: if either `Y` or `X` is `NaN`, the result is `NaN`. Propagates through any further arithmetic - guard with a `Scalar::Util::looks_like_number` check on inputs from untrusted sources.
- **Non-numeric arguments**: strings are coerced to numbers by the usual Perl rules. `atan2("3", "4")` works; `atan2("abc", 1)` is equivalent to `atan2(0, 1)` and emits an `Argument "abc" isn't numeric` warning under `use warnings`.
- **Argument order is `Y, X`, not `X, Y`**: this matches C's `atan2(3)`, FORTRAN's `ATAN2`, and every other language's two-argument arctangent. Reversing the arguments gives the complement angle - a subtle bug that tests in only one quadrant will miss.
- **`atan2` is not `atan`**: there is no one-argument form. `atan2($ratio)` is a syntax error at the call site. For the single-argument arctangent, use `Math::Trig::atan` or `POSIX::atan`, or write it as `atan2($ratio, 1)`.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `sin` - sine; paired with `cos` to reconstruct an angle's coordinates after `atan2` has extracted it
- `cos` - cosine; the X component when converting a radius and angle back to Cartesian coordinates
- `sqrt` - square root; used with `atan2` in the Cartesian → polar conversion `($r, $theta) = (sqrt($x**2+$y**2), atan2($y,$x))`
- - ships `tan`, `atan`, `asin`, `acos`, and the hyperbolic family; also exports `pi` as a constant so you do not have to write `atan2(1,0)*2`

Sockets

bind

Attach a local address to a socket.

`bind` is the Perl wrapper around the POSIX `bind(2)` system call. It associates the already-opened `SOCKET` filehandle with the local address `NAME`, so that subsequent `listen` / `accept` calls (for a server) or `connect` / `send` calls (for a client with a fixed local address) use that address as their source. `NAME` is a **packed** address structure in the form the socket's address family expects - the job of packing is yours, typically via `Socket` helpers such as `` `sockaddr_in` `` or `` `sockaddr_un` ``.

Synopsis

```
bind SOCKET, NAME
```

What you get back

A true value on success, false on failure (with `$!` set to the underlying errno: `EADDRINUSE`, `EACCES`, `EADDRNOTAVAIL`, etc.). Always check the return value - unlike `print`, a failed `bind` is a setup-time error that has to abort the program:

```
bind $sock, $addr
    or die "bind failed: $!";
```

Global state it touches

None directly. `bind` sets `$!` on failure; that is the only interpreter global involved. The socket itself must already be open - typically produced by `socket` or `socketpair` - and `bind` mutates its kernel-side state, not any Perl-level variable.

Examples

Bind a TCP server socket to port 9000 on all local interfaces and start listening. `INADDR_ANY` selects "any local address"; the packed form comes from `` `sockaddr_in` ``:

```
use Socket;

socket(my $srv, PF_INET, SOCK_STREAM, getprotobyname("tcp"))
    or die "socket: $!";
bind($srv, sockaddr_in(9000, INADDR_ANY))
    or die "bind: $!";
listen($srv, SOMAXCONN)
    or die "listen: $!";
```

Bind a client socket to a specific local address before connecting - used when the remote peer filters by source IP, or when a host has multiple interfaces and you need outgoing traffic on a specific one:

```
use Socket;

socket(my $cli, PF_INET, SOCK_STREAM, getprotobyname("tcp"))
    or die "socket: $!";
bind($cli, sockaddr_in(0, inet_aton("192.0.2.17")))
    or die "bind: $!";          # port 0 = kernel picks one
connect($cli, sockaddr_in(443, inet_aton("93.184.216.34")))
    or die "connect: $!";
```

Bind a UNIX-domain listener to a filesystem path. The path must not already exist - bind creates the socket node and leaves it behind when the process exits, so clean it up yourself:

```
use Socket;

my $path = "/tmp/myapp.sock";
unlink $path;                   # clear stale socket from prior run
socket(my $srv, PF_UNIX, SOCK_STREAM, 0)
    or die "socket: $!";
bind($srv, sockaddr_un($path))
    or die "bind $path: $!";
listen($srv, 5) or die "listen: $!";
```

Bind a UDP socket to a fixed local port so replies from the peer come back to a predictable address:

```
use Socket;

socket(my $udp, PF_INET, SOCK_DGRAM, getprotobyname("udp"))
    or die "socket: $!";
bind($udp, sockaddr_in(5300, INADDR_ANY))
    or die "bind: $!";
```

Rebinding a previously-released port in quick succession requires SO_REUSEADDR; otherwise bind fails with EADDRINUSE during the TIME_WAIT window:

```
use Socket;

```

(continues on next page)

(continued from previous page)

```
setsockopt($srv, SOL_SOCKET, SO_REUSEADDR, pack("l", 1))
    or die "setsockopt: $!";
bind($srv, sockaddr_in(9000, INADDR_ANY))
    or die "bind: $!";
```

Edge cases

- **`SOCKET` must already be open.** `bind` does not create the socket - call `socket` first. A closed or never-opened filehandle fails with `EBADF`.
- **`NAME` is a packed byte string, not a human-readable address.** Pass the output of `` `sockaddr_in` ``, `` `sockaddr_in6` ``, or `` `sockaddr_un` `` from `Socket`. Passing a plain string like `"127.0.0.1:9000"` produces `EINVAL` or worse - the kernel reads the bytes as a `struct sockaddr`.
- **Port 0 means "let the kernel choose".** After a successful `bind` with port 0, retrieve the actual port with `getsockname` and `` `sockaddr_in` `` in unpack mode.
- **Ports below 1024** require elevated privileges on Unix - `bind` fails with `EACCES` for an unprivileged process.
- **`EADDRINUSE`** is the most common failure. It fires when another process holds the address, when a prior socket on the same address is still in `TIME_WAIT`, or when you call `bind` twice on the same socket. `SO_REUSEADDR` (see `setsockopt`) addresses the `TIME_WAIT` case; the others are genuine conflicts.
- **UNIX-domain sockets leave a filesystem entry** that outlives the process. Remove it before a fresh `bind` (`unlink $path`) and on clean shutdown.
- **Binding after `connect` is an error.** Once a socket has been connected, its local address is fixed. Call `bind` first if you need a specific source address.
- **IPv6 sockets** use `` `sockaddr_in6` `` and `PF_INET6`; mixing families between the socket and the packed address fails with `EAFNOSUPPORT`.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `socket` - create the socket filehandle that `bind` attaches to; must be called first
- `connect` - the client-side counterpart; sets the remote address instead of the local one
- `listen` - mark a bound socket as willing to accept incoming connections
- `accept` - pull the next pending connection off a bound, listening socket
- `getsockname` - recover the actual address `bind` chose, especially after binding to port 0
- `setsockopt` - set `SO_REUSEADDR` and friends *before* `bind` to control address reuse

- `Socket` - packs and unpacks the `NAME` argument; source of `` `sockaddr_in` ``, `` `sockaddr_un` ``, `INADDR_ANY`, `PF_INET`, and related constants

I/O

binmode

Set the I/O layer stack on a filehandle - typically to make it deliver raw bytes, or to attach a character encoding.

`binmode` reconfigures `FILEHANDLE` so that subsequent reads and writes go through a specific PerlIO layer stack. In the one-argument form it puts the handle into raw binary mode (no CRLF translation, no character decoding). In the two-argument form it pushes (or, for a handful of pseudo-layers, replaces or pops) the named I/O layers onto the handle's stack. Call it after `open` and before any I/O on the handle, except for `:encoding`, which may be applied mid-stream.

Synopsis

```
binmode FILEHANDLE
binmode FILEHANDLE, LAYER
```

What you get back

1 on success, `undef` on failure, with `$!` set. Failure is rare in practice - passing a malformed layer string or an unknown layer name is the usual cause. The return value is worth checking when the layer argument is user-supplied:

```
binmode $fh, ":encoding($enc)"
    or die "bad encoding '$enc': $!";
```

If `FILEHANDLE` is an expression rather than a bareword or simple scalar, its value is taken as the name of the filehandle, matching the filehandle-resolution rules of `open` and `print`.

Global state it touches

`binmode` itself reads no special variables, but the layers it installs change how later I/O on the handle interacts with them. The layer stack governs how `$/` (input record separator) and `$\` (output record separator) are matched and emitted against the external byte stream - on platforms where text mode translates `\n` to a multi-byte sequence, the translation happens inside the layer, not in the variable.

PerlIO layers

A layer is a string beginning with `:` that names a filter in the PerlIO stack - see `PerlIO` for the full catalogue. Multiple layers may be given in one call by concatenating them:

```
binmode $fh, ":raw:utf8";
```

The order is bottom-up: `:raw` is applied first (closest to the OS file descriptor), then `:utf8` sits on top of it. Reads traverse the stack from bottom to top; writes traverse it from top to bottom.

The commonly used layers:

- `:raw` - strip the handle down to byte-for-byte I/O. Turns off CRLF translation, removes any character-encoding layers, and marks the handle as bytes. Despite occasional claims to the contrary, `:raw` is **not** simply the inverse of `:crlf`; it also disables any other layer that would alter the binary nature of the stream. This is what the one-argument form installs.
- `:crlf` - translate `\r\n` sequences to `\n` on input and `\n` to `\r\n` on output. This is the default on Windows / DOS-family systems for text-mode handles; applying it explicitly forces CRLF translation on any platform.
- `:utf8` - mark data on the handle as UTF-8. No validation on input: bytes are accepted and flagged as characters. On output, characters are encoded to UTF-8 bytes. Fast, but trusts the source.
- `:encoding(NAME)` - decode bytes to characters on input and encode characters to bytes on output, using the named encoding (`UTF-8`, `iso-8859-1`, `shiftjis`, etc.). Input that is not valid for the encoding triggers a warning and substitution. `:encoding` implicitly pushes `:utf8` on top of itself because Perl works on UTF-8 internally. See `PerlIO::encoding` for details and tuning options.
- `:bytes` - the inverse of `:utf8`; mark data on the handle as bytes, disabling any character-semantics flag that a higher layer set.

Two pseudo-layers control stack shape rather than filtering:

- `:pop` - remove the topmost layer from the handle.
- `:push` - used with `:via(...)` to stack a custom layer module.

`:raw` in the two-argument form acts as a reset: it pops layers until the handle is in its most minimal state, rather than pushing a new layer on top.

Examples

The one-argument form, for reading a binary file portably:

```
open my $img, "<", "photo.jpg" or die $!;
binmode $img;
my $bytes = do { local $/; <$img> };
```

Read a UTF-8 text file with validation:

```
open my $fh, "<", "notes.txt" or die $!;
binmode $fh, ":encoding(UTF-8)";
while (my $line = <$fh>) {
    # $line contains decoded characters, not bytes
}
```

Write Windows-style line endings on any platform:

```
open my $out, ">", "dos.txt" or die $!;
binmode $out, ":crlf";
print $out "one\ntwo\nthree\n";    # written as one\r\ntwo\r\nthree\
 ↪r\n
```

Strip a previously-applied encoding layer and drop back to bytes:

```
binmode $fh, ":pop";               # remove topmost layer
binmode $fh, ":raw";               # or: flatten to bare bytes
```

Validate a user-supplied encoding before committing to it:

```
my $enc = $ENV{INPUT_ENCODING} // "UTF-8";
binmode $fh, ":encoding($enc)"
    or die "unsupported encoding '$enc': $!";
```

Edge cases

- **Call after open, before I/O.** `binmode` on a handle that has already been read from or written to produces defined but implementation-dependent results; most layers flush pending buffers on install, which may lose data that crossed the layer boundary. `:encoding` is the documented exception - it may be pushed mid-stream without a flush.
- **One-argument form is `:raw`, not "do nothing".** On Unix the observable effect is often nil because handles are already in byte-oriented mode, but on Windows it disables CRLF translation. Write portable code as if `binmode $fh` is always meaningful.
- **CRLF on Windows / DOS.** Without `binmode`, the C runtime on those platforms converts `\r\n` to `\n` on input and the reverse on output. For binary formats (images, compressed data, serialised structures) this silently corrupts the stream. `binmode` is mandatory there.
- **Ctrl-Z as end-of-file.** On Microsoft-family systems, text-mode handles treat `\cZ` (byte 0x1A) as end-of-file. Binary data that happens to contain that byte will appear truncated unless `binmode` is used.
- **`:utf8` is trust-only.** An input handle with `:utf8` but no `:encoding(UTF-8)` will happily hand you invalid UTF-8 flagged as characters. Use `:encoding(UTF-8)` when the source is untrusted.
- **`:encoding` pulls in `:utf8`.** After `binmode $fh, ":encoding(shiftjis)"` the handle has both `:encoding(shiftjis)` and `:utf8` in its stack - perl's internal form is UTF-8 regardless of the wire encoding.
- **Filehandle expressions.** `binmode $handles[0], ":utf8"` works; any expression that yields a filehandle value is accepted as the first argument and is evaluated to name the handle.
- **Affects every I/O primitive on the handle.** The layer stack governs `read`, `sysread`, `syswrite`, `seek`, `tell`, `print`, and `readline` alike. In particular, `sysread` and `syswrite` on a `:utf8`-marked handle still go through the layer and may return partial characters.

- **Interaction with the open pragma.** `use open ":encoding(UTF-8)"` installs a default layer set for subsequently-opened handles. Explicit `binmode` overrides that default for a given handle.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `open` - create the filehandle in the first place; accepts layers directly in the mode string (`<:encoding(UTF-8)`) to skip a follow-up `binmode` call
- `Encode` - encode and decode strings in memory, independent of any filehandle
- `PerlIO` - full layer catalogue, including less common layers like `:scalar`, `:via`, and `:mmap`
- `read` / `sysread` - input primitives whose byte / character semantics depend on the layer stack
- `print` / `syswrite` - output primitives whose encoding and line-ending behaviour depend on the layer stack
- `$/` - input record separator, matched against the post-layer byte stream
- `$\` - output record separator, written through the layer stack

Classes and OO

bless

Mark the thing a reference points at as an object in a package.

`bless` associates the referent of `REF` with the package named by `CLASSNAME`, so that method calls through `REF` dispatch against that package's subroutines (and its `@ISA` ancestors). The reference itself does not change; what changes is the referent's class tag. `bless` returns `REF`, which is why it is almost always the last statement in a constructor.

Synopsis

```
bless REF, CLASSNAME
bless REF
```

What you get back

The same reference that was passed in as the first argument, now carrying the class tag of `CLASSNAME`. Returning the reference lets you tail-call `bless` from a constructor:

```
sub new {
    my ($class, %args) = @_;
    return bless { %args }, $class;
}
```

From that point on, `ref` on the returned reference yields the class name rather than the raw reference type (`HASH`, `ARRAY`, `CODE`, ...).

Forms

Two calling forms, one strongly preferred:

- **Two-argument - always use this.**

  ```
  bless $ref, $class;
  ```

  The referent becomes an object in the package named by `$class`. Using the first argument of the constructor (`$_[0]`, conventionally `$class`) keeps the constructor inheritable: a subclass calling `Parent::new('Child', ...)` gets back an object blessed into `Child`, not into `Parent`.

- **Two-argument with empty string - bless into `main`.**

  ```
  bless $ref, "";
  ```

  An empty `CLASSNAME` is treated as the `main` package. Rarely useful outside of deliberate tricks.

- **One-argument - discouraged.**

  ```
  bless $ref;
  ```

  The referent is blessed into the *current* package (the package the `bless` call is compiled in). This breaks inheritance: a subclass that inherits a one-argument `bless` constructor still ends up with objects in the parent's package, not the caller's. Use the two-argument form unless you have a concrete reason not to, and document the reason.

Examples

A minimal hash-based class:

```
package Point;

sub new {
    my ($class, $x, $y) = @_;
    return bless { x => $x, y => $y }, $class;
}

sub x { $_[0]->{x} }
sub y { $_[0]->{y} }

package main;

my $p = Point->new(3, 4);
print ref $p, "\n";       # Point
print $p->x, ",", $p->y;  # 3,4
```

Inheritable constructor - the same new works for a subclass without modification:

```
package Shape;
sub new {
    my ($class, %args) = @_;
    return bless { %args }, $class;
}

package Circle;
our @ISA = ('Shape');
sub area {
    my $self = shift;
    return 3.14159 * $self->{radius} ** 2;
}

package main;
my $c = Circle->new(radius => 5);
print ref $c, "\n";       # Circle
print $c->area, "\n";     # 78.53975
```

Array-based objects work the same way - any reference type can be blessed:

```
package Pair;
sub new {
    my ($class, $a, $b) = @_;
    return bless [$a, $b], $class;
}
sub first  { $_[0]->[0] }
sub second { $_[0]->[1] }

my $p = Pair->new("left", "right");
print $p->first, "/", $p->second;   # left/right
```

Re-blessing an existing object changes its class tag in place:

```
my $obj = bless {}, 'OldClass';
bless $obj, 'NewClass';
print ref $obj;            # NewClass
```

`bless` returns the reference, which is why chaining into method calls on the same line works:

```
Logger->new(file => $path)->info("started");
#                           ^ called on the blessed ref
```

Edge cases

- **The reference is what gets tagged, not the variable.** Copies of the reference share the class tag because they all point to the same referent:

```
my $a = bless {}, 'Foo';
my $b = $a;
print ref $b;            # Foo
```

- **Re-blessing is permitted and immediate.** A reference can be blessed as many times as you like; each call overwrites the previous class tag. There is no "unbless" built-in - rebless into a neutral package if you need to strip objecthood from a referent you still hold references to.
- **`CLASSNAME` of `""` means `main`.** Explicit and rarely wanted; usually a sign that a computed class name came out empty by accident.
- **Avoid `bless $ref, "0"`.** The class name `"0"` is a false value, so code that checks `if (ref $obj) { ... }` will misread the object as "not a reference". Pick any non-false class name.
- **Class-name casing convention.** Use mixed-case (`My::Package`). ALL-CAPS names are reserved for Perl built-in types (`SCALAR`, `ARRAY`, `HASH`, `CODE`, ...) and all-lowercase names are reserved for pragmas (`strict`, `warnings`, `utf8`). Blessing into either namespace invites confusion with language-level machinery.
- **Non-reference first argument is a fatal error.** `bless 42, 'Foo'` dies with `Can't bless non-reference value`. The first argument must be a reference.
- **One-argument `bless` picks the compile-time package.** It uses the package the `bless` call is *lexically* inside - not the caller's package, not the package of `$class`. That is exactly why it breaks inheritance and why the two-argument form exists.
- **Weak references and blessing interact as expected.** `Scalar::Util::weaken` on a blessed reference does not unbless it; the weakened slot still sees the class tag, and `ref` works until the referent is destroyed.
- **`DESTROY` fires on the blessed referent.** When the last reference to a blessed referent goes away, Perl calls `DESTROY` in the reference's class (following `@ISA`). Changing the class with a late `bless` changes which `DESTROY` runs.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `ref` - read the class tag back off a blessed reference, or the reference type (`HASH`, `ARRAY`, ...) for an unblessed one
- `package` - declare the package a `bless` call or a subroutine definition compiles into; the target of one-argument `bless`
- `class` - the experimental class-syntax built-in that replaces the hand-rolled `package` + `bless` constructor pattern with declarative syntax
- `tie` - a different form of per-variable magic; unrelated to `bless` despite the superficial "attach behaviour to a value" framing
- `@ISA` - the inheritance chain consulted when dispatching a method call on a blessed reference

- `DESTROY` - the destructor method Perl calls on the blessed referent when its last reference goes away; declared as a subroutine inside the class's package

Control flow

break

Break out of a `given` block.

`break` terminates the enclosing `given` block immediately and transfers control to the statement following it - the `given` analogue of what `last` does for loops. It takes no arguments, returns no value, and applies only to the innermost `given`; it cannot target an outer block by label.

Synopsis

```
break
```

Availability - read this first

`break` is a keyword of the `switch` feature, together with `given`, `when`, `default`, and (inside a `when`) `continue`. In Perl 5.44 the switch feature is **disabled by default** and is not part of any current feature bundle: `use v5.36` through `use v5.44` do **not** turn it on. The feature still exists and can be used, but only under explicit opt-in:

```
use feature 'switch';
no warnings 'experimental::smartmatch';
```

Without that opt-in, the bareword `break` is parsed as an ordinary subroutine call and fails at compile time with `Bareword "break" not allowed while "strict subs" in use`, or at runtime with `Undefined subroutine &main::break called`. The fully-qualified form `CORE::break` works from any scope without enabling the feature and is how you invoke it in code that otherwise keeps `switch` off.

`given`/`when` rely on smartmatch (`~~`), which is itself experimental and warns on every use unless `experimental::smartmatch` warnings are suppressed. New code should **not** reach for `given`/`when`/`break`; see *Replacements* below.

Behaviour

Inside a `given` block, `break` unwinds to the statement immediately after the block. It is equivalent to what falls through the bottom of a `when` branch implicitly - `when` clauses `break` at the end of their body unless the body ends in `continue`, so an explicit `break` is mostly used for early exit from a long `when` body:

```
use feature 'switch';
no warnings 'experimental::smartmatch';

given ($cmd) {
    when ('quit') {
        cleanup();
```

(continues on next page)

(continued from previous page)

```
        break if $fast_exit;        # skip the logging below
        log_exit();
    }
    when ('help') { show_help() }
    default       { warn "unknown: $cmd" }
}
```

Outside a `given` block, `break` raises `Can't "break" outside a given block` at runtime.

Replacements

For new code, prefer one of these forms. They are plain Perl, need no feature opt-in, and have no deprecation cloud over them:

- **`if` / `elsif` / `else` chain** - direct replacement for most `given`/`when` usage:

```
if    ($cmd eq 'quit') { cleanup(); log_exit() }
elsif ($cmd eq 'help') { show_help() }
else                   { warn "unknown: $cmd" }
```

- **Dispatch table** - a hash of coderefs keyed by the input, with a default branch. Scales better than a long `elsif` ladder and makes the set of handled cases data rather than control flow:

```
my %dispatch = (
    quit => sub { cleanup(); log_exit() },
    help => \&show_help,
);
($dispatch{$cmd} // sub { warn "unknown: $cmd" })->();
```

- **Labelled loop with `last`** - when you specifically want the "exit this block early" shape that `break` provided, wrap the body in a bare block and `last` out of it:

```
CMD: {
    if ($cmd eq 'quit') { cleanup(); last CMD if $fast_exit;
 ↪log_exit(); last CMD }
    if ($cmd eq 'help') { show_help(); last CMD }
    warn "unknown: $cmd";
}
```

Examples

Explicit feature opt-in, early-exit from a `when` body:

```
use feature 'switch';
no warnings 'experimental::smartmatch';

given ($n) {
    when ($_ < 0) { warn "negative"; break }   # skip the log below
```

(continues on next page)

(continued from previous page)

```
    when (0)      { warn "zero" }
    default       { warn "positive: $n" }
}
# control resumes here after break
```

Using `CORE::break` without enabling the feature globally:

```
sub classify {
    my ($n) = @_;
    given ($n) {
        when ($_ < 0) { return "neg"; CORE::break }
        when (0)      { return "zero" }
        default       { return "pos" }
    }
}
```

Edge cases

- **No argument form.** `break` takes nothing. `break LABEL` is a syntax error - unlike `last`/`next`, `break` has no way to target an outer block.
- **Outside `given`.** Calling `break` (or `CORE::break`) when no `given` block is on the call stack raises `Can't "break" outside a given block`.
- **`return` wins.** Inside a subroutine, `return` unwinds the whole sub regardless of any enclosing `given`. `break` only unwinds to the end of the `given`; control continues in the surrounding sub.
- **Implicit `break` at end of `when`.** A `when` clause's body ends with an implicit `break` unless the body is `continue`, so an explicit `break` at the end of a `when` body is redundant. `continue` suppresses the implicit `break` and lets control fall through to the next `when`.
- **Not a loop keyword.** `break` does not interact with `for`, `while`, `foreach`, `do`, or bare blocks. For those, use `last`.
- **Experimental warnings.** Running `given`/`when` code without `no warnings 'experimental::smartmatch'` prints the warning once per smartmatch site, which in a busy `given` block can be every iteration.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `last` - the loop counterpart; exits the innermost enclosing `for` / `while` / `foreach` / bare block, and unlike `break` can target an outer block by label
- `next` - start the next iteration of a loop rather than exiting it; useful when `break` is the wrong shape because you want continuation, not termination

- `return` - unwinds out of the enclosing subroutine entirely, not just the `given` block; use when the `given` is a dispatcher inside a sub and the answer is ready

Control flow · Scoping

caller

Return information about the subroutine, `eval`, or `require` that called the currently executing code.

`caller` walks the call stack. With no argument it returns a short three-element record identifying *who called us*. With a numeric argument `EXPR` it walks `EXPR` frames further back and returns a longer eleven-element record - the form the debugger, stack-trace modules, and `warn`/`die` machinery use to decorate diagnostics with file-and-line context. XS frames are invisible to `caller`; the next pure-Perl frame surfaces in their place.

Synopsis

```
caller
caller EXPR
```

What you get back

In scalar context, the caller's package name, or `undef` if there is no caller (i.e. we are at the top level of the file, not inside a `sub`, `eval`, or `require`):

```
my $pkg = caller;
```

In list context with no argument, three elements - the immediate caller's package, file, and line number:

```
# 0         1           2
my ($package, $filename, $line) = caller;
```

In list context with a numeric argument, eleven elements describing the frame `EXPR` levels up. `0` is the immediate caller, `1` is its caller, and so on:

```
# 0         1           2      3             4
my ($package, $filename, $line, $subroutine, $hasargs,

#   5           6           7            8       9         10
    $wantarray, $evaltext, $is_require, $hints, $bitmask, $hinthash)
  = caller($i);
```

The eleven fields, in order:

- `$package`, `$filename`, `$line` - location of the *call site* inside that frame's caller. Same three values the short form returns.
- `$subroutine` - fully qualified name of the sub the frame belongs to, e.g. `"Foo::bar"`. For an `eval` frame it is the literal string `"(eval)"`. If the sub has been removed from the symbol table since the call, it is `"(unknown)"`.

- `$hasargs` - true if a fresh `@_` was set up for the frame. False for `eval` and `require` frames.
- `$wantarray` - the context the frame was called in, as `wantarray` would have reported it: true for list, defined-but-false (`""`) for scalar, `undef` for void.
- `$evaltext` - for an `eval EXPR` frame, the source text that was compiled. Undef for `eval BLOCK` and for non-`eval` frames.
- `$is_require` - true if the frame was created by `require` or `use` (each `use` creates a `require` frame nested inside an `eval EXPR` frame).
- `$hints` - the value of `$^H` at the point the frame's code was compiled. Internal; layout is not stable across Perl versions.
- `$bitmask` - the value of `${^WARNING_BITS}` at the same point. Also internal.
- `$hinthash` - a reference to the `%^H` hash captured at compile time, or `undef` if it was empty. Do **not** modify it - the reference points at the live optree.

Global state it touches

- `@DB::args` - set as a side effect when `caller` is called with an argument *from inside the* `DB` *package*. The caller's argument list is aliased into this array for debugger use. See *Edge cases* below - this is best-effort and has sharp corners.
- `$^H`, `${^WARNING_BITS}`, `%^H` - read out of the target frame's compiled state to populate `$hints`, `$bitmask`, `$hinthash`.

Examples

Find the immediate caller's package and report to the right log:

```
sub log_for_caller {
    my $pkg = caller;
    warn "[$pkg] @_\n";
}
```

Produce a `warn`-style location tag. `caller` returns filename and line of the call site one frame up, which is almost always the information you actually want in a diagnostic:

```
sub note {
    my (undef, $file, $line) = caller;
    print STDERR "note at $file line $line: @_\n";
}
```

Walk the full stack. Iterate `caller($i)` until it returns an empty list:

```
sub stacktrace {
    my @frames;
    for (my $i = 0; my @f = caller($i); $i++) {
        push @frames, \@f;
    }
    return @frames;
}
```

Detect whether the current sub was called in list, scalar, or void context. This is `wantarray` at the current level, but for somebody else's frame you reach through `caller`:

```
sub describe_parent_context {
    my $wa = (caller(1))[5];
    return !defined $wa ? "void"
         :  $wa         ? "list"
         :               "scalar";
}
```

Tell a `require`/`use` frame apart from an ordinary `eval`:

```
sub called_from_require {
    my @f = caller(1);
    return @f && $f[7];            # field 7 is $is_require
}
```

Render a single frame the way the default `die` / `Carp` formatter does - `"at FILE line LINE"`:

```
sub location_tag {
    my $depth = shift // 0;
    my (undef, $file, $line) = caller($depth);
    return defined $file ? "at $file line $line" : "at top level";
}
```

Edge cases

- **Scalar context, no caller**: at the top level of a script, `scalar caller` returns `undef`, and list-context `caller` returns the empty list. A common idiom for "am I being `require`d as a module, or run as a script?" is `caller() ? ... : ...`.
- **XS frames are skipped**. `caller` only reports pure-Perl frames. If a pure-Perl sub was invoked by an XS sub which was invoked by another pure-Perl sub, `caller(0)` from the inner sub reports the outer pure-Perl sub - the XS layer is invisible. Stack depths measured with `caller` therefore do not include XS dispatchers.
- **`eval` frames have no args**. `$hasargs` is false for `eval` and `require` frames, and `$subroutine` is the literal `"(eval)"` - not the package of the surrounding code.
- **`eval BLOCK` vs `eval EXPR`**. Both give `$subroutine eq "(eval)"`. Only `eval EXPR` fills `$evaltext`; `eval BLOCK` leaves it `undef`.
- **`use` creates two frames**. A `use Foo` statement is compiled as `BEGIN { require Foo; Foo->import; }`, and `require` itself is implemented as an `eval EXPR`. Walking the stack through a `use` line therefore sees a `require`-flagged `eval` frame inside an ordinary `eval` frame. Check `$is_require` to distinguish.
- **`(unknown)` subroutines**. If the sub named in a frame has been deleted from the symbol table (symbol-table surgery, `delete $Foo::{bar}`) between the call and the `caller`, `$subroutine` reads `"(unknown)"`.
- **Loops and `try` blocks are not frames**. `while`, `for`, `foreach`, and `try` do not create call frames. `caller` only sees subroutine calls and `eval`s.

- **Optimiser-elided frames**. The optimiser may remove tail-call and similar frames before `caller` can observe them. For `caller(N)` with `N > 1`, the frame you get is not guaranteed to be the frame you expected. Do not encode stack depth as a load-bearing contract.
- **`@DB::args` is best-effort**. It is populated only when `caller` is called with an argument from inside package `DB`. The values are aliases into the target frame's `@_`, not copies - so:
  - Mutations the frame has already made to `@_` are visible; original call-time values are not preserved.
  - `shift @_` inside the frame is usually undone in `@DB::args`, but `pop @_`, splices, or having taken a reference to `@_` all break that.
  - Elements may have been freed and reused for other values by the time you read them.
  - Stale state from a previous `caller` call may still be in `@DB::args` if the current call didn't trigger a refresh. Use it for backtraces and debugger display; do not rely on it for program logic.
- **`$hinthash` is live**. The hash reference in field 10 points at the actual `%^H` snapshot baked into the optree. Modifying its contents mutates compiler state. Read it; don't write it.
- **Filename and line can be rewritten**. `#line` directives and anything else the "Plain Old Comments (Not!)" mechanism recognises (see `perlsyn`) change the values `caller` reports, just as they change `__FILE__` and `__LINE__`.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `die` - raises an exception; its default message uses the same file-and-line information `caller` exposes
- `eval` - creates frames that `caller` tags with `$subroutine eq "(eval)"` and may fill `$evaltext` / `$is_require`
- `wantarray` - reports the current frame's calling context; the same value appears as field 5 of `caller(N)` for the frame `N` levels up
- `$^H` - compile-time hint bits; surfaced as field 8
- `%^H` - compile-time hint hash; surfaced as field 10
- `${^WARNING_BITS}` - warnings bitmask; surfaced as field 9

Control flow

catch

Handle an exception thrown by a preceding `try` block.

`catch` is the second half of the `try`/`catch` construct. It is not a standalone statement: it follows a `try` block, declares a lexical `VAR` to receive the exception value, and runs its `BLOCK` only when the `try` block died. If the `try` block completed normally the `catch` block is skipped. Control then resumes at the statement following the whole construct (or the `finally` block, if one is present).

This syntax must be enabled with `use feature 'try'`. `try` and `catch` are non-experimental as of Perl 5.40.

Synopsis

```
try BLOCK catch (VAR) BLOCK
try BLOCK catch (VAR) BLOCK finally BLOCK
```

What you get back

`catch` is not a function call and has no return in the expression sense. Like other control-flow constructs, the value of the whole `try`/`catch` is the value of whichever block ran last - useful inside a `do` block or as the tail expression of a subroutine:

```
my $value = do {
    try {
        fetch_thing(@args);
    }
    catch ($e) {
        warn "fetch failed: $e";
        $DEFAULT;
    }
};
```

If `try` ran to completion, the construct yields the `try` block's last value and `VAR` is never bound. If `try` died, the construct yields the `catch` block's last value.

The VAR declaration

The parentheses after `catch` are mandatory and must contain exactly one scalar variable declaration. It behaves like a `my` declaration - the `my` keyword is implicit, as in subroutine signatures:

```
catch ($e)  { ... }         # $e is a new lexical
catch (my $e) { ... }       # syntax error - 'my' is already implied
```

`VAR` is scoped to the `catch` block only. It is not visible in the preceding `try` block, in any following `finally` block, or after the construct. Its value is whatever the `try` block threw - a string, a list that stringified into one, or a reference, typically a blessed exception object. See `die` for how the exception value is produced.

Unlike `eval`, `catch` does **not** read or clear `$@`; the exception travels through VAR directly. `$@` is untouched by the construct.

Global state it touches

`catch` itself touches no special variables. The `try`/`catch` construct as a whole leaves `$@` unchanged - a deliberate divergence from `eval` intended to stop callers from conflating the two idioms. If you need `$@`-style inspection, use `eval` instead.

Examples

Catch a simple string exception and report it:

```
use feature 'try';

try {
    risky();
}
catch ($e) {
    warn "risky failed: $e";
}
```

Inspect a structured exception object:

```
try {
    process($record);
}
catch ($e) {
    if (ref($e) && $e->isa('My::NotFound')) {
        return;                         # missing is fine
    }
    die $e;                             # re-throw everything else
}
```

Re-throw unchanged with `die`. Because `catch` does not touch `$@`, you re-throw the variable you bound, not `$@`:

```
try {
    step();
}
catch ($e) {
    log_error($e);
    die $e;                             # propagates to the next␣
↪handler
}
```

Produce a value from the construct without `return`. Using `return` inside `try` or `catch` returns from the **enclosing subroutine**, not from the construct - see *Edge cases*:

```
sub safe_parse {
    my ($text) = @_;
```

(continues on next page)

(continued from previous page)

```
    try {
        parse($text);
    }
    catch ($e) {
        warn "parse error: $e";
        undef;                          # value of the catch block
    }
}
```

A `catch` block that itself dies - the new exception propagates outward to the next enclosing handler (`try`/`catch`, `eval`, or program exit):

```
try {
    open_db();
}
catch ($e) {
    cleanup();
    die "open_db failed: $e";          # thrown from inside catch
}
```

Edge cases

- **Cannot stand alone.** `catch (VAR) { ... }` without a preceding `try` block is a syntax error. `catch` is parsed as part of the `try` construct, not as an independent keyword.
- **Exactly one scalar in parentheses.** The parentheses are mandatory, must declare exactly one scalar, and the `my` keyword is implied. `catch { ... }` without parentheses is a syntax error.
- **`VAR` is block-scoped.** It exists only inside the `catch` block. A `finally` block that follows does **not** see it.
- **Falsy exceptions still trigger the handler.** `try` distinguishes "died" from "completed normally" by a flag set at unwind time, not by truthiness of the exception value. `die 0`, `die ""`, and `die undef` all enter the `catch` block; `VAR` receives the value as thrown (with `die`'s usual coercion of `undef` to `"Died at ..."`).
- **`$@` is not set.** Code that reaches for `$@` inside a `catch` block is reading the caller's `$@`, not the exception just caught. Use `VAR`.
- **A `catch` block may die.** Exceptions raised from inside `catch` are **not** re-caught by the same construct. They propagate to the next enclosing handler. This is the mechanism for re-throw (`die $e`) and for reporting cleanup failures from the handler itself.
- **`return` inside `catch` returns from the enclosing subroutine**, not from the `catch` block. This differs from `eval` BLOCK, where `return` would leave only the `eval`. To yield a value *from* the construct, use the block's final expression instead of `return`.
- **Loop controls (`last`, `next`, `redo`) inside `catch`** target an enclosing loop, not the construct. The `try`/`catch` construct is not itself a loop.

- **`caller()` does not see the `catch` frame.** Because `catch` does not intercept `return`, it is invisible to `caller`, like `while` or `foreach` are.
- **Pairs only with the immediately preceding `try`.** You cannot have two `catch` blocks, nor place statements between `try` and `catch`. The optional `finally` block, if present, must come after `catch`.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `try` - the block whose failure this `catch` handles; always paired with exactly one `catch`
- `finally` - optional third block that runs regardless of whether the `try` block died; cannot `return` or use loop controls
- `die` - produces the exception value that `VAR` receives; use it inside `catch` to re-throw
- `eval` - older exception-handling idiom using `$@`; reach for `try`/`catch` instead in new code for clearer scoping and no `$@` footguns
- `$@` - left untouched by `try`/`catch`; inspect `VAR` instead

Filehandles, files, directories

chdir

Change the process's current working directory.

`chdir` is the Perl wrapper around the `chdir(2)` and - where the kernel supports it - `fchdir(2)` system calls. It changes the directory that relative paths are resolved against for the current process and, on success, persists that change for every subsequent file operation in the program.

Synopsis

```
chdir EXPR
chdir FILEHANDLE
chdir DIRHANDLE
chdir
```

What you get back

1 on success, `0` on failure. On failure `$!` is set to the underlying `errno` so you can distinguish "no such directory" (`ENOENT`) from "permission denied" (`EACCES`) from "not a directory" (`ENOTDIR`):

```
chdir $path
    or die "chdir $path failed: $!";
```

The return is always a plain boolean - there is no separate "changed to nowhere because no argument was given" value. The no-argument form either succeeds by changing to the home directory or fails like any other call.

Global state it touches

- `%ENV` - read when `chdir` is called with no argument. The lookup order is `$ENV{HOME}` first, then `$ENV{LOGDIR}`. If neither is set, `chdir` does nothing and returns `0`.
- `$!` - set on failure to the `errno` reported by the kernel.

`chdir` does **not** update `$ENV{PWD}`. Shells maintain `PWD` themselves; a Perl program that cares about `PWD` for child processes has to assign to `$ENV{PWD}` after a successful `chdir`.

Argument forms

- **`chdir EXPR`** - `EXPR` is stringified and passed to `chdir(2)` as a pathname. Relative paths are resolved against the current working directory at the moment of the call.
- **`chdir FILEHANDLE`** and **`chdir DIRHANDLE`** - on Linux (and any system with `fchdir(2)`) the kernel changes directory to the one the handle refers to. `FILEHANDLE` here means a filehandle opened on a directory, or a directory handle from `opendir`. Using the handle form avoids a TOCTOU race between resolving a pathname and changing into it.
- **`chdir`** (no argument) - changes to `$ENV{HOME}`, falling back to `$ENV{LOGDIR}` if `HOME` is unset. If both are unset the call fails and `$!` is left untouched; check the return value, not `$!`.

Examples

Change into a directory and bail out if it does not exist:

```
chdir "/var/log"
    or die "cannot enter /var/log: $!";
```

Run a block of work in a different directory and return to where you started:

```
use Cwd qw(getcwd);
my $origin = getcwd;
chdir $work_dir or die "chdir $work_dir: $!";
# ... do work relative to $work_dir ...
chdir $origin  or die "chdir $origin: $!";
```

Change via a directory handle (race-free on systems with `fchdir(2)`):

```
opendir my $dh, $path or die "opendir $path: $!";
chdir $dh              or die "fchdir $path: $!";
```

Go home:

```
chdir or die "no HOME/LOGDIR in environment";
```

Probe a list of candidates, stopping at the first that works:

```
for my $d ("/srv/app", "/opt/app", "/usr/local/app") {
    last if chdir $d;
}
```

Edge cases

- **Empty string or undef**: `chdir ""` and `chdir undef` both fail and set `$!` to `ENOENT`. They do **not** fall back to `$HOME` - the no-argument form is triggered by *absence* of an argument, not by an argument that happens to be empty.
- **Relative paths on a thread with its own cwd**: the kernel's current working directory is per-process, not per-thread, on Linux. Two threads calling `chdir` concurrently race.
- **Symlink to a directory**: `chdir` follows symlinks, landing in the target. Use `readlink` or `Cwd::abs_path` first if you need to know the real destination.
- **Deleted directory**: if the directory was removed after you entered it, relative operations from inside fail with `ENOENT` and most shells show the path as `(deleted)`. You can still `chdir` out with an absolute path.
- **Handle form on systems without `fchdir(2)`**: raises an exception. Linux always supports it; the restriction is a portability concern for code that also runs on older systems.
- **Untrusted input**: passing a user-supplied path to `chdir` without validation lets the caller redirect all subsequent relative-path operations in the program. Canonicalise with `Cwd::abs_path` and check against an allow-list before the call if the input crosses a trust boundary.
- **After `fork`**: the child inherits the parent's cwd. A `chdir` in the child does not affect the parent.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `mkdir` - create a directory; pair with `chdir` when building a workspace and stepping into it
- `rmdir` - remove a directory; do not `rmdir` the cwd, `chdir` out first
- `opendir` - open a directory handle usable as the argument to `chdir` for race-free directory changes
- `chroot` - stronger confinement than `chdir`; changes the filesystem root for the process, not just the cwd

- `$ENV{HOME}` - consulted by the no-argument form of `chdir` before falling back to `$ENV{LOGDIR}`
- `$!` - carries the `errno` when `chdir` returns `0`

Filehandles, files, directories

chmod

Change the permission bits of a list of files.

`chmod` sets each file in `LIST` to the permission `MODE`. `MODE` is a **numeric** value - the low twelve bits are the usual Unix permission triad plus the setuid, setgid, and sticky bits, exactly as accepted by `chmod(2)`. Both plain pathnames and open filehandles are accepted in `LIST`. Returns the number of files for which the underlying system call succeeded.

Synopsis

```
chmod MODE, LIST
```

What you get back

The number of files successfully changed, as an integer. Zero means every call failed; a value less than `scalar @files` means a partial failure. `chmod` does **not** throw on a per-file failure - it keeps going and tallies the successes.

`$!` reflects the error from the **last** failing file only. On a mixed success/failure run there is no built-in way to learn which files failed; if you need per-file error reporting, call `chmod` once per file and check the return value each time:

```
for my $f (@files) {
    chmod 0644, $f
        or warn "chmod $f: $!";
}
```

Writing MODE correctly

`MODE` is an integer. The three ways to spell it:

- **Octal literal with leading zero**: `0644`, `0755`, `0600`. This is the conventional form and matches the digits you would type into `chmod(1)`.
- **Octal literal with `0o` prefix**: `0o644`, `0o755`. Accepted since Perl 5.34; equivalent to the leading-zero form and easier to read as "octal" at a glance.
- **Symbolic constants from `Fcntl`**: `use Fcntl qw(:mode); S_IRWXU | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH` evaluates to `0755`. Useful when a permission set is built up from pieces.

A string argument is almost always a bug:

```
chmod "0644", "foo";     # WRONG - stringifies as decimal 644 (octal␣
↪1204),
                         #         grants mode --w----r-T
chmod 644,    "foo";     # WRONG - decimal 644 is octal 1204, same␣
↪result
chmod 0644,   "foo";     # right
chmod 0o644,  "foo";     # right (Perl 5.34+)
```

If the mode genuinely arrives as a string (configuration file, CLI argument), convert it with `oct` first:

```
my $mode = "0644";       # string from config
chmod oct($mode), "foo"; # oct("0644") == 0644
```

Global state it touches

Sets `$!` on any failure to match the last failing `chmod(2)` / `fchmod(2)` call. Reads no interpreter globals.

Examples

Change one file, check for success:

```
chmod 0755, "install.sh"
    or die "chmod install.sh: $!";
```

Change many files in a single call, capture the count:

```
my $cnt = chmod 0644, "a.txt", "b.txt", "c.txt";
print "updated $cnt of 3 files\n";
```

With an array of pathnames:

```
my @executables = glob "bin/*";
chmod 0755, @executables;
```

Preserve existing bits, add owner write. Reads the mode with `stat`, masks to the permission bits, ORs in the new bit, and writes it back:

```
open my $fh, "<", "foo" or die $!;
my $perm = (stat $fh)[2] & 07777;
chmod $perm | 0200, $fh;
```

Symbolic form via `Fcntl`:

```
use Fcntl qw(:mode);
chmod S_IRWXU | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH, @executables;
# identical to: chmod 0755, @executables;
```

Per-file error reporting (needed to learn *which* file failed on a multi-file call):

```
my @failed;
for my $f (@files) {
    push @failed, $f unless chmod 0644, $f;
}
warn "chmod failed on: @failed\n" if @failed;
```

Edge cases

- **Empty list**: `chmod 0644` returns `0`. The mode is evaluated but no system call is made.
- **String modes**: `chmod "0644", $f` and `chmod "+x", $f` are both wrong. `chmod` never parses symbolic mode strings - unlike `chmod(1)`, the built-in takes a number, not a specification.
- **Filehandles vs barewords**: a bareword in `LIST` is a filename, not a filehandle. Pass a filehandle as a glob or glob reference:

  ```
  chmod 0644, *FH;          # filehandle
  chmod 0644, \*FH;         # filehandle
  chmod 0644, $fh;          # filehandle (lexical)
  chmod 0644, "FH";         # filename "FH" - almost certainly a
   ↪bug
  ```

  On systems without `fchmod(2)`, passing a filehandle raises an exception. Linux supports `fchmod(2)`, so on the platforms pperl targets this always works.
- **Setuid / setgid / sticky bits**: the high three bits of the mode word (`04000`, `02000`, `01000`) set setuid, setgid, and the sticky bit respectively. `0o4755` makes a setuid executable; `0o2775` a setgid directory.
- **Symlinks**: `chmod` follows symlinks. Changing permissions on the symlink itself is not portable and is not exposed by this built-in - use `lchmod` from a module if the platform supports it.
- **Permission to change permission**: only the file's owner (and root) may call `chmod(2)` on a file. Failure sets `$!` to `EPERM`.
- **Non-existent paths**: `chmod` on a missing path sets `$!` to `ENOENT` and contributes `0` to the success count. It does not warn and does not die.
- **umask does not apply**: `umask` masks the mode of *newly created* files from `open`, `sysopen`, `mkdir`, and similar. `chmod` sets the mode literally - no masking.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `chown` - the sibling call for changing owner/group; same list-and-return-count shape
- `stat` - read the current mode so you can modify it rather than replace it wholesale

- `umask` - default permission mask applied to *new* files; unrelated to `chmod`, which sets bits literally
- `open` / `sysopen` - create files; the mode argument to `sysopen` is masked by `umask` at creation time and can then be tightened or relaxed with `chmod`
- `oct` - parse `"0644"` into the integer `0644` when the mode arrives as a string
- `$!` - system error from the last failing file; only meaningful for single-file calls or immediately after a failing per-file call in a loop

SCALARs and strings

chomp

Strip the trailing input record separator from a string in place.

`chomp` removes any trailing substring that matches the current value of `$/` from its argument and returns the total number of characters removed across every argument it touched. It mutates its argument; the return value is a count, not the chomped string. If no argument is given, it operates on `$_`. `chomp` is the safer counterpart to `chop`: it only removes the separator if it is actually there, and it respects `$/` so it does the right thing on files that end without a final newline or that use a non-newline record terminator.

Synopsis

```
chomp VARIABLE
chomp( LIST )
chomp
```

What you get back

The total number of characters removed, as an integer. Usually `0` (no separator was present) or `1` (a single `"\n"` was removed under the default `$/`). When `$/` is a multi-character string, a successful chomp returns its length. When chomping a list, the count is summed across every element.

The chomped value lives in the variable you passed; `chomp` is called for effect, not for its return value:

```
my $line = "hello\n";
my $n = chomp $line;        # $line is "hello", $n is 1
```

Global state it touches

- `$/` - **input record separator**. `chomp` reads `$/` on every call and strips a trailing occurrence of its current value. Changing `$/` changes what `chomp` removes; this is usually what you want when you set `$/` to read custom-delimited records.
- `$_` - consumed as the default argument when `chomp` is called with no argument. `chomp;` inside `while (<>) { ... }` is the canonical idiom.
- Hash iterator - chomping a hash resets its `each` iterator (see *Edge cases*).

Examples

The everyday use - strip the trailing newline from input:

```
while (my $line = <$fh>) {
    chomp $line;
    # process $line without the trailing "\n"
}
```

The no-argument form operates on `$_`:

```
while (<>) {
    chomp;                              # removes "\n" from $_
    my @fields = split /:/;
}
```

Chomp an lvalue assignment - parentheses are **required** so the argument to `chomp` is the whole assignment, not just the variable:

```
chomp(my $cwd    = `pwd`);              # strip "\n" from command output
chomp(my $answer = <STDIN>);            # strip "\n" from user input
```

Chomping a list - every element is chomped and the total count is returned:

```
my @lines = ("a\n", "b\n", "no-nl");
my $n = chomp @lines;                   # @lines = ("a", "b", "no-nl"),
 ↪$n = 2
```

Custom record separator - `chomp` follows whatever `$/` is set to:

```
local $/ = "\r\n";
my $line = "GET / HTTP/1.0\r\n";
chomp $line;                            # removes "\r\n", returns 2
```

Paragraph mode - `$/ = ""` makes `chomp` remove **all** trailing newlines from the string:

```
local $/ = "";
my $para = "line one\nline two\n\n\n";
chomp $para;                            # $para is "line one\nline two"
```

Edge cases

- **No separator present**: a no-op. Returns `0` and leaves the argument unchanged. This is the whole point of `chomp` over `chop` - safe to call on strings that may or may not end in `$/`.
- **`undef` argument**: chomping an undefined scalar leaves it undefined and returns `0`. Under `use warnings` this emits a `uninitialized` warning.
- **Hash argument**: `chomp %h` chomps every value (not the keys) and **resets the hash's `each` iterator**. A `while (my ($k, $v) = each %h)` loop in progress over `%h` will be disrupted.

- **Slurp mode (`$/ = undef`)** and **fixed-length record mode** (`$/` set to a reference to an integer): `chomp` removes nothing and returns `0`. There is no notion of a trailing separator in these modes.
- **Parenthesisation trap on assignment**: without parens, `chomp` binds tighter than `=`:

```
chomp  $cwd = `pwd`;                    # parsed as (chomp $cwd) =␣
 ↪`pwd`;
chomp( $cwd = `pwd` );                  # what you meant
```

  Same trap with a bare comma list:

```
chomp  $x, $y;                          # parsed as chomp($x), $y;
chomp( $x, $y );                        # chomps both
```

- **Any lvalue works**, not just plain scalars. Array and hash elements, slices, dereferences, and assignment expressions are all legal targets.
- **Constants and read-only values**: chomping a read-only string croaks with `Modification of a read-only value attempted`. String literals are read-only.
- **Multi-byte `$/`**: `chomp` compares bytes of the string against bytes of `$/`. If the string and `$/` disagree on encoding (one has the UTF-8 flag set, the other does not), the match is done on the internal byte representation and may surprise you. Keep both sides in the same encoding.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `chop` - unconditionally remove the **last character** and return it; older, blunter tool superseded by `chomp` for input cleanup
- `split` - often the next step after `chomp`, splits the now-trimmed record into fields
- `readline` / `<>` - reads records terminated by `$/`; `chomp` undoes the terminator that `readline` left in place
- `each` - the hash iterator that `chomp %hash` resets
- `$/` - input record separator; `chomp` reads this on every call
- `$_` - default argument when `chomp` is called with no explicit target

SCALARs and strings

chop

Remove the last character from a string and return it.

`chop` shortens its argument in place by exactly one character and returns the character that was removed. It does not scan the string and does not copy it - it simply moves the end. If `VARIABLE` is omitted, `chop` operates on `$_`. If `VARIABLE` is a hash, `chop` chops

every value of the hash (not the keys) and resets the hash's `each` iterator as a side effect. Any lvalue is a valid target, including an assignment expression.

Synopsis

```
chop VARIABLE
chop( LIST )
chop
```

What you get back

The character that was removed, as a string. If the target was empty or undefined, the empty string is returned and the target is left unchanged. When `chop` is applied to a list, every element is chopped but only the character removed from the **last** element is returned.

```
my $last = chop $line;     # one character, or "" if $line was empty
```

Global state it touches

Reads `$_` when called with no argument. No other special variables are consulted - in particular, `chop` is **not** affected by `$/` (the input record separator), which is `chomp`'s concern.

Examples

Chop a scalar:

```
my $s = "hello";
my $c = chop $s;            # $s is now "hell", $c is "o"
```

Chop `$_` when no argument is given - idiomatic inside a read loop that wants the terminator gone regardless of what it is:

```
while (<$fh>) {
    chop;                   # drops whatever character ends the line
    push @rows, $_;
}
```

Chop a list. Every element loses its last character; the return value is the character removed from the **last** element only:

```
my @lines = ("foo\n", "bar\n", "baz!");
my $last  = chop @lines;    # @lines is ("foo","bar","baz"), $last
↪is "!"
```

Chop a hash. Values are chopped, keys are not, and the `each` iterator is reset:

```
my %h = (a => "one\n", b => "two\n");
chop %h;                    # %h is (a => "one", b => "two")
```

Chop an assignment - chop works on any lvalue, and an assignment returns one:

```
chop(my $cwd = `pwd`);      # strip the trailing newline from pwd's␣
↪output
```

Keep all but the last character without modifying the original - not a job for chop, which mutates. Use substr with a negative length instead:

```
my $head = substr($s, 0, -1);   # $s is unchanged
```

Edge cases

- **Empty or undefined target**: returns the empty string "" and leaves the target alone. chop on undef does *not* raise a warning by itself; it simply produces "".
- **Read-only target**: raises Modification of a read-only value attempted. This includes string literals and values flagged read-only via Internals::SvREADONLY.
- **UTF-8 strings**: chop removes one **character**, not one byte. A multi-byte character at the end of the string is removed whole and returned as a UTF-8 string. The target's UTF-8 flag is preserved on what remains.
- **List return value**: only the character chopped from the last element is returned; the others are discarded. If you need every removed character, chop the elements individually in a loop.
- **One character vs. a line ending**: chop unconditionally removes **one** character, whatever it is - including spaces, digits, or letters. To conditionally remove a line terminator (what $/ says one looks like), use chomp.
- **Parser parenthesisation**: chop($x = $y) chops the result of the assignment, which is $x. chop $x, $y is parsed as chop($x), $y - the comma does not extend chop's argument list without parentheses. Write chop($x, $y) when you mean the list form.
- **Hash iteration reset**: because chop %h walks the values via the hash iterator, any in-progress each loop over %h is invalidated.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- chomp - conditional cousin: removes a trailing $/, not an unconditional last character; returns the number of characters removed rather than the character itself
- substr - non-destructive alternative; substr($s, 0, -1) returns all but the last character without mutating $s
- s///-$s =~ s/.\z//s is the regex equivalent and the one chop is explicitly faster than, because chop neither scans nor copies
- lc, uc - other in-place(ish) string transforms that accept any lvalue

- `$_` - the implicit target when `chop` is called with no argument
- `$/` - **not** consulted by `chop`; documented here only to flag the contrast with `chomp`

Filehandles, files, directories

chown

Change the owner and group of a list of files.

`chown` takes a numeric user id, a numeric group id, and a list of files, and reassigns ownership on every file that the calling process is allowed to touch. The first two elements of `LIST` are the new `UID` and `GID`, in that order; every element after that is a filename (or, on systems that support `fchown(2)`, a filehandle). A value of `-1` in either of the id slots means "leave that id unchanged" - useful when you want to change only the group, or only the owner.

Synopsis

```
chown UID, GID, LIST
chown $uid, $gid, $file
chown $uid, $gid, @files
chown -1, $gid, @files          # change group only
```

What you get back

The number of files whose ownership was successfully changed, as a plain integer. Not a boolean - a call over ten files that succeeds on seven returns 7. Check against the list length when you need all-or-nothing semantics:

```
my @files = ('a', 'b', 'c');
my $ok = chown $uid, $gid, @files;
$ok == @files
    or die "chown failed on ", scalar(@files) - $ok, " files: $!";
```

When any individual file fails, `$!` holds the errno from the **last** failing call. A return value less than `scalar @files` is the reliable "something went wrong" signal; `$!` tells you what the last failure was, not every failure.

Global state it touches

- `$!` - set to the errno of the last underlying `chown(2)` call that failed. Unchanged when every file succeeds.

Numeric ids only

`chown` does **not** accept user or group names. Pass the string `"root"` where a uid is expected and you get a numeric conversion of `0` (with an `Argument … isn't numeric` warning under `use warnings`) - almost never what you want. Translate names to numbers with `getpwnam` and `getgrnam`:

```
my $uid = getpwnam('alice')  // die "no such user";
my $gid = getgrnam('staff')  // die "no such group";
chown $uid, $gid, @files;
```

The list-context return of `getpwnam` gives the full passwd entry; in scalar context it gives just the uid, which is usually what `chown` wants.

Permission rules

On Linux and every other POSIX system, only the superuser can give a file away to a different owner. An unprivileged user can change the **group** of a file they own, but only to one of their own secondary groups. Trying to do more returns failure with `$!` set to `EPERM`.

```
chown 0, 0, 'passwd.new'
    or warn "need root to chown: $!";
```

The restriction is kernel-enforced, not Perl-enforced - there is no pre-check that would save you a failed system call. Rely on the return value.

Examples

Change both owner and group on a single file:

```
chown $uid, $gid, '/var/log/app.log'
    or die "chown failed: $!";
```

Change the group only, leaving the owner alone:

```
chown -1, $gid, @files;
```

Translate names to numeric ids before calling:

```
my ($login, $pass, $uid, $gid) = getpwnam('www-data')
    or die "www-data not in passwd file";
chown $uid, $gid, glob('/var/www/*');
```

Count how many files were actually reassigned:

```
my @targets = glob('/srv/data/*.dat');
my $changed = chown $uid, $gid, @targets;
warn "only $changed of ", scalar(@targets), " files changed: $!"
    if $changed != @targets;
```

Chown by filehandle on a system that supports `fchown(2)`. The filehandle must be a glob or a glob reference - a bareword is treated as a filename:

```
open my $fh, '>', '/tmp/out' or die $!;
chown $uid, $gid, $fh;
```

Edge cases

- **Empty file list**: `chown $uid, $gid;` - no files to act on, returns `0`, does not touch `$!`.
- **`-1` in id position**: standard POSIX sentinel meaning "leave this id unchanged." Works on every supported platform.
- **String id**: a non-numeric string silently becomes `0` (root / wheel) after the usual numeric conversion. Under `use warnings` you get an `Argument isn't numeric` warning, but there is no hard error. Always pass an integer.
- **Non-existent file**: counted as a failure, does not abort the call. Remaining files still get processed.
- **Symlink target**: `chown` follows symlinks and changes the pointed-to file. To chown the link itself, use `lchown(2)` via `syscall` or a module that exposes it - core Perl does not.
- **Filehandle form**: on systems without `fchown(2)` (rare in practice on Linux), passing a filehandle raises an exception. Barewords in the file list are always treated as filenames - use `\*FH` or a `my $fh` to mean "this handle."
- **Give-away restriction**: on POSIX systems with `_PC_CHOWN_RESTRICTED` set for the target path, only root can change the owner at all. Probe with:

```
use POSIX qw(pathconf _PC_CHOWN_RESTRICTED);
my $restricted = pathconf($path, _PC_CHOWN_RESTRICTED);
```

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `chmod` - change file permission bits; takes the same "mode, then file list" shape and returns the same number-of-files-changed count
- `stat` - read the current owner, group, and mode of a file before deciding what to change
- `umask` - set the default permission mask for files created by this process; complements `chown` for newly created files
- `getpwnam` - translate a user name to the numeric uid `chown` requires
- `getgrnam` - translate a group name to the numeric gid `chown` requires
- `$!` - holds the errno of the last failing per-file `chown(2)` when the return count is below the list length

SCALARs and strings

chr

Return the character whose codepoint is the given number.

`chr` takes a non-negative integer and returns a one-character string holding the character at that codepoint in the current character set. `chr(65)` is `"A"` in both ASCII and Unicode; `chr(0x263a)` is the Unicode smiley face ☺. If `NUMBER` is omitted, `chr` operates on `$_`. The inverse operation is `ord`.

Synopsis

```
chr NUMBER
chr
chr($n)
```

What you get back

A string of length one character. With Unicode semantics - the default for any codepoint above `0x7f` - that character may be encoded internally as several bytes of UTF-8, so `length` reports `1` while `bytes::length` can report up to four. For codepoints in the range `0..127` the result is a single-byte ASCII string and the two lengths agree.

Global state it touches

With no argument, `chr` reads `$_`. It writes nothing. The `bytes` pragma changes what `chr` does with out-of-range or negative input (see *Edge cases*); no other pragma or special variable affects it.

Examples

ASCII character from its codepoint:

```
print chr(65);              # A
print chr(97), "\n";        # a
```

Build a string from a list of codepoints with `map` and `join`:

```
my @code = (72, 101, 108, 108, 111);
print join("", map { chr } @code), "\n";   # Hello
```

Unicode codepoint above the ASCII range - the result is one character, four bytes in its internal UTF-8 form:

```
use utf8;
my $smiley = chr(0x263a);
print length($smiley);                      # 1
use bytes; print length($smiley);           # 3
```

Round-trip with `ord` - for any single character `$c`, the identity `chr(ord($c)) eq $c` holds:

```
my $c = "Z";
print chr(ord($c)) eq $c ? "yes" : "no";     # yes
```

Default-argument form inside a loop over `$_`:

```
for (65, 66, 67) {
    print chr;                               # prints ABC
}
```

Negative input yields the Unicode replacement character `U+FFFD`:

```
print chr(-1) eq "\x{fffd}" ? "repl" : "?";  # repl
```

Edge cases

- **Fractional argument**: truncated toward zero before lookup. `chr(65.9)` returns `"A"`.
- **Non-numeric string**: converted by the normal numeric-coercion rules, with a warning under `use warnings`. `chr("65abc")` returns `"A"`.
- **`chr(undef)`** returns `chr(0)`, the `NUL` character `"\0"`, and warns under `use warnings` about use of an uninitialized value.
- **Negative values**: outside the `bytes` pragma, negative input returns the Unicode replacement character `"\x{fffd}"`. Under `use bytes`, the low eight bits of the integer value are used: `chr(-1)` yields the byte `"\xff"`.
- **Values above `0x10ffff`** (the maximum Unicode codepoint) still return a single-character string under default semantics, but emit a `Unicode non-character is illegal for interchange` warning for the non-character ranges, and a `Unicode surrogate U+...` warning for the surrogate range `0xd800..0xdfff` when the string is later encoded.
- **The 128..255 range**: characters in this range are by default **not** stored as UTF-8 internally, for backward compatibility with byte strings. They compare equal to their single-byte form and to the same Unicode codepoint; the internal representation only matters when inspecting with `bytes::length` or interacting with PerlIO layers.
- **`chr` never croaks** on any numeric input. Errors surface only as warnings, never exceptions.
- **Not an lvalue**: `chr` returns a fresh scalar. `chr(65) = "B"` is a syntax error.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `ord` - inverse of `chr`: codepoint of the first character of a string
- `sprintf` - `sprintf "%c", $n` is equivalent to `chr $n`; use it to splice a character into a larger format at once
- `pack` - build a multi-character string from a list of codepoints in one call (`pack "U*", @code` for Unicode, `pack "C*", @code` for bytes)
- `hex` - parse a hex string into the integer you then pass to `chr`; pairs with `chr` when decoding `U+XXXX` notation
- `use bytes` - pragma that switches `chr` to byte-oriented semantics for values outside `0..0x7f`
- `perlunicode` - background on how Perl stores and compares Unicode strings

Filehandles, files, directories

chroot

Change the root directory of the current process and every child it later spawns.

After a successful `chroot`, any pathname that begins with `/` is resolved relative to the directory named by `FILENAME` instead of the real filesystem root. The process loses the ability to name files outside that subtree through absolute paths. The kernel enforces this for the current process and for every descendant - there is no per-thread scope and no way to undo it from inside the jail.

`chroot` does **not** change the current working directory. If the working directory was outside the new root when the call succeeded, it stays outside, and relative paths can still escape. Always follow a successful `chroot` with `chdir` to `/` (which now means inside the jail) before doing anything else.

The call is restricted to the superuser. An unprivileged process gets `EPERM` and `$!` is set accordingly.

Synopsis

```
chroot FILENAME
chroot
```

What you get back

`1` on success, `0` on failure. On failure `$!` holds the `errno` from the underlying `chroot(2)` system call - commonly `EPERM` (not root), `ENOENT` (path does not exist), or `ENOTDIR` (path is not a directory).

Always check the return value. A silent failure leaves the process running outside the intended jail, which is usually worse than the code path that made the call in the first place.

```
chroot "/var/empty"
    or die "chroot: $!";
chdir "/"
    or die "chdir after chroot: $!";
```

Global state it touches

- `$_` - when called with no argument, `chroot` uses the value of `$_` as the path. Inside a `while (<>)` loop this is rarely what you want; pass the path explicitly.
- `$!` - set to the underlying `errno` on failure.

Examples

The canonical safe sequence - change root, then change working directory to the new root, then drop privileges:

```
chroot "/srv/jail"   or die "chroot: $!";
chdir  "/"           or die "chdir: $!";
$) = $(= 65534;                         # drop to 'nobody' gids
$> = $< = 65534;                        # drop to 'nobody' uids
```

Open files and load shared libraries **before** `chroot`, not after. The new root typically has neither `/etc/resolv.conf` nor the dynamic linker's search paths:

```
open my $log, ">>", "/var/log/app.log" or die $!;
my $cfg = read_config("/etc/app.conf");

chroot "/srv/jail" or die "chroot: $!";
chdir  "/"         or die "chdir: $!";

# $log and $cfg are still usable inside the jail.
```

Checking failure modes separately - distinguishing "not root" from "bad path" matters for diagnostics:

```
use Errno qw(EPERM ENOENT ENOTDIR);

unless (chroot $path) {
    die "chroot: need root privileges\n"       if $! == EPERM;
    die "chroot: $path does not exist\n"       if $! == ENOENT;
    die "chroot: $path is not a directory\n"   if $! == ENOTDIR;
    die "chroot: $!";
}
```

Default-argument form reads the path from `$_`:

```
$_ = "/srv/jail";
chroot or die "chroot: $!";
```

Edge cases

- **Not root**: fails with `EPERM`. There is no capability-based relaxation in stock Linux unless the process holds `CAP_SYS_CHROOT`.
- **Relative path**: `chroot "jail"` resolves against the current working directory at the moment of the call. Prefer absolute paths so the behavior does not depend on the caller's `cwd`.
- **Symlinks in `FILENAME`**: resolved before the chroot takes effect. Symlinks *inside* the new root that point to absolute paths will resolve relative to the new root, not the real root - this is the intended jailing behavior, not a bug.
- **Open file descriptors survive**: any handle opened before the call remains usable afterward. This is how well-behaved daemons keep logs, config, and listening sockets across the transition.
- **No escape from inside**: a process cannot `chroot` its way out. The only way to leave a chroot is for a privileged process *outside* it to reap and replace it.
- **Working directory outside the new root**: the kernel allows the call, but relative paths and `..` traversal from there can still reach files outside the jail. The mandatory `chdir "/"` after `chroot` closes that gap.
- **Threads**: `chroot` on Linux is per-process (strictly, per fs struct). Under `use threads` all interpreter threads share the same root after the call - there is no per-thread jail.
- **Does not affect `$0`, `%ENV`, or open sockets**. Only path resolution changes.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `chdir` - the mandatory follow-up after a successful `chroot`; without it the working directory may still point outside the new root
- `fork` - children inherit the chroot; combine with `chroot` to run untrusted code in a restricted subtree
- `exec` - runs the replacement program inside the current root, so the target binary and its dynamic libraries must exist inside the jail
- `open` - opening files by absolute path after `chroot` looks up names inside the jail; pre-open anything you need from outside before the call
- `$_` - the default path when `chroot` is called with no argument
- `$!` - holds the `errno` from `chroot(2)` on failure

Scoping · Classes and OO

class

Declare a namespace that behaves as a native object class.

`class` introduces a new package just like `package`, but marks it as a *class*: the runtime attaches a constructor named `new`, enables the `field` and `method` keywords inside its body, and supports single inheritance through the `:isa` attribute. It is the core-language answer to hand-rolled `bless`-based OO - the same problem, without the boilerplate.

> **Experimental**
>
> The `class` feature (codename *Corinna*) is experimental in Perl 5.44 and may still change in incompatible ways. Enable it with `use feature 'class'` and silence the notice with `no warnings 'experimental::class'`.

Synopsis

```
class NAMESPACE;                            # rest-of-scope form
class NAMESPACE VERSION;
class NAMESPACE :ATTRIBUTES;
class NAMESPACE BLOCK                       # block form
class NAMESPACE VERSION BLOCK
class NAMESPACE VERSION :ATTRIBUTES BLOCK
```

Minimal working class:

```
use v5.38;
use feature 'class';
no warnings 'experimental::class';

class Point {
    field $x :param;
    field $y :param;

    method distance_from_origin {
        return sqrt($x ** 2 + $y ** 2);
    }
}

my $p = Point->new(x => 3, y => 4);
say $p->distance_from_origin;               # 5
```

What you get back

`class` is a declaration, not an expression - it does not return a value you can assign. What you get is:

- A new package in the symbol table, reachable as any other package.
- A generated `new` constructor. **Do not write your own `new`.** The generated one honours `:param` field attributes and runs every `ADJUST` block in declaration order.

- `$self` implicitly bound inside every `method` body.
- `__CLASS__` usable inside field initialisers and methods to name the class currently being constructed (which may be a subclass).

Forms

`class` accepts four syntactic shapes. The split between **block** and **statement** forms mirrors `package` exactly:

- **`class NAME BLOCK`** - the class definition is the contents of the block. Scope ends with the closing brace.
- **`class NAME;`** - the rest of the current scope (or file) up to the next `class` or `package` statement is the class body. Useful for single-class files.
- **`class NAME VERSION ...`** - the version is declared like `package Foo 1.234`. The version must precede any attributes.
- **`class NAME :ATTRIBUTES ...`** - one or more class attributes, separated by whitespace, each starting with `:`. The only currently defined class attribute is `:isa`.

```
class My::Thing 1.02 :isa(My::Base) {
    # body
}
```

Fields, methods, ADJUST

Inside a class body three new keywords are in scope:

- `field` declares per-instance storage. Fields behave like lexicals inside methods but each instance has its own slot. Attributes `:param`, `:reader`, and `:writer` generate constructor binding and accessors.
- `method` declares a subroutine that auto-binds `$self` and that can read and write the enclosing fields by name. Methods act as if `use feature 'signatures'` were in effect; `$self` does not appear in the signature.
- `ADJUST BLOCK` runs after field initialisers, during construction, with `$self` already bound. Use it for post-construction validation and for derived fields.

```
class Counter {
    field $value :param = 0;
    field $step  :param = 1;

    ADJUST {
        die "step must be positive" if $step <= 0;
    }

    method tick { $value += $step }
    method peek { $value }
}
```

Fields are **always lexical to the class** - they are not inherited. Subclasses that need access to a parent's state do so through methods, not through fields.

Inheritance: `:isa`

A class inherits from at most one other class:

```
class Shape { ... }

class Circle :isa(Shape) {
    field $r :param;
    method area { 3.14159265 * $r * $r }
}
```

Only single inheritance is supported. The `:isa` target is loaded automatically if it is not already in memory - equivalent to `use MODULE ()` before the class declaration. A minimum version can be required the same way `use MODULE VERSION` does:

```
class Circle :isa(Shape 2.000) { ... }
```

`:isa` only inherits **methods**. Fields are per-class private storage; a subclass has no implicit view into a parent's fields.

Global state it touches

- **`%INC`** - `:isa(Parent)` loads `Parent` via `require` on first use and records it here, just like `use`.
- **`@ISA`** - populated for the class from the `:isa` attribute, so existing `isa` / `UNIVERSAL` / `SUPER::` introspection keeps working.
- **The symbol table** - the class name becomes a stash in `%::` exactly as `package` would create one.

Examples

Plain class with a required parameter and a derived field:

```
class User {
    field $name :param;
    field $greeting;

    ADJUST { $greeting = "Hello, $name" }

    method greet { say $greeting }
}

User->new(name => 'Ada')->greet;          # Hello, Ada
```

Default parameter values - `=` applies when the caller omitted the parameter; `//=` also applies when the value was `undef`; `||=` also applies when the value was false:

```
class Window {
    field $title :param          = 'untitled';
    field $width :param          //= 640;
```

(continues on next page)

(continued from previous page)

```
    field $shown :param           ||= 1;
}
```

Single inheritance with a method override:

```
class Animal {
    field $name :param;
    method speak { "some sound" }
    method introduce { say $name, " says ", $self->speak }
}

class Dog :isa(Animal) {
    method speak { "woof" }
}

Dog->new(name => 'Rex')->introduce;       # Rex says woof
```

Private method via lexical `my method`, invoked through `->&`:

```
class Safe {
    field $secret :param;

    my method check ($n) { $n == $secret }

    method unlock ($n) {
        return $self->&check($n) ? "open" : "denied";
    }
}
```

Using `__CLASS__` so a base-class field initialiser picks up the subclass's override:

```
class Base {
    sub DEFAULT_X { 10 }
    field $x = __CLASS__->DEFAULT_X;
    method x { $x }
}

class Tuned :isa(Base) {
    sub DEFAULT_X { 99 }
}

say Tuned->new->x;                          # 99
```

Rest-of-file statement form:

```
use v5.38;
use feature 'class';
class Config;

field $path :param;
```

(continues on next page)

(continued from previous page)

```
method path { $path }
# File ends here; whole remainder is Config's body.
```

Edge cases

- **Do not define new**. The generated constructor is the only supported entry point. A user-written `sub new { ... }` inside a `class` body is a compile-time error.
- **No multiple inheritance**. `:isa(A, B)` is a syntax error. Multiple `:isa` attributes on one class are also rejected.
- **No `@ISA` manipulation**. Treat `@ISA` as read-only for classes declared with `class`. Pushing onto it at runtime is undefined.
- **`package` cannot re-open a class**. Once a namespace has been declared with `class`, subsequent `package` statements targeting the same name are a compile-time error, and vice versa.
- **Fields are not lexicals** in the usual sense. You cannot refer to a field from outside the class body, and you cannot `our` or `local` it. Only methods and `ADJUST` blocks see fields.
- **`$self` is read-only** inside a method. Assigning to it is a runtime error.
- **Field initialisers run before `$self` exists**. Inside `field $x = EXPR`, you can use `__CLASS__` and previously declared fields, but not `$self`. Anything that needs the fully built instance belongs in an `ADJUST` block.
- **No field access before declaration**. Within a field's initialiser expression, only fields declared *earlier* in the class body are visible.
- **Method resolution**. Dispatch still goes through the usual MRO (`UNIVERSAL::isa`, `SUPER::`), so `$self->SUPER::foo(@args)` works the way it does for `bless`-based classes.
- **Statement form scope**. `class Foo;` consumes the rest of the enclosing block - typically the rest of the file. A second `class` or `package` terminates it; there is no way to "close" a statement-form class early.
- **Experimental warning**. Every `class` declaration emits a `class is experimental` warning under the default warnings. Use `no warnings 'experimental::class'` once the feature is in use for real.

Differences from upstream

Fully compatible with upstream Perl 5.44. The feature is flagged experimental in upstream; pperl mirrors that status and emits the same warning category (`experimental::class`).

See also

- `field` - declares per-instance storage within a `class` body; supports `:param`, `:reader`, `:writer`
- `method` - declares a subroutine with implicit `$self` and direct access to the enclosing class's fields
- `package` - the non-class namespace declaration; `class` is the OO-aware sibling with the same syntactic forms
- `bless` - the pre-5.38 way to attach a class to a reference; still supported, and interoperates with `class`-built objects for `isa`/`can` checks
- `isa` - operator for testing class membership; works for both `class`-declared and `bless`-based objects
- `use` - how `:isa(Parent)` loads the parent class under the hood

I/O

close

Close a filehandle, flush its buffers, and release the underlying file descriptor.

`close` finishes whatever I/O was pending on `FILEHANDLE`, tears down the PerlIO layer stack, and asks the kernel to close the descriptor. The return value distinguishes a clean shutdown from one where either a buffered write failed on the way out, a PerlIO layer reported an error, or - for pipes - the child process exited non-zero. Without an argument, `close` operates on the currently selected filehandle (see `select`).

Synopsis

```
close FILEHANDLE
close EXPR
close
```

What you get back

Truthy (1) on a clean shutdown; false otherwise. A false return means one of the following went wrong:

- A pending buffered write could not be flushed (disk full, broken pipe, closed socket).
- A PerlIO layer (e.g. `:encoding(...)`, `:gzip` from a CPAN layer) reported an error.
- A pre-existing sticky error was already recorded on the handle.
- The handle came from a piped `open` and the child exited non-zero (or another syscall involved in tearing the pipe down failed).

When `close` returns false, `$!` holds the error from the failing operation and is safe to read. For pipe handles, if the only problem was a non-zero child exit, `$!` is set to `0` and the child's status shows up in `$?` (see below).

Always check the return value on writable handles: the final buffered block is flushed inside `close`, so `print` returning true earlier does **not** guarantee the data reached disk.

```
close $fh
    or die "close $path failed: $!";
```

Global state it touches

- `$!` - set to the failing `errno` when `close` returns false, including the flush error on the final buffered write. Cleared to `0` on a pipe close where the only problem was a non-zero child exit.
- `$?` - for handles from a piped `open`, set to the wait status of the child process: low 8 bits are the signal, high 8 bits are the exit code (same encoding as `system`).
- `${^CHILD_ERROR_NATIVE}` - for pipe handles, set to the raw OS-native status value (before Perl's portable encoding).
- `$.` - an **explicit** `close` on an input handle resets the current line number. An implicit close via reopening the same handle does not.
- Currently selected filehandle (via `select`) - `close` with no argument closes it.

Examples

Close a regular file and check for a flush failure:

```
open my $fh, ">", "out.txt" or die $!;
print {$fh} @records;
close $fh or die "flush to out.txt failed: $!";
```

Close a pipe opened for writing and inspect the child's exit status. `close` waits for the child, then populates `$?`:

```
open(my $sort, "|-", "sort", ">", "foo")
    or die "can't start sort: $!";
# ... print records to $sort ...
close $sort
    or warn $! ? "error closing sort pipe: $!"
               : "sort exited with status " . ($? >> 8);
```

Close a pipe opened for reading, then collect the command's exit status. Exit code is the top 8 bits of `$?`:

```
open(my $who, "-|", "who") or die $!;
my @lines = <$who>;
close $who;
my $exit = $? >> 8;
```

Bare `close` closes whatever handle `select` most recently made current, then restores `STDOUT` as the selected handle:

```
open my $log, ">>", "app.log" or die $!;
my $old = select $log;
print "entry\n";
close;                          # closes $log (the selected handle)
select $old;
```

Closing the read end of a pipe early makes the writer receive SIGPIPE. If the writer has no handler, it will be killed - drain the pipe first if you need the writer to exit cleanly:

```
open my $gen, "-|", "some-producer" or die $!;
while (<$gen>) { last if /^END/ }
1 while <$gen>;                 # drain so the child finishes cleanly
close $gen or warn "producer failed: $!";
```

Edge cases

- **Bare `close`** - with no argument, closes the currently selected filehandle. In a freshly started program that is STDOUT; after a `select($fh)` it is `$fh`. Rarely what you want outside tightly scoped code.
- **Reopening closes for you** - `open my $fh, ...` on a handle that is already open will close the old descriptor first. The implicit close does **not** reset `$.`; an explicit `close` does. Use an explicit `close` when line-number state matters.
- **Scope exit closes lexical handles** - when the last reference to a lexical filehandle goes out of scope, Perl closes it. If that implicit close fails (flush error, child exit non-zero), Perl emits a `Warning: unable to close filehandle %s properly` warning. Call `close` explicitly at a point where you can check the return value and silence that warning.
- **Indirect handles** - FILEHANDLE may be any expression producing a usable filehandle: a bareword (`close STDERR`), a scalar (`close $fh`), a glob ref (`close \*FH`), or an autovivified handle from `open my $fh, ...`.
- **Pipe close waits** - closing a pipe handle blocks until the child exits, so `$?` is populated by the time `close` returns. Under threads, if the same pipe handle is still open in another thread, `close` returns true immediately without reaping the child.
- **Pipe close with non-zero exit** - `close` returns false and sets `$!` to `0` so that `$! ? "$!" : "exit $?"` discriminates between an OS-level failure and an ordinary non-zero exit.
- **Closed or invalid handle** - closing a handle that is not open sets `$!` to `"Bad file descriptor"` and returns false. Under `use warnings`, a `close()` on unopened filehandle warning is emitted.
- **STDIN / STDOUT / STDERR** - can be closed like any other handle. The descriptor is freed and will be reused by the next open on the standard handle, which is how the classic `open STDOUT, ">", "log"` redirect works.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `open` - counterpart; an `open` on an already-open handle does an implicit `close` first (but does not reset `$.`)
- `print` - final buffered `print` is flushed inside `close`, so the write's real success is only known after `close`
- `fileno` - descriptor number of a handle; becomes invalid once `close` succeeds
- `binmode` - layer stack set by `binmode` is torn down during `close`, and its teardown can itself fail
- `$?` - child exit status populated by `close` on a pipe handle
- `$!` - error from the failing close operation

I/O

closedir

Close a directory handle opened by `opendir`.

`closedir` releases the operating-system directory stream associated with `DIRHANDLE` and frees the slot so the handle can be reused. It is the counterpart to `opendir`, and the only supported way to close a directory handle - `close` operates on file handles and does **not** accept a directory handle, because file handles and directory handles live in separate namespaces.

Synopsis

```
closedir DIRHANDLE
closedir $dh
```

What you get back

1 on success, `undef` on failure (with `$!` set to the error from the underlying `closedir(3)` call). The return value is worth checking on networked or FUSE-backed filesystems where the close may surface a deferred I/O error.

```
closedir $dh
    or warn "closedir $path failed: $!";
```

Closing a directory handle that is already closed, or one that was never successfully opened, returns `undef` and - under `use warnings` - emits a `closedir() attempted on invalid dirhandle` warning.

Lexical directory handles and auto-close

A directory handle stored in a lexical scalar - the form produced by `opendir my $dh, $path` - is closed automatically when the scalar goes out of scope (or is reassigned, or the program exits). Explicit `closedir` is then a matter of timing, not correctness:

```
sub entries {
    my ($path) = @_;
    opendir my $dh, $path or die "opendir $path: $!";
    return readdir $dh;
    # $dh is closed here as it leaves scope
}
```

Call `closedir` explicitly when you need the release to happen at a known point - before forking, before renaming the directory on platforms that hold it open, before checking the return value, or simply when the enclosing scope is long and holds many other resources.

Bareword directory handles (package globals, e.g. `opendir DH, $path`) are **not** auto-closed. They persist until the program ends or `closedir DH` is called.

Global state it touches

`closedir` does not read or write any of Perl's documented special variables. It sets `$!` on failure. It does **not** affect `$_`, the selected filehandle, or any of the output-separator globals.

Examples

Basic open/read/close cycle:

```
opendir my $dh, "." or die "opendir: $!";
my @names = readdir $dh;
closedir $dh;
```

Check the return value - useful on network filesystems:

```
opendir my $dh, "/mnt/nfs/project" or die "opendir: $!";
my @entries = readdir $dh;
closedir $dh
    or die "closedir failed: $!";
```

Reset to the start with `rewinddir` instead of `closedir` + `opendir` when you want to re-read the same directory:

```
opendir my $dh, $path or die $!;
my @first = readdir $dh;
rewinddir $dh;
my @again = readdir $dh;              # same set as @first
closedir $dh;
```

Bareword handle - must be closed explicitly:

```
opendir DH, "/etc" or die $!;
while (my $name = readdir DH) {
    last if $name eq "hosts";
}
closedir DH;                            # not optional for barewords
```

Using `close` on a directory handle does **not** work:

```
opendir my $dh, "." or die $!;
close $dh;                              # wrong - does not close the
↪dir
closedir $dh;                           # correct
```

Edge cases

- **`close` on a directory handle**: file handles and directory handles are distinct types. `close` on a directory handle is either a no-op or an error depending on the handle form; always use `closedir` to close what `opendir` opened.
- **Double close**: calling `closedir` twice on the same handle returns `undef` on the second call and sets `$!`. Under `use warnings` a `closedir() attempted on invalid dirhandle` warning is emitted.
- **Closed or never-opened handle**: same behaviour as double close - `undef` return and a warning under `use warnings`.
- **Pending `readdir` state**: `closedir` discards any unread entries. If you want to restart iteration, use `rewinddir` instead of close + reopen.
- **`seekdir` / `telldir` positions**: positions obtained from `telldir` are invalidated by `closedir`. They are only meaningful for the lifetime of a single open directory stream.
- **Scope exit during exception**: lexical directory handles are still closed when the scope unwinds via `die`, so explicit `closedir` in cleanup code is usually redundant.
- **Fork**: after `fork`, both parent and child share the directory stream's kernel state; each should call `closedir` in whichever process no longer needs it.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `opendir` - open a directory handle; `closedir` is its counterpart
- `readdir` - read the next entry (or all entries in list context) from an open directory handle
- `rewinddir` - reset the directory stream to the beginning without closing it
- `seekdir` - reposition the directory stream to a value previously returned by `telldir`

- `telldir` - return the current position within an open directory stream
- `close` - close a **file** handle; does not work on directory handles

Sockets

connect

Initiate a connection from a socket to a remote address.

`connect` is the client-side half of the socket handshake. Given a `SOCKET` that was created with `socket` and a packed address `NAME` in the form the socket's address family expects, it asks the kernel to establish a connection to that peer. It is a thin wrapper over the `connect(2)` system call and inherits every one of that call's blocking, non-blocking, and error semantics.

Synopsis

```
connect SOCKET, NAME
```

What you get back

Truthy (1) on success, false on failure with `$!` set to the system error. Always check the return value - there is no useful default behaviour when a connection attempt fails:

```
connect($sock, $peer)
    or die "connect to $host:$port failed: $!";
```

Global state it touches

- `$!` is set to the kernel's `errno` on failure. Interpret it against the `Errno` constants, not by string value - the stringified message is locale-dependent.
- The socket handle itself carries the connection state; no other special variables are involved.

Building `NAME`: it's a packed struct, not a string

The second argument is **not** a `"host:port"` string. It's a packed sockaddr in the exact byte layout the kernel expects for the socket's address family. Build it with the helpers from `Socket`:

- `sockaddr_in($port, $inaddr)` for IPv4 - `$inaddr` from `inet_aton` or `inet_pton`.
- `sockaddr_in6($port, $in6addr)` for IPv6 - `$in6addr` from `inet_pton(AF_INET6, ...)`.
- `sockaddr_un($path)` for Unix-domain sockets.

Hand-packing with `pack` works, but there is no reason to do so when `Socket` already exports the right builders.

Examples

A plain IPv4 TCP client:

```
use Socket;

socket(my $sock, PF_INET, SOCK_STREAM, getprotobyname("tcp"))
    or die "socket: $!";

my $addr = sockaddr_in(80, inet_aton("93.184.216.34"));
connect($sock, $addr)
    or die "connect: $!";

print $sock "GET / HTTP/1.0\r\nHost: example.com\r\n\r\n";
```

Modern address-family-agnostic client via `getaddrinfo`:

```
use Socket qw(getaddrinfo SOCK_STREAM);

my ($err, @res) = getaddrinfo("example.com", "80",
                              { socktype => SOCK_STREAM });
die "getaddrinfo: $err" if $err;

my $sock;
for my $ai (@res) {
    socket($sock, $ai->{family}, $ai->{socktype}, $ai->{protocol})
        or next;
    last if connect($sock, $ai->{addr});
    close $sock;
    undef $sock;
}
defined $sock or die "could not connect to any address";
```

A Unix-domain client:

```
use Socket;

socket(my $sock, PF_UNIX, SOCK_STREAM, 0) or die "socket: $!";
connect($sock, sockaddr_un("/tmp/my.sock"))
    or die "connect: $!";
```

Non-blocking connect with a timeout via `select` - the idiomatic way to bound a connect attempt in pure Perl:

```
use Socket;
use Fcntl qw(F_GETFL F_SETFL O_NONBLOCK);
use Errno qw(EINPROGRESS);

socket(my $sock, PF_INET, SOCK_STREAM, 0) or die "socket: $!";
my $flags = fcntl($sock, F_GETFL, 0);
fcntl($sock, F_SETFL, $flags | O_NONBLOCK);
```

(continues on next page)

(continued from previous page)

```
my $ok = connect($sock, sockaddr_in(80, inet_aton("93.184.216.34
 ↪")));
if (!$ok && $! == EINPROGRESS) {
    my $w = '';
    vec($w, fileno($sock), 1) = 1;
    my $n = select(undef, $w, undef, 5.0);   # 5-second timeout
    $n > 0 or die "connect timed out";
    # Check SO_ERROR to see whether the async connect succeeded:
    my $err = unpack("i", getsockopt($sock, SOL_SOCKET, SO_ERROR));
    $err == 0 or die "connect: " . do { local $! = $err; "$!" };
}
```

Edge cases

- **Address-family mismatch.** If `NAME` was packed for a different family than the socket was created with (e.g. `sockaddr_in` against a `PF_INET6` socket), `connect` fails with `EAFNOSUPPORT` or `EINVAL`. The packed bytes carry the family; the kernel checks.
- **Already connected.** Calling `connect` on a stream socket that is already connected fails with `EISCONN`.
- **Connectionless sockets.** On `SOCK_DGRAM` / `SOCK_RAW` sockets, `connect` does not exchange packets; it records a default peer so that subsequent `send` / `recv` calls can omit the address. It can be called again with a new `NAME`, or cleared by calling with an `AF_UNSPEC` sockaddr.
- **Non-blocking sockets.** If the socket is in non-blocking mode, a TCP connect typically returns false with `$!` set to `EINPROGRESS`. The connection is **not yet established**. Wait for the socket to become writable (via `select` or your event loop), then read `SO_ERROR` with `getsockopt` to obtain the final status. A zero `SO_ERROR` means the connection is up; anything else is the real failure code.
- **Signals interrupting the call.** A blocking `connect` can return false with `$!` set to `EINTR` if a signal handler ran. The socket is still connecting in the background; you must wait on writability and check `SO_ERROR` as in the non-blocking case - **do not** retry the `connect` call, that yields `EALREADY`.
- **Unbound source address.** If `bind` was not called first, the kernel picks a source address and ephemeral port automatically. The common case is to skip `bind` on the client.
- **Closed or unopened socket.** Fails with `EBADF`.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `socket` - create the endpoint that `connect` operates on
- `bind` - pin the local address before connecting; rarely needed on a client
- `accept` - the server-side counterpart to `connect`
- `getpeername` - recover the peer address of a connected socket
- `shutdown` - close one direction of a connected socket without releasing the descriptor
- `Socket` - packers (`sockaddr_in`, `sockaddr_un`, ...) and constants (`PF_INET`, `SOCK_STREAM`, ...) used to build arguments
- `Errno` - symbolic constants for comparing `$!` against `EINPROGRESS`, `EISCONN`, `ECONNREFUSED`, ...

Control flow

continue

Attach a block to a loop that runs after every iteration, just before the condition is re-tested.

`continue` is not a built-in function - it is a flow-control keyword. In its one remaining sense it introduces the trailing block of a `while`, `until`, or `foreach` loop. The block runs at the end of each iteration **and** whenever `next` transfers control out of the main body, but **not** when `last` or `redo` do. It plays the role that the third clause of a C `for (init; cond; step)` loop plays in C.

Synopsis

```
while (EXPR) { BODY } continue { STEP }
until (EXPR) { BODY } continue { STEP }
for VAR (LIST) { BODY } continue { STEP }
foreach VAR (LIST) { BODY } continue { STEP }
LABEL: while (EXPR) { BODY } continue { STEP }
```

What you get back

`continue` is a statement, not an expression. It has no return value and cannot appear in value position. The loop itself yields whatever its context demands (nothing in void context, the last evaluated expression under some constructs); the `continue` block does not contribute to that.

Execution model

For a loop with an attached `continue` block, each iteration runs in this order:

1. Evaluate the condition (or take the next list element, for `foreach`). If the loop should not run again, exit.
2. Run the main body.

3. Run the `continue` block.
4. Go back to step 1.

The three loop-control keywords interact with the `continue` block differently:

- `next` - jumps straight to step 3. The `continue` block **is** executed, then the condition is re-checked. This is the point of having a `continue` block: state that must advance on every iteration (a counter, a line number, a one-shot match reset) still advances when the body short-circuits via `next`.
- `last` - leaves the loop entirely. The `continue` block is **not** executed.
- `redo` - restarts step 2 without re-checking the condition and without running the `continue` block. Use it when the body needs another pass over the current iteration's data (for example, after splicing in more input).

Omitting the `continue` block is equivalent to an empty one.

Examples

A counter that advances even when the body skips:

```
my $n = 0;
for my $line (@lines) {
    next if $line =~ /^\s*#/;     # skip comments
    process($line);
} continue {
    $n++;                         # still counts skipped lines
}
print "scanned $n lines\n";
```

Equivalence to a C-style `for` loop - the two forms below are the same, because Perl rewrites the C-style header as a `while` with a `continue` block:

```
for (my $i = 1; $i < 10; $i++) { say $i }

my $i = 1;
while ($i < 10) {
    say $i;
} continue {
    $i++;
}
```

Resetting one-shot matches and line counters while streaming files with <>:

```
while (<>) {
    m?(fred)?   && s//WILMA/;
    m?(barney)? && s//BETTY/;
} continue {
    print "$ARGV $.: $_";
    close ARGV if eof;            # reset $.
    reset      if eof;            # reset ?pat?
}
```

The interaction with `redo` - note that `continue` runs on the `next` path but not on the `redo` path:

```
while (<>) {
    chomp;
    if (s/\\$//) {
        $_ .= <>;
        redo unless eof;         # merge continuation, re-enter body
    }                            # continue block skipped on redo
    process($_);
} continue {
    $seen++;                     # counts completed records only
}
```

Edge cases

- **No trailing block** - bare `continue` (without a following `{ ... }`) is a statement of the `switch` feature: inside a `when` block of `given` / `when` it falls through to the next `when`. The feature still exists but is off by default and in no feature bundle. Anywhere else, a bare `continue` compiles and then dies at run time with `Can't "continue" outside a when block`.
- **`last` skips the `continue` block.** If the block holds cleanup you need on every path out, put the cleanup after the loop or in a scope guard, not in `continue`.
- **`redo` skips the `continue` block.** If the `continue` block advances a counter, a `redo` iteration will not tick it - that is usually what you want (the iteration is not yet finished) but it can surprise code that relies on the counter matching the number of body entries.
- **`do { ... } while`** is **not** a real loop - `last`, `next`, and `redo` do not work in it, and a trailing `continue` block is a syntax error.
- **Labeled loops** - `LABEL:` attaches to the `while` / `for` / `foreach` keyword, not to `continue`. `next LABEL` and `last LABEL` target the loop as a whole; the `continue` block is part of that loop.
- **A block by itself** is a one-shot loop, so a bare `{ ... } continue { ... }` parses and runs the `continue` block once after the body - rarely useful, but legal.
- **Foreach with aliasing** - inside a `foreach` loop the loop variable aliases the current list element. The `continue` block still runs with that alias in scope, so you can inspect or mutate the element there just as in the body.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `next` - jump to the `continue` block (if any) and re-test the condition; the one loop-control keyword that **does** run the `continue` block
- `last` - leave the loop entirely; **skips** the `continue` block

- `redo` - restart the body without re-testing the condition; **skips** the `continue` block
- `return` - leave the enclosing subroutine; also skips the `continue` block because it leaves the loop
- `break` - companion keyword of the `switch` feature; like the bare `given`/`when` form of `continue`, available only under `use feature 'switch'`

Numeric functions

cos

Return the cosine of a number given in radians.

`cos` is the standard trigonometric cosine: it takes an angle measured in **radians** and returns the ratio of the adjacent side to the hypotenuse of the corresponding right triangle. If `EXPR` is omitted, `cos` operates on `$_`. The result is always a floating-point number in the closed interval `[-1, 1]`.

Synopsis

```
cos EXPR
cos
```

What you get back

A double-precision float in `[-1, 1]`. Perl forwards the call to the platform's C `cos(3)` routine, so the precision and rounding behaviour match your libm. Special values follow IEEE 754: `cos(0)` is exactly `1`, `cos("inf")` and `cos("nan")` are both `NaN`.

Radians, not degrees

The single most common mistake with `cos` is feeding it degrees. A full turn is `2 * π` radians, not `360`. Convert once and keep the converted value:

```
use POSIX qw(acos);
my $pi = acos(-1);                  # π to full double precision

sub deg2rad { $_[0] * $pi / 180 }

print cos( deg2rad(60) ), "\n";     # 0.5
print cos( 60 ), "\n";              # -0.952412980415 - wrong if you
 ↪meant degrees
```

exports `deg2rad` and `rad2deg` if you would rather not write the conversion yourself.

Examples

Walk the unit circle in eight steps and print `(cos θ, sin θ)`:

```
use POSIX qw(acos);
my $pi = acos(-1);

for my $k (0 .. 7) {
    my $theta = $k * $pi / 4;
    printf "%.3f  %+.3f  %+.3f\n",
           $theta, cos($theta), sin($theta);
}
```

Compare `cos` against its Taylor series around zero. For small `$x` the four-term expansion is already accurate to ten decimal places - a useful sanity check when porting numeric code:

```
sub cos_taylor {
    my $x = shift;
    my $x2 = $x * $x;
    1 - $x2/2 + $x2*$x2/24 - $x2*$x2*$x2/720;
}

my $x = 0.3;
printf "libm:   %.15f\n", cos($x);
printf "taylor: %.15f\n", cos_taylor($x);
```

Default argument. Inside a `while (<>)` loop `cos` without an argument reads the current line - almost always a bug, but the language allows it:

```
$_ = 0;
print cos, "\n";                    # 1
```

Inverse cosine via the identity from `perlfunc`:

```
sub acos { atan2( sqrt(1 - $_[0] * $_[0]), $_[0] ) }

print acos(0.5), "\n";              # 1.0471975511966 (= π/3)
```

Using for the full set of inverse and hyperbolic functions rather than rolling your own:

```
use Math::Trig;
print acos(0.5), "\n";              # same result, no hand-written␣
↪formula
```

Edge cases

- **Non-numeric arguments coerce to `0`.** `cos("hello")` returns 1 because the string is numified to `0` first. Under `use warnings` this produces an `Argument "hello" isn't numeric in cos` warning. Validate input with `looks_like_number` from `Scalar::Util` if the source is untrusted.
- **Very large arguments lose precision.** `cos` first reduces its argument modulo 2π, and for `|x|` on the order of `1e16` the nearest representable double differs from `x` by more than π. The result is still a number in `[-1, 1]`, but it is essentially

random. Reduce the angle yourself before the call if you are accumulating a phase over many iterations.

- **`undef` coerces to `0`** and returns 1, with the usual `uninitialized` warning under `use warnings`.
- **IEEE specials**: `cos("inf")`, `cos("-inf")`, and `cos("nan")` all return `NaN`. Compare with `$result != $result` (the only value not equal to itself) to detect `NaN`, or use `POSIX::isnan`.
- **No complex-number support** in the built-in. For complex arguments use , which overloads `cos` via operator overloading:

```
use Math::Complex;
my $z = cplx(1, 2);
print cos($z), "\n";              # 2.03272300701967-3.
↪0518977991518i
```

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `sin` - sine of an angle in radians; the companion function most often used alongside `cos`
- `atan2` - two-argument arctangent, the usual building block for inverse trig (`acos`, `asin`) in pure Perl
- `sqrt` - square root; appears in the classical `acos` identity `acos(x) = atan2(sqrt(1 - x*x), x)`
- - `acos`, `asin`, `atan`, hyperbolic variants, `deg2rad`/`rad2deg`, and great-circle helpers
- - overloads `cos` (and the rest of the trig family) for complex arguments
- `POSIX` - exposes `acos` directly, plus `isnan`/`isinf` for detecting the non-finite results discussed above

SCALARs and strings

crypt

One-way `passwd(5)`-style hashing of a plaintext string with a salt.

`crypt` is a thin wrapper over the system C library's `crypt(3)` function. It turns `PLAINTEXT` and `SALT` into a short string - a *digest* - and returns it. The same `PLAINTEXT` and `SALT` always produce the same digest; small changes in either produce a completely different digest. There is no inverse function: you cannot recover `PLAINTEXT` from a digest.

> **Warning**
>
> `crypt` is **not a general-purpose encryption function**, despite the name. It cannot be undone. It is also **not a modern password hash** on most traditional configurations - the historical Unix scheme considers only the first eight bytes of `PLAINTEXT` and produces a 13-byte digest over a 12-bit salt, which is unsuitable for protecting passwords against contemporary attackers. What `crypt` is good for is verifying a user-supplied string against an already-stored digest produced by the same system's `crypt`. For anything else - encryption, key derivation, or hashing large data - use a real cryptographic library. See *See also* below.

Synopsis

```
crypt PLAINTEXT, SALT
```

Both arguments are strings. The return value is a string digest.

What you get back

A string whose exact shape depends on the host's `crypt(3)` implementation. Traditionally it is 13 bytes: two bytes of salt followed by 11 bytes from the set `[./0-9A-Za-z]`. Modern glibc, musl, and BSD libcs accept extended salts that select stronger schemes (MD5, SHA-256, SHA-512, bcrypt, yescrypt) and return longer digests in the `$id$salt$hash` passwd-file format.

Treat the digest as opaque. Do not parse it, do not assume a length, and do not assume which bytes of `SALT` the implementation consumed. The correct way to verify a plaintext against an existing digest is to pass the digest itself back in as the salt:

```
if (crypt($plain, $digest) eq $digest) { ... }
```

The salt used to create the digest is embedded in the digest, so this re-runs the same scheme with the same parameters on the candidate plaintext. Code written this way works with the classical DES scheme and with every modern extension a given `crypt(3)` exposes, without having to know which one produced the stored digest.

Global state it touches

`crypt` does not read or write any Perl special variable. On Unicode input it may emit a fatal `Wide character in crypt` error (see *Edge cases*).

Examples

Verify a password against a stored `passwd`-style digest:

```
my $stored = (getpwuid($<))[1];       # encrypted field from passwd

print "Password: ";
chomp(my $word = <STDIN>);
```

(continues on next page)

(continued from previous page)

```
if (crypt($word, $stored) eq $stored) {
    print "ok\n";
} else {
    die "Sorry...\n";
}
```

Create a fresh digest for a new password using a random two-character salt from the classical alphabet. The set is only a recommendation; the exact characters accepted in a salt are whatever the host `crypt(3)` allows, and Perl imposes no restriction of its own:

```
my @chars = ('.', '/', 0..9, 'A'..'Z', 'a'..'z');
my $salt  = $chars[rand 64] . $chars[rand 64];
my $hash  = crypt($plaintext, $salt);
```

Request a specific modern scheme by using an extended salt (glibc and compatible libcs). The `$id$` prefix selects the algorithm - `$1$` is MD5-crypt, `$5$` is SHA-256-crypt, `$6$` is SHA-512-crypt, `$2b$` is bcrypt where supported:

```
my $hash = crypt($plaintext, '$6$rounds=100000$' . $random_salt);
```

Re-verify using the stored digest as the salt - scheme-agnostic:

```
my $candidate = crypt($plaintext, $stored_digest);
my $match     = $candidate eq $stored_digest;
```

Constant-time comparison of the two digests is still the caller's responsibility; plain `eq` leaks timing information. If the threat model cares, compare byte-by-byte with a loop that does not short-circuit, or use a comparison routine from a crypto module.

Edge cases

- **First-8-byte truncation on classical DES.** On traditional Unix `crypt(3)`, only the first eight bytes of `PLAINTEXT` affect the digest: `crypt("password123", $s)` and `crypt("password456", $s)` collide. Extended schemes (`$1$`, `$5$`, `$6$`, `$2b$`, ...) do not have this limit, but the classical scheme is what you get from an unprefixed two-byte salt.
- **Salt-length assumptions are wrong.** Do not assume `substr($digest,0,2)` is the salt. Extended digests carry the salt in a different position and length. Re-use the full digest as the salt argument (`crypt($plain, $digest)`) and let `crypt(3)` parse it.
- **Unicode input.** If `PLAINTEXT` or `SALT` contains characters above codepoint 255, Perl attempts to downgrade a copy of the string to bytes before calling `crypt(3)`. If every character fits in a byte, the downgrade succeeds silently. If any character does not, `crypt` dies with `Wide character in crypt`. Encode explicitly (e.g. `Encode::encode_utf8`) if your plaintext might contain wide characters.
- **Unsupported scheme returns failure.** Passing a `$id$` prefix the host `crypt(3)` does not understand typically returns `undef` or a short string starting with `*0` or `*1`. Always check the return shape before using it.

- **`crypt(3)` may be absent or restricted.** On a small number of hosts the library function has been removed or replaced with a stub for export-control reasons. `crypt` in Perl is only as capable as the underlying libc call; if the libc call is a stub, so is this.
- **Not suitable for hashing large data.** `crypt` ignores most of its input on the classical scheme and is slow per call on the stretched schemes. It is the wrong tool for checksumming files or building content-addressable storage.

Differences from upstream

Fully compatible with upstream Perl 5.44. pperl dispatches to the host `crypt(3)` the same way perl5 does, so the accepted salt formats and the exact digest shape match whatever the system library produces - including the availability of `$1$`/`$5$`/`$6$`/`$2b$` extended schemes.

See also

- `Digest::MD5` - MD5-128 over arbitrary byte strings; use for fingerprinting data, not for password storage
- `Digest::SHA` - SHA-1/2 family digests; use for integrity checks and as a building block for HMAC and key derivation
- `Crypt::Bcrypt`, `Crypt::Argon2`, `Crypt::Passphrase` - CPAN modules for modern password hashing; prefer these over `crypt` for any new password-verification code
- `Crypt::Eksblowfish`, `Crypt::ScryptKDF` - CPAN modules for bcrypt and scrypt-based key derivation
- `getpwuid` - fetch the encrypted-password field from the local `passwd` database; its second field is the digest you pass as `SALT` when verifying
- `chomp` - remove the trailing newline from a password read with `<STDIN>` before passing it to `crypt`

I/O · Classes and OO

dbmclose

Break the binding between a DBM file and a hash previously attached by `dbmopen`.

`dbmclose` is the teardown counterpart to `dbmopen`. It flushes any pending writes, releases the DBM library's handles, and leaves `HASH` as an ordinary, untied hash you can continue to use or discard. In modern code `dbmclose` has been largely superseded by `untie` - the two accomplish the same thing, but `untie` works for every tied variable, not just DBM-bound hashes. Prefer `untie` unless you are maintaining existing dbmopen/`dbmclose` pairs and want them to read symmetrically.

Synopsis

```
dbmclose HASH
dbmclose %hash
```

What you get back

True on success, false on failure. Failure modes are rare - the common one is passing a hash that was never tied to a DBM file in the first place, in which case `dbmclose` returns false and sets `$!`. Most programs do not check the return value; the interesting errors (disk full, quota exceeded) surface on the writes that preceded the close, not on `dbmclose` itself. Check it when the DBM file is on a filesystem where a delayed write failure at close time is plausible:

```
dbmclose(%cache)
    or warn "dbmclose failed: $!";
```

Global state it touches

`dbmclose` reads and mutates the tie state of `HASH`. It does not touch any of the usual I/O-related special variables (`$,`, `$\`, `$|`, the selected filehandle); DBM access does not flow through the PerlIO layers that `print` and friends use. On failure, `$!` is set by the underlying DBM library.

Examples

Open a DBM file, iterate it, close it:

```
dbmopen(%HIST, '/usr/lib/news/history', 0666)
    or die "dbmopen: $!";
while (my ($key, $val) = each %HIST) {
    print $key, ' = ', unpack('L', $val), "\n";
}
dbmclose(%HIST);
```

Equivalent using the modern `tie`/`untie` pair - this is what new code should look like:

```
use AnyDBM_File;
use Fcntl;
tie(my %HIST, 'AnyDBM_File', '/usr/lib/news/history', O_RDWR, 0666)
    or die "tie: $!";
# ... use %HIST ...
untie %HIST;
```

Guard against closing an unbound hash:

```
my %h;
dbmclose(%h)
    or warn "not tied to a DBM file: $!";   # warns
```

Pair with an `eval` block to localise any cleanup errors without aborting the surrounding code:

```
eval {
    dbmclose(%cache) or die "dbmclose: $!";
    1;
} or warn $@;
```

Edge cases

- **Hash not tied to a DBM file**: returns false, sets `$!`. No exception is raised. The hash is left unchanged.
- **Hash tied via `tie`, not `dbmopen`**: `dbmclose` still works and is equivalent to `untie` for that hash, because dbmopen is itself implemented as a `tie` under the hood. Mixing the two APIs is legal but confusing - pick one per hash and stay with it.
- **References still held to the tied object**: like `untie`, `dbmclose` severs the binding between the hash and the DBM file, but if other code still holds a reference to the underlying tied object the DBM library may not actually release its handles until that reference is dropped. The symptom is a DBM file that stays locked or open past the `dbmclose` call. A `use warnings` program receives the `untie attempted while N inner references still exist` warning in that case.
- **Calling `dbmclose` twice on the same hash**: the first call succeeds, the second returns false. Idempotent-style code should either track whether the hash is currently tied or ignore the failure.
- **Hash is `local`'d or goes out of scope without `dbmclose`**: the binding is broken implicitly when the hash is destroyed, and the DBM library closes the file as part of that teardown. Explicit `dbmclose` is preferable because it makes write-flush failures visible at a known point in the program.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `dbmopen` - the opening half of the pair; every `dbmclose` matches a prior dbmopen on the same hash
- `untie` - the modern, general-purpose replacement; works for every tied variable, not just DBM-bound hashes
- `tie` - the modern counterpart to `dbmopen`, used with `AnyDBM_File`, `DB_File`, `GDBM_File`, `NDBM_File`, or `SDBM_File` to bind a hash to a DBM file
- `close` - unrelated, but the analogous operation for filehandles: needed when you care about write-flush failures surfacing at a known point

I/O · Classes and OO

dbmopen

Bind a DBM file on disk to a hash so hash reads and writes become lookups and stores in the database.

dbmopen is the old-style way to tie a hash to a dbm, ndbm, sdbm, gdbm, or Berkeley DB file. It has been largely superseded by `tie` with an explicit DBM backend module (`DB_File`, `GDBM_File`, `SDBM_File`, `NDBM_File`, `AnyDBM_File`), which gives you control over which implementation is used and over per-backend options. Reach for dbmopen when you are reading legacy code or when the one-liner brevity matters more than backend choice.

Unlike `open`, the first argument is **not** a filehandle - it is the hash that will be bound. `HASH` is written with the `%` sigil.

Synopsis

```
dbmopen %HASH, $DBNAME, $MASK
dbmopen(%HASH, $DBNAME, 0)       # open existing only; never create
dbmclose %HASH                   # or: untie %HASH
```

What you get back

A true value on success, a false value on failure. On failure `$!` is set to the system error that prevented the bind. The common failure cases are "database does not exist and `MASK` is `0`" and permission errors on the backing files.

```
dbmopen(%cache, $path, 0)
    or die "cache $path missing or unreadable: $!";
```

After a successful call, `%HASH` behaves like any other hash: `$HASH{key} = $value` writes to the database, `$HASH{key}` reads from it, `delete` removes an entry, `exists` tests membership, and `each` / `keys` / `values` iterate. The data lives on disk; the hash is only a view.

Arguments

- **`HASH`** - the hash variable to bind. Must be written with `%`. Any existing contents are discarded. The hash stays bound until `dbmclose` or `untie` is called on it, or until it goes out of scope.
- **`DBNAME`** - the database pathname **without** the backend's file extension. Traditional dbm / sdbm stores use two files named `` `DBNAME.dir` `` and `` `DBNAME.pag` ``; gdbm and Berkeley DB use a single file. Passing `"/var/lib/app/cache"` opens `/var/lib/app/cache.dir` + `/var/lib/app/cache.pag` under sdbm, or `/var/lib/app/cache` under gdbm.
- **`MASK`** - creation mode for any files that have to be created, in the same form as the third argument to `chmod` (typically an octal literal like `0666` or `0644`). The active `umask` is applied to `MASK`, so real on-disk permissions are `MASK & ~umask`. **A `MASK` of `0` suppresses creation**: if the database does not already exist, dbmopen returns false and `$!` is set.

Global state it touches

- `$!` - set on failure (database missing, permission denied, backing store corrupt).
- The active `umask` - masks `MASK` when new backing files are created.

Examples

Open a history database, iterate with `each`, then close:

```
dbmopen(%HIST, '/usr/lib/news/history', 0666)
    or die "open history: $!";
while (my ($key, $val) = each %HIST) {
    print $key, ' = ', unpack('L', $val), "\n";
}
dbmclose %HIST;
```

Open read-only, refusing to create the database:

```
dbmopen(%cache, "$ENV{HOME}/.app/cache", 0)
    or die "no cache at $ENV{HOME}/.app/cache: $!";
my $hit = $cache{$key};
dbmclose %cache;
```

Pick a specific backend by loading its module before calling dbmopen. `AnyDBM_File` consults `@AnyDBM_File::ISA` to decide which implementation dbmopen uses:

```
use DB_File;
dbmopen(%NS_Hist, "$ENV{HOME}/.netscape/history.db", 0644)
    or die "open netscape history: $!";
```

Write-protected database - assignment silently fails (or croaks depending on the backend); guard with `eval` when you need to probe:

```
dbmopen(%db, $path, 0) or die $!;
my $writable = eval { $db{__probe__} = 1; delete $db{__probe__}; 1 }
 ↪;
warn "read-only\n" unless $writable;
```

Large database - avoid `keys` / `values`, which materialise the whole list in memory. Use `each` instead:

```
dbmopen(%big, $path, 0) or die $!;
while (my ($k, $v) = each %big) {
    process($k, $v);
}
dbmclose %big;
```

Edge cases

- **Hash sigil is required.** `dbmopen %h, $name, 0666` - writing `$h` or `\%h` does not work. The first slot is the hash's name as a hash, not a reference.
- **`MASK = 0` is the "open existing only" idiom.** A plausible-looking `0666` silently creates a fresh empty database if the file is missing, which is almost never what a reader wants on a cache lookup path.
- **Extensions are managed by the backend.** Do not pass `"cache.dir"` or `"cache.pag"` - pass `"cache"`. `sdbm` appends `.dir` / `.pag`; `gdbm` writes to the bare name.
- **One open per process for older DBM.** If the backend is the traditional `dbm(3)` library (rare today), a process may only have a single DBM open at a time. `sdbm` and `gdbm` do not have this limit.
- **Value size limits.** `sdbm` caps keys + values at roughly 1008 bytes per record; oversize writes are truncated or rejected. `gdbm` and Berkeley DB have much higher limits. When in doubt, use `tie` with `DB_File` and check the backend's own limits.
- **Numeric values are bytes on disk.** Storing `$h{k} = 42` writes the string `"42"`, not an integer. Pack explicitly with `pack` / `unpack` when the field needs a fixed binary layout (as in the news-history example above).
- **Closing is optional but recommended.** The bind is released when the hash goes out of scope, but explicit `dbmclose` or `untie` flushes pending writes and surfaces close-time errors.
- **Re-binding an already-bound hash.** Calling `dbmopen` on a hash that is already bound closes the previous binding first.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `dbmclose` - release a `dbmopen` binding and flush pending writes
- `tie` - the modern replacement; pair with `DB_File`, `GDBM_File`, `SDBM_File`, or `AnyDBM_File` to pick a backend explicitly
- `untie` - release a binding made with either `dbmopen` or `tie`
- `each` - stream entries without materialising the whole database in memory
- `umask` - masks the `MASK` argument when backing files are created
- `open` - the ordinary file-open built-in, which takes a filehandle and not a hash

Control flow

defer

Schedule a block to run when the enclosing scope exits, for any reason.

`defer` is a flow-control keyword, not a function. Writing `defer { ... }` inside a scope records the block for later execution; when control leaves the enclosing block - by falling

off the end, by `return`, by `die`, by `goto`, or by a loop-control statement (`next`, `last`, `redo`) - the stored block runs on the way out. It is the standard idiom for pairing acquisition with release (open/close, lock/unlock, push/pop) without relying on object destructors or wrapping every path in `eval`.

The feature is **experimental** and must be enabled. Either opt in explicitly or pick up the bundle:

```
use feature 'defer';    # explicit
use v5.36;              # bundle - enables defer among others
```

Under `use warnings 'experimental::defer'` (or plain `use warnings`) the first use emits an experimental warning. Silence it with `no warnings 'experimental::defer'` once you have decided to rely on the feature.

Synopsis

```
defer BLOCK
```

`defer` takes a brace-delimited block and nothing else. There is no expression form, no filehandle form, and no return value - the statement exists for its side effect on scope exit.

What you get back

`defer` is a statement, not an expression. It contributes no value to its surrounding context. The block's own return value is discarded when it runs at scope exit.

When the deferred block runs

The deferred block is invoked **once**, when control leaves the directly enclosing block, regardless of how control leaves:

- normal fallthrough off the end of the block,
- explicit `return` from the containing subroutine,
- an exception thrown by `die` or propagated from a called sub,
- `goto` out of the block,
- loop control via `next`, `last`, or `redo`.

If the flow of execution never reaches the `defer` statement itself, the block is **not** enqueued and does not run. This is the direct opposite of an `END` phaser, which the compiler enqueues unconditionally:

```
use feature 'defer';

{
    defer { say "This will run"; }
    return;
    defer { say "This will not"; }   # never reached, never␣
```

(continues on next page)

(continued from previous page)

```
 ↪scheduled
}
```

LIFO ordering

Multiple `defer` blocks in the same scope run in **last-in, first-out** order - the most recently scheduled block runs first. This matches the natural nesting of resource acquisition: the last thing acquired is the first thing released.

```
use feature 'defer';

{
    defer { say "1"; }
    defer { say "2"; }
    defer { say "3"; }
}
# prints 3, then 2, then 1
```

Global state it touches

None directly. The deferred block sees interpreter globals (`$_`, `$!`, `$@`, selected filehandle, etc.) with whatever values they hold **at the moment the block runs**, not at the moment it was scheduled. A `defer` that wants to preserve `$!` or `$@` across cleanup must save and restore them itself - otherwise a system call inside the block will clobber them and the caller of the enclosing sub will see the overwritten values.

```
defer {
    local ($!, $@);
    close $fh;          # clobbers $! on failure - local'd copy␣
 ↪absorbs it
}
```

Examples

Pair an open with its `close`, guaranteed to run even on `die`:

```
use feature 'defer';

sub read_config {
    open my $fh, "<", "config.ini" or die "open: $!";
    defer { close $fh; }

    while (<$fh>) { ... }
    # close $fh runs here on fallthrough, on return, and on die.
}
```

Unlock a shared resource on every exit path:

```
use feature 'defer';

sub critical_section {
    $lock->acquire;
    defer { $lock->release; }

    do_work();            # release runs on fallthrough
    return if $shortcut;  # release runs before the return completes
    die "boom" if $bad;   # release runs, then the exception
↪propagates
}
```

LIFO cleanup of nested resources - the outer resource is released last, which is usually what you want:

```
use feature 'defer';

sub copy_file {
    open my $in,  "<", $src or die "open $src: $!";
    defer { close $in; }

    open my $out, ">", $dst or die "open $dst: $!";
    defer { close $out; }

    print {$out} $_ while <$in>;
    # close $out runs first, then close $in.
}
```

Temporarily adjust interpreter state and restore it on exit without using `local`:

```
use feature 'defer';

sub with_separator {
    my $saved = $,;
    $, = " | ";
    defer { $, = $saved; }

    print @_, "\n";
}
```

Edge cases

- **Conditional scheduling.** A `defer` statement only schedules its block when control actually reaches it. An early `return` or `die` that happens before the `defer` line leaves nothing to run. Place the `defer` immediately after the paired acquisition so they cannot become separated by an intervening failure.
- **Once scheduled, always runs.** There is no mechanism to un-schedule a `defer` block. If the acquisition that the block releases has failed, either do not schedule the `defer` (keep it after the success check) or make the block idempotent so a redundant release is harmless.

- **No control-flow escape from the block**. A `defer` block may throw an exception, but it may **not** `return` from the enclosing sub, `goto` a label outside itself, or run `next`, `last`, or `redo` for a loop that encloses the `defer`. Those constructs are fine *inside* the `defer` when they act on loops **contained entirely within** the block's own body:

```
defer {
    foreach (1 .. 5) { last if $_ == 3; }   # permitted
}

foreach (6 .. 10) {
    defer { last if $_ == 8; }              # forbidden
}
```

- **Exception during exception unwind**. If the `defer` block is running because an exception is propagating and the block itself throws, the language does not specify the resulting exception - only that the caller will see one. Do not rely on which exception wins. Wrap risky cleanup in `eval` if you need both signals preserved.
- **Exceptions propagate normally**. An exception thrown by a `defer` block that is running for any other reason (normal exit, return, loop control) propagates to the caller exactly like any other exception. A failing `close` in cleanup can therefore turn a successful-looking function into a caller-visible `die`.
- **`$@` and `$!` inside the block**. On the exception-driven exit path, `$@` holds the in-flight exception while the `defer` block runs. Reading it is fine; replacing it changes what the caller sees after the unwind completes.
- **Scope granularity**. `defer` binds to the **immediately** enclosing block, not to the enclosing sub. A `defer` inside `if (...) { defer { ... } }` runs when the `if` block exits, not when the sub returns. Put the `defer` at subroutine scope if you want subroutine-exit semantics.
- **Experimental warning**. Under `use warnings` the first occurrence warns. Opt in deliberately with `no warnings 'experimental::defer'` once you have committed to the feature.
- **Not available without the feature guard**. Without `use feature 'defer'` (or a bundle that enables it), `defer { ... }` is parsed as a call to a subroutine named `defer` followed by a block - almost certainly not what you meant, and usually a compile-time error.

Differences from upstream

Fully compatible with upstream Perl 5.44. The feature remains marked experimental in pperl to mirror upstream's classification; behaviour (scheduling on statement execution, LIFO order, exception propagation, restrictions on control-flow escape) matches Perl 5.44.

See also

- `eval` - the established tool for turning exceptions into return values; `defer` complements it by guaranteeing cleanup without an explicit `eval` wrapper on every path
- `die` - one of the exit paths that triggers deferred blocks; read alongside `defer` when designing cleanup
- `return` - another exit path; a `defer` scheduled before a `return` runs before the caller resumes
- `local` - the other idiom for scoped state restoration; pick `local` for dynamic scoping of a single variable, `defer` for arbitrary cleanup actions
- `$@` - current exception; visible inside a `defer` block running on the exception-unwind path
- `$!` - system errno; easily clobbered by cleanup syscalls, `local`-ise inside the deferred block if the caller needs it preserved

Misc

defined

Test whether a value, variable, or subroutine is defined.

`defined` returns true when `EXPR` holds any value other than the undefined value `undef`, and false when it holds `undef`. It is the one and only way to tell `undef` apart from the other false values (`0`, `""`, `"0"`), which a plain boolean test cannot.

Synopsis

```
defined EXPR
defined
defined &SUBROUTINE
```

What you get back

A boolean: true if `EXPR` is defined, false if it is `undef`.

With no argument, `defined` operates on `$_`.

The result is context-sensitive only in the usual boolean sense; `defined` does not propagate list context into `EXPR`.

Why it exists, and why a plain boolean is not enough

Many operations return `undef` to signal failure, end of file, a system error, an uninitialised variable, or some other exceptional condition. `defined` lets you distinguish that sentinel from a legitimate result. A simple boolean test treats `undef`, `0`, `""`, and `"0"` the same - all four are false - so you cannot use truthiness to tell “nothing was returned” apart from “zero was returned”:

```
my $n = some_count();
if ($n)          { ... }   # false for 0 AND for undef
if (defined $n) { ... }   # false only for undef
```

Note that undef is itself a valid scalar value. Its presence does not *necessarily* mean something went wrong: pop @empty returns undef, but so does pop @array when the element popped happened itself to be undef. defined tells you the value is undef; it does not tell you why.

Default argument: $_

With no argument, defined tests $_:

```
while (<$fh>) {
    next unless defined;          # same as: defined $_
    chomp;
    ...
}
```

defined &subroutine

defined &func asks whether subroutine func has ever been defined. A forward declaration (sub func;) does **not** make it defined; an actual body does:

```
sub greet { "hello" }
print defined &greet   ? "yes\n" : "no\n";   # yes
print defined &missing ? "yes\n" : "no\n";   # no
```

A subroutine that is not defined may still be *callable*: the package's AUTOLOAD method can bring it into existence on first call. defined &func reflects the current state of the symbol table, not what a future AUTOLOAD might produce. See perlsub.

The return value is unaffected by forward declarations.

Hash elements: defined vs exists

On a hash element, defined asks about the **value**, not the key:

```
my %h = (a => 1, b => undef);

print exists  $h{a} ? 1 : 0, "\n";   # 1
print defined $h{a} ? 1 : 0, "\n";   # 1

print exists  $h{b} ? 1 : 0, "\n";   # 1   - key is present
print defined $h{b} ? 1 : 0, "\n";   # 0   - value is undef

print exists  $h{c} ? 1 : 0, "\n";   # 0   - key not present
print defined $h{c} ? 1 : 0, "\n";   # 0   - also autovivifies? no,␣
 ↪rvalue is safe
```

Use `exists` to ask "is this key in the hash?" and `defined` to ask "does this slot hold a usable value?". The two differ whenever a value was explicitly assigned `undef` or left `undef` by construction (e.g. `$h{b} = undef`, or `@h{qw(a b c)} = (1)`).

The same distinction applies to array elements: `exists $a[7]` reports whether the array is long enough and the slot has been assigned, while `defined $a[7]` reports whether that slot currently holds a non-`undef` value.

Examples

The common `pop`-until-empty idiom - stops only when the array is exhausted, not on a legitimate falsy element:

```
while (defined(my $val = pop @ary)) {
    process($val);
}
```

Check a system call for failure:

```
my $target = readlink $sym;
die "can't readlink $sym: $!" unless defined $target;
```

Call a coderef only if one was supplied:

```
sub dispatch {
    my ($cb, @args) = @_;
    return defined &$cb ? $cb->(@args) : die "no callback";
}
```

Provide a default without clobbering a legitimate `0` or `""`:

```
$debugging = 0 unless defined $debugging;
```

The modern form is the defined-or operator `//`, which does exactly this test inline:

```
$debugging //= 0;
my $name = $user // "anonymous";
```

Regex capture: `$1` being defined tells you the capture group participated in the match, even if it matched the empty string:

```
"ab" =~ /a(.*)b/;
print defined $1 ? "matched '$1'\n" : "no match\n";   # matched ''
```

Aggregates: `defined @array`, `defined %hash` are no longer supported

Historically `defined @array` and `defined %hash` reported whether memory for the aggregate had ever been allocated - an implementation detail that was rarely what callers actually wanted. Modern Perl rejects these forms. Test the aggregate's size instead:

```
if (@array) { print "array has elements\n" }
if (%hash)  { print "hash has members\n"   }
```

In boolean context an array evaluates to its element count and a hash evaluates to a true value iff it has any keys. That is the test you almost always wanted.

Testing *an element* of an aggregate is still fine and is the usual use: `defined $array[$i]`, `defined $hash{$key}`.

Overuse

`defined` is frequently overused. Zero and the empty string are defined values. A regex match that captures zero characters still sets `$1` to a defined empty string - the match did not fail; it matched something that happened to be of length zero. When a function returns `undef`, it is admitting it cannot give you a meaningful answer. Reach for `defined` when you are questioning the integrity of a value. For "is it zero?" or "is it empty?" a plain comparison to `0` or `""` is what you want.

Edge cases

- **Bare `defined`** with no argument tests `$_`. Inside `map`, `grep`, or a `while (<$fh>)` loop this is the idiomatic form.
- **`defined(&func)` vs `defined(&{$ref})`**: both work. The second dereferences a code reference and asks whether the referenced slot currently holds a defined subroutine body.
- **`defined $hash{$key}` does not autovivify the key** - `defined` is an rvalue context and leaves the hash unchanged. Contrast with `$hash{$key}{inner} = 1`, which does autovivify.
- **Chained hash/array access**: `defined $h{a}{b}` autovivifies `$h{a}` into an empty hash ref in order to perform the inner lookup, even though the outer `defined` reads the result. This is a consequence of how the dereference chain is evaluated, not of `defined` itself. To probe nested structure without autovivifying, check each level with `exists` first.
- **`defined $&`, `defined $1`**: after a successful match these are defined; after a failed match they retain whatever value they had from the previous successful match in the same scope. `defined` alone is not a safe "did the match succeed" test - check the match operator's return value instead.
- **`defined` is not an lvalue**. `defined $x = $y` is a syntax error; parenthesise the assignment: `defined($x = $y)`.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `undef` - the value `defined` tests against, and the operator that produces or clears it
- `exists` - ask whether a hash key or array index is present, independent of whether its value is `undef`
- `ref` - when the question is not "is it defined" but "what kind of reference is this"
- `//` *and* `//=` - defined-or operator; the modern inline form of `defined($x) ? $x : $default`
- `$_` - the default argument when `defined` is called with no expression

Hashes

delete

Remove the named key-value pair(s) from a hash, or element(s) from an array, and return what was removed.

`delete` reaches into an aggregate and removes the specified entries outright - not just by setting them to `undef`. After `delete`, `exists` on the same key or index returns false, `keys` no longer lists it, and iterating the hash with `each` skips it. The argument must be an element or slice of a hash or array; its final operation selects what to remove, and anything leading up to that operation (dereferences, method calls) is just navigation.

Synopsis

```
delete $hash{KEY}           # scalar key
delete @hash{LIST}          # hash slice: values of the listed keys
delete %hash{LIST}          # key/value hash slice (5.20+)
delete $array[INDEX]        # array element
delete @array[LIST]         # array slice
```

What you get back

In list context, `delete` returns the value or values removed, in argument order. A slice argument whose key or index did not exist yields `undef` at the matching position, so the length of the returned list matches the length of the key/index list.

In scalar context, `delete` returns the value of the last element removed, or `undef` if the key/index was not present.

The 5.20+ key/value hash-slice form `delete %hash{LIST}` returns a list of alternating key/value pairs - two elements for every deleted entry - which makes it the idiomatic way to pluck a named set of entries and keep both sides.

```
my %h = (foo => 11, bar => 22, baz => 33);
my $v = delete $h{foo};             # 11
my @v = delete @h{qw(bar quux)};    # (22, undef)
my %kv = delete %h{qw(baz quux)};  # (baz => 33, quux => undef)
```

Global state it touches

- **%ENV** - `delete $ENV{VAR}` unsets the environment variable for the current process and its future children. The effect is real, not just the Perl-side hash; system calls see the change.
- **tied aggregates** - `delete` dispatches to the `DELETE` method of the tying package. The return value is whatever the implementation chooses to hand back; some tied hashes return the deleted value, some do not.
- **$_** - unaffected. `delete` does not default to anything; `EXPR` is mandatory.

Examples

Remove one key, inspect what was there:

```
my %cfg = (host => "db1", port => 5432, user => "alice");
my $old_host = delete $cfg{host};
# $old_host is "db1"; exists $cfg{host} is false
```

Free memory by removing many entries in one call (hash slice):

```
delete @cache{@stale_keys};
```

Pluck a set of named entries and keep the pairs (5.20+ kv-slice):

```
my %h = (a => 1, b => 2, c => 3);
my %taken = delete %h{qw(a c)};
# %taken is (a => 1, c => 3); %h is (b => 2)
```

Delete through a nested structure. Leading navigation can be arbitrary - only the final operation decides what gets removed:

```
my $tree = { users => { alice => { roles => [qw(admin ro)] } } };
delete $tree->{users}{alice}{roles}[0];
# roles is now (undef, "ro")
delete $tree->{users}{alice};
# the whole user record is gone
```

Avoid autovivification on the read side while still removing on the write side. `delete` never vivifies the intermediate hashes it walks through:

```
delete $h{outer}{inner};    # does not create $h{outer} if absent
```

Compare with `splice` when you want indices to renumber. `delete` on an array element leaves a hole; `splice` closes the hole:

```
my @a = (10, 20, 30, 40);
delete $a[1];
# @a is (10, undef, 30, 40); $#a is still 3

my @b = (10, 20, 30, 40);
```

(continues on next page)

(continued from previous page)

```
splice(@b, 1, 1);
# @b is (10, 30, 40); $#b is 2
```

Edge cases

- **`delete` vs assigning `undef`**: `$h{k} = undef` leaves k in the hash with an undefined value; `exists $h{k}` is still true. `delete $h{k}` removes the key entirely.
- **Missing key in scalar context**: `delete $h{nope}` returns `undef`. That's indistinguishable from deleting a key whose value was `undef`; use `exists` first if the distinction matters.
- **`each` + `delete` inside the loop**: safe when you delete only the current key (the key just returned by `each`). Deleting *other* keys during iteration can skip entries or revisit them - Perl's hash iterator is not stable across structural change. The safe rewrite is to collect keys first:

```
for my $k (grep { $h{$_} < 0 } keys %h) {
    delete $h{$k};
}
```

- **Trailing array deletions shrink the array**: if the deleted elements are at the top end, `$#array` drops to the highest index still testing true under `exists`. Holes in the middle stay as holes.
- **`delete` on an array is discouraged**: the concept of "missing index" is not coherent for arrays - indices are positional, not named. Leave array shrinkage to `shift`, `pop`, or `splice`. `WARNING` in upstream POD restated: reach for an array `delete` only when you know you want a hole.
- **`delete local $h{k}`**: removes the entry for the enclosing block and reinstates it on scope exit. Useful for temporarily hiding configuration keys from code called inside the block without losing the original value:

```
{
    local $ENV{LANG};         # still present, but undef
    delete local $ENV{PATH};  # gone for this block only
    run_thing();
}   # both restored to original values here
```

- **Tied hash, tied array**: `delete` calls `DELETE`; the return value is whatever the tie implementation supplies. A tied DBM hash deletes the entry on disk.
- **Readonly target**: `delete` on a readonly element croaks with `Modification of a read-only value attempted`.
- **The expression must end in an element or slice**: `delete $h{k} + 1` is a compile-time error; `delete` has lower precedence than arithmetic and expects an lvalue-ish aggregate lookup, not a computed value.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `exists` - the query counterpart; use it before `delete` when you need to tell "was never there" apart from "was there with value `undef`"
- `each` - hash iterator; safe to combine with `delete` only when you delete the current key
- `keys` - snapshot of hash keys; the usual way to build a deletion list that won't interact with iterator state
- `values` - companion to `keys` when you want the deleted-value set without the keys
- `splice` - the right tool when you want to remove array elements *and* renumber remaining indices
- `undef` - clears a whole aggregate (`undef %h`, `undef @a`) or a single scalar; the customary way to empty a hash or array wholesale rather than deleting every key

I/O · Control flow

die

Raise an exception.

`die` throws an exception built from `LIST`. If the exception happens inside an `eval`, the `eval` terminates with `undef` and `$@` holds the exception value. If nothing catches it, Perl prints the message to `STDERR` and the interpreter exits with a non-zero status. The exception can be a plain string, a list that stringifies into one, or a reference - typically a blessed object carrying structured error state.

Synopsis

```
die LIST
die $message
die $exception_object
die                          # re-raise / propagate current $@
```

What you get back

`die` does not return. Control transfers to the nearest dynamically enclosing `eval` (which then returns `undef` in scalar context or the empty list in list context), or - if there is no enclosing `eval` - the interpreter unwinds, runs `END` and `DESTROY` handlers, and exits.

The exception value itself lands in `$@`:

- **String exceptions** - the stringified message, including any appended `at FILE line N.\n` (see below).
- **Reference exceptions** - the reference itself, unchanged. This is the idiomatic way to throw a blessed exception object.

On uncaught exit, the exit code is derived from `$!` and `$?`:

```
exit $! if $!;              # errno
exit $? >> 8 if $? >> 8;    # child exit status
exit 255;                   # last resort
```

Rely only on the exit code being non-zero; the exact value is a best-effort encoding of whatever global error state was live at the moment of the throw.

Global state it touches

- `$@` - written with the exception on unwind into an `eval`, read by the no-argument form of `die` to propagate an existing exception, and read by the uncaught-exception printer.
- `$!` - read when deriving an exit code for an uncaught exception, and commonly interpolated into the message itself (`die "open $f: $!"`).
- `$?` - read when deriving an exit code if `$!` is zero. `END` and `DESTROY` handlers can still mutate it before the process actually exits.
- `$.` - the current input line number, appended to string exceptions that lack a trailing newline.
- `$SIG{__DIE__}` - if set, the handler runs *before* the exception is propagated, with the exception as its argument. The handler can replace the exception by calling `die` again with a different value.
- `$^S` - readable by `$SIG{__DIE__}` handlers to tell whether the die is happening inside an `eval`.

The trailing-newline rule

Whether `die` appends location information depends on one character at the end of the message:

- **Message ends in `"\n"`** - used verbatim. Nothing is appended. Use this form when the message is already a complete, user-facing diagnostic.
- **Message does not end in `"\n"`** - Perl appends `" at FILE line N"`, and if a file is being read, `", <HANDLE> line M"`, then a final `"."` and newline. Use this form during development, or whenever locating the `die` call in the source is more useful than a tidy message.

```
die "bad config\n";                 # "bad config\n"
die "bad config";                   # "bad config at script.pl line␣
 ↪42.\n"
```

A common idiom is to append `", stopped"` to make the location suffix read naturally:

```
die "/etc/games is no good, stopped";
# /etc/games is no good, stopped at canasta line 123.
```

Reference exceptions never get location information appended - stringifying an object to inspect it would defeat the point of throwing an object. If you want file and line in-

side a structured exception, capture them at construction time (`__FILE__`, `__LINE__`, or `caller`).

Reference-valued exceptions

Passing a reference - typically a blessed object - lets `$@` carry structured state instead of a string you later have to parse:

```
package MyApp::Error;
sub new   { my ($c, %a) = @_; bless { %a }, $c }
sub code  { $_[0]{code}  }
sub where { $_[0]{where} }

eval {
    die MyApp::Error->new(code => 'EPARSE', where => 'line 12');
};
if (my $err = $@) {
    if (ref($err) && $err->isa('MyApp::Error')) {
        warn "parse error at ", $err->where;
    }
    else {
        die $err;   # not ours - re-raise
    }
}
```

Overload stringification on exception classes so that an uncaught exception still prints a useful message:

```
use overload '""' => sub { "[MyApp::Error ".$_[0]->code."]\n" };
```

The stringified form should end in a newline and be non-empty, matching the treatment of string exceptions.

Because `$@` is a global, copy it into a lexical before inspecting it - any method call, `ref`, or `isa` you run during handling can itself `eval` internally and clobber `$@`:

```
if (my $err = $@) { ... }       # safe
if ($@) { $@->method; ... }     # unsafe: method may reset $@
```

Propagation: `die` with no argument (or empty string)

When `LIST` is empty or stringifies to `""`, `die` does not throw a fresh exception - it re-raises whatever is in `$@`:

- If `$@` holds a **string**, `die` appends `"\t...propagated at FILE line N.\n"` and re-throws. This is the canonical way to let an exception continue up the call stack after a partial-match check:

  ```
  eval { risky() };
  die unless $@ =~ /Expected exception/;
  ```

- If `$@` holds an **object reference with a `PROPAGATE` method**, that method is invoked as `$@ = eval { $@->PROPAGATE(__FILE__, __LINE__) }`, and the re-

turn value replaces `$@` before the re-throw. Implement `PROPAGATE` on an exception class when you want the exception to record the chain of re-raises.

- If `$@` holds an object reference with **no** `PROPAGATE`, the reference is re-thrown unchanged.
- If `$@` is also empty, the string `"Died"` is used.

The `$SIG{__DIE__}` hook

Setting `$SIG{__DIE__}` installs a handler that runs when `die` fires, with the exception as its argument. The handler can inspect or replace the exception by calling `die` again with a different value. It runs **even inside `eval`**, which is the usual reason to want it out of the way:

```
local $SIG{__DIE__} = sub {
    die @_ if $^S;              # inside eval: do nothing special
    log_fatal(@_);              # outside eval: log before we exit
};
```

`$^S` is `undef` during compilation, true inside an `eval`, and false at the top level - use it to gate handler work that only makes sense for uncaught exceptions.

Examples

Bail out with a clean message (note the trailing newline suppresses file/line appending):

```
chdir '/var/spool' or die "can't cd to spool: $!\n";
```

Die without a newline during development so the location is appended:

```
die "unexpected empty record";
# unexpected empty record at parse.pl line 87.
```

Catch a string exception:

```
eval {
    parse($input);
};
if ($@) {
    warn "parse failed: $@";    # $@ already ends in "\n" if the
                                # die'd message did
}
```

Throw and catch a blessed exception object:

```
eval {
    die MyApp::Error->new(code => 'EIO', detail => $!);
};
if (my $err = $@) {
    if (ref($err) && $err->isa('MyApp::Error')) {
        handle($err);
```

(continues on next page)

(continued from previous page)

```
    }
    else {
        die $err;               # re-raise anything unexpected
    }
}
```

Conditional propagation - catch one class, let the rest through:

```
eval { work() };
die unless $@ =~ /^recoverable:/;
recover();
```

Edge cases

- **List with two or more items** is stringified and concatenated to form the message: `die "read failed on ", $file, ": $!"`. The newline rule applies to the concatenated result.
- **Empty list** (`die;` or `die "";`) triggers the propagation path described above, not a fresh exception.
- **Reference with trailing newline inside the stringified form** is irrelevant: location is never appended to a reference, regardless of what its `""` overload produces.
- **`die` inside `DESTROY` during global destruction** can still fire `$SIG{__DIE__}` but may not propagate as expected - the interpreter is already unwinding. Keep destructors defensive.
- **Uncaught `die` during `END` block** runs remaining `END` blocks but sets exit status per the `$!`/`$?` rule above.
- **Nested `eval` and `$SIG{__DIE__}`**: the handler runs for *every* die, including those caught by an inner `eval`. Guard with `$^S` if the handler has side effects.
- **Re-throwing the current `$@`**: prefer bare `die;` over `die $@;`. The bare form triggers the propagation machinery (`...propagated` suffix, `PROPAGATE` hook); `die $@` does not - it treats `$@` as a fresh exception value.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `eval` - the only construct that catches a `die`; its block form traps exceptions into `$@`
- `warn` - same formatting rules (trailing-newline suppresses location), but writes to `STDERR` and does not unwind
- `caller` - capture file and line at throw time when building a structured exception object
- `exit` - use when you want a specific exit code without the exception-handling machinery

- `$@` - where the exception lands after an `eval` catches it, and the source the no-arg `die` re-raises from
- `$SIG{__DIE__}` - pre-throw hook that can inspect or rewrite the exception
- `$^S` - tells a `$SIG{__DIE__}` handler whether it is running inside an `eval`

Control flow · Modules

do

Execute a block of code or run a Perl source file as if it were part of the current program.

do wears two unrelated hats that share only a keyword. do BLOCK groups statements into a single expression and returns the value of the last one - a syntactic tool, not a function. do EXPR treats the string as a filename and compiles-and-runs the file at runtime. Read whichever section matches the form you see; mixing them up is the most common source of confusion around this keyword.

Synopsis

```
do BLOCK                          # group statements, yield last value
do EXPR                           # load and run a Perl file
do { ... } while CONDITION;       # post-test loop idiom
```

What you get back

- do BLOCK - the value of the last expression evaluated in the block, in whatever context the surrounding expression imposes.
- do EXPR - the value of the last expression evaluated in the file on success; `undef` on failure, with `$@` set to the compile error or `$!` set to the I/O error.

do BLOCK - group statements into one expression

do BLOCK is not a function. It is a syntactic construct that turns a brace-delimited sequence of statements into a single term whose value is the value of the last statement. Reach for it when the language wants an expression but you have several statements to run:

```
my $x = do {
    my $tmp = fetch_raw();
    $tmp =~ s/\s+\z//;
    lc $tmp;
};                                # $x = cleaned, lower-cased value
```

When do BLOCK is modified by a trailing `while` or `until`, the block runs **once before** the condition is tested - this is the standard post-test loop idiom:

```
do {
    $line = <$fh>;
} while defined $line && $line !~ /^END\b/;
```

On any other statement, `while` and `until` as statement modifiers test the condition first. do is the exception.

next, last, redo do not work inside do BLOCK

A do BLOCK is **not** a loop, even when trailed by `while` or `until`. Loop-control keywords do not see it:

```
do {
    last if $done;              # WRONG - not a loop
} while $more;
```

If you need loop control, wrap the body in a bare block, which *is* a loopable scope:

```
{
    last if $done;
    redo if $retry;
}
```

Or use an explicit `while` loop.

return inside a bare do BLOCK

`return` in a do BLOCK returns from the **enclosing subroutine**, not from the block itself. That is occasionally what you want and occasionally a bug - either way, know which one you are writing:

```
sub classify {
    my $n = shift;
    my $kind = do {
        return "zero" if $n == 0;   # returns from classify(), not
↪the do
        $n > 0 ? "pos" : "neg";
    };
    "$kind ($n)";
}
```

To yield a value from the block without returning from the sub, just let the last expression be the value - that is the whole point of do BLOCK.

do EXPR - load and run a Perl source file

do EXPR evaluates EXPR, treats the result as a filename, reads the file, compiles it, and runs it in the current package. It is similar to, but not quite the same as,

```
eval `cat stat.pl`;
```

The do form runs no external process, keeps the original filename for error messages, and **cannot see lexicals in the enclosing scope** - which eval STRING can. Like eval STRING, the file is reparsed on every call, so do FILE inside a loop is almost always wrong.

Path resolution and @INC

The filename is interpreted like this:

- Absolute paths (`/foo/stat.pl`), paths beginning with `./`, and paths beginning with `../` are used as-is.
- Any other relative path is searched along `@INC`, and on success the found path is recorded in `%INC` under the key you passed.

```
do '/etc/myapp/stat.pl';        # exact file
do './stat.pl';                 # exact file, current directory
do 'stat.pl';                   # search @INC
do 'MyApp/config.pl';           # search @INC
```

Since Perl 5.26 `.` is not in `@INC` by default, so a bare `do 'stat.pl'` no longer falls back to the current directory. If the file is not found, Perl emits the hint:

```
do "stat.pl" failed, '.' is no longer in @INC;
did you mean do "./stat.pl"?
```

Pass `./stat.pl` when you mean the current directory.

Return value and error reporting

`do EXPR` has three outcomes you must distinguish:

Outcome	Return	`$@`	`$!`
File compiled and ran	value of last expression	`""`	unchanged
File read but did not compile	`undef`	compile error	may also be set
File could not be read	`undef`	`""`	I/O error

Because compilation failure can incidentally set `$!` as well, **always check `$@` first**:

```
my $return;
for my $file ("/etc/myapp.rc", "$ENV{HOME}/.myapprc") {
    unless ($return = do $file) {
        warn "couldn't parse $file: $@" if $@;
        warn "couldn't do $file: $!"    unless defined $return;
        warn "couldn't run $file"       unless $return;
    }
}
```

The three `warn`s correspond to the three failure shapes above plus the fourth case - the file ran but its last expression was false.

Why not use do FILE for modules

For loading library code, use `require` or `use` instead of do EXPR. `require` caches through %INC (so a file is loaded at most once), raises an exception on failure instead of returning `undef`, and integrates with @INC hooks. `use` adds compile-time loading and import handling on top. do EXPR exists for configuration files and ad-hoc script inclusion - not for libraries.

Examples

Group statements into an expression:

```
my $config = do {
    open my $fh, "<", $path or die $!;
    local $/;
    <$fh>;
};                                # $config = whole file as one string
```

Post-test loop with do … while:

```
my $n;
do {
    print "enter a positive number: ";
    chomp($n = <STDIN>);
} until defined $n && $n =~ /\A\d+\z/ && $n > 0;
```

Read a Perl-syntax configuration file, with full three-way error handling:

```
my $cfg = do '/etc/myapp/config.pl';
if (!defined $cfg) {
    die $@ ? "config parse error: $@"
           : "config read error: $!";
}
die "config did not return a true value" unless $cfg;
```

Conditional value selection where each branch is several statements:

```
my $greeting = do {
    if ($user) {
        my $name = $user->display_name;
        "hello, $name";
    }
    else {
        "hello, stranger";
    }
};
```

Temporary localisation that must span several statements and still yield a value:

```
my $dump = do {
    local $Data::Dumper::Sortkeys = 1;
    local $Data::Dumper::Indent   = 1;
```

(continues on next page)

(continued from previous page)

```
    Data::Dumper::Dumper(\%state);
};
```

Global state it touches

- `@INC` - searched when `do EXPR` is called with a relative path other than `./…` or `../…`.
- `%INC` - updated with the resolved path when `do EXPR` succeeds on an `@INC` search.
- `$@` - set by `do EXPR` on compile failure; cleared on success.
- `$!` - set by `do EXPR` on I/O failure (file could not be read).
- `$0`, `__FILE__`, `__LINE__` - the currently compiling filename is set to the file being loaded while `do EXPR` parses it, so warnings and `die` messages point at the right file.

`do BLOCK` touches no globals; it is a pure syntactic construct.

Edge cases

- **`do BLOCK` is not a loop.** `next`, `last`, and `redo` inside a `do … while` do not apply to the block. Use a bare block or a real `while` loop.
- **`return` in `do BLOCK` returns from the enclosing sub.** Not from the block. There is no way to return a value from a `do BLOCK` other than making it the last expression evaluated.
- **Statement modifiers are pre-test everywhere else.** `STATEMENT while COND` tests first; `do BLOCK while COND` tests after the first run. This is a deliberate language exception.
- **Relative paths without `./` search `@INC`.** Since 5.26, `.` is not in `@INC`, so `do 'stat.pl'` does not mean "current directory file" any more; pass `./stat.pl` if that was the intent.
- **`do EXPR` cannot see enclosing lexicals.** The file is parsed as if at the top level. If you need access to lexicals, use `eval STRING` - with the usual taint and injection caveats.
- **The file is re-parsed every call.** `do EXPR` is not cached via `%INC` the way `require` is. Calling it in a loop re-reads and re-compiles the file each iteration.
- **Check `$@` before `$!`.** A compilation failure can also set `$!` as a side effect; the compile error in `$@` is the one you actually want to report.
- **A file that compiles but returns a false value is ambiguous.** `unless ($ret = do $file) { … }` fires for both "file failed" and "file ran and returned `0`". If your config file is allowed to end in a false value, use `defined` and `$@` explicitly instead of truth-testing the return.
- **`do` with no argument is a syntax error.** Both forms require a term: a block or an expression.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `eval` - `eval BLOCK` traps exceptions without reading a file; `eval STRING` parses arbitrary code and **can** see enclosing lexicals, unlike do `EXPR`
- `require` - load a library file with `%INC` caching and automatic exception on failure; the right tool for modules
- `return` - note that `return` inside do `BLOCK` returns from the enclosing subroutine, not from the block
- `@INC` - module search path consulted by do `EXPR` on relative filenames
- `%INC` - records the resolved path after a successful `@INC`-searched do `EXPR`
- `$@` - compile-time error from do `EXPR`
- `$!` - I/O error from do `EXPR` when the file cannot be read

Control flow

dump

Cause the running program to produce a core dump immediately.

dump is a pre-historic Perl operator whose original purpose was to freeze a running interpreter to a core file so that the (separately supplied) undump utility could turn that core file back into an executable. The resulting binary, when run, would restart the program by jumping to `LABEL` via `goto`, skipping the startup cost of re-reading and re-compiling the source. That workflow has been dead for decades: undump was never shipped with Perl, modern operating systems make core-to-executable conversion essentially impossible, and the whole operator is now a historical curiosity. As of Perl 5.30 it must be spelled `CORE::dump()` to discourage accidental use.

There is almost no situation in modern Perl where dump is the right answer. If you reached for it because you want a stack trace, use `Carp::confess`. If you want to exit abnormally, use `die` or `exit`. If you want a snapshot of the program state, use a checkpointing tool at the OS level.

Synopsis

```
CORE::dump();
CORE::dump LABEL;
CORE::dump EXPR;
```

What you get back

Nothing. dump does not return: control leaves the Perl interpreter the way `abort(3)` leaves a C program. The process is terminated by the kernel's core-dump handling (`SIGABRT`-style), with the usual side effects: an exit status reflecting the signal, a possible

`core` file in the current directory if the user's `RLIMIT_CORE` and the kernel's `core_pattern` allow it, and no orderly shutdown of the Perl runtime. `END` blocks do **not** run. `DESTROY` methods do **not** run. Buffered output on any filehandle is lost.

Global state it touches

None directly. `dump` bypasses the normal end-of-program machinery entirely - it does not read or write special variables, does not flush buffers, and does not run destructors. Whatever state the interpreter happens to be in at the moment of the call is what lands in the core file.

Examples

Trigger a core dump from a running program (requires `CORE::` as of 5.30):

```
CORE::dump();
```

With a label - originally, the undump-produced binary would resume execution here. With undump gone this is purely decorative; the process still just aborts:

```
RESTART:
    # ... initialisation ...
CORE::dump RESTART;
```

Runtime-computed label name (the `dump EXPR` form, available since Perl 5.18):

```
my $label = pick_entry_point();
CORE::dump $label;
```

What you almost certainly want instead. For a post-mortem with a stack trace:

```
use Carp;
Carp::confess("unreachable state: $state");
```

For a clean abnormal exit with a status code:

```
die "fatal: $reason\n";      # exit via $@, runs END / DESTROY
exit 1;                      # plain exit, runs END / DESTROY
```

Edge cases

- **Spelling since 5.30**: a bare `dump` or `dump LABEL` at statement level is a compile-time error unless written as `CORE::dump` or `CORE::dump(...)`. This is deliberate - the core team made the operator hard to reach by accident.
- **`LABEL` is not resolved at call time**: historically the label was looked up only after undump reincarnated the program, so a typo would not be caught at dump-time. Today the point is moot - the program never resumes, so the label is pure documentation.
- **`dump EXPR`**: the expression is evaluated in scalar context and must yield a label name. It has no effect on the outcome, for the same reason.

- **Parser quirks**: dump has the same precedence as assignment and is exempt from the looks-like-a-function rule. `dump ("foo") . "bar"` parses with `"bar"` as part of the argument, not as a post-concatenation. In practice this only matters if you are writing dump at all, which you should not be.
- **Open files are lost**: any filehandle open at the moment of dump is gone when (in the historical workflow) the reincarnated program starts. The POD warning about "possible resulting confusion by Perl" dates from the era when someone might actually try this; today the warning is academic because nothing resumes.
- **Core files may not be produced**: whether a core file actually lands on disk depends on `ulimit -c`, the kernel's `kernel.core_pattern`, filesystem permissions, and whether the process is running under a supervisor that captures or discards core files. dump succeeding in aborting the process is not the same as a core file being available afterwards.
- **Portability**: on systems without meaningful core-dump support the operator reduces to "abort the process"; see `perlport` for historical notes.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `exit` - terminate the program normally, running `END` blocks and destructors; this is what you almost certainly want instead of dump
- `die` - raise an exception; with no handler it terminates the program but still runs `END` blocks and flushes output
- `goto` - the control-transfer operator whose `LABEL` form dump was historically meant to cooperate with after reincarnation
- `Carp::confess` - stack-trace-on-exit, the modern answer for "I want to see program state at the point of failure"
- `warn` - emit a diagnostic without terminating; combine with a stack-trace module when you want information without aborting
- `perlrun` - documents the historical `-u` command-line switch that performs a dump immediately after compilation

Arrays · Hashes

each

Step through a hash or array one entry at a time.

`each` advances a per-container iterator and hands back the next entry. For a hash that entry is a `(key, value)` pair; for an array it is an `(index, value)` pair. When the iterator has walked every entry, the next call returns the empty list (or `undef` in scalar context) and then rewinds, so a fresh `while (each ...)` loop starts from the beginning. Hash order is randomised per process and per hash; array order is strictly lowest index first.

Synopsis

```
while (my ($k, $v) = each %h) { ... }
while (my ($i, $v) = each @a) { ... }
while (each %h) { ... }              # sets $_ to the key
my $k = each %h;                     # scalar context: key/index only
```

What you get back

- **List context**: a two-element list `($key, $value)` for a hash, `($index, $value)` for an array. On exhaustion, the empty list.
- **Scalar context**: the key (for a hash) or the index (for an array). On exhaustion, `undef`.
- **Bare `each %h` in a `while`/`for` condition**: tests *definedness* of the returned key/index, not truth, and assigns the key to `$_`. This is what makes `while (each %h) { print "$_\n" }` work even when a key is the empty string or `0`.

After exhaustion the iterator is implicitly reset, so the call *after* the one that returned empty restarts iteration from the top.

Global state it touches

Every hash and every array carries **its own internal iterator**. `each`, `keys`, and `values` all advance and share that one iterator - they are three views of the same cursor, not three independent loops. This is the single most important fact about `each`:

- Calling `keys %h` or `values %h` (in any context, even as a one-off count like `scalar keys %h`) **resets** the hash's iterator. A `while (each %h)` loop interrupted by `scalar keys %h` in the body will restart from the top on the next iteration - or worse, loop forever.
- **Referencing a hash in list context resets its iterator too** (e.g. `my @copy = %h`, or passing %h to a sub). Array iterators are *not* reset by list-context reference.
- Two loops iterating the same %h share one cursor. Nesting `while (each %h) { ... while (each %h) { ... } ... }` visits each entry at most once total, not `n * m` times.
- An anonymous hash/array in the condition (`each %{ +{ a => 1 } }`) builds a *new* container every iteration, so the iterator is always fresh - the loop never terminates.
- The bare-`each` form assigns to `$_`; if you rely on `$_` elsewhere in the loop body, save it with `local`.

Every entry to a `while (each ...)` loop picks up whatever iterator position the container happens to be at. There is no way to isolate a loop's iterator from other code that touches the same container. If the loop body calls a function that might read the same hash, the safe pattern is a `foreach` loop over `keys` instead.

Examples

Basic hash iteration:

```
my %colour = (red => "#f00", green => "#0f0", blue => "#00f");
while (my ($name, $hex) = each %colour) {
    print "$name = $hex\n";
}
# order is unspecified
```

Array iteration (Perl 5.12+):

```
my @letters = ('a' .. 'e');
while (my ($i, $c) = each @letters) {
    print "[$i] $c\n";
}
# [0] a
# [1] b
# [2] c
# [3] d
# [4] e
```

Printing a hash in sorted order - `each` does **not** sort, so reach for `keys` plus `sort` instead:

```
for my $k (sort keys %colour) {
    print "$k = $colour{$k}\n";
}
```

Bare `each` in a `while` condition, which sets `$_` to the key every iteration (Perl 5.18+):

```
while (each %ENV) {
    print "$_=$ENV{$_}\n";
}
```

Safe deletion during iteration - you may delete the key *just returned* by `each` without disturbing the cursor:

```
while (my ($key, $value) = each %hash) {
    delete $hash{$key} if $value < 0;   # safe
}
```

Iterating a hash with a `foreach` loop when anything in the body might touch `%hash` (the robust pattern - snapshot the keys first):

```
for my $key (keys %hash) {
    my $value = $hash{$key};
    # body is free to read %hash, call subs that read it, even
    # mutate it cautiously - our iteration list is fixed already.
}
```

Multi-value `foreach` over a hash (Perl 5.36+), which makes `each` unnecessary for most loops and sidesteps the shared-iterator trap:

```
foreach my ($key, $value) (%hash) {
    print "$key => $value\n";
}
```

Edge cases

- **Iterator not reset between re-entries**: a `while (each %h)` loop that exits early via `last`, `return`, or `die` leaves the iterator mid-walk. The next `while (each %h)` on the same hash picks up where the first left off, not at the start. Call `keys %h` in void context - `keys %h;` - to force a reset before the new loop.
- **`keys`/`values` resets the iterator**: any call to `keys` or `values` on the same hash - including `scalar keys %h` inside the loop body - rewinds the cursor.
- **List-context reference resets a hash's iterator, not an array's**: `my @flat = %h` rewinds %h's cursor; the equivalent `my @copy = @a` does not touch @a's iterator.
- **Delete-during-iteration is unspecified** except for the entry most recently returned. Deleting any other key may cause entries to be visited twice or skipped entirely.
- **Add-during-iteration is unspecified** - inserting a new key mid loop may or may not be visited, and may disturb the order of entries not yet seen.
- **Anonymous container in the condition**: `each %{ +{ a => 1 } }` or `each @{ [1,2,3] }` constructs a fresh container each time the condition is evaluated, so the iterator is always new and the loop never terminates.
- **Tied hashes and arrays** delegate to `FIRSTKEY`/`NEXTKEY` (hashes) or `FETCHSIZE`/`FETCH` (arrays), so iteration order is whatever the tied class chooses and none of the guarantees above necessarily hold.
- **Definedness vs truth in the loop condition**: `while (each %h)` and `while (my $k = each %h)` both test *definedness* of the expression, so a key of `"0"` or `""` does not terminate the loop. This is special-cased for `each` specifically.
- **Scalar expressions**: `each $href` on a scalar holding a reference was an experimental feature in 5.14 and was **removed in 5.24**. Dereference explicitly: `each %$href`, `each @$aref`.
- **Shared state across subs**: passing %h to a sub that calls `each %h`, `keys %h`, or `values %h` on it perturbs your caller's iterator. If you publish a hash, assume callers may break your loop; prefer `foreach (keys ...)` for robustness.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `keys` - snapshot of all keys (hash) or indices (array); **resets the iterator** `each` uses
- `values` - snapshot of all values; **resets the iterator** `each` uses

- `delete` - safe for the entry just returned by `each`, unsafe for any other during iteration
- `exists` - test key/index membership without perturbing the iterator
- `for` / `foreach` - the safer loop when the body might touch the container; combine with `keys` to snapshot first
- `sort` - pair with `keys` when you need a defined iteration order; `each` never sorts

User and group info

endgrent

Close the group database after iterating it.

`endgrent` is the cleanup half of the group-database iteration trio `setgrent` / `getgrent` / `endgrent`. It tells the C library that you are done walking `/etc/group` (or the NSS source behind it), releasing any file descriptors or cached state the library opened on your behalf. After `endgrent`, the next `getgrent` call starts a fresh iteration from the top.

Synopsis

```
endgrent;
endgrent();
```

No arguments, no meaningful return value.

What you get back

The return value is unspecified and must not be relied on. `endgrent` is called for its side effect - closing the group database - not for a result. Treat it as a void call.

```
endgrent;                               # correct
my $ok = endgrent;                      # wrong: $ok is not a success␣
↪flag
```

If your platform does not implement `endgrent`, the call croaks with `The endgrent function is unimplemented` at runtime. This is effectively a compile-once check - if the function works on your system at all, it will continue to.

Global state it touches

`endgrent` operates purely on the libc-internal group-iteration state that `setgrent` and `getgrent` share. It reads and writes no Perl special variables. It does not alter `$!`: a failure path that sets `errno` in the underlying C call is not surfaced to Perl.

The iteration state is **per-process**, not per-thread. In a threaded program, one thread's `endgrent` closes the iteration for every thread - see *Edge cases*.

Examples

The canonical iterate-and-close pattern. Always pair a `setgrent` / `getgrent` loop with `endgrent` so long-running programs do not leak the file descriptor the C library keeps open:

```
setgrent;
while (my @g = getgrent) {
    my ($name, $passwd, $gid, $members) = @g;
    print "$name ($gid): $members\n";
}
endgrent;
```

Dropping into `endgrent` mid-iteration is legal - it simply abandons whatever cursor position the library was holding:

```
setgrent;
while (my @g = getgrent) {
    last if $g[0] eq 'wheel';       # found what we wanted
}
endgrent;                           # release the handle anyway
```

Restarting iteration. `endgrent` followed by `setgrent` is the portable way to rewind:

```
setgrent;
my @first_pass  = collect_groups();
endgrent;

setgrent;
my @second_pass = collect_groups();  # starts over from the top
endgrent;
```

Defensive cleanup in a `END` block for programs that might die mid-walk:

```
setgrent;
END { endgrent }
while (my @g = getgrent) { ... }    # any die still triggers END
```

Edge cases

- **Calling endgrent without a prior `setgrent` or `getgrent`** is harmless. The C library silently does nothing if no iteration is in progress.
- **Calling it twice in a row** is also harmless - the second call is a no-op.
- **Parser ambiguity.** `endgrent` takes no argument and has the precedence of a named unary operator. A bareword expression immediately to its right will be parsed as an argument and rejected:

```
endgrent or die;                    # WRONG: parses as␣
↪endgrent(or die)
endgrent() or die;                  # right: empty parens␣
↪disambiguate
```

In practice there is nothing to check the return value against, so this trap rarely bites - but the same rule applies to any chained expression.

- **Threads share the iteration.** `endgrent` from one thread ends iteration for all threads of the same process, because the underlying state is process-global. Do not interleave `getgrent` calls across threads expecting each to see every record.
- **NSS and caching.** On systems where the group database is served through NSS (nss_ldap, nss_sss, etc.), `endgrent` releases per-module caches that the NSS backend may hold. Long-lived daemons that walk the group database periodically should call `endgrent` after each walk to avoid accumulating backend state.
- **No effect on name-lookup calls.** `getgrnam` and `getgrgid` are independent of the iteration cursor. `endgrent` does not invalidate their results or cached data, and you do not need to call `endgrent` before or after them.
- **Unimplemented on some platforms.** Perl croaks `The endgrent function is unimplemented` if the host libc has no `endgrent(3)`. All modern Linux, BSD, and macOS systems provide it.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `setgrent` - open or rewind the group database before walking it; always the prologue to an iteration that ends with `endgrent`
- `getgrent` - pull the next record from the iteration `endgrent` closes
- `getgrnam` - look up one group by name, independent of the iteration cursor; no `endgrent` needed
- `getgrgid` - same as `getgrnam` but keyed on numeric gid
- `endpwent` - the parallel cleanup call for the passwd database; the two iterations are independent and both need closing if you opened both

Network info

endhostent

Close the hosts database after iteration.

`endhostent` releases any resources held by the hosts database after a walk driven by `gethostent`. It pairs with `sethostent`, which opens or rewinds the database, and is a direct wrapper around the C library call of the same name. Calling it when no iteration is in progress is harmless.

Synopsis

```
endhostent;
endhostent();
```

`endhostent` takes no arguments. Parentheses are optional.

What you get back

No useful return value. The call is made for its side effect - tearing down whatever per-process state the platform's `gethostent(3)` keeps between iterations (file descriptors on traditional `/etc/hosts` iteration, resolver sockets on NSS-backed systems).

Global state it touches

- The hosts-database iteration cursor shared by `gethostent`, `sethostent`, `gethostbyname`, and `gethostbyaddr`. After `endhostent`, the next `gethostent` call restarts from the beginning (typically after an implicit `sethostent`).
- `$?` carries the C-library `h_errno` value on systems where host lookups surface it. Most implementations of `endhostent` itself do not touch `h_errno`, but a subsequent `gethostent` can.

This state is per-process, not per-thread. Two threads iterating hosts concurrently will interleave records and neither will see the full set - pair `sethostent` / `gethostent` / `endhostent` within one thread.

Examples

Close the database after a full walk:

```
sethostent(1);                          # 1 = keep connection open
while (my @h = gethostent()) {
    print "$h[0]\n";                    # canonical hostname
}
endhostent();
```

Always pair `endhostent` with `sethostent` in a subroutine so the iteration state does not leak to the caller:

```
sub list_hosts {
    sethostent(0);
    my @names;
    while (my ($name) = gethostent()) {
        push @names, $name;
    }
    endhostent();
    return @names;
}
```

Defensive cleanup - `endhostent` is safe even when no iteration is in progress, so it is fine to call unconditionally on an error path:

```
eval {
    sethostent(1);
    process_hosts();
    1;
} or do {
    my $err = $@;
```

(continues on next page)

(continued from previous page)

```
    endhostent();
    die $err;
};
endhostent();
```

Using the by-name accessor object interface from `Net::hostent` does not change the rule - the underlying iterator is the same one `endhostent` closes:

```
use Net::hostent;
sethostent(1);
while (my $h = gethostent()) {
    print $h->name, "\n";
}
endhostent();
```

Edge cases

- **No-op when nothing is open**: calling `endhostent` without a prior `sethostent` or `gethostent` is defined and does nothing observable.
- **Interleaving with name / address lookups**: a `gethostbyname` or `gethostbyaddr` call in the middle of an iteration may reset the cursor on some platforms (any implementation that internally calls `sethostent(0)` for single-shot lookups). Do not mix the two styles against the same database without re-calling `sethostent` afterwards.
- **Platforms without a hosts iterator**: on systems whose resolver does not support enumeration (most modern NSS setups querying DNS only), `gethostent` returns the empty list immediately. `endhostent` is still callable and still a no-op.
- **Thread safety**: the host iterator is process-global. In a multi-threaded program, guard the `sethostent` / `gethostent` / `endhostent` trio with your own mutex.
- **Prototype / parsing**: `endhostent` has no arguments, so there is no parser-ambiguity trap. Writing `endhostent LIST` is a syntax error rather than being silently interpreted as a call with arguments.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `sethostent` - open or rewind the hosts database; the opening half of the iteration pair
- `gethostent` - read one record from the open database; returns the empty list when the iteration is exhausted
- `gethostbyname` - single-shot lookup by name; does not require `sethostent` / `endhostent`
- `gethostbyaddr` - single-shot lookup by address

- endnetent - same pattern for the networks database
- endservent - same pattern for the services database
- $? - receives h_errno from failed host lookups on platforms where the C library exposes it

Network info

endnetent

Close the networks database after iteration.

endnetent releases the resources held by the networks database - typically the /etc/networks file or its NSS-backed equivalent - that were opened implicitly by the first call to getnetent, getnetbyname, or getnetbyaddr, or explicitly by setnetent. After endnetent the iteration cursor is gone; a subsequent getnetent starts a fresh sweep from the top of the database. It is a thin wrapper over the system C library's endnetent(3).

Synopsis

```
endnetent;
endnetent();
```

What you get back

A system-defined true value. The C function returns void; Perl surfaces a placeholder so the call can stand as a statement. Do not compare the return value against anything - there is nothing useful in it.

Global state it touches

endnetent operates on the process-wide networks-database cursor owned by libc. That cursor is shared with getnetent, getnetbyname, getnetbyaddr, and setnetent; a call here invalidates the iteration state those functions depend on. No Perl special variable is touched.

Examples

Paired with setnetent and getnetent for a complete sweep of the networks database:

```
setnetent(1);                          # stayopen = 1
while (my @net = getnetent()) {
    my ($name, $aliases, $addrtype, $net) = @net;
    print "$name\t$net\n";
}
endnetent();                           # release the cursor
```

Close after the handful of lookups you actually needed, so the database file does not stay open for the rest of the program:

```
setnetent(0);
my @loopback = getnetbyname("loopback");
my @link     = getnetbyname("link-local");
endnetent();
```

Safe to call when the database was never opened - libc treats it as a no-op:

```
endnetent();                        # harmless on a fresh process
```

Edge cases

- **No arguments, list-operator precedence.** `endnetent` takes no arguments but parses as a list operator, so a stray term after it is swallowed: `endnetent $x` is `endnetent($x)` and produces a "Too many arguments" compile-time error. Write `endnetent;` or `endnetent();`.
- **Repeated calls.** Calling `endnetent` twice in a row is a no-op on the second call on every libc we target. Do not rely on it raising an error.
- **Interleaving with other DB iterators.** The networks cursor is independent of the cursors used by `getpwent`, `getgrent`, `gethostent`, `getprotoent`, and `getservent`. `endnetent` does not close those - call the matching `end*ent` for each iterator you opened.
- **Forking between `setnetent` and `endnetent`.** The cursor lives in the parent's libc state; a child inherits it at `fork` time but further `getnetent` calls in either process will desynchronise. Close and re-open in the child.
- **NSS backends.** On glibc the networks database routes through NSS, so the meaning of "close" depends on the configured service (`files`, `nis`, `ldap`, ...). User-visible behaviour - iteration restarts on the next `getnetent` - is the same; the cost of `endnetent` is not.
- **Thread safety.** The libc networks cursor is a single process-global resource. pperl does not serialize access to it; a program that iterates the networks database from multiple threads must do its own locking.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `setnetent` - open the networks database and choose whether it stays open across individual lookups
- `getnetent` - read the next entry from the cursor that `endnetent` closes
- `getnetbyname` - single lookup by network name; implicitly opens the database but does not close it
- `getnetbyaddr` - single lookup by numeric address, same implicit-open, no-implicit-close rule

- `endhostent` - the hosts-database analogue; same pattern, independent cursor
- `endservent` - the services-database analogue for completeness of the `end*ent` family

Network info

endprotoent

Close the protocols database after iteration.

`endprotoent` is the terminator for the `getprotoent` / `setprotoent` / `endprotoent` iteration triad that walks the system protocols database (typically `/etc/protocols`). It releases whatever handle or state the system C library kept open for the iteration. You call it once the sequential walk is done; `getprotobyname` and `getprotobynumber` do not need it - they open and close the database on each call.

Takes no arguments, returns nothing useful. This is a thin wrapper over the system `endprotoent(3)` call.

Synopsis

```
endprotoent
```

What you get back

No meaningful return value. Calling it in list or scalar context yields the empty list / undef-equivalent; nothing you'd want to check. Its value is the side effect of closing the protocols database.

Examples

Walk every protocol, then close the database:

```
setprotoent(1);                          # 1 = keep the file open
 ↪between calls
while (my @p = getprotoent()) {
    my ($name, $aliases, $proto) = @p;
    print "$proto\t$name\n";
}
endprotoent();
```

Short-circuit mid-walk - `endprotoent` still closes cleanly:

```
setprotoent(0);
while (my @p = getprotoent()) {
    last if $p[0] eq 'tcp';
}
endprotoent();
```

Paired with a direct lookup, `endprotoent` is unnecessary - name and number lookups manage their own state:

```
my @tcp = getprotobyname('tcp');        # no endprotoent needed
```

Edge cases

- **Calling without a prior `setprotoent` / `getprotoent`**: harmless. The system call is idempotent; closing an already-closed database is a no-op on every supported platform.
- **No arguments accepted**: `endprotoent $x` is a syntax error - the built-in has arity zero.
- **Not reentrant**: the protocols database is a single global resource inside the C library. Two concurrent iterations in the same process step on each other's state. Use per-thread or per-subprocess walks if you need parallelism.
- **Platform quirks**: behaviour is whatever the host system's `endprotoent(3)` does. See `perlport` for the list of platforms where the network-database routines diverge or are absent entirely.
- **Stubbed on platforms without `/etc/protocols`**: on systems where the C library has no protocols database, the call is a no-op rather than an error.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `setprotoent` - open the protocols database and optionally keep it open across `getprotoent` calls; pair with `endprotoent` to bracket a walk
- `getprotoent` - read the next entry during an iteration; drives the loop that `endprotoent` terminates
- `getprotobyname` - single-shot lookup by protocol name; self-contained, no `endprotoent` needed
- `getprotobynumber` - single-shot lookup by protocol number; same independence from the iteration triad
- `endservent` - the analogous terminator for the services database; same idiom, different table
- `endhostent` - the analogous terminator for the hosts database

User and group info

endpwent

Close the passwd-database iterator opened by `getpwent` or `setpwent`.

`endpwent` is the companion teardown call for a passwd-file walk. After iterating the password database with repeated `getpwent` calls, `endpwent` releases whatever per-process state the C library allocated to hold the iteration cursor (an open file descriptor on `/etc/passwd`, a cached buffer for NSS, a connection to `nscd` / `sssd`, an LDAP

session, depending on the host's name-service configuration). It takes no arguments and wraps the system `endpwent(3)` call directly.

Synopsis

```
endpwent
```

What you get back

A single boolean value. The return is rarely checked - on every platform PetaPerl supports, `endpwent` cannot meaningfully fail, and real code treats it as a void call.

Global state it touches

`endpwent` mutates no Perl-visible variable. The state it clears is entirely inside the C library: the per-process passwd iterator, shared with `getpwent` and `setpwent` (and, on some systems, with `getpwnam` / `getpwuid` when those happen to reuse the same underlying stream). Calling `endpwent` in one thread affects every other thread in the same process - the iterator is per-process, not per-thread.

Examples

Standard iterate-then-close idiom:

```
setpwent;
while (my @ent = getpwent) {
    my ($name, undef, $uid) = @ent;
    print "$uid\t$name\n";
}
endpwent;
```

Ensure the iterator is closed even if the loop exits early:

```
setpwent;
eval {
    while (my @ent = getpwent) {
        my ($name, undef, $uid, $gid) = @ent;
        die "found it\n" if $uid == $target;
    }
};
endpwent;       # always run, regardless of how the loop ended
```

Pair with a `local` guard so nested code cannot leave the iterator half-walked:

```
sub each_user {
    my $cb = shift;
    setpwent;
    my $guard = Guard::guard { endpwent };   # or a manual eval-and-
↪rethrow
```

(continues on next page)

(continued from previous page)

```
    while (my @ent = getpwent) { $cb->(@ent) }
}
```

endpwent with no prior iteration is harmless:

```
endpwent;                                   # no-op on a fresh␣
↪process
```

Edge cases

- **No arguments, no parentheses needed.** endpwent is a named unary operator with zero arguments; write it bare. endpwent() is accepted but adds nothing.
- **Idempotent.** Calling endpwent twice in a row, or calling it before any getpwent / setpwent, is defined as a successful no-op on every supported platform.
- **Does not invalidate records you already fetched.** Arrays and hashes returned by earlier getpwent / getpwnam / getpwuid calls are plain Perl data and remain valid after endpwent - only the iterator cursor is closed.
- **Thread safety.** The iterator is process-wide. If two threads are walking the passwd database simultaneously, endpwent closes the shared cursor for both - neither thread will get the full record set. Use a lock around the whole setpwent / loop / endpwent block when iterating from threads.
- **Name-service backends.** On hosts using LDAP, NIS, sssd, or any other non-flat-file backend, endpwent may release a network handle. A long-running daemon that iterates the passwd database periodically should call endpwent at the end of each pass rather than leaking one handle per pass.
- **Not a forced cache flush.** endpwent closes the iterator; it does **not** flush the C library's getpwnam / getpwuid caches. A subsequent getpwnam may still return a cached record.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- setpwent - rewind or open the passwd-database iterator before a walk; pairs with endpwent as the open/close bracket
- getpwent - fetch the next record from the iterator that endpwent closes
- getpwnam - direct lookup by login name, independent of the iterator in most implementations
- getpwuid - direct lookup by numeric UID, same independence note as getpwnam
- endgrent - analogous teardown for the group-database iterator; the two are almost always called together

Network info

endservent

Close the services database after a walk with `getservent`.

`endservent` is the teardown half of the services-database iteration triplet. After `setservent` opens the database (typically `/etc/services` on Linux) and a sequence of `getservent` calls walks it entry by entry, `endservent` releases the underlying handle and any buffered state the C library is holding. It wraps the POSIX `endservent(3)` call one-to-one.

Synopsis

```
endservent;
endservent();
```

What you get back

Nothing useful. The function is called for its side effect; it does not return a meaningful value. Do not test the return value.

Global state it touches

- The system-wide services-database iterator opened by `setservent` or implicitly by the first `getservent` / `getservbyname` / `getservbyport` call. `endservent` closes that iterator.
- `$!` is **not** meaningfully set by this call; the POSIX contract for `endservent(3)` defines no error return.

Examples

Paired with `setservent` and `getservent` to enumerate every entry in `/etc/services`:

```
setservent 0;
while (my @e = getservent) {
    last unless @e;
    my ($name, $aliases, $port, $proto) = @e;
    print "$name/$proto = $port\n";
}
endservent;
```

Close a walk early. The database was opened implicitly by the first `getservent`; `endservent` is still the right way to release it:

```
while (my @e = getservent) {
    last if $e[0] eq 'http';
}
endservent;
```

Defensive cleanup around a block that may leave the iterator open. Harmless when no iteration is in progress:

```
eval {
    setservent 1;
    scan_services();
    1;
} or do {
    endservent;          # always release the handle
    die $@;
};
endservent;
```

Edge cases

- **Calling without a preceding `setservent` or `getservent`** is harmless. The underlying C call tolerates a no-op close.
- **Calling twice** is harmless for the same reason. Repeated `endservent` does not report an error.
- **Does not affect the name and port lookup functions** - `getservbyname` and `getservbyport` open and close their own short-lived handles and are unaffected by the iterator state that `endservent` manages.
- **No prototype, no arguments**. `endservent $x` is a compile-time error. Use `endservent;` or `endservent();`.
- **Not portable beyond Unix-like systems**. On platforms where the C library lacks `endservent(3)`, the call croaks with `endservent not implemented`.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `setservent` - open the services database and, with a non-zero `STAYOPEN`, keep the handle across subsequent lookups
- `getservent` - read the next entry from the open services database; pair every iteration loop with `endservent`
- `getservbyname` - one-shot lookup by service name; does not interact with the iterator `endservent` closes
- `getservbyport` - one-shot lookup by port number; same independence from the iterator
- `endprotoent` - the protocols-database analogue, same shape and same rules
- `endhostent` - the hosts-database analogue

I/O

eof

Test a filehandle for end-of-file.

`eof` reports whether the next read on a filehandle will return end-of-file. The three call forms look similar but mean different things: `eof FILEHANDLE` asks about a specific handle, bare `eof` asks about the handle most recently read from, and `eof()` with empty parentheses asks about the virtual `ARGV` stream fed by `<>`. The forms are not interchangeable - picking the wrong one is the classic bug with this built-in.

Synopsis

```
eof FILEHANDLE
eof()
eof
```

What you get back

1 if the next read on the handle would return end-of-file, or if the handle is not open. `0` otherwise. `eof` never blocks waiting for more input on a regular file, but on a pipe or terminal it may have to read and push back one character to decide - see *Edge cases*.

Global state it touches

- `ARGV` - the magical filehandle backing the `<>` operator. `eof()` examines and may advance it; `eof` inside a `while (<>)` loop queries it indirectly.
- `@ARGV` - the list of file names `<>` iterates over. `eof()` called before `<>` has been used triggers `@ARGV` inspection and, if empty, arranges for `STDIN` to be read.
- `$.` - the current input line number, attached to the handle `eof` queries. Not written by `eof` itself, but relevant in the idioms below where `close ARGV` resets it at file boundaries.

The three forms, spelled out

`eof FILEHANDLE` - asks about that one handle. `FILEHANDLE` may be a bareword, a scalar holding a handle, or an expression evaluating to one. Returns true for a closed or never-opened handle.

`eof` (no parentheses, no argument) - asks about the *last file read*. Inside a `while (<>) { ... }` loop this means "the current `ARGV` file" - the physical file currently being streamed, not the whole `<>` sequence. `eof` becomes true at every file boundary, once per input file.

`eof()` (empty parentheses) - asks about the `<>` stream as a whole. It becomes true only at the end of the *last* file in `@ARGV`. If `<>` has not been used yet, `eof()` will examine `@ARGV`, fall back to `STDIN` when `@ARGV` is empty, and may actually open the first file to answer the question.

Bare `eof` and `eof()` look almost identical and do completely different things. The parentheses matter.

Examples

Check a specific handle before the next read:

```
open my $fh, "<", "data.txt" or die $!;
while (not eof $fh) {
    my $line = <$fh>;
    # process $line
}
close $fh;
```

The classic `while (<>)` idiom, resetting `$.` at each file boundary. Bare `eof` (no parens) is what you want here - it fires once per input file:

```
while (<>) {
    next if /^\s*#/;
    print "$.\t$_";
} continue {
    close ARGV if eof;        # not eof() - resets $. per file
}
```

Insert a marker before the last line of the last file. `eof()` with parentheses catches the end of the *entire* <> stream, not each file:

```
while (<>) {
    if (eof()) {
        print "--------------\n";
    }
    print;
    last if eof();            # needed when reading from a terminal
}
```

Guard against a closed handle. `eof` on a closed or never-opened handle returns true without warning:

```
my $fh;                       # never opened
if (eof $fh) {
    warn "handle not ready";
}
```

Edge cases

- **`eof` before any read returns false, not true.** Bare `eof` with no prior read has no "last file read" to refer to and Perl treats it as false. Writing `if (eof) { ... }` at the top of a program does not mean "if there is no input" - use `eof()` for that.
- **`eof()` before <> is used has side effects.** It examines `@ARGV`, and if `@ARGV` is empty it arranges for <> to read from `STDIN`. A stray `eof()` at program start can therefore silently bind the program to `STDIN`.
- **`eof()` after <> has exhausted its input** assumes you are starting another `@ARGV` pass. If `@ARGV` is empty at that point, the next <> reads `STDIN`. This is usually not what you want; set `@ARGV` explicitly or test the bare-`eof` form instead.

- **`eof` on a terminal consumes a character.** The implementation reads one byte and ungetcs it to tell whether more input exists. On terminals the pushback may not survive after end-of-file is reached - do not call `eof FH` on a terminal after you have already hit EOF there, or the handle can lose its end-of-file condition.
- **Closed or never-opened handle** - `eof FILEHANDLE` returns true. There is no way to distinguish "closed" from "at EOF" using `eof` alone; check `fileno` or track the open state yourself if that matters.
- **`eof(ARGV)` equals bare `eof` inside `while (<>)`.** Both query the current ARGV file. Prefer bare `eof` for brevity and to make the parenthesised `eof()` form visually distinct.
- **You rarely need `eof` at all.** The input operators return `undef` at end-of-file, so `while (my $line = <$fh>) { ... }` and `while (<>) { ... }` handle termination without any `eof` test. Reach for `eof` only when you need to act *at* the boundary - last line, per-file reset, final-file marker.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `readline` - read a line from a handle; returns `undef` at end-of-file, which replaces most uses of `eof`
- `read` - block read of a fixed byte count; returns `0` at end-of-file
- `getc` - read a single character; returns `undef` at end-of-file
- `close` - close a handle; the common partner of `eof` in the `close ARGV if eof` per-file-reset idiom
- `ARGV` - the magical handle fed by `<>` that `eof` and `eof()` query

Control flow

eval

Run a piece of Perl code with any fatal error trapped instead of killing the program.

`eval` is Perl's exception-catching mechanism and its runtime code-loader in one. It comes in two real forms: **block eval** (`eval { ... }`), which catches exceptions thrown by already-compiled code, and **string eval** (`eval EXPR`), which compiles and runs a string of Perl source at runtime. Either way, a `die` that would otherwise terminate the program becomes an assignment to `$@` and a normal return.

Synopsis

```
eval BLOCK                  # catch exceptions in compiled code
eval EXPR                   # compile and run source at runtime
eval                        # short for: eval $_
```

What you get back

The value of the last expression evaluated inside the mini-program, or the value handed to an explicit `return`. The evaluation happens in the same context as the surrounding `eval` - void, scalar, or list - so `wantarray` inside an `eval` reports the caller's context, not "some `eval` context."

If anything inside raises an exception (`die`, a division by zero, a string-eval syntax error, any other trappable fatal), `eval` returns `undef` in scalar context and the empty list in list context, and the exception lands in `$@`:

```
my $n = eval { risky() };   # undef on failure, $@ holds the error
if ($@) { ... }
```

On success, `$@` is set to the empty string. Every completed `eval` touches `$@` - there is no "leave it alone" option. A control-flow operator that bypasses the normal exit (`last`, `goto` out of the block) can leave `$@` unchanged, which is a corner case rather than a feature to rely on.

Global state it touches

- `$@` - written on every normal exit of the `eval`: the thrown exception on failure, the empty string on success. Read to determine whether the `eval` caught something.
- `$^S` - reflects whether Perl is currently inside an `eval` (or `try`). `undef` during parsing, true while executing inside an `eval`/`try` block, false at the top level. Read primarily by `$SIG{__DIE__}` handlers that want to act differently for caught vs. uncaught exceptions.
- `$SIG{__DIE__}` - fires on **every** `die`, including those that an enclosing `eval` is about to catch. Use `local $SIG{__DIE__}` inside an `eval` to suppress a globally installed hook for the duration of the trap.
- `$SIG{__WARN__}` - unaffected by `eval`. Warnings emitted inside the block still go through the normal warning path; `eval` does not redirect them into `$@`.
- `$_` - the source of the no-argument form (`eval` with nothing after it is exactly `eval $_`).

Block eval vs. string eval

The two forms look similar but do very different things.

- **Block eval** (`eval { ... }`) is parsed and compiled together with the surrounding program. At runtime it is just a marker that says "catch any `die` thrown from inside this block." Because the code is already compiled, block eval is cheap and safe - the compiler already checked it for syntax errors. This is the form to reach for whenever the goal is exception handling.
- **String eval** (`eval EXPR`) evaluates `EXPR` in scalar context, then parses and executes the resulting string as if it were a block in the lexical scope of the surrounding code. The parse happens every time the `eval` is reached. A syntax error in the string becomes a runtime `$@`, not a compile-time failure of the enclosing program. This is the form for loading code decided at runtime - optional features,

user-supplied expressions, dynamically constructed subs - and it carries the costs and hazards that implies.

Both forms run in the lexical scope of the surrounding code: outer `my` variables are visible, and package variable assignments, subroutine definitions, and format definitions made inside the `eval` persist after it returns.

The $@ contract

After an `eval` completes normally, `$@` is always one of:

- The empty string - the block ran to completion with no exception.
- A string ending in a newline - a `die "message\n"` or a string exception the callee already terminated.
- A string with appended location (`... at FILE line N.\n`) - a `die "message"` without a trailing newline, or a runtime error Perl itself raised.
- A reference, typically a blessed exception object - `die` called with a reference.

Check `$@` immediately after the `eval` and nowhere else. Any method call, `ref` test, `isa` check, or regex match done during handling can itself invoke an `eval` internally (for instance, during autoloading or overload resolution) and overwrite `$@` before you finish inspecting it. Copy into a lexical first:

```
eval { work() };
if (my $err = $@) {         # safe snapshot
    if (ref $err && $err->isa('MyApp::Error')) { handle($err) }
    else                                       { die $err      }
}
```

Nested `eval` and `local $@`

The outer `eval` sees whatever `$@` holds when the outer block finishes - not what it held partway through. If inner code has its own `eval`, that inner `eval` sets `$@` (to the empty string on success), which can clobber an error the outer code wanted to preserve. The guard is `local $@` around the inner work:

```
eval {
    # ... some work that might fail ...
    {
        local $@;           # shield outer $@
        eval { optional_step() };
        # ignore whatever optional_step did to $@
    }
    die "still needed\n" if $condition;
};
warn "outer caught: $@" if $@;
```

Before Perl 5.14, `$@` was assigned before localized variables were restored, so code that wanted to filter errors had to stage the value through a temporary. On 5.14+ (which 5.44 inherits) the order is fixed and the temporary is unnecessary.

String eval security

String eval runs whatever is in the expression. If any part of that string came from untrusted input - a file, a network peer, a user form - the `eval` hands the attacker the full Perl interpreter. Rules of thumb:

- Never pass user input through `eval EXPR` unmodified. "It's only a number" is not a safeguard - `"1; system 'rm -rf /'"` is also only a string.
- If runtime-chosen code is genuinely needed, constrain the input against a strict allow-list *before* interpolation, and prefer data-driven dispatch (a hash of coderefs) over generating source.
- Floating-point values interpolated into source are fragile. Under `use locale` the decimal separator may not be a dot, and values like `"NaN"` or `"Infinity"` stringify to letters that the parser reads as barewords, not numbers.
- Source filters activated inside a string `eval` leak out into the surrounding file scope (unless `evalbytes` is used). This is a long-standing quirk, not a feature.

Block eval has none of these problems - the code was compiled with the rest of the program.

`BEGIN` inside string eval

`BEGIN { ... }` blocks embedded in a string `eval` execute immediately, before the rest of the `eval`'d code runs, matching their behaviour in a normal compilation unit. Set `${^MAX_NESTED_EVAL_BEGIN_BLOCKS}` to `0` to make any embedded `BEGIN` throw instead:

```
local ${^MAX_NESTED_EVAL_BEGIN_BLOCKS} = 0;
eval $untrusted;            # any BEGIN inside $untrusted dies
```

This does not make a string eval safe - it only removes one specific attack on compile-time side effects.

`try/catch`

Under `use feature 'try'` (stable since 5.40, available in 5.44), the `try/catch/finally` construct provides dedicated exception-handling syntax:

```
use feature 'try';

try {
    risky();
}
catch ($e) {
    warn "caught: $e";
}
finally {
    cleanup();
}
```

Differences from `eval { ... }; if ($@) { ... }`:

- The caught value is bound to a fresh lexical `$e` (or whatever name is given), not to `$@`. No need for the `my $err = $@` snapshot dance.
- `$@` is not set as a side effect - the `try` block has no global after-effect on it.
- `try` is not itself a loop, but control-flow operators behave analogously to `eval BLOCK`: `return` inside `try` returns from the enclosing sub, and `next`/`last`/`redo` target an outer loop.
- `finally` runs on every exit path and cannot itself use `return`/`next`/`last`/`redo`. It is marked experimental in 5.44 and emits a warning under `experimental::try`.

Reach for `try`/`catch` in new code. The `eval`-plus-`$@` idiom remains correct and is still the right tool whenever the surrounding code must run on older Perls or must not require the feature guard.

Examples

Trap a runtime error and keep going:

```
my $q = eval { $x / $y };
if ($@) { warn "division failed: $@"; $q = 0 }
```

Detect whether a feature is available without aborting at startup:

```
my $have_socket = eval { socket(my $s, PF_INET, SOCK_STREAM, 0); 1 }
 ↪;
die "no sockets on this build" unless $have_socket;
```

Load a module whose absence should be tolerated:

```
my $json = eval { require JSON::XS; JSON::XS->new } //
           do { require JSON::PP; JSON::PP->new };
```

Catch a blessed exception, re-throw anything else:

```
eval {
    parse($input);
};
if (my $err = $@) {
    if (ref($err) && $err->isa('MyApp::ParseError')) {
        report($err);
    }
    else {
        die $err;
    }
}
```

Nested `eval` with `local $@` so an inner optional step does not erase an outer diagnostic:

```
eval {
    do_main_work();
    { local $@; eval { maybe_cleanup() } }   # ignore cleanup errors
```

(continues on next page)

(continued from previous page)

```
    die "main still failed\n" if $something_else_wrong;
};
warn $@ if $@;
```

Suppress a globally installed `$SIG{__DIE__}` hook for the duration of a private trap:

```
eval {
    local $SIG{__DIE__};            # don't trigger hooks while we
↪probe
    $answer = $x / $y;
};
warn $@ if $@;
```

A `$SIG{__DIE__}` hook that logs only uncaught exceptions, using `$^S` to tell the two cases apart:

```
$SIG{__DIE__} = sub {
    return if $^S;              # inside eval/try - let it propagate
    log_fatal(@_);              # top level - record before exit
};
```

Runtime-built dispatch (string eval used deliberately, input is not user-supplied):

```
for my $op (qw(add sub mul div)) {
    eval "sub ${op}_of { \$_[0] "
       . { add=>'+', sub=>'-', mul=>'*', div=>'/' }->{$op}
       . " \$_[1] }";
    die $@ if $@;
}
```

Edge cases

- **No argument** - `eval;` is `eval $_`. Rarely what you want; write `eval { ... }` or `eval EXPR` explicitly.
- **Final semicolon** - may be omitted from the end of `EXPR` or `BLOCK`. `eval "1+2"` and `eval "1+2;"` are equivalent.
- **`eval BLOCK` is not a loop** - `next`, `last`, and `redo` cannot leave or restart the block. They look for an enclosing loop and will croak if there is none.
- **`return` inside `eval BLOCK`** returns from the *enclosing subroutine*, not from the `eval` - the `eval` block is not a function body. To produce a value from the block, put the value as the final expression.
- **Warnings are not trapped.** `warn` inside an `eval` still writes to `STDERR`. Route them into `$@` only if you set `$SIG{__WARN__}` to do so yourself.
- **Control flow out of `eval`** - a `goto`, `last`, or `next` that jumps out of the block skips the normal `eval` exit and therefore may leave `$@` untouched. Do not build logic that depends on this.

- **Quotes on EXPR - `eval $x` vs. `eval "$x"`** behave identically: both run the code contained in `$x`. The double quotes add visual noise, not semantics.
- **`eval '$x'` vs. `eval { $x }`** - both return the value of `$x`. The block form is compiled with the surrounding program; prefer it unless the code genuinely needs to be constructed at runtime.
- **`eval ''` inside the `DB` package** - does not see the usual surrounding lexical scope but the scope of the first non-`DB` caller. Only debugger authors hit this.
- **XS load failures** - some binary-interface problems during `require` of an XS module are fatal even inside `eval`, unless `$ENV{PERL_DL_NONLAZY}` is set. Version skew between the running interpreter and a pre-built `.so` is the common case. pperl does not load XS; the caveat applies when `eval`'d code delegates loading to a classic perl5.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `die` - the counterpart: throws the exception that `eval` catches. The trailing-newline rule on the message shapes what lands in `$@`.
- `do` - `do FILE` reads and evaluates a file the way string eval reads and evaluates a string; useful for configuration files written in Perl.
- `return` - inside `eval BLOCK` returns from the enclosing sub, not from the block; use a trailing expression to hand a value back from the `eval` itself.
- `require` - commonly wrapped in `eval` to probe for optional modules without aborting at compile time.
- `evalbytes` - string eval that treats its argument as a byte string and allows source filters to work normally; use when byte-level semantics matter.
- `$@` - where a caught exception lands, cleared to the empty string on successful `eval`. The error family page also covers the classic `$@`-clobbering hazard.
- `$^S` - tells `$SIG{__DIE__}` and `$SIG{__WARN__}` handlers whether an `eval` or `try` is currently active.
- `$SIG{__DIE__}` - pre-throw hook; fires even for exceptions an enclosing `eval` is about to catch, which is the usual reason to `local`-ize it for the duration of a trap.

Control flow

evalbytes

Compile and run a string of Perl source, forcing the source to be interpreted as **bytes** rather than characters.

`evalbytes` is the byte-oriented twin of string `eval`. It takes an EXPR (or `$_` when EXPR is omitted), treats the string as a raw byte stream, and parses it as Perl source. The parse is insulated from the caller's `use utf8` pragma: whatever encoding rules the string itself

declares win. Reach for it when the code you are about to compile was read from disk, the wire, or a build step, and you need the parser to see exactly the bytes you handed it.

Synopsis

```
evalbytes EXPR
evalbytes
CORE::evalbytes EXPR    # when the feature is not enabled
```

What you get back

In scalar context, the value of the last expression evaluated; in list context, a list. On a compile or run-time error, `undef` (or an empty list) and `$@` is set to the error message; `$@` is the empty string on success. Return shape matches string `eval` exactly - the only difference is how the source text itself is decoded.

```
my $result = evalbytes $source;
die $@ if $@;
```

How it differs from string `eval`

Two concrete behaviours distinguish `evalbytes` from `eval EXPR`:

- **The source is parsed as bytes.** Every octet in the string is a separate input character to the lexer, regardless of whether the string is internally marked as `utf8`. A multi-byte UTF-8 sequence in the source shows up to the parser as the individual bytes, not as the decoded code point.
- **Code points above 255 are a hard error.** If the string contains any character whose ordinal exceeds `0xFF`, it cannot be a byte stream and the compile fails with `Wide character in eval` stored in `$@`. Downgrade with `utf8::encode` first if the source currently holds decoded text.

`use utf8` and `no utf8` **inside** the evaluated code have their usual effect: the pragma switches on once the parser reaches it and affects the remainder of that compilation. It is the source's outer framing that is fixed to bytes, not what the source itself declares.

Enabling the name without `CORE::`

`evalbytes` is feature-gated. To call it as a bareword:

```
use feature 'evalbytes';
```

A `use v5.16` (or higher) declaration enables the `evalbytes` feature automatically, along with the rest of the `:5.16` feature bundle. Without either, the unqualified name is not recognised and you must write `CORE::evalbytes` explicitly - useful inside generated code or one-liners that do not carry a version declaration.

```
use v5.16;
my $n = evalbytes '1 + 2';        # 3
```

```
# no feature, no version bundle:
my $n = CORE::evalbytes '1 + 2';  # 3
```

Global state it touches

- `$@` - cleared before the compile, then set to the error message on failure or the empty string on success. Same contract as `eval`.
- `$_` - read as the source text when EXPR is omitted.

Source filters activated *inside* the evaluated code apply to the code itself, exactly as they would inside a string `eval`. The filter sees the same bytes the parser sees.

Examples

Evaluate a literal byte string - no surprises, no encoding choices to make:

```
use feature 'evalbytes';
my $n = evalbytes 'my $x = 40; $x + 2';   # 42
```

Evaluate source read from a file that is known to be UTF-8 encoded but whose declared pragma inside the source is what should govern parsing:

```
use feature 'evalbytes';
open my $fh, '<:raw', 'script.pl' or die $!;
my $src = do { local $/; <$fh> };
my @values = evalbytes $src;
die "compile failed: $@" if $@;
```

Reading with `:raw` keeps `$src` a byte string. `evalbytes` then feeds those bytes to the parser verbatim; a `use utf8;` line at the top of the script will do its own work once the parser reaches it.

Compare with string `eval` when the caller has `use utf8`:

```
use utf8;
use feature 'evalbytes';

my $src = "length('caf\xc3\xa9')";   # bytes for 'café' in UTF-8

print eval $src;         # 3  (characters: c, a, f, é)
print "\n";
print evalbytes $src;    # 5  (bytes:     c, a, f, 0xC3, 0xA9)
print "\n";
```

Under `use utf8` the caller's own literal `"length('caf\xc3\xa9')"` is a **character** string to `eval`, so `length` inside the evaluated code sees one é character. `evalbytes` forces the same source to be parsed as the byte sequence `c a f 0xC3 0xA9`, so `length` reports the byte count.

Catching a wide-character error:

```
use feature 'evalbytes';
my $src = "print \x{2603}";          # contains U+2603 SNOWMAN
evalbytes $src;
warn "rejected: $@" if $@;           # "Wide character in eval ..."
```

Omitting EXPR - source comes from `$_`:

```
use feature 'evalbytes';
for ('1+1', '2*3', '10-7') {
    my $r = evalbytes;
    print "$_ = $r\n";
}
```

Edge cases

- **Wide character in source**: any code point above `0xFF` aborts the compile and leaves `Wide character in eval` in `$@`. Run `utf8::encode` on the source (or read the file in `:raw`) before calling.
- **`use utf8` is local to the evaluated code.** It affects string literals and identifiers inside the evaluated text once the parser reaches the pragma line. It does **not** change how `evalbytes` frames the outer source - that framing is always "bytes".
- **Feature not enabled**: without `use feature 'evalbytes'` and without a `use v5.16+` bundle, the bareword is unrecognised. Write `CORE::evalbytes` to bypass the feature gate.
- **`evalbytes` with no argument and `$_` undef**: compiles an empty string, which succeeds and returns `undef` in scalar context, the empty list in list context, with `$@` empty.
- **Source filters** inside the evaluated code see the byte stream. A filter that expects decoded characters will misbehave; write filters for `evalbytes` in terms of bytes.
- **Lexicals and package state**: same rules as string `eval`. The evaluated code shares the caller's current package and sees the caller's in-scope lexicals; its own `my` variables vanish when the eval finishes.
- **`die` inside `evalbytes`**: caught the same way as for `eval`. The thrown value lands in `$@`; execution continues after the `evalbytes` call.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `eval` - the character-mode counterpart; parses EXPR according to the caller's Unicode framing, so `use utf8` in the caller changes how the source is interpreted
- `die` - throw an exception from within evaluated code; sets `$@` just like any other run-time error

- do - compile and run a file; takes its source from disk rather than a string, and is not affected by the caller's utf8 framing
- require - load and compile a file once; the standard tool when the source lives on disk and should only run on first reference
- utf8 - the pragma controlling character vs byte semantics of string literals; evalbytes exists precisely to pin the outer framing to bytes regardless of this pragma
- $@ - the error slot populated on evalbytes failure, cleared on success

Processes

exec

Abandon this program and run another in the same process.

exec replaces the current process image with another program. It does **not** return on success - execution continues in the new program, with the same process id, open filehandles subject to the close-on-exec flag, and whatever environment was in effect. It returns false (and sets $!) only when the target could not be launched *and* it was executed directly rather than through the shell. Reach for system when you want the child to run and control to come back.

Synopsis

```
exec LIST
exec PROGRAM LIST
exec { PROGRAM } LIST
```

What you get back

Nothing, on success - the process has been replaced and this Perl interpreter is gone. On failure exec returns false and leaves the reason in $!. Failure is only detectable for the direct-exec path; when the argument is handed to the shell, a missing program is reported by the shell itself and exec still counts as having "succeeded" in launching /bin/sh.

```
exec '/usr/bin/vi', $file
    or die "couldn't exec vi: $!";
```

Because exec normally does not return, Perl emits a warning under use warnings if it appears in void context followed by another statement that is not die, warn, or exit. Either handle the failure inline with or die, or wrap the call in a block to silence the warning:

```
exec('foo') or die "couldn't exec foo: $!";
{ exec('foo') }; die "couldn't exec foo: $!";
```

Shell or no shell - the single-string rule

`exec` picks between two launch mechanisms based on the shape of `LIST`:

- **Two or more arguments** → `exec` calls `execvp(3)` directly with the list. No shell is involved, no metacharacter interpretation, no word splitting.
- **Exactly one argument** → `exec` scans that single string for shell metacharacters. If any are found, the whole string is passed to `/bin/sh -c` for parsing. If none are found, the string is split on whitespace and handed to `execvp` directly.

The rule matters for both correctness and security. A one-argument string containing a space but no metacharacters is split into words and executed directly - which is often what you want. A one-argument string containing `|`, `>`, `$`, `;`, glob characters, or quotes goes through the shell, and the shell will happily expand wildcards, interpret variables, and split on `$IFS`.

```
exec '/bin/echo', 'Your arguments are:', @ARGV;  # direct execvp
exec "sort $outfile | uniq";                     # pipeline → shell
exec "ls /tmp";                                  # no metachars →␣
↪split + execvp
```

To **force** the no-shell path even with a single logical command, use the indirect-object form described next.

Indirect-object form - set argv[0], skip the shell

Placing a block or scalar before `LIST` (with no comma) tells `exec` two things at once: run *this* program, and pass *those* arguments as `argv`. The first element of `LIST` becomes `argv[0]` - the name the child sees for itself - which need not match the program path.

```
exec { '/bin/echo' } 'echo', 'hi';   # argv[0] is "echo"
my $shell = '/bin/csh';
exec $shell '-sh';                   # login-shell convention:␣
↪argv[0] = "-sh"
exec { '/bin/csh' } '-sh';           # same, more direct
```

The indirect-object form always treats `LIST` as a multi-valued list, even if it has only one element. That disables the metacharacter scan entirely:

```
my @args = ('echo surprise');
exec @args;              # one element → shell scan → /bin/sh -c
↪'echo surprise'
exec { $args[0] } @args; # always direct execvp, "echo surprise" is␣
↪argv[0]
```

In the second form there is no program on disk literally named `echo surprise`, so `execvp` fails, `exec` returns false, and `$!` holds the reason. That's the safe behaviour: the caller gets to decide what to do, rather than the shell running something unexpected.

Global state it touches

- `$!` - set on failure to the `errno` from the underlying `execvp(3)` call. Only meaningful when `exec` actually returns.
- `$?` - left unchanged by `exec` itself, but a failed shell-mediated attempt may leave the shell's exit status visible to a wrapping `system`.
- The process's open filehandles, environment (`%ENV`), current working directory, `umask`, signal dispositions, and resource limits all carry into the replacement program - `exec` replaces the program, not the process.

Examples

Run a program with an explicit argument list, direct `execvp`:

```
exec '/usr/bin/git', 'status', '--short'
    or die "exec git: $!";
```

Hand a command line to the shell deliberately, using metacharacters:

```
exec "find . -name '*.tmp' -print0 | xargs -0 rm"
    or die "exec pipeline: $!";
```

Set a distinctive `argv[0]` so the child's `ps` line identifies its role:

```
exec { '/usr/local/bin/worker' } 'worker[queue=email]', @flags
    or die "exec worker: $!";
```

Fork-and-exec - the classic pattern for launching a child without replacing yourself:

```
my $pid = fork // die "fork: $!";
if ($pid == 0) {
    exec '/usr/bin/gzip', '-9', $file
        or die "exec gzip: $!";
}
waitpid $pid, 0;
```

Guarantee the no-shell path when the argument list was assembled from data:

```
my @cmd = build_command();          # may contain one element or
↪many
exec { $cmd[0] } @cmd
    or die "exec $cmd[0]: $!";
```

Edge cases

- **Never returns on success.** Any code after a bare `exec` is reachable only on failure. Under `use warnings`, a following statement other than `die`, `warn`, or `exit` triggers a warning - Perl's hint that you probably meant `system`.
- **Single-argument trap.** `exec $cmd` where `$cmd` came from user input is a shell-injection vector if `$cmd` contains metacharacters. Use the indirect-object form or pass a pre-split list.

- **Output buffering.** Perl attempts to flush handles opened for output before handing control to the new program, but this is not guaranteed on every platform. If you have pending output on an unflushed handle, set `$|` on the selected handle - or call the `autoflush` method on each `IO::Handle` - before `exec`.
- **`END` blocks and `DESTROY`.** `exec` does **not** run `END` blocks, and it does **not** invoke `DESTROY` on your objects. The process image is replaced; Perl-level cleanup never happens. Anything that needs a clean shutdown (temp files, lockfile removal, buffered log flushes) must happen before `exec`.
- **Close-on-exec.** Filehandles with the `FD_CLOEXEC` flag set are closed by the kernel during `execvp`; others stay open. Control this with `fcntl` or the `$^F` system filehandle threshold.
- **Shell-mediated failure is invisible.** When `exec` routes through `/bin/sh -c`, a missing target program is reported by the shell and `exec` itself has already been replaced by `/bin/sh`. Your Perl code will not see a failure return.
- **One-arg list with no metacharacters** is *still* split on whitespace and passed to `execvp` - so `exec "ls /tmp"` is a direct exec of `/bin/ls` with one argument, not a shell call.
- **Empty `LIST`** is a runtime error; there is no program to run.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `system` - run a program and wait for it; what you almost certainly wanted if `exec` is at the top of the file
- `fork` - pair with `exec` in the child to launch a program without replacing the parent
- `wait` - collect a child produced by `fork` + `exec`
- `die` - terminate the current program with an error; the idiomatic partner of `exec ... or die`
- `$!` - error reason set when direct-exec fails
- `$|` - autoflush flag; set before `exec` so no buffered output is lost when the process image is replaced

Hashes

exists

Test whether a hash or array element, or a named subroutine, is present - without creating it and without caring what it holds.

`exists` asks a structural question: *is there an entry at this location?* It does not ask what the entry's value is. An entry whose value is `undef` still exists. An entry that has never been assigned does not - looking it up with a plain read would return `undef`, but `exists` distinguishes the two cases.

Synopsis

```
exists $hash{KEY}
exists $array[INDEX]
exists &subroutine
exists $ref->{A}{B}          # autovivifies $ref->{A}!
```

What you get back

A boolean: 1 (a true scalar) if the element or subroutine is present, the empty string "" (a false scalar) if not. The result is always a plain scalar - there is no list-context variant.

```
my %h = (a => undef);
exists $h{a};                # 1   (key exists)
defined $h{a};               # ""  (value is undef)
exists $h{b};                # ""  (key does not exist)
```

`exists` is not `defined`

The two answer different questions and they disagree exactly when an existing entry holds `undef`:

State	`exists`	`defined`	Plain read
Never assigned	false	false	`undef`
Assigned `undef`	**true**	false	`undef`
Assigned any other value	true	true	that value

Use `exists` when you need to know whether a key has been placed in the structure - e.g. an option hash where a bare key means "the caller opted in, value doesn't matter." Use `defined` when you need to know whether a value is usable.

Examples

Hash-key test. Classic option-hash idiom - presence, not truth:

```
sub configure {
    my %opt = @_;
    my $verbose = exists $opt{verbose};     # caller passed verbose
 ↪=> anything
    my $color   = $opt{color} // "auto";    # caller-supplied or
 ↪default
    ...
}
```

Distinguishing "missing" from "present but undef":

```
my %cache = (user_42 => undef);             # cached: "we looked,
 ↪nothing there"
```

(continues on next page)

(continued from previous page)

```
if (exists $cache{user_42}) {
    # hit - even though the value is undef, we already did the␣
 ↪lookup
} else {
    # miss - have to query the database
}
```

Sparse-array test. Deleting an array element leaves a hole that `exists` reports as false:

```
my @a = (10, 20, 30);
delete $a[1];
exists $a[0];                # 1
exists $a[1];                # ""   (hole)
exists $a[2];                # 1
exists $a[99];               # ""   (past the end)
```

Subroutine existence. True once the sub has been declared, even if the body is a forward declaration with no definition yet:

```
sub later;                   # forward declaration
exists &later;               # 1
defined &later;              # ""   (no body yet)

sub later { 42 }
defined &later;              # 1
```

Nested structure - the **safe** way. Check each level before descending, because the arrow-chain form has a side effect (see Edge cases):

```
if (exists $tree{users}
    && exists $tree{users}{$id}
    && exists $tree{users}{$id}{email}) {
    send_mail($tree{users}{$id}{email});
}
```

Edge cases

- **Autovivification of intermediate levels.** This is the single most important pitfall. `exists $ref->{A}{B}{C}` tests the innermost key, but evaluating the expression autovivifies every **intermediate** hash - `$ref->{A}` and `$ref->{A}{B}` spring into existence as empty hashes even when the `exists` test returns false:

  ```
  my %h;
  exists $h{a}{b}{c};          # returns "" (false)
  exists $h{a};                # now returns 1 - $h{a} was␣
   ↪created!
  exists $h{a}{b};             # likewise - now an empty hash
  ```

  The innermost element is the only one that does not autovivify. The same happens through references: `exists $ref->{A}` on an undef `$ref` vivifies `$ref` into

a hash reference. If you need to probe a deep structure without mutating it, walk it one level at a time with `exists` at each step, or use a dedicated helper like `Data::Diver` semantics (a chained `eval` / `no autovivification` pragma scope).

- **`exists` on a key with value `undef` is true.** This is the defining difference from `defined`. `delete` removes both the key and the value; assigning `undef` keeps the key.
- **Array bounds.** `exists $a[$i]` is false when `$i` is past the end of the array **or** when the slot has been `delete`d mid-array. Negative indices count from the end: `exists $a[-1]` tests the last element if the array is non-empty.
- **Array `exists` is discouraged upstream.** The notion of deleting an array element is not conceptually clean - `delete $a[1]` leaves a hole rather than shifting. Most code that reaches for `exists $a[$i]` is better served by a bounds check (`$i <= $#a`) or by using a hash.
- **`exists &sub()` is a syntax error.** The argument must name a subroutine, not call one. `exists &sub` asks "is `sub` declared?"; `exists &sub()` would mean "does the return value of calling `sub` exist?", which is meaningless.
- **`AUTOLOAD` does not count.** `exists &sub` is false for a sub that would spring into existence via `AUTOLOAD` on first call. A failing `exists` therefore does not guarantee a call would fail.
- **Tied variables.** For tied hashes and arrays, `exists` calls the `EXISTS` method on the tied object. The tie implementation decides what "existing" means - a `DBM` tie checks the on-disk record, a lazy tie might materialise the entry. See `perltie`.
- **The argument must be an lvalue expression whose final operation is a key or index lookup, or a subroutine name.** `exists func()`, `exists $scalar`, `exists @array`, `exists %hash` are all errors. Only element access and `&sub` are accepted.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `defined` - asks whether the value is usable, not whether the key is present; differs from `exists` only when the stored value is `undef`
- `delete` - removes an entry so that `exists` on it becomes false; the counterpart to `exists`
- `each` - iterates key/value pairs of a hash; only visits keys for which `exists` is true
- `keys` - returns the list of existing keys; `exists $h{$k}` is equivalent to asking whether `$k` is among `keys %h`, but O(1) instead of O(n)
- `ref` - classifies what a reference points to; useful before `exists` on `$ref->{...}` if you are not sure `$ref` is a hash reference
- `scalar` - force scalar context; note that `exists` is already scalar-only and does not need it

Control flow

exit

Terminate the program with a status value.

`exit` evaluates `EXPR`, runs end-of-program cleanup (`END` blocks and object destructors), and then exits the process with that value as its status code. With no argument it exits `0` - success.

Synopsis

```
exit EXPR
exit
```

What you get back

Nothing. `exit` does not return to its caller. It is a statement in the control-flow sense even though the grammar lets you write it inside an expression. Code after `exit` in the same block is unreachable:

```
exit 1;
print "never\n";            # dead code
```

Status values and the 8-bit truncation

The operating system stores an exit status in **8 bits**. Whatever integer you pass to `exit` is taken modulo 256 by the kernel before the parent process sees it. `exit 256` looks like `exit 0` to the shell. `exit -1` becomes `255`. Pass only values in `0..255` and you avoid the surprise.

The only universally portable values are `0` for success and `1` for a generic error. Other numbers have conventions - the `sysexits.h` range `64..78` on Unix, specific codes expected by `sendmail` filters, and so on - but those are agreements between your program and a specific caller, not guarantees of the language. Exiting `69` (`EX_UNAVAILABLE`) tells a `sendmail` incoming-mail filter to defer the message; the same `69` from a script run by `cron` means nothing in particular.

```
exit 0;                     # success
exit 1;                     # generic failure
exit 2;                     # conventional "misuse" (grep, diff)
exit 77;                    # conventional "skip" (Automake test)
```

Cleanup: `END` blocks and destructors

`exit` is **not** an immediate syscall. Before the process goes away, Perl:

1. Runs every `END` block in **reverse order of compilation** (LIFO).
2. Calls `DESTROY` on any object whose refcount drops to zero as global destruction tears down pads, the symbol table, and package globals.

Both kinds of cleanup can observe **and change** `$?`. The value you passed to `exit` is placed in `$?` on entry to the `END`/`DESTROY` phase; whatever `$?` holds when the last cleanup returns is the status the process exits with.

```
END { $? = 0 if $? == 2; }  # rewrite "misuse" to success

exit 2;                     # process actually exits 0
```

This is occasionally useful and regularly astonishing. If a module you did not write installs an `END` block that touches `$?`, your carefully chosen status code can arrive at the parent shell altered.

When you need a hard, unconditional exit that bypasses `END` and `DESTROY` entirely, call `POSIX::_exit`:

```
use POSIX ();
POSIX::_exit(3);            # no END, no DESTROY, status = 3
```

Reach for `_exit` in child processes after `fork` where the parent owns the cleanup, and in signal handlers where running arbitrary Perl code is unsafe.

`exit` is not `die`

`exit` is the wrong tool for reporting an error from a library or a subroutine. A caller higher up the stack may want to catch the failure, log it, and continue - an `eval` block or a `Try::Tiny` wrapper cannot recover from `exit`. The process is gone.

```
sub load_config {
    my ($path) = @_;
    open my $fh, "<", $path
        or die "open $path: $!";   # correct - trappable
    ...
}

# NOT:
# open my $fh, "<", $path or exit 1;
```

Use `die` for error conditions. Reserve `exit` for the top level of the program where you have decided, as the application, that the run is over.

`exit` inside `eval`

`exit` is **not** caught by `eval`. Unlike `die`, which unwinds to the nearest `eval` and sets `$@`, `exit` walks straight past every `eval` frame on the stack, runs `END`/`DESTROY`, and terminates.

```
eval { exit 5 };            # process exits 5; the eval block
print "after eval\n";       # never runs
```

If you are writing a plugin host or a test harness that must survive misbehaved user code calling `exit`, there is no language-level catch. The conventional workaround is to fork, let the child call `exit`, and inspect the child's status in the parent:

```
my $pid = fork // die "fork: $!";
if ($pid == 0) {
    run_user_code();        # may call exit
    exit 0;
}
waitpid $pid, 0;
my $status = $? >> 8;
```

Examples

Early return from the program after a prompt:

```
my $ans = <STDIN>;
exit 0 if $ans =~ /^[Xx]/;
```

Exit with a status derived from a child process - extract the high byte of `$?` so the shell sees the same code the child produced:

```
system @cmd;
exit $? >> 8 if $?;
```

Rewrite a status in an `END` block - whatever `$?` holds at the end of cleanup is the exit status delivered to the parent:

```
END {
    $? = 0 if $? == 2 && $ENV{IGNORE_MISUSE};
}
```

Bypass cleanup in a forked child so the parent's destructors do not run twice:

```
use POSIX ();
my $pid = fork // die "fork: $!";
if ($pid == 0) {
    exec @cmd;
    POSIX::_exit(127);      # exec failed; do NOT run parent's END
}
```

Edge cases

- **`exit` with no argument** is `exit 0`. Many scripts rely on falling off the end of the file to get an implicit `0`; writing `exit;` makes the intent explicit.
- **Non-integer argument**: `EXPR` is evaluated in scalar context and converted to an integer. `exit "3x"` exits 3; `exit "hello"` exits `0` with a warning under `use warnings`.
- **Negative values** are taken modulo 256 by the kernel. `exit -1` appears to the shell as 255. Pass only `0..255` to avoid the coercion.
- **Values above 255** lose their high bits. `exit 256` looks like success; `exit 300` looks like `exit 44`. If you need to convey a larger value, write it to a file or to `STDERR` and exit with a small status code.

- **`END` blocks can cancel your status but not the exit itself**. An `END` block cannot stop the process from terminating - it can only change `$?`, and therefore the status seen by the parent.
- **Global destruction order is not the reverse of construction**. During the final sweep Perl walks package globals in an implementation-defined order. Do not rely on one `DESTROY` running before another; if ordering matters, tear the objects down explicitly before calling `exit`.
- **Inside a `BEGIN` block**, `exit` terminates the compilation phase and exits the process. `END` blocks already registered run; later `BEGIN`/`END`/code in the file does not.
- **Threads**: in a program with multiple Perl interpreter threads, `exit` terminates the whole process, not just the calling thread. Use `threads->exit` for per-thread exit.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `die` - raise a trappable exception; the right choice for reporting errors from a subroutine
- `warn` - emit a diagnostic without terminating; pairs with `die` the way `print` pairs with `exit`
- `END` blocks - compile-time-registered cleanup code run in LIFO order on any exit, including `exit EXPR`
- `DESTROY` methods - per-object cleanup called during global destruction after `END` blocks have run
- `$?` - child-process / final exit status; visible to and mutable by `END` blocks and destructors
- `POSIX::_exit` - immediate termination that bypasses `END` and `DESTROY`; use in forked children and signal handlers

Numeric functions

exp

Raise *e* to a power.

exp returns *e* (the base of natural logarithms, roughly `2.718281828459045`) raised to the power of `EXPR`. It is the inverse of `log`: for any finite positive `$x`, `exp(log($x))` reproduces `$x` to within floating-point rounding. With no argument, `exp` operates on `$_`.

Synopsis

```
exp EXPR
exp
```

What you get back

A double-precision floating-point number. The result is always non-negative: `exp(0) == 1`, exp of any positive value is greater than 1, and exp of any negative value lies in the open interval `(0, 1)`.

On every Linux target pperl supports, the underlying type is IEEE 754 binary64, so the range of finite results is roughly `5e-324` up to `1.7976931348623157e+308`. Arguments above about `709.78` overflow to `Inf`; arguments below about `-745` underflow to `0.0`.

Examples

Get *e* itself, and verify the round-trip against `log`:

```
my $e = exp(1);                         # 2.718281828459045
printf "%.15f\n", exp(log(42));         # 42.000000000000000
```

No-argument form reads `$_`:

```
for (1, 2, 3) {
    printf "exp(%d) = %.6f\n", $_, exp;
}
# exp(1) = 2.718282
# exp(2) = 7.389056
# exp(3) = 20.085537
```

Compose with other numeric built-ins. The hyperbolic sine, for example, has no dedicated operator but falls out of exp:

```
sub sinh { (exp($_[0]) - exp(-$_[0])) / 2 }
printf "%.6f\n", sinh(1);               # 1.175201
```

Overflow at the top of the IEEE 754 range:

```
print exp(709), "\n";                   # 8.21840746949e+307
print exp(710), "\n";                   # Inf
```

Underflow at the bottom:

```
print exp(-745), "\n";                  # 5e-324  (smallest subnormal)
print exp(-746), "\n";                  # 0
```

Edge cases

- **Very large positive argument**: Any argument greater than roughly `709.78` produces `Inf`. `Inf` propagates through subsequent arithmetic (`Inf + 1 == Inf`, `Inf - Inf` is `NaN`) and stringifies as `"Inf"`.
- **Very negative argument**: Arguments below roughly `-745` underflow to `0.0`. Values in between render as denormal floats with progressively reduced precision.
- **`NaN` input**: `exp(NaN)` is `NaN`. `NaN` compares unequal to everything including itself - test with `Scalar::Util::looks_like_number` or `$x != $x` rather than `==`.

- **`exp(0)`** is exactly 1, not a rounded approximation.
- **Non-numeric argument**: Perl coerces the argument through its standard string-to-number rules before calling the C `exp(3)` routine. `exp("")` and `exp("abc")` are therefore `exp(0) == 1`, with an `Argument "…" isn't numeric` warning under `use warnings`. A leading-numeric string like `"3.5xyz"` becomes `3.5`, again with a warning.
- **`undef` argument**: Coerced to `0`, yielding 1, with an `uninitialized` warning under `use warnings`.
- **List context**: `exp` is a unary numeric function. It ignores list context and always returns a single scalar. `my @r = exp(2)` puts one element into `@r`.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `log` - natural logarithm, the inverse of `exp`; `exp(log($x)) == $x` for positive `$x`
- `sqrt` - square root; `sqrt($x) == exp(log($x)/2)` for positive `$x`
- `sin` - sine, the other transcendental unary built-in
- `cos` - cosine; with `sin` and `exp` covers the standard transcendentals available without a module
- `abs` - absolute value, useful when feeding a signed quantity into `log` after `exp`-based manipulation
- `$_` - default argument when `exp` is called without an explicit operand

SCALARs and strings

fc

Return the Unicode casefolded form of a string for case-insensitive comparison.

`fc` is the string operation that powers case-insensitive string equality. It produces a form of the string where case distinctions have been erased, so two strings are considered case-insensitively equal when - and only when - their `fc` results are byte-for-byte identical. It is also the function behind the `\F` escape in double-quoted strings.

Synopsis

```
use feature 'fc';           # or: use v5.16;

fc EXPR
fc                          # operates on $_
"\F...\E"                   # same casefold, inside a double-quoted
 ↪string
```

`fc` is gated behind the `fc` feature. Enable it explicitly with `use feature 'fc'`, pull it in via a version bundle (`use v5.16` or newer, `use feature ':5.16'`), or call it fully qualified as `CORE::fc` without any pragma.

What you get back

A string containing the casefolded form of EXPR. The result is a fresh string; the argument is never modified. Length may change - casefolding `"\x{1E9E}"` (LATIN CAPITAL LETTER SHARP S) normally expands to `"ss"`, two characters from one.

Treat the return value as opaque: it is a key suitable for equality comparison, not for display. `fc("Hello")` is `"hello"` for ASCII, but in the general case the output is not something you'd show to a user.

Why not `lc` or `uc`?

Lowercasing and uppercasing are **not** reliable for case-insensitive comparison. Both of these are wrong:

```
lc($a) eq lc($b)            # Wrong
uc($a) eq uc($b)            # Also wrong
```

They fail on characters whose lower/upper mapping is not symmetric - most famously the German sharp S. `lc("\x{1E9E}")` is `"\x{1E9E}"` (no lowercase form), but `uc("ß")` is `"SS"`. Casefolding sidesteps this by mapping both sides into a dedicated equality form:

```
fc($a) eq fc($b)            # Right
```

The regex-based equivalent that was correct before `fc` existed:

```
$a =~ /^\Q$b\E\z/i
```

`fc` is the direct, non-regex way to get the same answer.

Global state it touches

- `$_` - used as the argument when EXPR is omitted.
- `use locale` - inside a `use locale` scope, casefolding of characters crossing the 255/256 boundary is disabled (see *Edge cases* below for the U+1E9E rule).
- `use feature 'unicode_strings'` - affects `fc` the same way it affects `lc`: forces full Unicode semantics on byte strings that would otherwise be treated under legacy 8-bit rules.

Examples

Case-insensitive equality, the canonical use:

```
use feature 'fc';
fc("Hello") eq fc("HELLO");             # true
fc("café")  eq fc("CAFÉ");              # true
```

The German sharp S - the textbook case where `lc` and `uc` both fail but `fc` succeeds:

```
use feature 'fc';
my $a = "straße";
my $b = "STRASSE";
fc($a) eq fc($b);                       # true
lc($a) eq lc($b);                       # false - "straße" ne
 ↪"strasse"
```

Using `fc` as a hash key for case-insensitive lookup:

```
use feature 'fc';
my %seen;
for my $word (@words) {
    $seen{ fc $word }++;                # groups "Foo", "FOO", "foo"
}
```

No argument - operates on `$_`:

```
use feature 'fc';
for (@lines) {
    next unless fc eq "quit";           # matches "QUIT", "Quit", ..
 ↪.
    last;
}
```

Inside a double-quoted string via `\F` (same operation, inline):

```
use feature 'fc';
my $name = "Alice";
my $key  = "\F$name\E";                 # "alice" - same as fc(
 ↪$name)
```

Calling without the feature pragma, fully qualified:

```
my $folded = CORE::fc($input);          # works in any scope
```

Edge cases

- **No argument**: `fc` with no argument folds `$_`.
- **`undef` argument**: stringifies to the empty string, which folds to the empty string. Emits an `uninitialized` warning under `use warnings`.
- **Feature not enabled**: `fc EXPR` without `use feature 'fc'` (or a `use v5.16+` bundle) is a compile-time error - the parser does not recognise `fc` as a keyword. `CORE::fc(EXPR)` always works.
- **Length can change**: full casefolds may expand one character to several. `fc("\x{1E9E}")` is `"ss"`. Never rely on `length(fc $s) == length($s)`.
- **Not a round-trip**: `fc` is one-way. There is no "uncasefold" operation; the original case is gone.

- **U+1E9E under `use locale`**: `fc` of *LATIN CAPITAL LETTER SHARP S* (U+1E9E) normally folds to `"ss"`. Under `use locale` that mapping is suppressed because it crosses the 255/256 codepoint boundary, which locale rules do not handle cleanly. Instead `fc` returns `"\x{17F}\x{17F}"` (two *LATIN SMALL LETTER LONG S*). Since each *long s* itself folds to `"s"`, two of them compare equal to a single U+1E9E folded outside the locale scope - so equality semantics are preserved even though the byte form differs.
- **Turkic and "simple" folds are not provided**: Perl implements only the *full, non-Turkic* form of casefolding. For the simple form or the Turkic variant, use `Unicode::UCD::casefold` or the CPAN module `Unicode::Casing`.
- **Not the same as NFKC or NFC**: `fc` erases case, not compatibility differences. `fc("ffi")` (*LATIN SMALL LIGATURE FFI*) is `"ffi"` after fold, not `"ffi"`. Combine with `Unicode::Normalize` if you need both.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `lc` - lowercases a string; use for display, not for case-insensitive comparison
- `uc` - uppercases a string; same caveat as `lc`
- `lcfirst` - lowercases only the first character
- `ucfirst` - uppercases only the first character
- `index` - substring search; pair with `fc` on both operands for a case-insensitive variant
- `$_` - the default argument when `EXPR` is omitted

Filehandles, files, directories

fcntl

Perform a `fcntl(2)` file-control operation on a filehandle.

`fcntl` is the direct Perl binding for the POSIX `fcntl(2)` system call. It reads or changes per-descriptor state - non-blocking mode, close-on-exec, advisory record locks, descriptor flags - by invoking `FUNCTION` against `FILEHANDLE`, with `SCALAR` acting as either the input argument, the output buffer, or both, depending on which `FUNCTION` you pass.

The `FUNCTION` codes are not Perl-level literals; they are C preprocessor macros exposed by the `Fcntl` module. Load it first, or `FUNCTION` will be an undefined bareword:

```
use Fcntl;
```

Synopsis

```
use Fcntl;
fcntl FILEHANDLE, FUNCTION, SCALAR
my $flags = fcntl($fh, F_GETFL, 0);
fcntl($fh, F_SETFL, $flags | O_NONBLOCK);
```

What you get back

On success, the integer the kernel returned. When that integer is `0`, Perl substitutes the dual-valued string `"0 but true"` - true in boolean context, numeric `0` in numeric context, and exempt from the `Argument "..." isn't numeric` warning. This lets you test the result with `or die` even for operations whose success value is literally zero:

```
fcntl($fh, F_SETFD, FD_CLOEXEC)
    or die "F_SETFD: $!";
```

On failure, `fcntl` returns `undef` and sets `$!` to the relevant `errno`. You do **not** need to wrap the return in `defined` - the `"0 but true"` convention makes a plain truth test sufficient.

For `FUNCTION` codes that write back through `SCALAR` (such as `F_GETLK`), the updated bytes land in the scalar you passed; the return value still follows the rule above.

How SCALAR is used

`SCALAR` plays three different roles depending on `FUNCTION`:

- **Input integer.** For flag-setting calls like `F_SETFD` or `F_SETFL`, `SCALAR` is the integer bitmask to install.
- **Ignored placeholder.** For flag-getting calls like `F_GETFD` or `F_GETFL`, `SCALAR` is unused by the kernel; pass `0` by convention.
- **In/out struct buffer.** For record-locking calls like `F_GETLK`, `F_SETLK`, and `F_SETLKW`, `SCALAR` must hold a packed `struct flock` - build it with `pack` before the call, and for `F_GETLK` decode the returned bytes with `unpack` afterward.

When `SCALAR` is used as an output buffer, pre-size it to at least the length of the struct the kernel will write; an undersized scalar is grown automatically, but pre-sizing makes the intent obvious.

Global state it touches

- `$!` - set on failure to the `errno` from `fcntl(2)`.
- Descriptor-level flags on `FILEHANDLE` (e.g. `O_NONBLOCK`, `FD_CLOEXEC`) persist for the lifetime of the descriptor, survive dup and `fork` according to POSIX rules, and are **not** Perl-visible through any other variable. The only way to read them back is another `fcntl` call with `F_GETFL` / `F_GETFD`.

Examples

Make a socket non-blocking - the canonical use:

```
use Fcntl qw(F_GETFL F_SETFL O_NONBLOCK);

my $flags = fcntl($remote, F_GETFL, 0)
    or die "F_GETFL: $!";
fcntl($remote, F_SETFL, $flags | O_NONBLOCK)
    or die "F_SETFL: $!";
```

Set close-on-exec so the descriptor is not inherited by child processes across `exec`:

```
use Fcntl qw(F_SETFD FD_CLOEXEC);

fcntl($fh, F_SETFD, FD_CLOEXEC)
    or die "F_SETFD: $!";
```

Read the current descriptor flags and test a bit:

```
use Fcntl qw(F_GETFD FD_CLOEXEC);

my $fd_flags = fcntl($fh, F_GETFD, 0);
defined $fd_flags or die "F_GETFD: $!";
if ($fd_flags & FD_CLOEXEC) {
    # descriptor will close on exec
}
```

Place an advisory write lock on the whole file. The `struct flock` layout is platform-specific; on Linux `s s l l i` covers `l_type l_whence l_start l_len l_pid`:

```
use Fcntl qw(F_SETLK F_WRLCK SEEK_SET);

my $lock = pack('s s l l i', F_WRLCK, SEEK_SET, 0, 0, 0);
fcntl($fh, F_SETLK, $lock)
    or die "lock refused: $!";
```

Toggling off a single flag - mask out `O_NONBLOCK` to return a handle to blocking mode:

```
use Fcntl qw(F_GETFL F_SETFL O_NONBLOCK);

my $flags = fcntl($fh, F_GETFL, 0);
fcntl($fh, F_SETFL, $flags & ~O_NONBLOCK)
    or die "F_SETFL: $!";
```

Edge cases

- **Forgot `use Fcntl`**: `F_GETFL` and friends parse as barewords and evaluate to a warning-level nuisance under `use strict` / `use warnings`, or to the string `"F_GETFL"` otherwise - either way the call fails with `EINVAL`. Always load `Fcntl` first.

- **"0 but true" return**: do not compare the return with `== 0` as a failure test. Use a plain boolean (`or die`) or `defined` - the former is the idiom.
- **F_SETLK vs F_SETLKW**: `F_SETLK` returns `undef` and sets `$!` to `EAGAIN` / `EACCES` when another process holds a conflicting lock. `F_SETLKW` blocks until the lock is granted. Pick deliberately; mixing them up is the usual source of hangs.
- **F_GETLK output**: on success the `struct flock` bytes in `SCALAR` are overwritten with information about the conflicting lock (or `l_type == F_UNLCK` if the region is free). `unpack` back with the same template you packed with.
- **Not every system implements every FUNCTION**: `fcntl` raises an exception on a machine that does not implement `fcntl(2)` at all. On systems that do, unsupported `FUNCTION` codes fail with `EINVAL` via `$!`. The `Fcntl` module only exports constants the current platform actually defines.
- **`fcntl` vs `ioctl`**: both share the argument-processing and return-value convention, but they target different kernel interfaces. Use `fcntl` for descriptor flags and advisory locks; use `ioctl` for device-specific control codes.
- **`fcntl` vs `flock`**: `flock` uses whole-file BSD-style `flock(2)` locks, which on Linux interoperate with `fcntl` POSIX locks only in limited ways. If you need byte-range locking or POSIX semantics (lock released on *any* `close` of the file by the owning process), use `fcntl` with `F_SETLK`; otherwise `flock` is simpler.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `Fcntl` - the module that exports the `F_*` / `O_*` / `FD_*` / `SEEK_*` constants `fcntl` expects; always `use` it first
- `ioctl` - same calling and return-value convention as `fcntl`, but for device control codes rather than file-control flags
- `flock` - simpler whole-file advisory locks; reach for it when you don't need byte-range or POSIX semantics
- `open` - opens the filehandle in the first place; some flags `fcntl` adjusts (e.g. `O_NONBLOCK`, `O_APPEND`) are also settable at open time
- `sysopen` - low-level open that takes the same `O_*` flags from `Fcntl`; use it when you want the flags applied atomically rather than via a follow-up `fcntl(F_SETFL)`
- `pack` / `unpack` - required for building and decoding the `struct flock` buffer used by `F_GETLK` / `F_SETLK`

Scoping · Classes and OO

field

Declare a per-instance variable inside a `class` block.

`field` introduces a new variable whose storage is private to each object of the enclosing class. Every method and every `ADJUST` block of that class sees the field as if it were a lexical in scope at that point - but the value each sees is the value belonging to the *current* instance. Fields are the class feature's replacement for hash-slot bookkeeping on a blessed reference.

The feature is still marked **experimental** in Perl 5.44 (see *Differences from upstream*); enable it with `use feature 'class';` and silence the warning category with `no warnings 'experimental::class';` if you don't want the noise.

Synopsis

```
field $scalar;
field $scalar = EXPR;
field @array  : ATTRIBUTES;
field %hash   : ATTRIBUTES = EXPR;
```

What you get back

`field` is a declaration, not an expression. It produces no value and cannot appear on the right-hand side of anything. Each `field` statement allocates one slot per instance in the class's layout; the slot is the field variable itself, bound to a per-object storage location on entry to any method or `ADJUST` block.

Scalars, arrays, and hashes are all allowed:

```
class Thing {
    field $scalar = 42;
    field @array  = qw(this is just an array);
    field %hash   = (species => 'Martian', planet => 'Mars');
}
```

Where field declarations are legal

Only directly inside a `class` block. `field` is not a general-purpose declarator - it is not legal inside a `method` body, an `ADJUST` block, a nested `sub`, or plain `package` code. The compiler rejects it anywhere else:

```
class C {
    field $ok;                  # OK
    method m {
        field $nope;            # compile error
    }
}
```

A field must be declared before it can be referred to by any method or initializer in the same class body.

Field initializers

If a `= EXPR` is present, the expression runs once per constructor call, after all fields declared earlier in the class body have been initialized. This matters when a later field's default depends on an earlier one:

```
class WithACounter {
    my $next_count = 1;
    field $count = $next_count++;
}
```

Inside an initializer, `$self` does **not** exist - the object is still being built. Use the `__CLASS__` token when the initializer needs the class name (for example, to call a class method that subclasses can override):

```
class WithCustomField {
    use constant DEFAULT_X => 10;
    field $x = __CLASS__->DEFAULT_X;
}

class DifferentCustomField :isa(WithCustomField) {
    sub DEFAULT_X { rand > 0.5 ? 20 : 30 }
}
```

An instance of `DifferentCustomField` will see `__CLASS__` resolve to its own name, so the initializer picks up the overridden `DEFAULT_X`.

Field attributes

Attributes after a colon control how the field participates in construction and whether accessor methods are generated for it.

`:param`

Take the field's value from a named argument to the constructor:

```
field $x :param;
field $y :param(the_y_value);
```

By default the parameter name is the field's name with the sigil stripped (`$x` → `x`). An explicit name in parentheses overrides that. Without a defaulting expression the parameter is **required** - omitting it from `->new( . . . )` throws. With a defaulting expression the parameter is optional, and three default operators are available:

- `=` - default applies only when the caller omitted the parameter.
- `//=` - default also applies when the caller passed `undef`.
- `||=` - default also applies when the caller passed a false value.

```
class Point {
    field $x :param = 0;          # default if omitted
    field $y :param //= 0;        # default if omitted or undef
```

(continues on next page)

(continued from previous page)

```
    field $z :param ||= 0;       # default if omitted, undef, or 0
}
```

:reader

Generate a zero-argument accessor method that returns the field. Without an explicit name the method is named after the field (sigil stripped):

```
field $s :reader;

# equivalent to:
field $s;
method s () { return $s }
```

An explicit name is allowed:

```
field $x :reader(get_x);        # method get_x () { return $x }
```

Readers can be applied to array and hash fields too; in list context the method yields the contents, and in scalar context the element count - the usual context behaviour of the underlying variable:

```
field @users :reader;
...
scalar $instance->users;        # count of users
```

:writer

Generate a one-argument setter method that assigns its argument to the field and returns the invocant (to support chaining). The default method name is the field name (sigil stripped) prefixed with `set_`:

```
field $s :writer;

# equivalent to:
field $s;
method set_s ($new) { $s = $new; return $self }
```

An explicit name is allowed:

```
field $x :writer(write_x);      # method write_x ($new) { ... }
```

`:writer` currently only works on scalar fields; applying it to an array or hash field is a compile-time fatal error. Build array or hash writers by hand.

Global state it touches

None directly. Field storage lives on the instance, and initializer expressions run in the constructor's scope - they can read any variable visible at the point of declaration, but `field` itself touches no documented special variable.

Examples

Minimal class with one field, initialized in `ADJUST`:

```
use feature 'class';
no warnings 'experimental::class';

class Greeter {
    field $greeting;

    ADJUST {
        $greeting = "Hello";
    }

    method say_to ($name) {
        say "$greeting, $name";
    }
}

Greeter->new->say_to("world");   # Hello, world
```

Constructor parameters via `:param`, with a mix of required and defaulted fields:

```
class Point {
    field $x :param;             # required
    field $y :param = 0;         # optional, defaults to 0

    method as_string { "($x, $y)" }
}

Point->new(x => 3)->as_string;          # (3, 0)
Point->new(x => 3, y => 4)->as_string;  # (3, 4)
```

Auto-generated accessors cover the common boilerplate:

```
class Person {
    field $name :param :reader;
    field $age  :param :reader :writer;
}

my $p = Person->new(name => "Ada", age => 36);
$p->name;            # "Ada"
$p->age;             # 36
$p->set_age(37);     # returns $p
```

Array and hash fields with defaults:

```
class Bag {
    field @items = ();
    field %counts;

    method add ($item) {
        push @items, $item;
        $counts{$item}++;
        return $self;
    }

    method unique { keys %counts }
}
```

Initializer that depends on an earlier field:

```
class Rect {
    field $w :param;
    field $h :param;
    field $area = $w * $h;       # evaluated after $w, $h are set
}
```

Edge cases

- **Lexical visibility is method-scoped, not file-scoped.** A field is visible from any `method` or `ADJUST` block of the same class, regardless of source file order. It is *not* visible from plain `sub` definitions inside the class block - only from `method`.
- **Per-instance, not per-class.** Two instances of the same class have independent storage for every field. Assigning in one instance does not affect another. If you want class-wide state, use a file-scoped `my` variable outside the field list.
- **Sigil determines type.** `field $x` is a scalar slot, `field @a` an array slot, `field %h` a hash slot. Unlike hash-based objects, there is no way to retype a field at runtime; the slot's container type is fixed by the sigil at declaration.
- **`$self` is unavailable in initializers.** The object is not yet constructed when the initializer runs. Use `__CLASS__` for the class name, and defer anything requiring the full object to `ADJUST`.
- **Initializer order is declaration order.** Later initializers may refer to earlier fields; the reverse is a reference to an uninitialized variable.
- **`:writer` on non-scalar fields is fatal at compile time.** The attribute is defined only for scalars in 5.44; applying it to `@arr` or `%h` aborts compilation of the class.
- **`:param` without a default is mandatory.** `Class->new()` with the parameter missing throws. Add `= EXPR`, `//= EXPR`, or `||= EXPR` to make it optional.
- **No direct external access.** Without `:reader` / `:writer` (or a hand-written method) a field is unreachable from outside the class - that is the encapsulation the class feature provides.
- **Experimental warning.** Parsing a `field` declaration emits an `experimental::class` warning unless that category is suppressed; see *Differences from up-*

stream.

Differences from upstream

Fully compatible with upstream Perl 5.44. Upstream marks the `class` feature - and therefore `field` - as experimental; pperl tracks the same status and emits the same `experimental::class` warning. The known upstream bugs listed in `perlclass` (segfaults around in-file inheritance, `refaliasing` interactions, and leaky encapsulation) apply equally to pperl.

See also

- `class` - declares the class whose body is the only place `field` is legal
- `method` - the only subroutine form that sees field variables by name
- `my` - ordinary lexical declaration; use it for class-wide state that should *not* be per-instance
- `our` - package-global declaration; use it for genuinely global state a class wants to expose
- `bless` - the pre-5.38 way to build objects; `field` replaces the hash-slot bookkeeping that `bless`-based classes needed
- `ref` - check an object's class at runtime; still works on instances produced by a `class` constructor

I/O

fileno

Return the OS-level file descriptor number behind a filehandle.

`fileno` looks past Perl's buffered filehandle abstraction and gives you the raw integer descriptor the kernel uses. It is the bridge between Perl I/O and system calls that speak descriptors directly - `select`, `POSIX::dup2`, `fcntl`, or anything in `IO::Poll` / `IO::Select`. On a directory handle it asks the same question against `dirfd(3)`, which not every platform provides.

Synopsis

```
fileno FILEHANDLE
fileno DIRHANDLE
fileno EXPR             # EXPR evaluated as an indirect handle
```

What you get back

- An integer `>= 0` - the OS file descriptor - for a normal open handle backed by a real kernel object.
- `-1` when the handle is open but has **no** real descriptor. This is what you get for in-memory handles created by `open` with a scalar reference as the third argument (`open my $fh, ">", \$buf`).

- `undef` when the handle is closed, never opened, or - on a directory handle - when the platform lacks `dirfd(3)`. In the dirhandle case `$!` is set.

The three return classes matter. Code that checks `fileno($fh)` for truth conflates "no such fd" with "fd number 0" (STDIN). Always compare explicitly:

```
my $fd = fileno $fh;
die "handle not open"      unless defined $fd;
die "handle has no real fd" if $fd == -1;
```

Global state it touches

- `$!` - set when `fileno` on a directory handle fails because the platform has no `dirfd(3)` equivalent.

`fileno` does not read or write any other I/O globals. It does not flush, it does not seek, it does not change the currently selected handle.

Examples

The common case - turn a Perl filehandle into the fd number a system call wants:

```
open my $fh, "<", "/etc/hostname" or die $!;
print fileno $fh, "\n";             # e.g. "3"
```

Standard handles always have fixed descriptors `0`, 1, 2:

```
print fileno STDIN,  "\n";          # 0
print fileno STDOUT, "\n";          # 1
print fileno STDERR, "\n";          # 2
```

Check whether two handles are duplicates of the same underlying descriptor (the idiom from the upstream POD, adjusted for the `-1` case):

```
if (fileno($this) != -1 && fileno($this) == fileno($that)) {
    print "\$this and \$that are dups\n";
} elsif (fileno($this) != -1 && fileno($that) != -1) {
    print "\$this and \$that have different "
        . "underlying file descriptors\n";
} else {
    print "at least one handle has no real file descriptor\n";
}
```

In-memory handles have no kernel object, so `fileno` returns `-1`:

```
my $buf = "";
open my $mh, ">", \$buf or die $!;
print fileno $mh, "\n";             # -1
```

Build a descriptor bitmap for `select` - the textbook use for `fileno`:

```
my $rin = "";
vec($rin, fileno($sock),  1) = 1;
vec($rin, fileno(STDIN),  1) = 1;
my $n = select(my $rout = $rin, undef, undef, 5.0);
```

Directory handles, where the platform supports `dirfd(3)`:

```
opendir my $dh, "." or die $!;
my $dfd = fileno $dh;
if (defined $dfd) {
    # use with fstat, openat, fchdir, ...
} else {
    warn "no dirfd on this platform: $!";
}
```

Edge cases

- **Closed handle**: returns `undef`. No warning under `use warnings` - `fileno` is deliberately quiet so it can be used as a probe.
- **Never-opened bareword**: `fileno NOSUCH` returns `undef`; strict mode will refuse the bareword at compile time unless it's declared.
- **In-memory scalar handles**: return `-1`, not `undef`. Treating `-1` as "open but unusable for system calls" is the intended contract - the handle itself still works for `print`, `readline`, etc.
- **`fileno(-1)` trap**: an explicit `if (fileno $fh) { ... }` is a bug waiting to happen: `fd 0` is STDIN. Use `defined` and compare against `-1` explicitly.
- **Indirect filehandle via expression**: if `FILEHANDLE` is any expression other than a bareword or simple scalar, its value is taken as an **indirect handle name** (a string), not as a handle object. To pass a handle held in a data structure, dereference first:

```
fileno $handles[0];                 # indirect: uses the string
fileno ${ \$handles[0] };           # still wrong
fileno $handles[0]->fileno;         # wrong: calling a method
 ↪on ""
my $fh = $handles[0]; fileno $fh;   # right
```

- **Directory handle on platforms without `dirfd(3)`**: returns `undef` and sets `$!`. Linux has `dirfd(3)`, so on pperl's supported targets this path is not exercised.
- **Tied handles**: `fileno` calls the tie class's `FILENO` method if defined; otherwise returns `undef`. The tied class is responsible for returning a sensible value (`-1` for "no fd" is the convention).
- **Dup'd handles**: two handles opened with `open` from the same underlying fd (e.g. `open my $copy, ">&", $orig`) return the **new** descriptor, not the original - the dup created a fresh fd.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `open` - creates the filehandles that `fileno` inspects; the `>&` and `<&=` modes take descriptor numbers that round-trip through `fileno`
- `opendir` - source of directory handles for the dirhandle form of `fileno`
- `close` - after which `fileno` on the same handle returns `undef`
- `select` - the four-argument form consumes bitmaps built from `fileno` values
- `binmode` - works on the PerlIO layer above the fd; `fileno` reaches beneath it

Control flow

finally

Run cleanup code on the way out of a `try`/`catch`, whether the body succeeded, threw, or jumped away.

`finally` introduces a third block that can follow a `try`/`catch` pair. Its body runs exactly once, after the `try` body completes - however it completes. A normal fall-through, an exception caught by the `catch`, an exception that escapes because there is no matching `catch`, a `return` from the enclosing sub, a `goto`, a `last`/`next`/`redo` leaving the enclosing loop: every one of these routes through the `finally` block before control continues. The pattern is equivalent to scheduling a `defer` block at the top of the `try`, with a syntax that keeps the cleanup visually attached to the try/catch it belongs to.

Synopsis

```
use feature 'try';
no warnings 'experimental::try';

try    { BODY }
catch ($e) { HANDLE }
finally { CLEANUP }
```

What you get back

`finally` has no return value. The final expression of the block is evaluated for its side effects and then discarded - it does **not** contribute to the value of the enclosing `try`/`catch` expression, even if the whole construct is the last statement of a sub or do block being used to produce a value.

```
my $v = do {
    try    { compute() }
    catch ($e) { $DEFAULT }
    finally { log("done"); 999 }   # 999 is discarded
```

(continues on next page)

(continued from previous page)

```
};
# $v is either compute()'s value or $DEFAULT
```

Use `finally` for cleanup that has to happen, not for computing a result.

Feature status

`finally` is part of the `try` feature, but unlike `try` and `catch` (which became non-experimental in Perl 5.40) it remains experimental in Perl 5.44. Using it without silencing the warning produces:

```
finally is experimental at ... line N.
```

Enable the feature and silence the warning together:

```
use feature 'try';
no warnings 'experimental::try';
```

Because the warning category is `experimental::try` - the same one that `try`/`catch` used to emit - silencing it enables `finally` without re-enabling warnings on the now-stable parts of the feature you do want diagnostics for.

When the block runs

The contract is "on the way out, no matter how." Concretely, `finally` runs after each of the following:

- **Normal completion of `try`.** The `try` body fell off the end or returned its last expression value; `finally` runs next, then the statement after the `try` construct.
- **Exception caught by `catch`.** The `try` body threw, the `catch` matched and handled it; `finally` runs after the `catch` body.
- **Exception not caught.** There is no `catch`, or the `catch` itself re-threw with `die`; `finally` still runs, then the exception continues propagating to the next dynamic `eval`/`try` outward.
- **`return` inside `try` or `catch`.** The enclosing sub is about to return; `finally` runs, then the return takes effect.
- **Loop control (`last`, `next`, `redo`) inside `try` or `catch`.** `finally` runs, then the loop control takes effect in the enclosing loop.
- **`goto &sub` or `goto LABEL` leaving the `try`.** `finally` runs before the jump completes.

The ordering matters: if the `catch` itself raises a new exception, `finally` still runs, and the new exception is the one that propagates (the original is replaced, just as with any `die` inside a handler).

Restrictions on the block body

A `finally` block is a cleanup point, not a control-flow redirect. Perl rejects any control-flow statement that would try to override how the construct is already unwinding:

- **`return`** inside `finally` is a syntax / compile-time error.
- **`goto`** inside `finally` is a syntax / compile-time error.
- **Loop controls** (`last`, `next`, `redo`) targeting a loop *outside* the `finally` block are rejected.

Inside a plain loop that is *contained* in the `finally` body, the loop controls apply to that inner loop as normal - the restriction is on leaving the `finally` block, not on looping inside it.

`die` is permitted. A `finally` block that throws replaces any in-flight exception from the `try`/`catch` with the new one. This is usually a bug (you lose the original failure), so guard cleanup code that might itself fail:

```
finally {
    eval { $fh->close };        # swallow close errors during
 ↪cleanup
}
```

Relationship to `defer`

`finally BLOCK` is "`defer BLOCK` placed at the top of the `try`, attached instead to the try/catch pair." Both run on every exit path from their containing scope; both forbid the same control-flow statements in their body; both discard their return value.

The difference is scope attachment:

- `defer` attaches to the *innermost enclosing block* and fires on that block's exit. It is the general-purpose cleanup primitive and works in any block.
- `finally` attaches to the `try`/`catch` *construct* and fires when that construct is done. It is syntactic sugar for the common case where the cleanup is specifically about what the `try` was doing.

```
# These two are behaviourally equivalent:

try {
    my $fh = acquire();
    defer { release($fh) }
    work($fh);
}
catch ($e) { warn $e }

try {
    my $fh = acquire();
    work($fh);
}
```

(continues on next page)

(continued from previous page)

```
catch ($e) { warn $e }
finally { release_last() }      # if $fh were available here
```

Note one practical consequence: variables declared inside the `try` body go out of scope before `finally` runs. If the cleanup needs a resource allocated inside the `try`, either declare it in an outer scope, or use `defer` inside the `try` where the variable is live.

Examples

Always close a handle, even on error:

```
use feature 'try';
no warnings 'experimental::try';

open my $fh, '<', $path or die "open $path: $!";
try {
    process($fh);
}
catch ($e) {
    warn "processing failed: $e";
}
finally {
    close $fh;
}
```

Release a lock regardless of outcome:

```
my $lock = $mutex->lock;
try {
    critical_section();
}
finally {
    $lock->release;
}
```

Observability: log every exit path:

```
try {
    run($job);
}
catch ($e) {
    $job->mark_failed($e);
    die $e;                     # re-throw; finally still runs
}
finally {
    metrics->tick('job.exit', $job->id);
}
```

Value-producing `try`/`catch`, with `finally` for cleanup only:

```
my $result = do {
    try    { fetch($url) }
    catch ($e) { $CACHED }
    finally { close_connection() }
};
# $result is fetch() or $CACHED; finally's value is never used
```

Edge cases

- **`catch` can be omitted.** `try { ... } finally { ... }` is legal and runs `finally` on both normal and exceptional exit; the exception then propagates as if no `try` were there.
- **Nested `try`/`finally`**: `finally` blocks fire innermost-first as scopes unwind, matching `defer` semantics.
- **Exception inside `finally`**: replaces any in-flight exception. The original is lost unless you captured it in the `catch` first.
- **`$@` during `finally`**: not reliable for "what just happened." If the `finally` needs to know whether it is running after success or failure, set a flag in the `catch` block and inspect it:

  ```
  my $failed;
  try   { risky() }
  catch ($e) { $failed = $e; die $e }
  finally {
      if ($failed) { rollback() } else { commit() }
  }
  ```

- **Global destruction**: `finally` blocks registered by a `try` that is still active when the interpreter starts tearing down run, but the same caveats as `die` inside `DESTROY` apply - keep the body defensive.
- **Parse requires the feature**: without `use feature 'try'`, the word `finally` is not a keyword and parses as a bareword function call, which usually produces a confusing unrelated error. Always enable the feature before using the syntax.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `try` - introduces the block whose exit `finally` hooks; see *Try Catch Exception Handling* in perlsyn
- `catch` - the optional middle block that handles exceptions thrown by `try`; `finally` runs after it either way
- `defer` - the general-purpose "run on scope exit" block; `finally` is `defer` attached to a `try`/`catch` pair

- `die` - the only way to cause the `catch` branch to fire; a `die` inside `finally` replaces any in-flight exception
- `eval` - the older exception-catching mechanism; `try`/`catch`/`finally` supersede the common `eval { ... }; if ($@)` idiom with cleaner scoping and proper control-flow semantics

I/O

flock

Place an advisory lock on an open file.

`flock` is Perl's portable file-locking primitive. It calls the underlying `flock(2)`, `fcntl(2)`, or `lockf(3)` - whichever the platform provides - on `FILEHANDLE` and blocks by default until the requested lock can be granted. Locks cover the **whole file**, never a byte range, and are **advisory**: they coordinate only among processes that also call `flock` on the same file. Code that ignores the lock can still read or modify the contents.

Synopsis

```
flock FILEHANDLE, OPERATION
```

What you get back

True on success, false on failure (with `$!` set). Always check the return value - with `LOCK_NB` a failure is the normal "someone else holds it" signal, not an error:

```
flock($fh, LOCK_EX | LOCK_NB)
    or die "already locked: $!";
```

On a platform that implements none of `flock(2)`, `fcntl(2)` locking, or `lockf(3)`, `flock` raises a fatal error rather than returning false. On Linux (pperl's only supported platform) this never happens.

OPERATION values

`OPERATION` is one of the mutually exclusive request types, optionally bitwise-or'ed with the non-blocking flag. Traditional numeric values are 1, 2, 4, 8 but portable code imports the symbolic names from `Fcntl`, either individually or as a group via the `:flock` tag:

```
use Fcntl qw(:flock);
```

- `LOCK_SH` - shared lock. Multiple processes may hold a shared lock on the same file simultaneously. Used for readers.
- `LOCK_EX` - exclusive lock. Only one process may hold it; no other process may hold any lock on the file at the same time. Used for writers.
- `LOCK_UN` - release whatever lock this filehandle currently holds.
- `LOCK_NB` - bitwise-or with `LOCK_SH` or `LOCK_EX` to make the call non-blocking; `flock` returns false immediately if the lock cannot be granted, instead of waiting.

An `flock` call with a new mode atomically replaces an existing lock on the same filehandle (for example, upgrading `LOCK_SH` to `LOCK_EX`).

Global state it touches

- Sets `$!` on failure.
- Flushes `FILEHANDLE`'s output buffer before locking or unlocking, so locked-region data hits the OS before another process can see the lock state change.

Examples

Exclusive lock for a writer, blocking until granted:

```
use Fcntl qw(:flock);
open my $fh, ">>", "counter.log" or die $!;
flock($fh, LOCK_EX)               or die "lock failed: $!";
print $fh "tick\n";
flock($fh, LOCK_UN)               or die "unlock failed: $!";
close $fh;
```

Non-blocking attempt - skip the work if another process is already running:

```
use Fcntl qw(:flock);
open my $fh, ">", "/var/run/myjob.lock" or die $!;
flock($fh, LOCK_EX | LOCK_NB)
    or do { warn "another instance running\n"; exit 0 };
# ... work ...
# lock auto-released when $fh is closed / program exits
```

Classic mailbox appender. Note the `seek` to `SEEK_END` after locking - on very old systems `>>` append mode is not atomic, so the write position must be re-established once the lock is held:

```
use Fcntl qw(:flock SEEK_END);

sub lock_mbox   { flock($_[0], LOCK_EX) or die "lock: $!" }
sub unlock_mbox { flock($_[0], LOCK_UN) or die "unlock: $!" }

open my $mbox, ">>", "/var/mail/$ENV{USER}" or die $!;
lock_mbox($mbox);
seek($mbox, 0, SEEK_END) or die "seek: $!";
print $mbox $msg, "\n\n";
unlock_mbox($mbox);
close $mbox;
```

Shared read lock while loading a config file, letting other readers proceed concurrently:

```
use Fcntl qw(:flock);
open my $fh, "<", "config.dat" or die $!;
flock($fh, LOCK_SH)             or die "lock: $!";
```

(continues on next page)

(continued from previous page)

```
my @lines = <$fh>;
close $fh;                      # releases the lock
```

Upgrading a shared lock to exclusive - the second `flock` atomically swaps modes:

```
flock($fh, LOCK_SH) or die $!;
# ... inspect contents ...
flock($fh, LOCK_EX) or die $!;  # now we intend to write
```

Edge cases

- **Locks are per-filehandle, not per-file**. Two independent opens of the same path each have their own lock slot. This is why locking your own file twice from the same process through different handles can deadlock or behave surprisingly on `fcntl(2)`-based systems.
- **Closing the filehandle releases the lock.** There is no separate "unlock" step required at program exit; `LOCK_UN` is useful only when you want to keep the handle open after.
- **`LOCK_NB` failure is not a croak.** It returns false and sets `$!` to `EWOULDBLOCK`. Treat it as an expected outcome, not an error.
- **Advisory only.** A process that never calls `flock` sees no locks. Use `flock` as a cooperation protocol among your own programs, not as a security mechanism.
- **Whole files only.** There is no byte-range locking through `flock`. For record-level locking use `fcntl` with `F_SETLK` / `F_SETLKW`.
- **NFS and other network filesystems.** Some older NFS implementations cannot honour `flock` across hosts; the call may silently become a no-op or return success without providing exclusion. Use `fcntl` locking if network coordination matters.
- **Fork inheritance.** When the platform uses real `flock(2)` (Linux does), locks are inherited across `fork`; the child shares them with the parent. On `fcntl(2)`-emulated platforms locks do not survive the fork, which makes writing locking servers considerably harder.
- **Buffer flushing.** Perl flushes `FILEHANDLE`'s output buffer on every `flock` call. Code that mixes buffered `print` with locking does not need an explicit flush before the lock boundary.
- **Interaction with dup-style opens.** An open of the form `open(my $A, ">>&=", $B)` shares the underlying file descriptor with `$B`, so `flock($A)` and `flock($B)` act on the same lock slot. A `>>&` dup gives each handle its own descriptor and its own lock slot.

Differences from upstream

Fully compatible with upstream Perl 5.44. pperl runs on Linux only, where `flock(2)` is always available, so the "platform has no locking primitive - fatal error" path documented upstream never triggers in practice.

See also

- `fcntl` - byte-range and advisory locking with finer control; use it when whole-file granularity is too coarse or when network filesystems require it
- `open` - produces the filehandle `flock` operates on; remember that the required open mode (< vs >) matters for `fcntl`-emulated `LOCK_SH` / `LOCK_EX`
- `close` - releases any `flock` still held on the handle
- `seek` - pair with `flock` when appending to files that other processes may have extended while you waited for the lock
- `fork` - lock inheritance across child processes depends on the underlying locking primitive
- `Fcntl` - source of the `LOCK_SH`, `LOCK_EX`, `LOCK_UN`, `LOCK_NB` constants and the `:flock` import tag

Processes

fork

Create a new process running the same program at the same point.

`fork` issues the `fork(2)` system call. After it returns, **two** processes are executing the same Perl program at the same statement - the original (parent) and a near-identical copy (child). The two are distinguished only by `fork`'s return value. File descriptors and, on some systems, the locks held on them are shared between parent and child; everything else is copied. On Linux the copy is cheap: data pages are shared copy-on-write, so the actual duplication happens only as each side writes to memory.

Synopsis

```
my $pid = fork;
```

What you get back

Three return values, one per role:

- **Parent**: the child's PID (a positive integer).
- **Child**: `0`.
- **Failure**: `undef`, with `$!` set to the `errno` from the underlying `fork(2)` (typically `EAGAIN` - resource limit hit - or `ENOMEM`).

The canonical dispatch is a three-way branch. Test for `undef` first, then for `0`, so the parent path is the fall-through:

```
my $pid = fork // die "fork failed: $!";
if ($pid == 0) {
    # child
    exec $cmd, @args;
    die "exec failed: $!";
```

(continues on next page)

(continued from previous page)

```
}
# parent continues here, with $pid = child's PID
```

The two-line "fork or die" idiom:

```
defined(my $pid = fork) or die "fork: $!";
```

Writing `fork or die` (without `defined`) is a bug: it treats the child's legitimate `0` return as failure and dies in every child.

Flush before you fork

Perl attempts to flush all output handles before the `fork(2)` call, but you should not rely on that alone - any unflushed buffered data that does slip through is duplicated in both processes, so the same bytes get written twice. The canonical defence is to disable buffering on any handle you intend to use across a fork:

```
STDOUT->autoflush(1);           # or: $| = 1 while STDOUT is␣
↪selected
STDERR->autoflush(1);

print "about to fork\n";        # now actually on the wire
my $pid = fork // die "fork: $!";
```

Loading `IO::Handle` is not required for `autoflush` in modern Perl - the method is available on all filehandles. Setting `$|` on the currently selected handle has the same effect for that one handle.

Reaping children - or `$SIG{CHLD}`

Every child that terminates before its parent reaps it becomes a **zombie**: a kernel process-table entry held open so the parent can read the exit status. Ignore them and the process table fills up.

Two ways to keep it clean:

- Actively reap with `wait` or `waitpid`:

```
my $pid = fork // die "fork: $!";
if ($pid == 0) {
    # ... child work ...
    exit 0;
}
my $reaped = waitpid $pid, 0;     # block until this child exits
my $status = $?;                  # exit status of the child
```

- Tell the kernel you do not care:

```
$SIG{CHLD} = 'IGNORE';            # children auto-reaped, no␣
↪status
```

Useful for fire-and-forget workers. You lose the exit status - `wait` and `waitpid` will also no longer see the children.

For long-running parents that do care about exit status, install a reaper handler that drains all pending children without blocking:

```
$SIG{CHLD} = sub {
    while ((my $pid = waitpid -1, POSIX::WNOHANG) > 0) {
        # optionally inspect $? here
    }
};
```

See `perlipc` for fuller patterns.

What the child inherits

- **Open file descriptors**: shared. A `print` in either process advances the same underlying file offset. Close the descriptors the child does not need, and reopen handles the child should not share with the parent - particularly `STDIN`/`STDOUT`/`STDERR` when they are connected to a pipe or socket driving the parent's caller. A backgrounded CGI script that forks and exits without closing its inherited `STDOUT` will leave the HTTP client hanging, because the child still holds the socket open:

  ```
  open STDIN,  '<', '/dev/null' or die $!;
  open STDOUT, '>', '/dev/null' or die $!;
  open STDERR, '>', '/dev/null' or die $!;
  ```

- **Memory**: logically copied, physically copy-on-write. Mutating a large data structure in one side materialises only the touched pages.
- **Process ID**: changes. `$$` (also `$PID` / `$PROCESS_ID` under `English`) is re-read from the kernel on access after a fork, so both parent and child see their own PID.
- **Signal handlers, %ENV, working directory, umask, process group, controlling terminal**: all copied.
- **Pending alarms and timers**: not inherited by the child (per POSIX).
- **Locks held via `flock`**: shared with the parent - both processes hold the same lock on the same open file description. Locks acquired via `fcntl` record locks behave differently; consult `fcntl(2)`.

Examples

Classic fork/exec - launch an external command without going through the shell:

```
my $pid = fork // die "fork: $!";
if ($pid == 0) {
    exec '/usr/bin/gzip', '-9', $file;
    die "exec: $!";                  # only reached if exec fails
}
waitpid $pid, 0;
die "gzip failed: $?" if $?;
```

Fan out N workers, then reap them all:

```
my @kids;
for my $i (1 .. 4) {
    my $pid = fork // die "fork: $!";
    if ($pid == 0) {
        do_work($i);
        exit 0;
    }
    push @kids, $pid;
}
waitpid $_, 0 for @kids;
```

Parent/child communication through a pipe. Open the pipe **before** forking so both sides inherit the two ends:

```
pipe(my $reader, my $writer) or die "pipe: $!";
my $pid = fork // die "fork: $!";
if ($pid == 0) {
    close $reader;
    $writer->autoflush(1);
    print $writer "hello from $$\n";
    exit 0;
}
close $writer;
chomp(my $line = <$reader>);
waitpid $pid, 0;
print "got: $line\n";
```

Note the `close` on the unused end in each process - leave both ends open in both processes and the reader will never see EOF.

Detach a daemon-style child and let the kernel reap it:

```
$SIG{CHLD} = 'IGNORE';
my $pid = fork // die "fork: $!";
if ($pid == 0) {
    # child: will be auto-reaped on exit
    run_background_task();
    exit 0;
}
# parent moves on, never calls wait
```

Edge cases

- **fork inside a thread or after DESTROY-sensitive code**: only the calling thread survives in the child; anything waiting on a mutex or joined thread in another thread is gone. Keep fork points before you spawn threads.
- **Buffered filehandles in the child**: any data still buffered at the moment of fork is present in both processes' buffers. When each side eventually flushes, the bytes are written twice. Autoflush before forking.

- **`exit` in the child runs `END` blocks and `DESTROY`-ers** written by the parent. That is rarely what you want - prefer `POSIX::_exit` in the child after an `exec` failure, or ensure `END` blocks check `$$` against the PID recorded at program start.
- **Error vs child confusion**: `fork` returns `0` legitimately in the child and `undef` on failure. Use `//` and `defined`, never `||` / truth tests.
- **`EAGAIN`**: transient - the kernel's per-user process limit is reached. Retry with a short sleep, or lower your concurrency.
- **Working directory and `chdir` in the child** do not affect the parent. The same is true of `%ENV`, `umask`, and signal handlers: the child has its own copy from the moment of fork on.
- **Seeks on shared descriptors are shared**: if parent and child both write to the same inherited `STDOUT`, their output interleaves at whatever the OS scheduler decides. Coordinate with explicit locking or give each child its own output handle.

Differences from upstream

Fully compatible with upstream Perl 5.44.

pperl targets Linux only, so `fork` is always a real `fork(2)`. Unlike traditional perl5 on Windows - where `fork` is emulated with interpreter-level *pseudo-processes* sharing a single OS process, with the quirks documented in `perlfork` - the pperl `fork` creates an independent OS process with its own PID, address space, and process table entry. Code that assumed pseudo-process semantics (shared global state across "forked" sides) will not port to pperl.

See also

- `exec` - replace the current process image; the typical next call in a child right after `fork`
- `wait` - block until any child exits and return its PID and status in `$?`
- `waitpid` - reap a specific child, optionally non-blocking via `WNOHANG`
- `exit` - terminate the current process; use in the child to end it without falling through into the parent's code
- `pipe` - create a pair of connected filehandles before forking to give parent and child a communication channel
- `$$` - the current process ID, re-read after `fork` so each side sees its own PID
- `%SIG` - in particular `$SIG{CHLD}`, controlling whether children are auto-reaped or must be collected with `wait` / `waitpid`
- `perlipc` - fuller treatment of forking, signals, pipes, and reaping moribund children

I/O

format

Declare a picture-based report template for use by `write`.

`format` is a **declaration**, not an executable statement. It binds a named picture - a block of fixed-column output lines with embedded field markers - to the symbol table, where `write` later looks it up by name. Pictures do not interpolate variables; each picture line is followed by an argument line whose expressions fill the fields in order. `format` is a legacy feature of Perl; upstream notes its use is limited and points at `perlform` for the full field grammar.

Synopsis

```
format NAME =
FORMLIST
.

format =          # declares the "STDOUT" format
FORMLIST
.
```

A `FORMLIST` is a sequence of lines, each one of three kinds: a comment (`#` in column 1), a **picture line**, or an **argument line** supplying values for the picture line immediately above it. A single `.` in column 1 terminates the declaration.

What you get back

`format` is a declaration and has no runtime value. It installs the compiled picture into the current package's format table under `NAME` (or `STDOUT` if `NAME` is omitted). It is not called; it is triggered indirectly by `write`.

The pieces of a picture

Each picture line mixes literal text with **field markers**. A field begins with `@` (regular field) or `^` (special field, used for fill mode and for blankable numeric fields) and extends through pad characters that set both the width and the justification:

- `<` - left-justified text
- `>` - right-justified text
- `|` - centered text
- `#` - right-justified numeric; a leading `0` pads with zeroes; `.` places the decimal point
- `...` - at the tail of a text field, renders three dots when the value was truncated
- `@*` - variable-width field that emits a multi-line scalar verbatim
- `^*` - variable-width field that emits the next line of a scalar and consumes that portion of the source variable

Two suppression tokens may appear anywhere on a picture line and are not part of any field:

- ~ - if every field on this line is empty, the line is suppressed entirely; the ~ itself renders as a space otherwise
- ~~ - like ~, and additionally repeats the line until every field on it is exhausted (every source scalar is empty or every `shift`-style expression returns undef)

`perlform.pod` gives the full grammar of picture lines including fill mode, multi-line fields, overflow (####), and locale-dependent decimal points.

The argument line

The line immediately below a picture line is the **argument line**: a comma-separated list of expressions, one per field, evaluated in list context before the picture is rendered. A single list-returning expression may fill multiple fields. The argument list may be broken across lines if and only if the first token on the first line is an opening brace:

```
format Report =
@<<<<<<<< @>>>>>>
{ $name,
  $score }
.
```

Globals that control `write`

Output through `format` is driven entirely by per-filehandle globals. Each is attached to the **currently selected filehandle**; changing them for another handle requires a `select` dance (see below):

- `$~` - name of the current format used for each record (defaults to the filehandle's own name)
- `$^` - name of the current top-of-page format (defaults to the filehandle's name with `_TOP` appended)
- `$=` - page length in lines (default 60)
- `$-` - lines remaining on the current page; `write` decrements it and triggers the top-of-page format when it reaches zero
- `$%` - current page number, incremented automatically
- `$:` - characters that fill mode (^) may break on (default `" \n-"`)
- `$^L` - string emitted before each top-of-page except the first (default `"\f"`)

Under `use English` these are `$FORMAT_NAME`, `$FORMAT_TOP_NAME`, `$FORMAT_LINES_PER_PAGE`, `$FORMAT_LINES_LEFT`, `$FORMAT_PAGE_NUMBER`, `$FORMAT_LINE_BREAK_CHARACTERS`, and `$FORMAT_FORMFEED`.

Because these are per-handle, setting them for a handle other than the currently selected one needs `select`:

```
my $ofh = select $fh;
$~ = 'Report';
$^ = 'Report_TOP';
select $ofh;
```

Top-of-form

When `write` decrements `$-` to zero, it emits `$^L` (except on the very first page), resets `$-` to `$=`, increments `$%`, and invokes the format named by `$^`. The default top-of-page format name is the filehandle's name with `_TOP` appended, so declaring `STDOUT_TOP` suffices for writes through `STDOUT`.

Examples

A minimal report with a header:

```
format STDOUT_TOP =
        Inventory
Name            Qty   Price
---------------------------
.
format STDOUT =
@<<<<<<<<<<<<<  @>>  @##.##
$name,          $qty, $price
.

for my $row (@rows) {
    ($name, $qty, $price) = @$row;
    write;
}
```

Switching formats for a non-selected handle:

```
open my $fh, '>', 'report.txt' or die $!;
my $ofh = select $fh;
$~ = 'Report';
$^ = 'Report_TOP';
select $ofh;
write $fh;
```

Fill mode with ~~ to drain a multi-line scalar one line at a time:

```
format STDOUT =
Note: ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
      $note
~~    ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
      $note
.
```

The ^ fields consume `$note` as they render; the ~~ line repeats until `$note` is empty.

Edge cases

- **Lexical visibility**: a format sees a `my` variable only when declared inside that variable's scope. A top-level format cannot see `my` variables declared inside a subroutine; use package globals or declare the format inside the sub.

- **Quoted pictures**: if you build a picture with `eval` from a double-quoted string, `@name` looks like an array interpolation. Escape it or use single quotes around the template.
- **A lone `.` is the terminator**: a picture whose literal text would begin with `.` in column 1 must indent that line. This also bites format code sent through naive mail transports that cut at `.` on a line of its own.
- **Numeric overflow**: a #-field narrower than the formatted number renders as #### rather than truncating silently.
- **Locale decimal point**: if `use locale` was in effect when the format was **declared**, `LC_NUMERIC` picks the decimal character. Toggling `use locale` at `write` time does not change the picture.
- **Control characters** inside a fixed-width text field are replaced by spaces to preserve column alignment.
- **Mixing `print` and `write`**: legal on the same handle, but `$-` only tracks lines emitted by `write`. Adjust `$-` manually if interleaved `print` output consumes page lines you care about.
- **No footer mechanism**: there is no `$FORMAT_BOTTOM_NAME`. Footers are done by hand - test `$-` before each `write` and emit the footer yourself when it reaches a threshold.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `write` - render one record through the current format into the selected filehandle; the only way a declared format produces output
- `select` - change which filehandle is current, required when setting `$~`, `$^`, `$=`, `$-`, `$%`, `$:`, or `$^L` for a handle other than `STDOUT`
- `$~` - current format name
- `$^` - current top-of-page format name
- `$=` - page length
- `$-` - lines left on the current page
- `$%` - current page number
- `$:` - fill-mode break characters
- `perlform.pod` - full picture-line grammar, fill-mode rules, `formline` / `$^A` internals, and worked report examples

Misc

formline

Format a list of values into a picture string and append the result to the format accumulator `$^A`.

`formline` is the low-level engine behind Perl's `format` / `write` report mechanism. It takes a `PICTURE` string describing field widths and alignment, interpolates the values from `LIST` into those fields, and appends the resulting text to the global `$^A` accumulator (also known as `$ACCUMULATOR` under `use English`). Every call to `write` flushes `$^A` to the associated filehandle and clears it; outside of `write`, you manage `$^A` yourself - read it for the formatted text, then reset it to `""` before the next call.

`PICTURE` is the same mini-language that `format` uses: `@<<<<` for a left-justified text field, `@>>>>` for right-justified, `@||||` for centered, `@####.##` for a right-justified numeric field with a decimal point, `@*` for an unbounded multi-line text field, and `^<<<` (and friends) for the fill-mode field that consumes its scalar argument incrementally. See `perldoc perlform` for the full picture vocabulary.

Synopsis

```
formline PICTURE, LIST
formline $picture, @values
```

What you get back

`formline` always returns 1. The useful output is the side effect: the formatted text is **appended** to `$^A`. If you need the text as a value, read `$^A` after the call and then clear it:

```
$^A = "";
formline '@<<<<<<<<<  @>>>>', $name, $score;
my $line = $^A;
$^A = "";
```

Because the result goes to an accumulator, consecutive `formline` calls build up multi-line output without any intermediate join.

Global state it touches

- `$^A` - the **format accumulator**. `formline` appends to it; it is never cleared automatically. Also known as `$ACCUMULATOR` under `use English`.
- `$:` - the set of characters on which fill-mode fields (`^*`, `^<<<`) are allowed to break. Defaults to `" \n-"`. Also known as `$FORMAT_LINE_BREAK_CHARACTERS` under `use English`.
- `LC_NUMERIC` locale - if `use locale` is in effect when the picture is compiled, the decimal point character in numeric fields (`@###.##`) follows the locale.
- Fill-mode fields (`^...`) **mutate their scalar arguments in place**, chopping off the portion that was consumed so the next `formline` call picks up where the previous one left off. Pass a copy if you need the original preserved.

Examples

Build one formatted line and read it back:

```
$^A = "";
formline '@<<<<<<<<  @>>>>>', "widget", 42;
print $^A, "\n";            # "widget       42\n"
$^A = "";
```

Build a multi-line block in one call - the picture may contain newlines, and each line of the picture consumes its own values from `LIST`:

```
$^A = "";
formline <<'END', "Alice", 30, "Bob", 25;
@<<<<<<<<<<  @###
@<<<<<<<<<<  @###
END
print $^A;                  # "Alice         30\nBob           25\n"
$^A = "";
```

An `swrite` helper - `formline` is to `write` what `sprintf` is to `printf`:

```
sub swrite {
    my $picture = shift;
    local $^A = "";
    formline $picture, @_;
    return $^A;
}

my $row = swrite '@<<<<<<<<  @>>>>', "total", 1234;
```

Fill-mode field consumes its scalar across repeated calls - note that `$text` shrinks each time:

```
my $text = "the quick brown fox jumps over the lazy dog";
$^A = "";
formline '^<<<<<<<<<<<<<<', $text;
formline "\n";
formline '^<<<<<<<<<<<<<<', $text;
print $^A;                  # two lines, word-broken on spaces
$^A = "";
```

Numeric field with decimal places and overflow filling - a value that does not fit is rendered as # characters:

```
$^A = "";
formline '@##.##', 3.14159;
formline "\n";
formline '@##.##', 9999.9;  # overflow
print $^A;                  # "  3.14\n######\n"
$^A = "";
```

Edge cases

- **Quoting the picture**: inside double quotes, `@` introduces an array interpolation. Write pictures as single-quoted strings or heredocs with a quoted terminator (`<<'END'`) to keep `@` literal. The classic trap: `formline "@<<<<<<<", $name` interpolates `@<` before `formline` ever sees the string.
- **Newlines in the picture**: `formline` does not care how many newlines `PICTURE` contains. The `~` (suppress-if-all-empty) and `~~` (repeat-until-exhausted) tokens treat the whole picture as one logical line, so a record format with `~~` typically needs one `formline` per physical line. A `format` declaration splits a multi-line form into one `formline` call per output line; when calling `formline` directly, you make that split yourself.
- **Accumulator is not cleared**: the most common mistake is forgetting to reset `$^A` between logical uses. Appending is the defined behaviour, not a bug. Use `local $^A = "";` inside helpers.
- **Field count vs. value count**: surplus values in `LIST` are silently ignored; missing values render as empty (regular field) or blanked (special `^` numeric field).
- **`undef` in a regular field** stringifies to the empty string under `use warnings` and triggers a `uninitialized` warning. A special field (`^###`) blanks the whole field on undef without warning.
- **Using a format scalar - capturing `write` output**: opening a filehandle onto a scalar reference is often easier than driving `formline` manually:

```
open my $fh, ">", \my $out or die $!;
format FH =
@<<<<<<<<  @>>>>
$name,     $score
.
$~ = "FH";
select((select($fh), $~ = "FH")[0]);
write $fh;
```

  Reach for `formline` directly when you want the picture-to-string path without involving a filehandle or a named `format` declaration.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `write` - drives one or more `formline` calls from a named `format` and flushes `$^A` to a filehandle
- `format` - declaration syntax for the picture/value pairs that `write` feeds into `formline`
- `sprintf` - the other "format into a string" primitive; use `sprintf` for `printf`-style %-directives, `formline` for fixed-width report columns and fill-mode text blocks

- `$^A` - the accumulator `formline` writes into; read it for the result, clear it before the next logical use
- `$:` - characters that fill-mode fields (^*, ^<<<) may break on

I/O

getc

Read the next single character from a filehandle.

`getc` returns one character of input from `FILEHANDLE`, advancing the handle's read position by that character. If `FILEHANDLE` is omitted, `getc` reads from `STDIN`. At end-of-file or on a read error, `getc` returns `undef` and - in the error case - sets `$!`. The unit is a **character**, not a byte: under a `:utf8` or `:encoding(...)` PerlIO layer a single call consumes however many bytes make up one decoded codepoint.

Synopsis

```
getc FILEHANDLE
getc
```

What you get back

A one-character string on success, or `undef` at end-of-file or on error. Distinguish the two by checking `$!`:

```
my $ch = getc $fh;
if (!defined $ch) {
    die "read error: $!" if $!;
    # otherwise clean EOF
}
```

Because `undef` collapses with the empty string `""` under stringy comparisons, always test with `defined`, never with `length` or a string match.

Global state it touches

- `$!` - set when `getc` returns `undef` because of a read error (not set for a clean end-of-file).
- The current input filehandle - the no-argument form reads from `STDIN`, not from the handle selected with `select`. `select` governs default output, not default input.
- The handle's PerlIO layer stack - determines whether the returned scalar is a byte or a decoded character, and whether CRLF translation is applied on the way in.

Examples

Read one character from `STDIN`:

```
my $ch = getc;
print "got: $ch\n" if defined $ch;
```

Read from an explicit filehandle and detect end-of-file:

```
open my $fh, "<", "input.txt" or die "open: $!";
while (defined(my $ch = getc $fh)) {
    print $ch;
}
close $fh;
```

Read characters, not bytes, from a UTF-8 file. Each call returns one decoded codepoint even when the underlying byte stream uses multi-byte sequences:

```
open my $fh, "<:encoding(UTF-8)", "utf8.txt" or die $!;
my $ch = getc $fh;      # e.g. "ä" as one character, even though
                        # it occupies two bytes on disk
```

Consume a yes/no prompt without waiting for a whole line - requires turning off terminal line buffering first (see *Edge cases*):

```
print "continue? [y/n] ";
system "stty", "-icanon", "eol", "\001";
my $key = getc(STDIN);
system "stty", "icanon", "eol", "^@";
print "\n";
exit unless defined $key and lc $key eq "y";
```

Edge cases

- **Terminal line buffering**: on an interactive `STDIN`, the terminal driver hands Perl nothing until the user presses Enter, so `getc` blocks until a full line is available and then returns only its first character. `getc` itself cannot bypass this. To read a single keypress, put the terminal into non-canonical ("cbreak") mode first - shell out to `stty`, use `POSIX::termios` via the `POSIX` module for a portable in-process solution, or use the CPAN module `Term::ReadKey` for a friendlier interface. Remember to restore the terminal mode on exit, including from signal handlers.
- **Bytes vs characters under I/O layers**: the return value is whatever the handle's layer stack produces. A bytes-mode handle yields one byte per call; a `:utf8` or `:encoding(...)` handle yields one decoded character per call, consuming as many bytes off the stream as that character takes. Malformed input under a decoding layer triggers the layer's usual warning or error rather than silently returning garbage.
- **End-of-file vs error**: both return `undef`. `$!` is cleared before the call is entered but only set when an actual read error occurred, so `defined $ch or $! and die ...` is the idiom.
- **Closed or unopened filehandle**: returns `undef` and sets `$!` to `Bad file descriptor`. Under `use warnings` a `getc() on closed filehandle` warning is emitted.

- **Efficiency**: `getc` is one-character-at-a-time, which crosses the PerlIO layer stack per call. For bulk scanning, `read`, `sysread`, or `readline` are substantially faster; reserve `getc` for interactive prompts and protocol parsers that genuinely need character-at-a-time control.
- **Default handle is `STDIN`, not `ARGV`**: unlike the diamond operator `<>`, `getc` without an argument does not walk `@ARGV` or fall back to the magic `ARGV` handle - it reads `STDIN` directly.
- **Wide terminal input**: reading a character from a terminal in raw mode still returns one codepoint only when the input layer decodes UTF-8. A paste containing a multi-byte character under a bytes-mode handle surfaces as several separate `getc` calls, one per byte.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `read` - read a fixed number of characters or bytes in one call, much faster than looping `getc`
- `readline` - read one whole record (line) at a time, governed by `$/`
- `sysread` - unbuffered read straight from the OS, bypassing the PerlIO layer stack
- `eof` - test whether the next `getc` / `readline` would return `undef` for end-of-file
- `$!` - errno set when `getc` returns `undef` because of a read error

User and group info

getgrent

Read the next entry from the system group database.

`getgrent` walks the host's group database one entry at a time, the way a plain loop over `/etc/group` would read it line by line, and returns a Perl-shaped view of each record. It is the iterator form of the group lookups - use it when you want every group, or every group matching some predicate. For single-entry lookup by name or gid, reach for `getgrnam` or `getgrgid` instead.

A first call to `getgrent` implicitly opens the database. Iteration continues across further calls until the database is exhausted, at which point `getgrent` returns the empty list (or `undef` in scalar context). Call `setgrent` to rewind to the start, or `endgrent` to close the database and free any resources the C library is holding.

Synopsis

```
my ($name, $passwd, $gid, $members) = getgrent;
my $name                            = getgrent;        # scalar␣
↪context
```

What you get back

In **list context**, a four-element list - one entry per group:

Index	Field	Meaning
0	`$name`	Group name (e.g. `"wheel"`)
1	`$passwd`	Group password, typically `"x"` on shadow-password hosts
2	`$gid`	Numeric group id
3	`$members`	Space-separated string of member login names

When iteration runs off the end, the return is the empty list. If you write `if (my @gr = getgrent)` the loop terminates cleanly on end-of-database.

In **scalar context**, `getgrent` returns just `$name`. When the database is exhausted, it returns `undef` - which is the signal most scripts test against to end a `while` loop.

`$members` is a plain string, not an array. Split it on whitespace when you need the list:

```
my @logins = split ' ', $members;
```

Global state it touches

`getgrent` keeps **per-process iteration state** inside the C library (conceptually a file position inside the group database). Two calls in the same process share that state; a nested loop that calls `getgrent` inside another `getgrent` loop will interleave records and miss most of them. Serialise the passes, or capture all entries into an array first:

```
my @all;
while (my @gr = getgrent) { push @all, [ @gr ] }
endgrent;
```

The iterator is not reset when a subroutine returns. If library code you do not control calls `getgrent`, it moves your position. Guard with `setgrent` when you need a clean start.

Examples

Walk every group and print the name and member count:

```
while (my ($name, undef, $gid, $members) = getgrent) {
    my $n = length $members ? scalar(split ' ', $members) : 0;
    printf "%-16s gid=%d members=%d\n", $name, $gid, $n;
}
endgrent;
```

Collect all groups a given login belongs to:

```
my $user = 'alice';
my @groups;
while (my ($name, undef, undef, $members) = getgrent) {
```

(continues on next page)

(continued from previous page)

```
    push @groups, $name
        if grep { $_ eq $user } split ' ', $members;
}
endgrent;
```

Scalar context - list every group name, one per line:

```
while (defined(my $name = getgrent)) {
    print $name, "\n";
}
endgrent;
```

Rewind and re-read, for example to build two indexes in one pass each:

```
setgrent;
my %by_name = map { $_->[0] => $_ }
              map { [ getgrent ] } 1 .. ();   # collect…
# …or just loop twice with an explicit setgrent before each pass.
setgrent;
while (my @gr = getgrent) { ... }
endgrent;
```

Convert a numeric gid from `stat` to a group name without a second lookup by caching the iterator output:

```
my %name_of;
while (my ($name, undef, $gid) = getgrent) {
    $name_of{$gid} = $name;
}
endgrent;

my $gid = (stat $path)[5];
print "group: $name_of{$gid}\n";
```

Edge cases

- **End of iteration vs. a real entry**: in list context, an exhausted database returns the empty list - which assigns as zero elements, so `my @gr = getgrent; @gr or last;` is the idiomatic termination test. In scalar context, test with `defined` rather than truthiness, since a legitimate group name could in principle be the string `"0"`.
- **No password field on modern Linux**: `$passwd` is almost always `"x"` (real hashes live in `/etc/gshadow`). Do not treat a non-`"x"` value as "password set" without consulting the shadow database.
- **Empty `$members`**: groups with no explicit members (users listed only by primary gid) have `$members` as the empty string. `split ' ', ""` returns an empty list - the idiomatic way to get members as an array without a special case.
- **Implicit primary-gid membership is not reported**: `$members` lists only the users explicitly named in the group's `/etc/group` record. A user whose primary gid

matches this group but who is not also listed in the `members` field does **not** appear. Cross-check with `getpwent` if you need both kinds of membership.

- **Nested iteration clobbers position**: a second `getgrent` loop started before the first one finishes shares the same cursor. Use `setgrent` between passes, or snapshot the database into memory up front.
- **Taintedness**: the `$passwd` field is tainted under `-T`, same as for `getpwent` - treat it as untrusted input even though the usual value is `"x"`.
- **Process-wide, not thread-wide**: on systems without thread-safe variants the iteration state is per process, not per thread. Two threads walking `getgrent` at once will miss records.
- **`endgrent` is optional but polite**: leaving the database open to program exit works, but if you are done iterating, call `endgrent` so the C library can drop its buffers.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `getgrnam` - single lookup by group name; use when you already know the name and want just one record
- `getgrgid` - single lookup by numeric gid; the usual partner of `stat` when you need a group name for a file
- `setgrent` - rewind the iterator to the start of the group database before another full-scan pass
- `endgrent` - close the database and release any resources the C library holds open
- `getpwent` - the parallel iterator over the passwd database; pair with `getgrent` when resolving full user-to-group relationships
- `scalar` - force scalar context when you only need the group name and want to skip the other fields

User and group info

getgrgid

Look up a group record by numeric group ID.

`getgrgid` asks the system group database for the entry whose `gid` is `GID` and returns what it finds. It is the numeric counterpart of `getgrnam` and mirrors the C library call `getgrgid(3)`. In list context you get the unpacked record fields; in scalar context you get the group name.

Synopsis

```
my ($name, $passwd, $gid, $members) = getgrgid($gid);
my  $name                           = getgrgid($gid);
```

What you get back

In **list context**, four values unpacked from `struct group`:

```
#    0        1         2      3
my ( $name,  $passwd,  $gid,  $members ) = getgrgid($gid);
```

- `$name` - the group name.
- `$passwd` - the group password field. On modern systems this is typically `"x"` or empty; real hashes live in a shadow database.
- `$gid` - the numeric group ID (the same number you passed in, echoed back from the record).
- `$members` - a **single space-separated string** of member login names, not a list. Split it yourself when you need an array:

  ```
  my @users = split / /, (getgrgid($gid))[3];
  ```

In **scalar context**, just the group name:

```
my $name = getgrgid(0);    # "root" on most Linux systems
```

If no group with the given `GID` exists, the function returns the empty list in list context and `undef` in scalar context. A system-level failure (e.g. NSS backend unreachable) is reported the same way; check `$!` if you need to distinguish.

Global state it touches

- `$!` - set on system-level failure of the underlying `getgrgid(3)` call. Not set when the `GID` simply has no matching entry; that is a normal "not found", not an error.
- The process-wide group-database iteration cursor maintained by libc. `getgrgid` does **not** advance or reset the cursor used by `getgrent`, `setgrent`, and `endgrent`, but it shares the same underlying file descriptor on most implementations - interleaving direct lookups with iteration is portable in theory and surprising in practice.

Examples

Look up the primary group name for the current process:

```
my $group = getgrgid($();           # $( is the real GID
print "running as group: $group\n";
```

Resolve a file's group ID to a name, with a sensible fallback for orphaned GIDs (files owned by a group that no longer exists):

```
my $gid   = (stat $file)[5];
my $name  = getgrgid($gid) // "<unknown gid $gid>";
```

Unpack the full record and list the members:

```
if (my ($name, undef, $gid, $members) = getgrgid(100)) {
    my @users = split / /, $members;
    print "group $name ($gid) has ", scalar(@users), " members\n";
}
```

Guard against the "no such group" case explicitly - the empty list in boolean/list context is already falsy, so an `if` suffices:

```
unless (my @gr = getgrgid($gid)) {
    die "no group with gid $gid\n";
}
```

Pair with `getpwuid` to render a directory listing's owner and group columns:

```
my @st   = stat $path                 or die "stat $path: $!";
my $user = getpwuid($st[4]) // $st[4];
my $grp  = getgrgid($st[5]) // $st[5];
printf "%-8s %-8s %s\n", $user, $grp, $path;
```

Edge cases

- **Scalar vs list context matters.** `getgrgid($gid)` in scalar context yields only the group name; wrap in parentheses or assign to a list to get the full record:

  ```
  my $name    = getgrgid(0);              # "root"
  my @record  = getgrgid(0);              # four-element list
  my $gid_out = (getgrgid(0))[2];         # the gid field
  ```

- **`$members` is a string, not an array.** The whitespace-joined form is historical and matches perl5's `pp_gpwent` / `pp_ggrent` output. Splitting on `/ /` (single space) is the canonical idiom; splitting on `/\s+/` is slightly more forgiving if a backend ever uses tabs.

- **Empty groups** return `$members` as the empty string. `split / /, ""` yields the empty list, which is what callers usually want.

- **Numeric coercion.** `GID` is passed to the C call as an integer. A non-numeric scalar is coerced: `"0root"` becomes `0`. Pass a clean integer.

- **GID namespace overlap.** Multiple group entries with the same `gid` are legal in `/etc/group` and in NSS. `getgrgid` returns **one** of them - which one is backend-defined. Do not rely on which name comes back when GIDs collide; use `getgrnam` for the reverse direction when the name matters.

- **NSS / nsswitch.conf.** The lookup follows the system's name service switch - `files`, `sss`, `ldap`, etc. A successful lookup in a test harness on one host may fail on another with a different NSS configuration. This is a property of the C library, not of Perl.

- **Large groups.** Very large group-member lists (hundreds of members) can arrive truncated if the backend's internal buffer is smaller than the rendered line. This is a libc / NSS limitation; Perl faithfully surfaces whatever the system returned.
- **Threaded builds.** On systems that provide `getgrgid_r`, perl transparently calls the reentrant variant. Callers see no difference.
- **Taint.** Under `-T`, `$passwd` and `$members` come back tainted - the first because it may encode credentials, the second because administrators can change it. `$name` and `$gid` are not tainted.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `getgrnam` - the inverse lookup: group record by name rather than by `gid`
- `getgrent` - iterate every group entry when you need to scan rather than look up
- `setgrent` and `endgrent` - reset and release the iteration cursor used by `getgrent`
- `getpwuid` - the passwd-side counterpart; pair with `getgrgid` to render owner/group columns
- `User::grent` - by-name accessor object that overrides `getgrgid` when you prefer `$gr->name` over positional unpacking

User and group info

getgrnam

Look up a Unix group by name and return its `/etc/group` record.

`getgrnam` wraps the C library's `getgrnam(3)`. Given a group name like `"wheel"` or `"staff"`, it returns the matching record from the system's group database - the name, the encrypted password placeholder, the numeric GID, and the list of member login names. Use it when you have a group name in hand and need the GID or the membership list; use `getgrgid` for the inverse lookup.

Synopsis

```
my ($name, $passwd, $gid, $members) = getgrnam NAME
my $gid = getgrnam NAME
```

What you get back

List context yields a four-element list:

```
my ($name, $passwd, $gid, $members) = getgrnam "wheel";
```

Index	Field	Meaning
0	`$name`	canonical group name (same as `NAME` on hit)
1	`$passwd`	group password field, typically `"x"` or empty
2	`$gid`	numeric group ID
3	`$members`	space-separated string of member login names

`$members` is a single scalar, **not** an array. Split it yourself when you want a list:

```
my @logins = split /\s+/, (getgrnam("wheel"))[3];
```

Scalar context yields the numeric GID (the "other thing", since the lookup was by name):

```
my $gid = getgrnam "wheel";
```

No such group: list context returns the empty list; scalar context returns `undef`. Test scalar returns for definedness rather than truth - GID `0` (`root`) is a valid hit that is also false:

```
defined(my $gid = getgrnam $group) or die "no group '$group'";
```

Global state it touches

None directly. `getgrnam` is a stateless lookup - unlike `getgrent`, it does not advance the group-database iterator, and `setgrent` / `endgrent` do not affect it. It does, however, share the underlying C library's static result buffer with other `getgr*` calls on platforms without thread-safe variants; copy the fields you care about before calling another `getgr*` function.

Examples

Resolve a group name to its GID:

```
my $gid = getgrnam "wheel";
defined $gid or die "no 'wheel' group on this system";
print "wheel GID = $gid\n";
```

List the members of a group:

```
my @fields = getgrnam "staff"
    or die "no 'staff' group";
my @members = split /\s+/, $fields[3];
print "staff has ", scalar(@members), " members: @members\n";
```

Use a group name to set the effective GID (requires privilege):

```
my $gid = getgrnam "video"
    // die "no 'video' group";
$) = "$gid $gid";                   # effective + supplementary
```

Check whether a user is listed as a group member. Note that the primary group (the user's GID in `/etc/passwd`) is **not** in `$members` - check both:

```
sub in_group {
    my ($user, $group) = @_;
    my @g = getgrnam $group or return 0;
    return 1 if grep { $_ eq $user } split /\s+/, $g[3];
    my @u = getpwnam $user or return 0;
    return $u[3] == $g[2];          # primary group match
}
```

Object-oriented form via `User::grent`, which overrides the built-in with a by-name record:

```
use User::grent;
my $g = getgrnam "wheel";
printf "wheel: gid=%d members=%s\n", $g->gid, "@{$g->members}";
```

Edge cases

- **Missing group**: list context returns `()`, scalar returns `undef`. A bare `if (getgrnam $g)` is fine in list context but wrong in scalar context when the GID could be `0`:

  ```
  if (defined(my $gid = getgrnam $g)) { ... }   # correct
  if (my $gid = getgrnam $g)           { ... }  # wrong for GID 0
  ```

- **Members is a string, not a list**. Forgetting this and writing `scalar @members = getgrnam $g` (or indexing `$members[0]`) silently gives wrong results. Always `split /\s+/` the fourth field.
- **Primary-group members are absent from the list**. A user whose `/etc/passwd` GID matches the group is a member in the kernel's eyes but will not appear in `$members`. Lookups that classify users by group must consult `getpwnam` too.
- **Password field is not a password**. On every modern Linux the second field is `"x"` (shadow redirection) or empty. It is never the plaintext or hash of a group password. Do not rely on its format.
- **Case sensitivity**: group names are case-sensitive on Linux. `getgrnam "Wheel"` does not find `wheel`.
- **Name with embedded whitespace or colon**: the group database format does not allow either, so such names cannot exist; the lookup simply fails.
- **NSS backends**: on systems configured with LDAP, SSSD, or other name-service switch modules, `getgrnam` consults those as well as `/etc/group`. Results therefore depend on `/etc/nsswitch.conf` and the configured backends, not on `/etc/group` alone.
- **Taint**: the fields come from outside the program, so under `-T` they are tainted and must be laundered before use in sensitive operations.
- **Thread safety**: pperl automatically substitutes the thread-safe `getgrnam_r` when the platform provides it. On platforms without it, concurrent `getgr*` calls across

threads can clobber each other's results.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `getgrgid` - inverse lookup by numeric GID; reach for this when you have the GID and want the group name or member list
- `getgrent` - iterate every group in the database when you do not know the name or GID in advance
- `setgrent` - rewind the `getgrent` iterator to the start of the group database
- `endgrent` - close the group-database file descriptor opened by iteration
- `getpwnam` - the parallel call for the user database; use alongside `getgrnam` when you need to resolve a user's primary group
- `User::grent` - object-oriented wrapper that overrides `getgrnam` to return a record object with named accessors (`name`, `passwd`, `gid`, `members`) instead of a positional list

Network info

gethostbyaddr

Look up a host record by its packed IP address.

`gethostbyaddr` performs a reverse DNS lookup: given a packed binary address (four bytes for IPv4, sixteen for IPv6) and the matching address family, it calls the system's C `gethostbyaddr(3)` routine and returns the host record. The typical use case - *"whose machine is this packet from?"* - wants the scalar-context form: a single hostname. The list-context form exposes the full `struct hostent`: canonical name, aliases, address family, address length, and every address the resolver knows for that host.

Synopsis

```
my $name = gethostbyaddr $packed_addr, $addrtype;
my ($name, $aliases, $addrtype, $length, @addrs) = gethostbyaddr
 ↪$packed_addr, $addrtype;
```

What you get back

Scalar context - the canonical hostname as a plain string, or `undef` if the reverse lookup fails.

List context - a five-element list mirroring the C `struct hostent`:

- `$name` - the canonical hostname the resolver mapped the address to.
- `$aliases` - a space-separated string of alternate names. Single scalar, not a list.

- `$addrtype` - the address family of the returned addresses, typically `AF_INET` (2) or `AF_INET6` (10).
- `$length` - the length of each address in bytes (4 for IPv4, 16 for IPv6).
- `@addrs` - every address returned for the host, each a packed binary string of `$length` bytes. Often just one, but multi-homed hosts can yield several.

On lookup failure in list context, the return is the empty list.

Global state it touches

- `$?` - set to the C `h_errno` value on failure, when the platform exposes it. Test with `$? != 0` to distinguish `HOST_NOT_FOUND` from `TRY_AGAIN` and similar transient errors.
- `$!` - not meaningfully set by DNS failures. Reverse lookups fail via `h_errno`, not `errno`; do not check `$!` after a failed call.
- The host database iteration cursor shared with `gethostent`, `sethostent`, and `endhostent`. A call to `gethostbyaddr` may reset a cursor left open by `gethostent`.

Examples

Scalar form - map a dotted-quad to a hostname:

```
use Socket;
my $packed = inet_aton("127.0.0.1");
my $name   = gethostbyaddr($packed, AF_INET);
print defined $name ? "$name\n" : "no PTR record\n";
```

Round-trip - forward lookup, then reverse the first address:

```
use Socket;
my $packed = gethostbyname("www.perl.org");
my $name   = gethostbyaddr($packed, AF_INET);
print "$name\n";
```

Identify the peer on an accepted socket connection:

```
use Socket;
my $peer = getpeername($sock);
my ($port, $iaddr) = sockaddr_in($peer);
my $host = gethostbyaddr($iaddr, AF_INET);
printf "connection from %s:%d\n", $host // inet_ntoa($iaddr), $port;
```

List form - every name and address the resolver knows for the host:

```
use Socket;
my $packed = inet_aton("8.8.8.8");
my ($name, $aliases, $type, $len, @addrs) = gethostbyaddr($packed,␣
↪AF_INET);
```

(continues on next page)

(continued from previous page)

```
print "canonical: $name\n";
print "aliases:   $_\n" for split ' ', $aliases;
print "address:   ", inet_ntoa($_), "\n" for @addrs;
```

By-name interface via the `Net::hostent` override - accessor methods instead of positional unpacking:

```
use Net::hostent;
use Socket;
my $h = gethostbyaddr(inet_aton("127.0.0.1"), AF_INET);
print $h->name, "\n" if $h;
```

Edge cases

- **`ADDR` is a packed binary string, not a dotted-quad.** Passing `"127.0.0.1"` directly does not work - Perl will call the resolver with nine raw bytes and get nothing useful back. Always go through `Socket::inet_aton`, `Socket::inet_pton`, or an equivalent packer first.
- **`ADDRTYPE` must match the address width.** `AF_INET` wants a four-byte `ADDR`; `AF_INET6` wants sixteen. A mismatch either returns `undef` or, on some platforms, reads past the buffer. Use the constant from `Socket`, not a hand-typed integer.
- **Check for `undef` on the scalar form.** A missing PTR record returns `undef`, not an empty string. Many addresses on the public internet have no reverse mapping at all.
- **`@addrs` may not include the address you queried.** The resolver returns the full record for the canonical host, which may contain additional addresses that happen to share the hostname.
- **Scalar context runs the full resolver call anyway.** The C routine always populates the whole `hostent`; Perl just discards the list. There is no performance advantage to the scalar form beyond clearer code.
- **Reverse of a forward lookup is not guaranteed to round-trip.** `gethostbyaddr(gethostbyname($h), AF_INET)` may return a different name from `$h` - reverse DNS maps to the canonical hostname, which often differs from the alias the forward query used.
- **IPv6 support is platform-dependent.** Not every libc's `gethostbyaddr(3)` handles `AF_INET6` cleanly. For portable dual-stack reverse resolution use `Socket::getnameinfo`.
- **Thread safety.** The underlying C routine keeps per-process static state for the host cursor. Concurrent calls from multiple threads may interleave results. Scripts that need thread-safe reverse resolution should use `Socket::getnameinfo`.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `gethostbyname` - the forward lookup: hostname → packed address
- `gethostent` - iterate every entry in the host database instead of looking up one address
- `sethostent` - rewind the host iterator and control whether the file stays open across lookups
- `Socket::inet_aton` - build the packed `ADDR` argument from a dotted-quad string
- `Socket::inet_ntoa` - render a packed address from `@addrs` back to dotted-quad form
- `Socket::getnameinfo` - modern, protocol-agnostic reverse resolver that handles IPv6 and is thread-safe

Network info

gethostbyname

Look up a host record by DNS name.

`gethostbyname` resolves a hostname to its IP address(es) and canonical record by calling the system's C `gethostbyname(3)` routine. The typical use case - *"give me the packed IP of this host"* - wants the scalar-context form and nothing more. The list-context form exposes the full host record: canonical name, aliases, address family, address length, and every address on file.

Synopsis

```
my $packed_ip = gethostbyname $name;
my ($name, $aliases, $addrtype, $length, @addrs) = gethostbyname
 ↪$name;
```

What you get back

Scalar context - the first IP address as a packed binary string (four bytes for IPv4, sixteen for IPv6), or `undef` if the lookup fails. Convert to a dotted-quad with `Socket::inet_ntoa`:

```
use Socket;
my $packed = gethostbyname("www.perl.org");
my $dotted = inet_ntoa($packed) if defined $packed;
```

List context - a five-element list mirroring the C `struct hostent`:

- `$name` - the canonical hostname.
- `$aliases` - a space-separated string of alternate names.

- `$addrtype` - the address family, typically `AF_INET` (2) or `AF_INET6` (10).
- `$length` - the length of each address in bytes (4 for IPv4, 16 for IPv6).
- `@addrs` - every address returned for the name, each a packed binary string of `$length` bytes.

On lookup failure in list context, the return is the empty list.

Global state it touches

- `$?` - set to the C `h_errno` value on failure, when the platform exposes it. Test with `$? != 0` to distinguish "no such host" (`HOST_NOT_FOUND`) from "try again" (`TRY_AGAIN`) and similar.
- The host database iteration cursor shared with `gethostent`, `sethostent`, and `endhostent`. A call to `gethostbyname` may reset a cursor left open by `gethostent`.

Examples

Scalar form - the common case:

```
use Socket;
my $packed = gethostbyname("localhost");
defined $packed or die "cannot resolve localhost: $!";
print inet_ntoa($packed), "\n";      # 127.0.0.1
```

List form - enumerate every address for a multi-homed host:

```
use Socket;
my ($name, $aliases, $type, $len, @addrs) = gethostbyname("www.
↪example.com");
for my $addr (@addrs) {
    print inet_ntoa($addr), "\n";
}
```

Unpack an IPv4 address by hand, without `Socket`:

```
my $packed = gethostbyname("127.0.0.1");
my ($a, $b, $c, $d) = unpack("W4", $packed);
print "$a.$b.$c.$d\n";               # 127.0.0.1
```

Forward then reverse - resolve a name, then look the address back up:

```
use Socket;
my $packed = gethostbyname("www.perl.org");
my $back   = gethostbyaddr($packed, AF_INET);
print "$back\n";
```

By-name interface via the `Net::hostent` override - accessor methods instead of positional unpacking:

```
use Net::hostent;
my $h = gethostbyname("www.perl.org");
print $h->name, " ", join(",", @{$h->aliases}), "\n";
```

Edge cases

- **Always call in scalar context when you want "the IP".** The list-context form is never what a quick resolver wants and allocates four extra return values. The upstream docs flag this explicitly: *"Make sure* `gethostbyname` *is called in SCALAR context and that its return value is checked for definedness."*
- **Check for `undef` on the scalar form.** A failed lookup returns `undef`, not an empty string. A literal `0.0.0.0` address, by contrast, is a valid four-byte packed string and stringifies to the zero byte sequence, not to `undef`.
- **`@addrs` may be empty** in list context even when the first four return values are defined - some resolvers produce a record with no addresses. Always iterate, don't index `[0]` blindly.
- **Aliases are one scalar, not a list.** `$aliases` is a single space-separated string. Split on whitespace if you need a list.
- **Numeric IP strings are accepted.** Passing `"127.0.0.1"` returns the packed form of that address without a DNS round-trip - the resolver short-circuits numeric input.
- **IPv6.** `gethostbyname` queries only the IPv4 (`AF_INET`) address family. For dual-stack resolution use `Socket::getaddrinfo`, which returns records for every address family the host offers.
- **Thread safety.** The underlying C routine keeps per-process static state for the host cursor. Concurrent calls from multiple threads may interleave results. Scripts that need thread-safe resolution should use `Socket::getaddrinfo`.
- **`h_errno` vs `$!`.** DNS failures set `h_errno` (exposed via `$?`), not `errno` / `$!`. Checking `$!` after a failed `gethostbyname` is meaningless.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `gethostbyaddr` - the reverse lookup: packed address → host record
- `gethostent` - iterate every entry in the host database instead of looking up one name
- `sethostent` - rewind the host iterator and control whether the file stays open across lookups
- `Socket::inet_ntoa` - turn the packed scalar-context result into a dotted-quad string

- `Socket::inet_aton` - the pure name-to-packed helper when you don't need the full host record
- `unpack` - decode the packed address manually when `Socket` is not available

Network info

gethostent

Fetch the next entry from the host database.

`gethostent` walks the host database - on traditional Unix systems the `/etc/hosts` file, on modern systems whatever NSS source is configured - returning one record per call until the database is exhausted. Use it when you want to enumerate every known host, not when you want to resolve a single name: `gethostbyname` is the right tool for a targeted lookup.

The iteration cursor is shared with `sethostent` (which rewinds it) and `endhostent` (which closes it). Between those two calls `gethostent` yields each record in order, then returns the empty list (or `undef` in scalar context) once the end of the database is reached.

Synopsis

```
my ($name, $aliases, $addrtype, $length, @addrs) = gethostent;
my $name = gethostent;
```

What you get back

List context - a five-element list mirroring the C `struct hostent`:

- `$name` - the canonical hostname for this entry.
- `$aliases` - a space-separated string of alternate names.
- `$addrtype` - the address family, typically `AF_INET` (2) or `AF_INET6` (10).
- `$length` - the length in bytes of each address (4 for IPv4, 16 for IPv6).
- `@addrs` - every address recorded for this host, each a packed binary string of `$length` bytes.

At end-of-database the return is the empty list. A distinction worth keeping in mind: the upstream perlfunc note that a missing entry may return *"a single meaningless true value"* applies to the by-name and by-address lookups, not to `gethostent` - iteration past the end is a clean empty list.

Scalar context - the canonical `$name` only, or `undef` at end-of-database. Use list context whenever you need anything beyond the name; reconstructing the other fields is not possible after the fact.

Convert a packed address from `@addrs` with `Socket::inet_ntoa` for IPv4 or `unpack` directly:

```
use Socket;
print inet_ntoa($addrs[0]), "\n";
```

Global state it touches

- The **host database cursor**, shared with `sethostent`, `endhostent`, `gethostbyname`, and `gethostbyaddr`. A by-name or by-address lookup may reset or invalidate a cursor left mid-iteration by `gethostent`; don't interleave the two styles without a fresh `sethostent` between them.
- `$?` - set to the C `h_errno` value on failure, where the platform exposes it. A plain end-of-database is not a failure and leaves `$?` alone.
- `$!` - `gethostent` is a database iterator, not a syscall wrapper in the `open`/`read` sense. DNS and NSS errors surface through `h_errno` (via `$?`), not through `$!`.

Examples

Enumerate every host in the database:

```
use Socket;
sethostent(1);
while (my ($name, $aliases, $type, $len, @addrs) = gethostent) {
    my @dotted = map { inet_ntoa($_) } @addrs;
    print "$name  @dotted\n";
}
endhostent;
```

Scalar context - list just the canonical names:

```
sethostent(0);
while (defined(my $name = gethostent)) {
    print "$name\n";
}
endhostent;
```

Build a name-to-address hash in one pass, then look up entries locally without further DNS calls:

```
use Socket;
my %ip_of;
sethostent(1);
while (my ($name, $aliases, undef, undef, @addrs) = gethostent) {
    next unless @addrs;
    $ip_of{$name} = inet_ntoa($addrs[0]);
    $ip_of{$_}    = inet_ntoa($addrs[0]) for split ' ', $aliases;
}
endhostent;

print $ip_of{"localhost"} // "not in hosts DB", "\n";
```

By-record interface via the `Net::hostent` override - accessor methods instead of positional unpacking:

```
use Net::hostent;
while (my $h = gethostent) {
    printf "%-30s %s\n", $h->name, join(",", @{$h->aliases});
}
```

Edge cases

- **No implicit rewind.** `gethostent` does not open or rewind the database on its own. First call after program start works because the C library lazily opens the database, but once the end is reached further calls keep returning end-of-database until a `sethostent` rewinds the cursor. Always pair enumeration with an explicit `sethostent` / `endhostent`.
- **`STAYOPEN` argument to `sethostent`.** Passing a true value asks the library to keep the database file open across lookups, which matters for NSS sources that otherwise reconnect on every call. Passing `0` permits the library to close and reopen between calls.
- **End-of-database is the empty list, not `undef`.** In list context the loop condition `while (my @r = gethostent)` works because the empty list is false. In scalar context use `while (defined(my $name = gethostent))` - a hostname of `"0"` is theoretically valid and would falsely terminate a truthiness-based loop.
- **`@addrs` may be empty** even for a returned record on some NSS backends. Guard the address-unpacking step:

  ```
  next unless @addrs;
  ```

- **Aliases are one scalar, not a list.** `$aliases` is a single space-separated string. Split on whitespace if you want a list.
- **NSS source mix.** On glibc-based systems the hosts database may draw from `/etc/hosts`, DNS, mDNS, and other NSS modules depending on `nsswitch.conf`. DNS-backed NSS modules typically do **not** enumerate - `gethostent` against a DNS-only hosts database returns the empty list immediately. On most real deployments only the `files` source contributes entries to iteration.
- **Interleaving with by-name / by-address lookups.** A call to `gethostbyname` or `gethostbyaddr` can reset the shared cursor. Finish the iteration first, or restart it with `sethostent` after the lookup.
- **Thread safety.** The underlying C routine keeps per-process static state for the iteration cursor. Concurrent calls from multiple threads interleave results unpredictably. Confine enumeration to a single thread.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `sethostent` - rewind the iterator and control whether the database stays open across calls
- `endhostent` - close the iterator and release the database handle
- `gethostbyname` - targeted lookup by name instead of full enumeration
- `gethostbyaddr` - targeted reverse lookup by packed address
- `getnetent` - same iteration pattern for the networks database
- `Socket::inet_ntoa` - turn a packed address from `@addrs` into a dotted-quad string

User and group info

getlogin

Return the login name of the user associated with the controlling terminal.

`getlogin` wraps the C library function of the same name. On Linux it consults the kernel's record of who owns the current controlling terminal - traditionally recorded in `/var/run/utmp` - and returns that name as a string. It answers the question *"who logged in on this tty?"*, which is **not** the same question as *"who is running this process?"*. A process that has changed its real or effective UID, or that has no controlling terminal at all (a daemon, a cron job, a container entrypoint), will usually see `getlogin` return the empty string or `undef`.

Synopsis

```
my $name = getlogin;
```

What you get back

A string - the login name recorded for the controlling terminal - or the empty string / `undef` when no such record is available. Treat both empty-string and undefined as "unknown" and fall back to a UID-based lookup:

```
my $login = getlogin || getpwuid($<) || "Kilroy";
```

Using `||` here is deliberate: the empty string and `undef` both mean "no answer" and both should trigger the fallback.

Global state it touches

- Reads `$<` indirectly only via the idiomatic `getpwuid` fallback above; `getlogin` itself takes no arguments and consults no Perl variables.
- Sets `$!` on the underlying system-call failure paths exposed by the C library.

Examples

The canonical who-is-this-user pattern, robust against detached sessions and UID changes:

```
my $login = getlogin || getpwuid($<) || "Kilroy";
print "hello, $login\n";
```

Bare call in a shell-attached script:

```
my $name = getlogin;
print defined $name && length $name
    ? "tty owner: $name\n"
    : "no controlling tty\n";
```

Contrast with `getpwuid`, which answers a different question - *"who is the effective user running this code?"*:

```
my $tty_user  = getlogin           // "";
my $proc_user = getpwuid($<);
print "tty=$tty_user proc=$proc_user\n";
# After `sudo -u bob script.pl`, tty_user is your login,
# proc_user is "bob".
```

Inside a daemon or a cron job there is no controlling terminal, so `getlogin` has nothing to report:

```
# running from cron
my $name = getlogin;
# $name is "" or undef; always pair with a getpwuid fallback
```

Edge cases

- **No controlling terminal**: daemons, systemd services, cron jobs, container entry-points, and anything started without a tty see the empty string or `undef`. This is the common case outside interactive shells - always provide a fallback.
- **sudo and su**: `getlogin` returns the **original** login name (the tty owner), not the target user. After `sudo -u bob script.pl` you get your own login, while `getpwuid` with `$<` returns bob. Pick the one that matches the question you are actually asking.
- **Not an authentication primitive**: the record `getlogin` reads can be forged or stale; a process that inherits a tty from an unrelated session can see a name that has nothing to do with its own credentials. Never gate privileged behaviour on the return value. Use `getpwuid` on `$<` / `$>` instead.
- **Empty string vs `undef`**: platforms differ on which they return when there is no record. The `||` fallback handles both; a `defined`-only check does not.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `getpwuid` - resolves `$<` or `$>` to a name, and is the correct building block for *"who is running this process?"*
- `getpwnam` - the reverse lookup when you already have a login name and need the passwd entry
- `getpwent` - iterate the whole passwd database when you need more than one entry
- `$<` - real UID; the natural argument to `getpwuid` in the `getlogin` fallback idiom
- `$>` - effective UID; use this instead of `$<` when you care about the identity under which privileged operations will actually run

Network info

getnetbyaddr

Look up a network record by its numeric network address.

`getnetbyaddr` translates a numeric network number and address family into a network database entry by calling the system's C `getnetbyaddr(3)` routine. It is the reverse of `getnetbyname` and consults the same system network database (typically `/etc/networks` or an NSS backend). In scalar context it returns just the canonical network name; in list context it returns the full record.

Synopsis

```
my $name = getnetbyaddr $addr, $addrtype;
my ($name, $aliases, $addrtype, $net) = getnetbyaddr $addr,
 ↪$addrtype;
```

What you get back

Scalar context - the canonical name of the network, or `undef` if no entry matches.

List context - a four-element list mirroring the C `struct netent`:

- `$name` - the canonical network name.
- `$aliases` - a space-separated string of alternate names.
- `$addrtype` - the address family of the record, typically `AF_INET` (2).
- `$net` - the network number as an unsigned integer, in host byte order.

On lookup failure in list context, the return is the empty list. If the entry exists but is empty, the return is a single meaningless true value per the shared `get*` family convention.

Arguments

- ADDR - the network number as an **integer in host byte order**, not a packed string. This is the native-integer form you get back from a prior `getnetbyname` call, or that you build with `Socket::inet_aton` followed by `unpack("N", ...)` to convert from network to host order.
- ADDRTYPE - the address family, almost always `AF_INET` from the `Socket` module.

Global state it touches

- The network database iteration cursor shared with `getnetent`, `setnetent`, and `endnetent`. A call to `getnetbyaddr` may reset a cursor left open by `getnetent`.
- `$!` - the C `errno` value on failure, where the platform sets it. Most `/etc/networks` lookups fail silently with `errno` untouched, so treat `undef` as the primary failure signal.

Examples

Look up the loopback network by its integer number:

```
use Socket;
my $name = getnetbyaddr(127 << 24, AF_INET);
print "$name\n";                    # loopback
```

List context - the whole record:

```
use Socket;
my ($name, $aliases, $type, $net)
    = getnetbyaddr(127 << 24, AF_INET);
printf "%s (net=%08x)\n", $name, $net;
```

Round-trip from a dotted string, via `Socket`:

```
use Socket;
my $packed = inet_aton("127.0.0.0");     # packed, network order
my $net    = unpack("N", $packed);        # integer, host order
my $name   = getnetbyaddr($net, AF_INET);
```

By-name interface via the `Net::netent` override - accessor methods instead of positional unpacking:

```
use Net::netent;
my $n = getnetbyaddr(127 << 24, AF_INET);
print $n->name, "\n" if defined $n;
```

Forward then reverse - resolve a name, then look the address back up:

```
use Socket;
my $net  = (getnetbyname("loopback"))[3];
my $back = getnetbyaddr($net, AF_INET);
print "$back\n";                         # loopback
```

Edge cases

- **Integer, not packed string.** `ADDR` is a plain integer in host byte order. Passing the four-byte packed form that `gethostbyaddr` consumes here will silently look up the wrong number. This is the most common mistake with this function.
- **Classful only, on most systems.** The underlying `/etc/networks` database predates CIDR. Entries are classful network numbers (e.g. `127`, `10`, `192.168`), and the system call is not useful for arbitrary CIDR subnets. For modern subnet reasoning use `Socket` or a CPAN module like `NetAddr::IP`.
- **Empty `/etc/networks`.** Many Linux distributions ship a nearly empty `/etc/networks`; expect most lookups to return `undef` on such a host. This is a configuration fact, not a bug in the call.
- **List-context empty database entry.** Per the shared `get*` family rule, an entry that exists but has no fields returns a single meaningless true value, not the four-element list. In practice no platform produces such an entry for networks; guard with `defined` and element count if you care.
- **`ADDRTYPE` other than `AF_INET`.** The C library accepts the argument but has no defined behavior for non-Internet families here. Pass `AF_INET` unless the target system documents otherwise.
- **Thread safety.** The underlying C routine keeps per-process static state for the network cursor. Concurrent calls from multiple threads may interleave results. Where the platform offers `getnetbyaddr_r`, Perl substitutes it automatically.
- **List-operator precedence.** Like the rest of the `get*` family, `getnetbyaddr` parses as a list operator. `getnetbyaddr $n, AF_INET or die` binds the `or die` to the whole argument list, which is usually what you want; add parens when in doubt.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `getnetbyname` - the forward lookup: canonical or alias name → network record
- `getnetent` - iterate every entry in the network database instead of looking up one address
- `setnetent` - rewind the network iterator and control whether the database stays open across lookups
- `endnetent` - close the network database when done iterating
- `gethostbyaddr` - the host-level analogue; takes a packed address, not an integer
- `Net::netent` - by-name wrapper that returns an object with `name` / `aliases` / `addrtype` / `net` accessors

Network info

getnetbyname

Look up a network record by name.

`getnetbyname` resolves a symbolic network name (as listed in the system's networks database, typically `/etc/networks`) to its numeric network address by calling the C `getnetbyname(3)` routine. The scalar-context form returns the network number; the list-context form exposes the full record: canonical name, aliases, address family, and network number.

Synopsis

```
my $net = getnetbyname $name;
my ($name, $aliases, $addrtype, $net) = getnetbyname $name;
```

What you get back

Scalar context - the network number as a plain integer, or `undef` if the name is not in the networks database. This is the host-byte-order network portion of an IPv4 address, not a packed string. For the class-C network `192.168.0`, the returned value is `12625920` (that is, `0xC0A800`).

List context - a four-element list mirroring the C `struct netent`:

- `$name` - the canonical network name.
- `$aliases` - a space-separated string of alternate names.
- `$addrtype` - the address family, typically `AF_INET` (2).
- `$net` - the network number, same integer as the scalar-context form.

On lookup failure in list context, the return is the empty list.

Global state it touches

- The networks database iteration cursor shared with `getnetent`, `setnetent`, and `endnetent`. A call to `getnetbyname` may reset a cursor left open by `getnetent`.

Examples

Scalar form - look up a named network and print it as dotted-quad:

```
my $net = getnetbyname("loopback");
defined $net or die "no networks entry for loopback";
printf "%d.%d.%d.%d\n",
    ($net >> 24) & 0xFF, ($net >> 16) & 0xFF,
    ($net >> 8)  & 0xFF,  $net        & 0xFF;   # 127.0.0.0
```

List form - every field of the record:

```
my ($name, $aliases, $type, $net) = getnetbyname("loopback");
print "name=$name aliases=$aliases type=$type net=$net\n";
```

Round-trip against `getnetbyaddr` - name to number, then number back to name:

```
use Socket qw(AF_INET);
my $net  = getnetbyname("loopback");
my $back = getnetbyaddr($net, AF_INET);
print "$back\n";                          # loopback
```

By-name interface via the `Net::netent` override - accessor methods instead of positional unpacking:

```
use Net::netent;
my $n = getnetbyname("loopback");
print $n->name, "  ", $n->net, "\n";
```

Guard against a missing entry - a name that is not in the database returns `undef`, not zero:

```
my $net = getnetbyname("no-such-network");
if (defined $net) {
    print "found: $net\n";
} else {
    print "not in /etc/networks\n";
}
```

Edge cases

- **Check for `undef`, not falsehood.** A network number of `0` is a legal (if unusual) record; treating the return value as a boolean will misread that case as failure. Always test with `defined`.
- **The returned number is host-byte-order, not a packed string.** Unlike `gethostbyname`, which returns a four-byte packed address, `getnetbyname` returns a plain integer. Do not pass it to `Socket::inet_ntoa` - that function expects a packed string and will produce garbage on an integer.
- **Aliases are one scalar, not a list.** `$aliases` in the list form is a single space-separated string. Split on whitespace if you need a list.
- **Networks are typically class-boundary numbers.** The entries in `/etc/networks` are traditionally stored as the network portion only - `127` for loopback, `10` for the private class-A, not full four-octet addresses. On modern CIDR-routed networks the utility of this database is limited; most deployments keep only a handful of legacy entries.
- **No DNS fallback.** Unlike `gethostbyname`, this call consults only the local networks database. There is no equivalent of DNS for network names.
- **Empty list vs. undef.** Scalar context returns `undef` on failure; list context returns the empty list. The two are not interchangeable - code written for one context will misbehave in the other.
- **Thread safety.** The underlying C routine keeps per-process static state for the networks cursor. Concurrent calls from multiple threads may interleave results.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `getnetbyaddr` - the reverse lookup: network number → network record
- `getnetent` - iterate every entry in the networks database instead of looking up one name
- `setnetent` - rewind the networks iterator and control whether the file stays open across lookups
- `endnetent` - close the networks database after an iteration
- `gethostbyname` - the host-level counterpart, for resolving a hostname to an IP address
- `Net::netent` - by-name object wrapper returning accessor methods instead of a positional list

Network info

getnetent

Read the next entry from the networks database.

`getnetent` wraps the C library routine of the same name. It walks the system networks database (usually `/etc/networks`, sometimes augmented by NIS or other resolver backends) one record at a time, returning the next entry on each call. Pair it with `setnetent` to rewind or control the connection, and `endnetent` to close it.

Synopsis

```
my ($name, $aliases, $addrtype, $net) = getnetent;
my $name                              = getnetent;   # scalar␣
↪context
```

What you get back

In list context, a four-element list:

- `$name` - canonical network name (e.g. `"loopback"`).
- `$aliases` - space-separated string of alternative names.
- `$addrtype` - address family, typically `AF_INET` (2).
- `$net` - network number as an integer in host byte order. Convert to dotted form with `Socket::inet_ntoa(pack("N", $net))`.

In scalar context, just `$name`.

When the iterator is exhausted (no more records), list context returns the empty list and scalar context returns `undef`. A missing entry during a direct lookup returns a single

meaningless true value per perlfunc convention, but since `getnetent` takes no key, the only terminal condition is "end of database."

Global state it touches

`getnetent` reads from a **process-wide iterator** maintained by the C library. Every call advances that iterator; there is no per-scope state, and concurrent iteration from two places in the same process will interleave records. `setnetent` rewinds it and optionally asks the library to keep the backing connection open for the duration of iteration; `endnetent` closes it.

Examples

Walk the entire networks database:

```
setnetent(1);
while (my ($name, $aliases, $addrtype, $net) = getnetent) {
    printf "%-20s %s\n", $name, join(" ", split / /, $aliases);
}
endnetent();
```

Scalar context gives just the name - useful for a quick enumeration:

```
setnetent(1);
while (defined(my $name = getnetent)) {
    print "$name\n";
}
endnetent();
```

Decode the network number into dotted form:

```
use Socket;
setnetent(1);
while (my ($name, undef, undef, $net) = getnetent) {
    printf "%-20s %s\n", $name, inet_ntoa(pack "N", $net);
}
endnetent();
```

Object-oriented access through `Net::netent` - the by-name interface from the standard library replaces the positional return list with a blessed object:

```
use Net::netent;
while (my $n = getnetent()) {
    printf "%s => %d\n", $n->name, $n->net;
}
```

Edge cases

- **Iterator exhaustion**: list context returns `()`, scalar context returns `undef`. The loop forms above terminate naturally.

- **No `setnetent` before the loop**: the first call implicitly opens the database. Calling `setnetent(1)` is still recommended - without the "stay open" flag, some C libraries reopen the file on every record and performance collapses.
- **Multiple concurrent walks**: the iterator is process-global. Two loops over `getnetent` in the same process will split the records between them. Collect into an array first if both need the full set.
- **Unsupported on some platforms**: Windows has no `/etc/networks` equivalent and the underlying C routine is absent. Attempting to call `getnetent` there raises `The getnetent function is unimplemented`. On Linux, glibc provides the function but the database is often empty - most systems rely on DNS instead, which this interface does not consult.
- **Thread safety**: the C library's iterator is not thread-safe. Perl may transparently substitute `getnetent_r` on systems that provide it, but do not assume this is universal. In a threaded program, serialise access through a single thread.
- **Tainting**: when tainting is enabled, all four return fields are tainted - they come from a file (or a remote resolver) outside the program's control.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `setnetent` - rewind the iterator and optionally keep the backing connection open across calls
- `endnetent` - close the iterator when you're done
- `getnetbyname` - look up one network by name instead of walking the whole database
- `getnetbyaddr` - look up one network by its address
- `gethostent` - the analogous walk over the hosts database, with the same `set`/`end` companion pair
- `Net::netent` - by-name object interface that replaces the positional return list

Sockets

getpeername

Return the address of the remote end of a connected socket.

`getpeername` asks the kernel who is on the other side of `SOCKET` and returns the answer as a packed `sockaddr` byte string - the same shape `connect` took and the same shape `accept` returned. You then hand that byte string to an unpacker from `Socket` (or `Socket::unpack_sockaddr_in`, `Socket::unpack_sockaddr_in6`, `Socket::unpack_sockaddr_un`) to pull out the port, address, or path.

This is the companion to `getsockname`, which returns the **local** end of the same socket.

Synopsis

```
use Socket;
my $peer = getpeername SOCKET;
my ($port, $iaddr) = unpack_sockaddr_in($peer);
```

What you get back

A packed `sockaddr` byte string on success, `undef` on failure (with `$!` set - typically `ENOTCONN` when the socket has no peer, or `EBADF` when `SOCKET` is not an open file descriptor).

The byte string is opaque. Its layout depends on the socket's address family; always decode it with an unpacker that matches the family you expect:

- `AF_INET` → `Socket::unpack_sockaddr_in` returns `($port, $iaddr)` where `$iaddr` is the 4-byte packed IPv4 address.
- `AF_INET6` → `Socket::unpack_sockaddr_in6` returns `($port, $iaddr, $scope_id, $flowinfo)`.
- `AF_UNIX` → `Socket::unpack_sockaddr_un` returns the peer's filesystem path.

If you do not know the family at the call site, ask `Socket::sockaddr_family` first:

```
my $peer   = getpeername $sock;
my $family = sockaddr_family($peer);
```

Global state it touches

`getpeername` reads the file-descriptor state of `SOCKET`. It sets `$!` on failure and does not otherwise touch interpreter globals. It does not interact with `$_`, the selected filehandle, or output separators.

Examples

Look up the remote IPv4 peer of a client-side TCP socket and render it in dotted-quad form:

```
use Socket;
my $peer = getpeername($sock)
    or die "getpeername: $!";
my ($port, $iaddr) = unpack_sockaddr_in($peer);
printf "peer is %s:%d\n", inet_ntoa($iaddr), $port;
```

Server side: every socket returned by `accept` is already connected, and `accept` itself hands back the peer address. `getpeername` is useful later, when that packed address has not been kept around:

```
while (my $client = accept(my $conn, $listen)) {
    handle($conn);
}
```

(continues on next page)

(continued from previous page)

```
sub handle {
    my ($conn) = @_;
    my $peer = getpeername($conn) or return;
    my ($port, $iaddr) = unpack_sockaddr_in($peer);
    warn "request from ", inet_ntoa($iaddr), ":$port\n";
    # ...
}
```

Resolve the peer back to a hostname - note that this issues a blocking DNS lookup:

```
use Socket;
my $peer = getpeername($sock);
my ($port, $iaddr) = unpack_sockaddr_in($peer);
my $name = gethostbyaddr($iaddr, AF_INET);
print "connected to ", $name // inet_ntoa($iaddr), "\n";
```

Mixed IPv4 / IPv6 code - decode generically via the address family:

```
use Socket qw(sockaddr_family unpack_sockaddr_in unpack_sockaddr_in6
              AF_INET AF_INET6 inet_ntop);

my $peer = getpeername($sock) or die "getpeername: $!";
my $fam  = sockaddr_family($peer);

if ($fam == AF_INET) {
    my ($port, $iaddr) = unpack_sockaddr_in($peer);
    printf "v4 %s:%d\n", inet_ntop(AF_INET, $iaddr), $port;
}
elsif ($fam == AF_INET6) {
    my ($port, $iaddr) = unpack_sockaddr_in6($peer);
    printf "v6 [%s]:%d\n", inet_ntop(AF_INET6, $iaddr), $port;
}
```

Probing whether a socket is connected at all - the `ENOTCONN` path is the well-defined way to ask:

```
if (defined getpeername($sock)) {
    # connected
} else {
    # $! == ENOTCONN for an unconnected socket
}
```

Edge cases

- **Unconnected socket**: a socket that has been `socket`'d but not yet `connect`'d or `accept`'d has no peer. `getpeername` returns `undef` and sets `$!` to `ENOTCONN`.
- **Non-socket filehandle**: passing a plain file or pipe yields `undef` with `$!` set to `ENOTSOCK`.

- **Closed or invalid filehandle**: returns `undef` with `$!` set to `EBADF`. Under `use warnings` a `getpeername() on closed socket` warning is emitted.
- **Datagram sockets**: `getpeername` only returns an address when the `SOCK_DGRAM` socket has been `connect`'d to a default peer; otherwise it fails with `ENOTCONN` even though the socket is usable via `recv` / `send`.
- **`AF_UNIX` with an unbound peer**: a Unix-domain socket whose peer never called `bind` returns a packed `sockaddr_un` with an empty path. `Socket::unpack_sockaddr_un` gives back the empty string.
- **After `shutdown`**: `getpeername` still succeeds as long as the descriptor is open; shutdown tears down data flow, not the kernel's record of the peer.
- **SOCKET is a parsed filehandle token**: like other socket built-ins, the argument is parsed as a filehandle, not evaluated as a general expression. `getpeername $handles[0]` is a syntax error; use a scalar holding the handle, or `*{...}` dereferencing.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `getsockname` - mirror call that returns the **local** end of the same socket; pairs with `getpeername` when logging both ends of a connection
- `accept` - already hands you the peer address for a freshly accepted server-side socket; use `getpeername` later when that address was not retained
- `connect` - the call that establishes the peer in the first place; the address you pass to `connect` is what `getpeername` will hand back afterwards
- `Socket` - provides the unpackers (`unpack_sockaddr_in`, `unpack_sockaddr_in6`, `unpack_sockaddr_un`) and `sockaddr_family` needed to interpret the returned byte string
- `gethostbyaddr` - turns the packed IPv4 address from the unpacker into a hostname
- `$!` - carries the reason on failure (`ENOTCONN`, `ENOTSOCK`, `EBADF`)

Processes

getpgrp

Return the POSIX process-group ID that a process belongs to.

Every Unix process is a member of exactly one process group, which in turn belongs to a session. Process groups are how the kernel addresses related processes as a unit - most visibly, they are the unit that receives a signal from the controlling terminal (`SIGINT` on Ctrl-C, `SIGTSTP` on Ctrl-Z). `getpgrp` asks the kernel for the group ID of a given process; with no argument or a `PID` of `0` it answers for the current process.

Synopsis

```
getpgrp
getpgrp PID
getpgrp 0
```

What you get back

A non-negative integer - the process-group ID in scalar context. Process-group IDs share the PID numbering space, and the group leader's PID equals the group ID, so the value returned by `getpgrp 0` is the PID of the current process's group leader.

On a system that does not implement `getpgrp(2)` at all, the call raises a fatal exception. That is vanishingly rare on modern Unix; Linux, the BSDs, macOS, and Solaris all support it.

Global state it touches

None. `getpgrp` reads kernel state for the named process and sets `$!` only on failure (which, on a POSIX system, means "no such process" for a non-zero `PID` that does not exist).

Examples

Get the current process's own group:

```
my $pgid = getpgrp;
print "my group is $pgid\n";
```

`PID` of `0` means "myself" and is the portable form - some very old systems do not accept any other value:

```
my $pgid = getpgrp 0;                  # same as getpgrp with no args
```

Detect whether the current process is its own group leader - the usual test a daemon runs before calling `setpgrp` or `POSIX::setsid` to detach from its controlling terminal:

```
if (getpgrp(0) == $$) {
    print "already a group leader\n";
}
else {
    print "group leader is ", getpgrp(0), "\n";
}
```

Look up another process's group, e.g. a child just returned by `fork`:

```
my $pid = fork // die "fork: $!";
if ($pid == 0) {
    # child
    exec @cmd;
```

(continues on next page)

(continued from previous page)

```
}
print "child $pid is in group ", getpgrp($pid), "\n";
```

Send a signal to every member of a process group. `kill` with a negative PID targets the group, and `getpgrp` is how you discover the group ID to negate:

```
my $grp = getpgrp($child);
kill 'TERM', -$grp;                    # SIGTERM to the whole group
```

Edge cases

- **Omitted argument equals `PID` of `0`**: `getpgrp` and `getpgrp 0` both report the current process's group. Prefer the explicit `0` form when portability to historical systems matters.
- **Non-existent `PID`**: returns `undef` and sets `$!` to `"No such process"` (`ESRCH`). Check `$!`, not the return value's truthiness - a legitimate group ID of `0` is falsy in Perl but is not an error condition (it is, on Linux, the idle/swapper group).
- **Not-permitted lookups**: on systems that restrict cross-session lookups, querying a `PID` in a different session can fail with `$!` set to `"Operation not permitted"` (`EPERM`). This is unusual on Linux, where any process may query any `PID`'s group.
- **`getpgrp` vs. `getpgrp PID`**: the BSD-style one-argument form was not portable to all Unixes historically; POSIX standardised both. On the platforms pperl runs on (Linux, all cranelift targets), both forms work.
- **Signed vs. unsigned**: the kernel's PID type is signed, so the returned value fits comfortably in a Perl integer. A return of `-1` is not used for success on any platform pperl targets; treat `-1` as the error sentinel only if you are porting code from a C codebase that checked it that way.
- **Not a shell alias for job control**: the shell's `jobs` and `%` notation operate on job tables the shell keeps, not on kernel process groups directly. `getpgrp` tells you the kernel truth, which can drift from a subshell's idea of the job tree.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `setpgrp` - put a process into a new or existing group; the write counterpart to `getpgrp`
- `getppid` - parent process ID; often wanted alongside the group ID when walking process relationships
- `getpriority` - read scheduling priority, which also accepts a process-group selector (`PRIO_PGRP`)
- `fork` - the child inherits its parent's process group, so `getpgrp` on a fresh child reports the parent's group until a `setpgrp` call changes it

- `kill -kill SIGNAL, -$pgid` signals every member of the group; `getpgrp` is how you find `$pgid`
- `$$` - current PID; equal to `getpgrp(0)` when the process is its own group leader

Processes

getppid

Return the process ID of the current process's parent.

`getppid` is a thin wrapper over the POSIX `getppid(2)` system call. It takes no arguments and returns an integer - the PID of whatever process `fork`ed (or `exec`ed into) the running program. Combined with `$$` (the current process's own PID), it is the standard way for a child to identify the process that spawned it.

Synopsis

```
my $parent = getppid;
```

What you get back

A positive integer - the PID of the parent process. On Linux this is `1` (init / systemd) once the original parent has exited and the child has been reparented; there is no special "orphaned" return value, the reparenting is transparent. Never returns `undef` or `0` on a supported platform; the underlying syscall is specified to always succeed.

Global state it touches

None. `getppid` reads the kernel's process table for the current process; it does not touch any Perl special variable. Note that `$$` holds *this* process's PID, not the parent's - fetch the parent with `getppid` explicitly.

Examples

Identify the parent that launched this script:

```
printf "my pid is %d, my parent is %d\n", $$, getppid;
```

Detect orphaning - a common idiom for daemons that want to exit when their controlling process goes away:

```
while (1) {
    last if getppid == 1;       # parent died, we've been reparented
↪to init
    do_work();
    sleep 1;
}
```

Child process reports back to its parent after `fork`:

```
my $pid = fork // die "fork failed: $!";
if ($pid == 0) {
    warn "child $$ spawned by $>, parent is " . getppid . "\n";
    exit 0;
}
waitpid $pid, 0;
```

Compare before and after `fork` to confirm the relationship:

```
my $before = $$;
my $pid = fork // die $!;
if ($pid == 0) {
    die "parent mismatch" unless getppid == $before;
    exit;
}
waitpid $pid, 0;
```

Edge cases

- **No arguments, no parentheses needed**: `getppid` is a niladic built-in. Write `getppid` or `getppid()`; both parse identically. Do not try to pass a PID argument - unlike `getpgrp`, there is no "parent of another process" form.
- **Orphan reparenting is invisible**: if the original parent exits, the kernel reparents the process (to `init`/`systemd` on Linux, or to the nearest ancestor with `PR_SET_CHILD_SUBREAPER` set). `getppid` silently starts returning the new parent's PID. Code that caches the return value at startup and compares later is the standard way to detect this.
- **Race at startup**: calling `getppid` extremely early, before the kernel has finished setting up the process, is not a concern in user Perl code - by the time the interpreter is running, the parent PID is stable.
- **PID wraparound**: PIDs are recycled. A stored parent PID can, over long-running time, refer to a *different* process than the original parent even without reparenting. Use `getppid == 1` (or the equivalent subreaper PID), not equality against a saved value, when the intent is "has my parent died?".
- **Threads**: since Perl 5.16 the Linux LinuxThreads workaround has been removed. `getppid` now always returns what the kernel reports for the current thread group.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `$$` - the current process's own PID; pair with `getppid` to log both ends of the parent/child relationship
- `fork` - spawn a child process; the child's `getppid` returns the forking process's PID
- `waitpid` - reap a specific child by PID; the mirror operation from the parent side

- `wait` - reap any child; same parent-side operation, unspecified PID
- `getpgrp` - process *group* ID rather than parent PID; related but distinct kernel concept
- `kill` - send a signal to a PID, including one obtained from `getppid` (e.g. notifying the parent)

Processes

getpriority

Return the current scheduling nice value of a process, a process group, or a user.

`getpriority` is a thin wrapper around the `getpriority(2)` system call. Nice values run from `-20` (most favourable to the process) to `19` (least favourable); a freshly-spawned process inherits its parent's value, typically `0`. Only the superuser can *lower* a nice value; any user may raise their own.

Synopsis

```
getpriority WHICH, WHO
```

`WHICH` selects what `WHO` means:

- `PRIO_PROCESS` - `WHO` is a process ID (`0` means the current process).
- `PRIO_PGRP` - `WHO` is a process-group ID (`0` means the current group).
- `PRIO_USER` - `WHO` is a numeric user ID (`0` means the real UID of the calling process).

The three constants come from `POSIX`; they are not built-ins and are not exported by default.

```
use POSIX qw(PRIO_PROCESS PRIO_PGRP PRIO_USER);
```

What you get back

The current nice value as an integer in the range `-20 .. 19`.

There is no distinguished error return: `-1` is a perfectly valid nice value. To detect failure, clear `$!` before the call and inspect it afterwards:

```
$! = 0;
my $nice = getpriority PRIO_PROCESS, $pid;
die "getpriority: $!" if $!;
```

On a kernel that implements `getpriority(2)` but returns an error (typically `ESRCH` for a non-existent target or `EINVAL` for a bogus `WHICH`), the function returns `-1` with `$!` set. On a platform that does not implement the call at all, `getpriority` raises a fatal exception; catch it with `eval` if the script must run on such a host.

Global state it touches

- `$!` - set when the underlying `getpriority(2)` call fails. Must be cleared before the call to distinguish a legitimate `-1` result from an error.

Examples

Read the current process's own nice value:

```
use POSIX qw(PRIO_PROCESS);
my $nice = getpriority PRIO_PROCESS, 0;
print "nice = $nice\n";             # nice = 0
```

Check another process by PID:

```
use POSIX qw(PRIO_PROCESS);
my $nice = getpriority PRIO_PROCESS, $pid;
```

Read the priority of the calling process's process group:

```
use POSIX qw(PRIO_PGRP);
my $nice = getpriority PRIO_PGRP, 0;
```

Read the priority cap associated with a user ID. The value returned is the *lowest* (most favourable) nice value held by any of that user's processes:

```
use POSIX qw(PRIO_USER);
my $nice = getpriority PRIO_USER, $uid;
```

Safe read that distinguishes `-1`-as-value from `-1`-as-error:

```
use POSIX qw(PRIO_PROCESS);
$! = 0;
my $nice = getpriority PRIO_PROCESS, $pid;
if ($!) {
    warn "cannot read priority of $pid: $!";
} else {
    print "pid $pid: nice=$nice\n";
}
```

Edge cases

- **`-1` is a valid return**, not an error indicator. Always clear `$!` before the call and test it after.
- **`WHO == 0`** is the "self" shorthand for each `WHICH`: current PID, current PGID, or real UID. Passing your own PID / PGID / UID explicitly gives the same result.
- **`PRIO_USER` with a UID that owns multiple processes** returns the numerically lowest nice value across those processes, reflecting the most favourable scheduling that user currently enjoys.
- **Missing target.** A PID, PGID, or UID with no matching process yields `-1` and sets `$!` to `ESRCH`.

- **Bad `WHICH`.** A value outside `PRIO_PROCESS` / `PRIO_PGRP` / `PRIO_USER` yields `-1` and sets `$!` to `EINVAL`.
- **No `getpriority(2)` on the host.** The call raises a fatal exception rather than returning a value. Guard with `eval` in portable code.
- **No `POSIX` import, no constants.** Writing `getpriority 0, 0` works by accident because `PRIO_PROCESS` happens to be `0` on Linux, but it is not portable. Always import and use the named constants.
- **Raising priority needs root.** `getpriority` itself is unprivileged; the matching `setpriority` call fails with `EACCES` when a non-root caller tries to *decrease* the nice value.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `setpriority` - write side of the same syscall pair; same `WHICH` constants, same caveats about `-1`
- `POSIX` - source of `PRIO_PROCESS`, `PRIO_PGRP`, and `PRIO_USER`; none of them are built-in
- `getpgrp` - process-group ID you would pass as `WHO` under `PRIO_PGRP`
- `fork` - child inherits the parent's nice value; read it back with `getpriority` after the fork if you need to confirm
- `$!` - the only channel for distinguishing a real `-1` priority from a failed call

Network info

getprotobyname

Look up an IP protocol by its textual name and return its database entry.

`getprotobyname` is the Perl wrapper around the system C library's `getprotobyname(3)`. Given a protocol name such as `"tcp"`, `"udp"`, or `"icmp"`, it returns the matching entry from the system's `/etc/protocols` database (or whatever backend `nsswitch.conf` directs to). The typical use is turning a human-readable protocol name into the numeric value that `socket` and `setsockopt` want.

Synopsis

```
getprotobyname NAME
my ($name, $aliases, $proto) = getprotobyname NAME;
my $proto                    = getprotobyname NAME;   # scalar␣
↪context
```

What you get back

In list context, a four-element-looking list with three fields:

```
my ($name, $aliases, $proto) = getprotobyname("tcp");
# $name    => canonical name, e.g. "tcp"
# $aliases => space-separated alias list, e.g. "TCP"
# $proto   => numeric protocol number, e.g. 6
```

In scalar context, just the numeric protocol number (`$proto`). That is the common form, because the protocol number is the whole reason the lookup was performed:

```
my $tcp = getprotobyname("tcp");   # 6
```

If the protocol is not found, the return is an empty list in list context and `undef` in scalar context. Always check definedness before using the result - `0` is a valid protocol number (`"ip"`), so truth-testing alone is wrong.

```
defined(my $tcp = getprotobyname("tcp"))
    or die "tcp protocol not in /etc/protocols";
```

Global state it touches

- Shares the protocol-database iteration cursor with `getprotobynumber`, `getprotoent`, `setprotoent`, and `endprotoent`. A call to `getprotobyname` may reposition or reset that cursor depending on the system C library; do not interleave name lookups with a `getprotoent` iteration loop.
- On some platforms the underlying `getprotobyname(3)` sets `$!` / `errno` on failure. Portable code should not rely on it - treat an undefined return as "not found" and move on.

Examples

Turn a name into a protocol number for `socket`:

```
use Socket;
my $proto = getprotobyname("tcp")
    // die "tcp not configured";
socket(my $sock, PF_INET, SOCK_STREAM, $proto)
    or die "socket: $!";
```

List form, to see the canonical name and aliases a system uses:

```
my ($name, $aliases, $proto) = getprotobyname("TCP");
print "name=$name aliases=$aliases number=$proto\n";
# name=tcp aliases=TCP number=6
```

Defensive lookup with a fallback for the two protocols every script actually needs:

```
my %num_for = (tcp => 6, udp => 17, icmp => 1);
sub proto_num {
```

(continues on next page)

(continued from previous page)

```
    my ($name) = @_;
    return getprotobyname($name) // $num_for{lc $name};
}
```

Scalar context is required when the return value feeds an expression; list context would flatten into the surrounding list:

```
my @args = (SOCK_STREAM, scalar getprotobyname("tcp"));
```

Edge cases

- **Case sensitivity is system-dependent.** glibc's `getprotobyname(3)` matches the canonical name and every alias case-insensitively; other C libraries may not. If portability matters, pass the lowercase form (`"tcp"`, not `"TCP"`).
- **Protocol number `0` is valid** (`"ip"`). Use `defined(...)`, not boolean truth, when checking for success.
- **Not the same as a port number.** `getprotobyname("http")` returns `undef` - `"http"` is a *service* name, not a protocol. Use `getservbyname` for service lookups.
- **Empty alias string is normal.** Many entries have no aliases; the second return value is then the empty string, not `undef`.
- **Iteration cursor side-effects.** If your program is in the middle of a `getprotoent` loop, calling `getprotobyname` may reset the cursor and skip or repeat entries. Finish the iteration, or save state, before doing name lookups.
- **Thread safety.** The underlying C call is not thread-safe on every platform - it typically uses a per-process static buffer. Perl may use the reentrant `getprotobyname_r` form when available, but do not assume it.
- **Missing `/etc/protocols` entry.** On minimal containers the database may be absent, in which case every call returns `undef`. Build protocol tables into your code when the target environment is not guaranteed to ship a protocol database.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `getprotobynumber` - the inverse lookup; use when you have the numeric protocol and want its name
- `getprotoent` - iterate every entry in the protocol database one record at a time
- `setprotoent` / `endprotoent` - control the iteration cursor shared with this function
- `getservbyname` - the equivalent lookup for services (ports), not protocols; the two are often confused
- `socket` - the usual consumer of the returned protocol number

- `Net::protoent` - object-oriented wrapper that returns a blessed record with named accessors instead of a positional list

Network info

getprotobynumber

Look up a network protocol entry by its assigned protocol number.

`getprotobynumber` wraps the system `getprotobynumber(3)` call. Given a numeric protocol identifier - the same values named by constants like `IPPROTO_TCP` (6), `IPPROTO_UDP` (17), `IPPROTO_ICMP` (1) - it returns the matching entry from the system's protocol database (typically `/etc/protocols`). The common use is turning a protocol number pulled off the wire, out of a routing table, or returned by `getsockopt` into a human-readable name.

Synopsis

```
getprotobynumber NUMBER
```

What you get back

In list context, a three-element list:

```
my ($name, $aliases, $proto) = getprotobynumber($number);
```

- `$name` - canonical protocol name (`"tcp"`, `"udp"`, `"icmp"`).
- `$aliases` - space-separated string of alternative names for the protocol, or the empty string if none.
- `$proto` - the protocol number you passed in, echoed back from the database entry.

In scalar context, you get `$name` alone. If the number is not in the database, the scalar return is `undef` and the list return is empty.

```
my $name = getprotobynumber(6);          # "tcp"
my @rec  = getprotobynumber(6);          # ("tcp", "", 6)
my $none = getprotobynumber(9999);       # undef
```

Global state it touches

Shares the protocol-database cursor with `getprotobyname`, `getprotoent`, `setprotoent`, and `endprotoent`. Calling `getprotobynumber` does not disturb an iteration in progress on most systems, but portable code that interleaves single lookups with `getprotoent` iteration should call `setprotoent` afterwards to reset the cursor.

On failure, `$!` is set to the underlying `errno`, though most implementations simply return "not found" without setting `errno`.

Examples

Resolve the canonical name of a protocol number:

```
my $name = getprotobynumber(6);
print "$name\n";                          # "tcp"
```

Full record with aliases:

```
my ($name, $aliases, $proto) = getprotobynumber(1);
print "$name ($proto): $aliases\n";     # e.g. "icmp (1): ICMP"
```

Reverse-lookup a protocol pulled out of a packet header:

```
sub protocol_label {
    my ($num) = @_;
    my $name = getprotobynumber($num);
    return defined $name ? $name : "proto-$num";
}
```

Round-trip against `getprotobyname` - useful as a sanity check that your system's protocol database knows the name you expect:

```
my $num  = getprotobyname("tcp");       # 6
my $back = getprotobynumber($num);      # "tcp"
```

Pretty-print a `getsockopt` level. The `level` argument to `getsockopt` is a protocol number; `getprotobynumber` turns it back into a label for logging:

```
my ($level) = unpack("I", $packed_level);
printf "socket option at level %s\n", getprotobynumber($level) //
 ↪$level;
```

Edge cases

- **List-operator precedence trap.** Even though `getprotobynumber` takes a single argument, it parses with list-operator precedence. The function name swallows everything to its right as one big list expression:

  ```
  getprotobynumber $number eq 'icmp'    # WRONG - equivalent to:
  getprotobynumber($number eq 'icmp')   # looks up protocol 0 or 1
  getprotobynumber($number) eq 'icmp'   # what you meant
  ```

  Always parenthesise the argument when comparing the result, or assign the lookup to a variable first.

- **Unknown number**: scalar return is `undef`; list return is the empty list. Guard with `defined`:

  ```
  defined(my $name = getprotobynumber($n))
      or die "unknown protocol number $n";
  ```

- **Non-integer argument**: converted to an integer via the usual numeric coercion before the lookup. A string like `"tcp"` coerces to `0` and you get whatever entry (if any) is registered for protocol `0`, which is usually `"ip"`. For name-based lookup, use `getprotobyname`.
- **Negative or out-of-range numbers**: no entry, returns empty / `undef`. No warning is raised.
- **The `Net::protoent` object interface**. If you import `Net::protoent`, `getprotobynumber` is overridden to return a blessed object with named accessors (`$p->name`, `$p->aliases`, `$p->proto`). Handy when you do not want to remember positional field order.
- **Database lookup, not protocol stack introspection**. This function reads `/etc/protocols` (or the equivalent NSS source). It tells you what name the system assigns to a number; it does not tell you whether the kernel actually supports that protocol or whether a socket of that type can be opened.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `getprotobyname` - the inverse: look up a protocol entry by name when you have a label like `"tcp"` and need the number
- `getprotoent` - iterate every entry in the protocol database, when you want to list or index them all
- `setprotoent` - reopen / rewind the protocol-database cursor; pair with `endprotoent` around an iteration
- `getservbyport` - the service-table analogue, turning a port number into a service name
- `getsockopt` - returns a protocol number as its `level` argument; feed it to `getprotobynumber` for readable logs
- `Socket` - exports the `IPPROTO_*` constants whose numeric values are what you pass in

Network info

getprotoent

Fetch the next entry from the protocols database.

`getprotoent` walks the system protocols database - on traditional Unix-likes that's `/etc/protocols`, on modern systems whatever the C library and NSS stack map to it - returning one record per call until the database is exhausted. It is the iterating leg of the `setprotoent` / `getprotoent` / `endprotoent` triad. Reach for `getprotobyname` or `getprotobynumber` when you want a single targeted lookup; `getprotoent` is for enumerating every protocol known to the system.

The iteration cursor is shared process-wide and owned by the C library. `setprotoent` rewinds it and optionally pins the backing descriptor open; `endprotoent` closes it. Between those two calls `getprotoent` yields each record in order, then returns the empty list (or `undef` in scalar context) once there is nothing left.

Synopsis

```
my ($name, $aliases, $proto) = getprotoent;
my $name = getprotoent;
```

What you get back

List context - a three-element list mirroring the C `struct protoent`:

- `$name` - the canonical protocol name, e.g. `"tcp"`, `"udp"`, `"icmp"`.
- `$aliases` - a space-separated string of alternate names for the same protocol. Often empty.
- `$proto` - the numeric protocol number as used in IP headers and accepted by `socket` / `getsockopt`.

At end-of-database the return is the empty list. The upstream perlfunc note about *"a single meaningless true value"* for missing entries applies to the by-name and by-number lookups, not to `getprotoent` - running off the end of the iteration is a clean empty list.

Scalar context - the canonical `$name` only, or `undef` at end-of-database. Use list context whenever you need the number or the aliases; you cannot reconstruct them from the name after the fact.

Split the aliases into a list with a plain `split`:

```
my @alts = split ' ', $aliases;
```

Global state it touches

- The **protocols database cursor**, a per-process C-library-owned resource shared with `setprotoent`, `endprotoent`, `getprotobyname`, and `getprotobynumber`. A by-name or by-number lookup may reset the cursor mid-iteration on some platforms - see *Edge cases*.
- No Perl-level special variables are read or written. Unlike `gethostent`, the protocols routines have no `h_errno` equivalent, so `$?` is not touched. End of database is not a failure and does not set `$!`.

Examples

Enumerate every protocol the system knows about:

```
setprotoent(1);
while (my ($name, $aliases, $proto) = getprotoent) {
    print "$proto\t$name\t$aliases\n";
```

(continues on next page)

(continued from previous page)

```
}
endprotoent;
```

Scalar context - list just the canonical names:

```
setprotoent(0);
while (defined(my $name = getprotoent)) {
    print "$name\n";
}
endprotoent;
```

Build a name-to-number map in one pass so later lookups don't hit the database at all:

```
my %proto_num;
setprotoent(1);
while (my ($name, $aliases, $num) = getprotoent) {
    $proto_num{$name} = $num;
    $proto_num{$_}    = $num for split ' ', $aliases;
}
endprotoent;

print $proto_num{"tcp"}, "\n";          # 6
```

By-record interface via the `Net::protoent` override - accessor methods instead of positional unpacking:

```
use Net::protoent;
while (my $p = getprotoent) {
    printf "%-10s %3d\n", $p->name, $p->proto;
}
```

Edge cases

- **No implicit rewind.** `getprotoent` does not open or rewind the database on its own. The first call after program start works because the C library lazily opens the file, but once the end is reached further calls keep returning end-of-database until a `setprotoent` rewinds the cursor. Always bracket enumeration with an explicit `setprotoent` and `endprotoent`.
- **End-of-database is the empty list, not `undef`.** In list context the loop condition `while (my @r = getprotoent)` works because the empty list is false. In scalar context use `while (defined(my $name = getprotoent))` - a protocol name of `"0"` is theoretically valid and would falsely terminate a truthiness-based loop.
- **Aliases are one scalar, not a list.** `$aliases` is a single space-separated string; split on whitespace if you want a list. An empty aliases field is the common case, not an error.
- **Interleaving with by-name / by-number lookups.** A call to `getprotobyname` or `getprotobynumber` during a walk can reset the shared cursor on some platforms. Finish the iteration first, or restart it with `setprotoent` after the lookup.

- **Not reentrant.** The database cursor is a single global resource inside the C library. Two concurrent iterations in the same process - across threads, signal handlers, or nested callbacks - corrupt each other's state. Confine each walk to one thread at a time.
- **Platforms without a protocols database.** On systems with no `/etc/protocols` equivalent, the call is a no-op that returns the empty list immediately rather than raising an error. See `perlport` for the full list of network-database quirks.
- **Tainting.** Records come from a file the kernel reads on your behalf and are not marked tainted. Treat them as untrusted anyway when the file is user-writable.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `setprotoent` - rewind the iterator and optionally pin the backing descriptor open for the full walk
- `endprotoent` - close the iterator and release the database handle once enumeration is done
- `getprotobyname` - targeted lookup by protocol name instead of full enumeration
- `getprotobynumber` - targeted lookup by numeric protocol number
- `getservent` - same iteration pattern for the services database, which references protocol names from this one
- `gethostent` - same idiom for the hosts database

User and group info

getpwent

Return the next entry from the system password database.

`getpwent` walks the password database one record at a time. The first call opens the database (typically `/etc/passwd`, possibly augmented by NSS with LDAP, SSSD, or similar); each subsequent call returns the next entry. When the iteration is exhausted the call returns an empty list in list context or `undef` in scalar context. Pair it with `setpwent` to rewind and `endpwent` to release any resources the database library is holding.

Synopsis

```
my @fields = getpwent;
my $name   = getpwent;      # scalar context: login name only
```

What you get back

In list context, a 10-element list describing one account:

```
my ( $name,   $passwd,   $uid,   $gid,   $quota,
     $comment, $gcos,    $dir,   $shell, $expire ) = getpwent;
#    0         1          2       3       4
#    5         6          7       8       9
```

- `$name` - login name.
- `$passwd` - the encrypted password field. On systems with shadow passwords this is usually `"x"` or `"*"` unless the process is privileged; the real hash lives in `/etc/shadow`. Tainted under `-T`.
- `$uid`, `$gid` - numeric user and primary group ids.
- `$quota` - disk quota on systems that support it, otherwise an empty string. On some systems this slot carries `$change` or `$age` (password-aging data).
- `$comment` - administrative comment on systems that support it, otherwise empty. On some systems this slot carries `$class`.
- `$gcos` - the GECOS field, conventionally the user's real name. User-writable on many systems, therefore tainted under `-T`.
- `$dir` - home directory.
- `$shell` - login shell. Tainted under `-T`.
- `$expire` - account or password expiration on systems that support it, otherwise absent.

In scalar context, only the login name is returned. When the iteration is exhausted the scalar value is `undef` and the list value is empty.

Whether `$quota`, `$comment`, and `$expire` carry meaningful data on the current build depends on how Perl was configured; inspect `Config` values `d_pwquota`, `d_pwage`, `d_pwchange`, `d_pwcomment`, and `d_pwexpire` to find out.

Global state it touches

`getpwent` keeps its cursor in a **process-wide**, library-owned state block inside the C library's pwd routines. Consequences:

- Two independent loops over the database in the same process interfere with each other. Each `getpwent` advances the one shared cursor.
- `setpwent` rewinds that cursor to the beginning.
- `endpwent` closes the database and drops any resources (open file descriptors, NSS module state) the library was holding.
- Threads share the cursor too. `getpwent` is not safe to call concurrently from multiple threads; if two threads iterate in parallel, neither sees every record.

`getpwent` does not read or write any Perl special variables.

Examples

Dump every account name and home directory:

```
while ( my @pw = getpwent ) {
    print "$pw[0]\t$pw[7]\n";
}
endpwent;
```

Named-field destructuring reads more clearly than numeric indexing:

```
while ( my ($name, undef, $uid, $gid, undef, undef, $gcos, $dir) =
 ↪getpwent ) {
    next if $uid < 1000;                # skip system accounts
    print "$name ($gcos) uid=$uid home=$dir\n";
}
endpwent;
```

Scalar context yields the login name, useful for enumerating users without unpacking the rest:

```
my @users;
while ( defined( my $name = getpwent ) ) {
    push @users, $name;
}
endpwent;
```

Rewind in the middle of a walk with `setpwent`:

```
my @first_pass = (getpwent)[0];
setpwent;                                # back to the top
while ( my @pw = getpwent ) { ... }
endpwent;
```

Collect all entries into a hash keyed by uid, then release the database:

```
my %by_uid;
while ( my @pw = getpwent ) {
    $by_uid{ $pw[2] } = \@pw;
}
endpwent;
```

Edge cases

- **End of iteration**: returns an empty list in list context and `undef` in scalar context. `while (my @pw = getpwent)` and `while (defined(my $name = getpwent))` are the idiomatic loop forms.
- **Forgetting `endpwent`**: leaks whatever file descriptors or NSS state the library opened until the process exits. Not a correctness bug, but noticeable in long-lived programs and in test harnesses.

- **Nested iteration is broken**: two loops in the same process share the library's single cursor. Collect one walk into a data structure before starting the next.
- **Shadow passwords**: on Linux and Solaris the real password hash is in `/etc/shadow`, readable only by root. Non-privileged processes see `"x"` or `"*"` in `$passwd` and must call the shadow-aware routines (outside of core Perl) to get the hash.
- **NSS surprises**: the database is whatever `/etc/nsswitch.conf` says it is. On a host bound to LDAP or SSSD, `getpwent` may return thousands of directory accounts, take a noticeable amount of time, and fail intermittently if the network is flaky. Consider looking accounts up by name with `getpwnam` when you know which one you want.
- **Tainting under -T**: `$passwd`, `$gcos`, and `$shell` come back tainted because users can influence them. Untaint with a deliberate regex match if you need to use them in sensitive operations.
- **Thread safety**: unlike `getpwnam` and `getpwuid`, there is no `getpwent_r` in POSIX that Perl can transparently substitute. Concurrent iteration from multiple threads is unsafe.
- **By-name interface**: `User::pwent` overrides `getpwent` with a version that returns an object exposing fields as methods (`$pw->name`, `$pw->uid`, ...). Use it when the numeric-slot unpacking gets unwieldy.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `getpwnam` - look up one account by login name; the right call when you know which user you want
- `getpwuid` - look up one account by numeric uid
- `setpwent` - rewind the `getpwent` cursor to the first entry
- `endpwent` - close the password database and release library resources; call it when the walk is done
- `getgrent` - the group-database counterpart, same iteration model
- `User::pwent` - by-name interface that returns an object with named accessors instead of a 10-element list

User and group info

getpwnam

Look up a user's passwd record by login name.

`getpwnam` wraps the system C library's `getpwnam(3)` call. Pass a login name like `"root"` or `"alice"`; you get back the passwd entry for that user, or an empty list / `undef` if no such user exists. The return shape depends on context: list context gives every field of the record; scalar context gives a single useful value - for `getpwnam` specifically, the numeric UID.

Synopsis

```
getpwnam NAME
```

What you get back

List context returns a 10-element list mirroring `struct passwd` augmented with the BSD extras:

```
my ($name,    $passwd, $uid,  $gid,   $quota,
    $comment, $gcos,   $dir,  $shell, $expire) = getpwnam($login);
```

- `$name` - login name (same string you passed in)
- `$passwd` - encrypted password, or `"x"` / `"*"` on shadow-password systems. Tainted under `-T`.
- `$uid` - numeric user ID
- `$gid` - numeric primary group ID
- `$quota` - disk quota, or `$change` / `$age` on some systems. Empty string when unsupported.
- `$comment` - administrative comment, sometimes `$class`. Empty string when unsupported.
- `$gcos` - GECOS field (usually the real name plus other user info). Tainted under `-T` because users can edit it.
- `$dir` - home directory
- `$shell` - login shell. Tainted under `-T`.
- `$expire` - account / password expiration, when supported.

Scalar context returns just `$uid`. For `getpwnam` - a lookup *by name* - scalar context gives you "the other thing," matching the rule upstream documents for all `getpw*` / `getgr*` / `gethost*` / `getnet*` / `getproto*` / `getserv*` functions.

On lookup failure both contexts return the empty list / `undef`. Always check the return before unpacking.

Global state it touches

- Sets `$!` on system-call failure (rare - "user not found" is not a failure, it's an empty return).
- Under `-T` (taint mode), `$passwd`, `$gcos`, and `$shell` come back tainted. See `perlsec` for how to launder them.
- Shares the same underlying iterator state as `getpwent`, `setpwent`, and `endpwent`. A `getpwnam` call does **not** disturb an in-progress `getpwent` loop on glibc, but portable code should not assume that.

Examples

Basic name-to-UID lookup in scalar context:

```
my $uid = getpwnam("root");          # 0
```

Full record in list context, with a defensive check:

```
my @pw = getpwnam($user)
    or die "$user: not in passwd database";
my ($name, undef, $uid, $gid, undef, undef, $gcos, $home, $shell) =
 ↪@pw;
print "$name ($uid) -> $home, shell $shell\n";
```

Assignment pattern from upstream - unpacking directly with `or die`:

```
my ($login, $pass, $uid, $gid) = getpwnam($user)
    or die "$user not in passwd file";
```

Comparing file ownership against a named user without hand-written field indexing, via the `User::pwent` override:

```
use User::pwent;
use File::stat;
my $is_theirs = (stat($filename)->uid == getpwnam($whoever)->uid);
```

`use User::pwent` replaces `getpwnam` with a version returning a blessed object with `->name`, `->uid`, `->gid`, `->dir`, `->shell` accessors - handy when you want one field and don't want to count commas. The two forms do not mix: inside one scope either use the override or the built-in, not both.

Fallback chain for "who am I running as":

```
my $login = getlogin() || getpwuid($<) || "unknown";
```

`getpwuid` is the by-UID counterpart; `getlogin` reads the controlling-tty record and can lie on detached processes, so it's the first choice, not the only one.

Edge cases

- **No such user** returns the empty list in list context and `undef` in scalar context. This is not an error - `$!` is not set. Distinguish "user missing" from "lookup failed" by checking `$!` only when you want to report an unexpected failure, not when deciding whether the user exists.
- **`$passwd` is rarely useful.** On modern Linux and BSD systems the real hash lives in `/etc/shadow`, readable only by root. You get `"x"` or `"*"` here. Use PAM or the `Authen::PAM` module for authentication rather than comparing against this field.
- **`$quota`, `$comment`, `$expire` portability.** These fields are empty or meaningless on Linux. The upstream POD lists `Config` values (`d_pwquota`, `d_pwage`, `d_pwchange`, `d_pwcomment`, `d_pwexpire`) you can check if you need to know at runtime whether the platform populates them.

- **Tainted fields under `-T`.** `$passwd`, `$gcos`, `$shell` are tainted because users can modify them (via `chsh`, `chfn`, or direct `/etc/passwd` edits in privileged contexts). Pass them through a regex capture to launder before using in a system call.
- **Return value is a "single meaningless true value"** in Boolean-ish scalar context *when the entry exists but you wanted list context.* The scalar-context documented return for `getpwnam` is the UID, which for `root` is `0` - a false value. `if (getpwnam("root")) { ... }` therefore fails for root. Use `defined` or list context:

```
if (defined getpwnam("root"))     { ... }   # correct
if (my @pw = getpwnam("root"))    { ... }   # also correct
```

- **Name is a string, not a UID.** `getpwnam(0)` stringifies `0` and looks up the literal user named `"0"`, which almost never exists. Use `getpwuid` for numeric lookups.
- **Large directories (LDAP / NIS / SSSD).** `getpwnam` goes through NSS, so the cost scales with however the local resolver answers. A miss against a misconfigured LDAP server can block for seconds. Cache results in a hash if you are looking up many users in a tight loop.
- **Threads.** Upstream notes that `getpwnam_r` is used automatically where the platform provides it; `getpwent` iteration, by contrast, remains per-process. Not a concern under pperl (no ithreads), but worth knowing if you share code with perl5.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `getpwuid` - same record, looked up by numeric UID; pair with `getpwnam` when you need to go UID → name
- `getpwent` - iterate the whole passwd database; use when you need every user, not a specific one
- `endpwent` - close the passwd iterator after `getpwent`
- `getgrnam` - group-database counterpart; needed to resolve the `$gid` field from this record back to a group name
- `getlogin` - cheap "who is on this tty" lookup; fall back to `getpwnam`/`getpwuid` when it returns empty
- `User::pwent` - object wrapper with named accessors; reach for it when you want `->uid` instead of counting list positions

User and group info

getpwuid

Look up a user's passwd record by numeric UID.

`getpwuid` wraps the system C library's `getpwuid(3)` call. Pass a numeric user ID like `0` or `$<`; you get back the passwd entry for that user, or an empty list / `undef` if no such UID

exists. The return shape depends on context: list context gives every field of the record; scalar context gives a single useful value - for `getpwuid` specifically, the login name.

Synopsis

```
getpwuid UID
```

What you get back

List context returns a 10-element list mirroring `struct passwd` augmented with the BSD extras - the same shape `getpwnam` returns:

```
my ($name,    $passwd, $uid,  $gid,   $quota,
    $comment, $gcos,   $dir,  $shell, $expire) = getpwuid($uid);
```

- `$name` - login name
- `$passwd` - encrypted password, or `"x"` / `"*"` on shadow-password systems. Tainted under `-T`.
- `$uid` - numeric user ID (same number you passed in)
- `$gid` - numeric primary group ID
- `$quota` - disk quota, or `$change` / `$age` on some systems. Empty string when unsupported.
- `$comment` - administrative comment, sometimes `$class`. Empty string when unsupported.
- `$gcos` - GECOS field (usually the real name plus other user info). Tainted under `-T` because users can edit it.
- `$dir` - home directory
- `$shell` - login shell. Tainted under `-T`.
- `$expire` - account / password expiration, when supported.

Scalar context returns just `$name` (the login name). For `getpwuid` - a lookup *by UID* - scalar context gives you "the other thing," matching the rule upstream documents for all `getpw*` / `getgr*` / `gethost*` / `getnet*` / `getproto*` / `getserv*` functions.

On lookup failure both contexts return the empty list / `undef`. Always check the return before unpacking.

Global state it touches

- Sets `$!` on system-call failure (rare - "UID not found" is not a failure, it's an empty return).
- Reads `$<` and `$>` implicitly only when you pass them in, but the common idiom `getpwuid($<)` is what turns the real-UID special variable into a useful record.
- Under `-T` (taint mode), `$passwd`, `$gcos`, and `$shell` come back tainted. See `perlsec` for how to launder them.

- Shares the same underlying iterator state as `getpwent`, `setpwent`, and `endpwent`. A `getpwuid` call does **not** disturb an in-progress `getpwent` loop on glibc, but portable code should not assume that.

Examples

Basic UID-to-name lookup in scalar context:

```
my $name = getpwuid(0);              # "root"
```

"Who am I running as" - the canonical `getpwuid` idiom, resolving the real UID to a login name:

```
my $me = getpwuid($<);               # e.g. "alice"
```

Pair with `getlogin` for a reliable fallback chain that survives detached processes and `su` sessions:

```
my $login = getlogin() || getpwuid($<) || "unknown";
```

`getlogin` reads the controlling-tty record and can lie (or return empty) when there is no tty; `getpwuid($<)` answers from the kernel's recorded UID and always works.

Full record in list context, with a defensive check:

```
my @pw = getpwuid($uid)
    or die "uid $uid: not in passwd database";
my ($name, undef, undef, $gid, undef, undef, $gcos, $home, $shell) =
 ↪@pw;
print "$name ($uid) -> $home, shell $shell\n";
```

Go UID → group name via the `$gid` field:

```
my $gid      = (getpwuid($<))[3];
my $groupnam = getgrgid($gid);
```

Object-style access via the `User::pwent` override, useful when you want named accessors instead of counting list positions:

```
use User::pwent;
my $pw = getpwuid($<);
printf "%s lives in %s, runs %s\n", $pw->name, $pw->dir, $pw->shell;
```

`use User::pwent` replaces `getpwuid` with a version returning a blessed object with `->name`, `->uid`, `->gid`, `->dir`, `->shell` accessors. The two forms do not mix: inside one scope either use the override or the built-in, not both.

Edge cases

- **No such UID** returns the empty list in list context and `undef` in scalar context. This is not an error - `$!` is not set. Distinguish "UID missing" from "lookup failed" by checking `$!` only when you want to report an unexpected failure, not when deciding whether the UID exists.

- **`$passwd` is rarely useful.** On modern Linux and BSD systems the real hash lives in `/etc/shadow`, readable only by root. You get `"x"` or `"*"` here. Use PAM or the `Authen::PAM` module for authentication rather than comparing against this field.
- **`$quota`, `$comment`, `$expire` portability.** These fields are empty or meaningless on Linux. The upstream POD lists `Config` values (`d_pwquota`, `d_pwage`, `d_pwchange`, `d_pwcomment`, `d_pwexpire`) you can check if you need to know at runtime whether the platform populates them.
- **Tainted fields under `-T`.** `$passwd`, `$gcos`, `$shell` are tainted because users can modify them (via `chsh`, `chfn`, or direct `/etc/passwd` edits in privileged contexts). Pass them through a regex capture to launder before using in a system call.
- **Scalar-return on UID 0 is never false.** Unlike `getpwnam`, whose scalar return for `root` is the UID `0` (a false value), `getpwuid`'s scalar return is the *name* `"root"` - always a true string for existing users. `if (getpwuid($uid)) { ... }` therefore works as you would expect, but reach for `defined` anyway if you care about the distinction between "missing" and "empty name":

```
if (defined getpwuid($uid))       { ... }   # unambiguous
if (my @pw = getpwuid($uid))      { ... }   # also correct
```

- **UID is a number, not a name.** `getpwuid("root")` stringifies `"root"` to `0` (with a warning under `use warnings`) and looks up UID `0`. Use `getpwnam` for name lookups.
- **Negative / oversized UIDs.** The system call accepts whatever `uid_t` accepts on the platform (typically 32-bit unsigned). Perl converts negative values via two's-complement on the way in; the result is platform-specific and almost never what you want.
- **Large directories (LDAP / NIS / SSSD).** `getpwuid` goes through NSS, so the cost scales with however the local resolver answers. A miss against a misconfigured LDAP server can block for seconds. Cache results in a hash if you are resolving many UIDs in a tight loop (e.g. annotating `ls -l`-style output).
- **Threads.** Upstream notes that `getpwuid_r` is used automatically where the platform provides it; `getpwent` iteration, by contrast, remains per-process. Not a concern under pperl (no ithreads), but worth knowing if you share code with perl5.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `getpwnam` - same record, looked up by login name; pair with `getpwuid` when you need to go name → UID
- `getpwent` - iterate the whole passwd database; use when you need every user, not a specific one
- `endpwent` - close the passwd iterator after `getpwent`
- `getgrgid` - group-database counterpart; needed to resolve the `$gid` field from this record back to a group name

- `getlogin` - cheap "who is on this tty" lookup; fall back to `getpwuid($<)` when it returns empty
- `User::pwent` - object wrapper with named accessors; reach for it when you want `->name` instead of counting list positions

Network info

getservbyname

Look up a network service by its textual name and protocol.

`getservbyname` consults the system service database (typically `/etc/services`, or whatever `nsswitch.conf` routes the `services` source to) and returns the entry whose name or alias matches `NAME` under the transport protocol named by `PROTO`. It is the Perl wrapper around the C library's `getservbyname(3)` call, and it is how you turn `"https"` into 443 without hardcoding the number.

Both arguments are strings. `NAME` is the service name or alias (`"http"`, `"ssh"`, `"imaps"`); `PROTO` is the protocol name (`"tcp"`, `"udp"`, `"sctp"`) - **not** a number, and **not** a `SOCK_STREAM`-style constant.

Synopsis

```
my $port                                = getservbyname NAME, PROTO;   #␣
↪scalar
my ($name, $aliases, $port, $proto) = getservbyname NAME, PROTO;   #␣
↪list
```

What you get back

In **scalar context**, the port number as a plain integer in host byte order - ready to feed to `pack` with `"n"` or to a `sockaddr_in` constructor. If no entry matches, the return is `undef`.

In **list context**, the four fields of the `servent` record:

- `$name` - canonical service name (the primary name, not whichever alias you looked up by).
- `$aliases` - space-separated string of alternate names.
- `$port` - port number, host byte order.
- `$proto` - canonical protocol name as returned by the resolver, normally equal to the `PROTO` you passed.

If no entry matches in list context the return is the empty list.

The port is **host byte order**. Network APIs expect network byte order, so pass it through `pack` with the `"n"` template - or let `Socket` do it via `sockaddr_in`.

Global state it touches

- Holds an open handle to the services database between calls when the database has been opened with `setservent` using a non-zero STAYOPEN. `endservent` closes it.
- Sets `$!` on lookup failures that stem from a system-level error (database file missing, NSS backend unreachable). A plain "no such service" is reported by returning `undef` / the empty list, not by setting `$!`.

Examples

Resolve a well-known service to its port:

```
my $port = getservbyname("https", "tcp");
print $port, "\n";                    # 443
```

Full record in list context:

```
my ($name, $aliases, $port, $proto) = getservbyname("http", "tcp");
print "$name ($aliases) -> $port/$proto\n";
                                      # www (www-http http) -> 80/
↪tcp
                                      # (exact aliases depend on /
↪etc/services)
```

Build a `sockaddr_in` for `connect`, converting the host-order port to network order with `pack`:

```
use Socket;
my $port  = getservbyname("smtp", "tcp")
    or die "smtp/tcp not in services db";
my $addr  = sockaddr_in($port, inet_aton("mail.example.com"));
socket(my $s, AF_INET, SOCK_STREAM, getprotobyname("tcp")) or die $!
↪;
connect($s, $addr) or die "connect: $!";
```

Distinguish "not found" from "wrong protocol" - a service that exists under `tcp` but not udp returns `undef` when the protocol does not match:

```
defined(getservbyname("https", "tcp")) or die "expected 443";
defined(getservbyname("https", "udp")) or warn "no https/udp entry";
```

Look up by alias - aliases resolve the same as the canonical name, but the returned `$name` is always the canonical one:

```
my ($name) = getservbyname("www", "tcp");
print $name, "\n";                    # "http" on most systems
```

Edge cases

- **Unknown service**: returns `undef` in scalar context, the empty list in list context. `$!` is **not** set for a plain miss - only for underlying system errors.
- **Unknown protocol**: same as an unknown service. `getservbyname("ssh", "xyzzy")` returns nothing; no separate error is raised.
- **Case sensitivity**: service and protocol names are compared case-insensitively by the system resolver on every platform we support; `"HTTPS"` and `"https"` give the same answer.
- **Port is host byte order**: passing it straight into a wire format is a bug. Use `pack("n", $port)` or `sockaddr_in($port, $addr)` - both handle the conversion.
- **`NAME` and `PROTO` are both required**. There is no one-arg form; omitting `PROTO` is a compile-time error.
- **Precedence trap**: like every named-unary-lookalike in this family, `getservbyname` is a list operator. Parenthesise when combining with operators:

```
getservbyname "http", "tcp" == 80      # WRONG: parses as
                                       # getservbyname("http", (
 ↪"tcp" == 80))
getservbyname("http", "tcp") == 80     # right
```

- **Threads**: on systems with `getservbyname_r(3)`, Perl uses the reentrant variant transparently. On the handful of platforms that lack it, concurrent lookups from multiple threads can clobber each other's results.
- **By-name interface**: `Net::servent` (core) wraps this call and returns an object with named accessors, at the cost of one allocation per lookup.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `getservbyport` - inverse lookup: port + protocol → service record
- `getservent` - walk every entry in the services database, one record at a time
- `setservent` - rewind (or persistently open) the services database before a walk
- `endservent` - close the services database opened by `setservent`
- `getprotobyname` - companion lookup for the `PROTO` argument when you also need its numeric protocol number
- `Socket` - `sockaddr_in` and `pack_sockaddr_in` consume the host-order port returned here

Network info

getservbyport

Look up a network service by its numeric port and protocol.

`getservbyport` consults the system services database (`/etc/services` on Linux, plus whatever NSS sources are configured) and returns the record for the service that listens on `PORT` under protocol `PROTO`. `PORT` is an integer **in network byte order** - the value you would pass to `bind` or receive from `sockaddr_in`, not the decimal shown in `/etc/services`. `PROTO` is a lowercase string such as `"tcp"` or `"udp"`. The call wraps the C library's `getservbyport(3)`.

Synopsis

```
getservbyport PORT, PROTO          # list context: full record
scalar getservbyport PORT, PROTO   # scalar context: service name
```

What you get back

In **list context**, a four-element list describing the service:

```
my ($name, $aliases, $port, $proto) = getservbyport($port_n, "tcp");
```

Index	Name	Meaning
0	`$name`	Canonical service name, e.g. `"http"`
1	`$aliases`	Space-separated alias list, e.g. `"www www-http"`
2	`$port`	Port number in **network byte order**
3	`$proto`	Protocol name, e.g. `"tcp"`

If the entry does not exist the list is empty, so a list-context assignment leaves every target `undef`. A bare list-context call returns a single meaningless true value on miss - always assign to a list and test the first element, or call in scalar context.

In **scalar context**, you get the canonical service name (the same `$name` as above), or `undef` if no service is registered on that port for that protocol:

```
my $name = getservbyport($port_n, "tcp")
    or warn "port $port not registered for tcp";
```

Global state it touches

- `$!` is **not** reliably set on a miss - the C `getservbyport(3)` function has no standard errno contract for "not found". Treat an empty list or `undef` return as the only trustworthy failure signal.
- Iteration state held by the services database (shared with `getservent`, `setservent`, and `endservent`) is process-global and not thread-safe.

Byte-order trap

PORT must be network byte order. The port column in `/etc/services` is a human decimal; the value the kernel and the C library use is the big-endian 16-bit encoding. Convert with `pack` / `unpack` or with the `Socket` helpers:

```
use Socket;

# Host decimal 80 -> network-order short:
my $port_n = pack("n", 80);            # two bytes, big-endian
my @rec    = getservbyport($port_n, "tcp");

# Or going the other way, a port extracted from sockaddr_in is
# already network order:
my ($port_n_from_peer, $iaddr) = sockaddr_in(getpeername($sock));
my $name = getservbyport($port_n_from_peer, "tcp");
```

Passing a host-order integer such as `80` works on little-endian hardware only by accident for low ports whose encoding happens to be non-zero in the low byte - most ports will simply miss. This is the single most common reason a `getservbyport` call "silently returns nothing".

Examples

Identify the service on port 443 over TCP:

```
use Socket;
my $port_n = pack("n", 443);
my ($name) = getservbyport($port_n, "tcp");
print $name, "\n";                     # https
```

Pair with `getpeername` to name an incoming connection's port:

```
use Socket;
my ($peer_port_n, $peer_addr) = sockaddr_in(getpeername($client));
my $service = getservbyport($peer_port_n, "tcp") // "unknown";
print "client connected from service $service\n";
```

Full record dump - useful when you want the protocol or aliases:

```
use Socket;
my @rec = getservbyport(pack("n", 53), "udp");
if (@rec) {
    my ($name, $aliases, $port_n, $proto) = @rec;
    printf "%s (%s) on %d/%s\n",
        $name, $aliases, unpack("n", pack("n", $port_n)), $proto;
}
```

Scalar context, guard clause style:

```
my $svc = getservbyport(pack("n", $port), "tcp")
    or die "no TCP service registered on port $port";
```

Iterate every registered service (the underlying database is shared with `getservent`):

```
while (my ($n, $a, $p, $proto) = getservent()) {
    next unless $proto eq "tcp";
    # …do something with $n / $p…
}
endservent();
```

Edge cases

- **Missing entry**: empty list in list context, `undef` in scalar context. A bare `if (getservbyport(...))` in list context is truthy even for a missing entry because of the "single meaningless true value" legacy return - always assign to a list or force scalar context.
- **Unknown protocol**: an unrecognised `PROTO` string (e.g. `"sctp"` on a host whose database does not include it) is treated by the C library as no-match. The call returns empty/undef, not an error.
- **Case of `PROTO`**: case-sensitive on most Linux libc implementations. Use lowercase `"tcp"` / `"udp"`; `"TCP"` will miss.
- **Host-order port**: passing `80` instead of `pack("n", 80)` is the classic bug - see *Byte-order trap* above.
- **Numeric vs string `PORT`**: Perl numifies `PORT` before passing to libc. A two-byte packed string works because `pack` with `"n"` produces bytes that numify deterministically on the supported platforms; if in doubt, `unpack("n", $packed)` first and pass the integer to keep the intent explicit.
- **Precedence with list operators**: like other named unary / list operators, `getservbyport` binds loosely. Parenthesise when the result feeds another operator:

  ```
  getservbyport $port_n, "tcp" eq "https"     # WRONG
  (getservbyport($port_n, "tcp")) eq "https"  # correct
  ```

- **Threads**: the underlying lookup shares state with `getservent` / `setservent` / `endservent`. Two threads iterating concurrently will interleave records. Use thread-local wrappers when that matters.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `getservbyname` - inverse lookup, service name to port; reach for it when you know the name and want the number
- `getservent` - iterate every service record; use when you need a full sweep rather than a single lookup
- `setservent` - rewind the services iterator, optional `STAYOPEN` flag to keep the file open across `fork`

- endservent - close the services database after an iteration
- getprotobynumber - the protocol-side companion lookup; often used right after getservbyport to translate the $proto field
- pack - build the network-order port value getservbyport expects as its first argument
- Socket - constants and sockaddr_in helpers that produce network-order ports straight from kernel structures

Network info

getservent

Read the next entry from the system services database.

getservent walks the /etc/services database one record at a time, yielding the same list of fields as getservbyname and getservbyport but without a lookup key. Each call advances an internal cursor; the cursor is opened on the first call, rewound by setservent, and released by endservent. It mirrors the POSIX getservent(3) C library routine and inherits every portability quirk that implies.

Synopsis

```
my ($name, $aliases, $port, $proto) = getservent;
my $name = getservent;          # scalar context
```

What you get back

In list context, four fields matching the order documented for every getserv* variant:

- $name - canonical service name, e.g. "http".
- $aliases - space-separated string of alternate names, e.g. "www www-http". Not an array; split it yourself if you need one.
- $port - port number as a native (host-byte-order) integer, e.g. 80.
- $proto - transport protocol, typically "tcp" or "udp".

In scalar context, just $name. When the cursor reaches end of file, getservent returns the empty list in list context and undef in scalar context - the idiomatic while (my @e = getservent) loop relies on this.

To convert the aliases into a list:

```
my @alt = split / /, $aliases;
```

Global state it touches

- **Internal database cursor** for /etc/services - shared with getservbyname, getservbyport, setservent, and endservent. Any by-name or by-port lookup on some platforms rewinds or closes this cursor as a side effect; iterate in a tight loop or wrap the iteration in setservent / endservent to be safe.

- `$!` - set when the underlying C call reports an error.
- Not thread-safe. `getservent` keeps per-process, not per-thread state; two threads iterating concurrently will interleave records and neither will see the full list.

Examples

Walk the full services database and print each entry:

```
while (my ($name, $aliases, $port, $proto) = getservent) {
    print "$name/$proto $port\n";
}
endservent;
```

Reset the cursor before iterating so earlier calls elsewhere in the program cannot leave it mid-stream:

```
setservent(0);
while (my @e = getservent) {
    # @e is ($name, $aliases, $port, $proto)
}
endservent;
```

Collect only TCP services into a hash keyed by port:

```
my %tcp;
while (my ($name, undef, $port, $proto) = getservent) {
    $tcp{$port} = $name if $proto eq 'tcp';
}
endservent;
```

Use the object-oriented by-name interface from `Net::servent`, which overrides the built-in and returns a blessed object with accessor methods:

```
use Net::servent;
while (my $s = getservent) {
    printf "%-20s %5d/%s\n", $s->name, $s->port, $s->proto;
}
```

Edge cases

- **End of file returns empty / undef, not an error.** A single-scalar false value in list context (`my @e = getservent` giving an empty list) is the terminator. Do not check `$!` to detect end of iteration.
- **Missing `/etc/services`**: on systems without the file, the first call returns the empty list immediately and `$!` is set. Nothing in the call signals the difference between an empty database and a missing one - inspect `$!` before assuming the database was simply empty.
- **Cursor sharing with by-name / by-port lookups**: calling `getservbyname` or `getservbyport` in the middle of a `getservent` loop may reset the cursor on some platforms. Either finish the walk first or re-open with `setservent`.

- **`$aliases` is a string, not a list**. `split / /, $aliases` gives you the array form. Treating `$aliases` as a list directly silently gives you the length of a one-element list.
- **Threaded programs**: iteration is per-process. Coordinate access with a lock, or prefer `getservbyname` / `getservbyport` for point queries in threaded code.
- **Scalar context returns only the name.** If you need the port, call in list context even when you only care about one field.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `getservbyname` - look up a single service by its textual name instead of iterating
- `getservbyport` - look up a single service by its numeric port
- `setservent` - rewind the shared cursor before iterating so prior calls cannot leave it mid-stream
- `endservent` - release the cursor when the walk is done, freeing the underlying file descriptor
- `getprotoent` - same iterator pattern for `/etc/protocols`, typically paired when resolving a `(port, proto)` tuple
- `Net::servent` - object-oriented wrapper whose accessors remove the need to remember field order

Sockets

getsockname

Return the local address of a connected or bound socket.

`getsockname` asks the kernel which address and port `SOCKET` is bound to on this end of the connection. The result is the same packed sockaddr structure the kernel hands to `accept` or that you would pass to `bind` - unpack it with the appropriate helper from `Socket` (for example `sockaddr_in` for IPv4, `sockaddr_in6` for IPv6). Useful when a host has multiple interfaces and you need to know which one a client reached you on, or when you asked the kernel for an ephemeral port via `bind` with port `0` and now want to know the port it picked.

Synopsis

```
getsockname SOCKET
```

What you get back

On success, the packed sockaddr for the local end of SOCKET as a byte string whose layout matches the socket's address family. On failure, undef with $! set (typically ENOTSOCK if SOCKET is not a socket, or EBADF if it is not an open filehandle).

The return value is **opaque bytes**. Do not inspect it directly - hand it to the family-appropriate unpacker:

```
use Socket;
my $packed = getsockname($sock) or die "getsockname: $!";
my ($port, $addr) = sockaddr_in($packed);      # IPv4
```

For address-family-agnostic code, call sockaddr_family on the result first and dispatch on the returned AF_* constant.

Global state it touches

None. getsockname is a pure system-call wrapper - it does not read or write $_, $! aside, or any other interpreter global.

Examples

Discover the ephemeral port the kernel assigned after bind(…, 0):

```
use Socket;
socket(my $srv, AF_INET, SOCK_STREAM, 0) or die $!;
bind($srv, sockaddr_in(0, INADDR_ANY))   or die $!;
my ($port) = sockaddr_in(getsockname($srv));
print "listening on port $port\n";
```

Identify which local IP a connected client reached you on, on a multi-homed host:

```
use Socket;
my $mysockaddr = getsockname($sock);
my ($port, $myaddr) = sockaddr_in($mysockaddr);
printf "Connect to %s [%s]\n",
    scalar gethostbyaddr($myaddr, AF_INET),
    inet_ntoa($myaddr);
```

IPv6 variant - unpack with sockaddr_in6:

```
use Socket;
my ($port, $addr) = sockaddr_in6(getsockname($sock));
printf "local: [%s]:%d\n", inet_ntop(AF_INET6, $addr), $port;
```

Family-agnostic dispatch when the socket could be either IPv4 or IPv6:

```
use Socket qw(sockaddr_family sockaddr_in sockaddr_in6
              inet_ntop AF_INET AF_INET6);
my $name = getsockname($sock) or die "getsockname: $!";
if (sockaddr_family($name) == AF_INET) {
```

(continues on next page)

(continued from previous page)

```
    my ($port, $addr) = sockaddr_in($name);
    printf "%s:%d\n", inet_ntoa($addr), $port;
} else {
    my ($port, $addr) = sockaddr_in6($name);
    printf "[%s]:%d\n", inet_ntop(AF_INET6, $addr), $port;
}
```

Edge cases

- **Unbound socket**: A socket that has been created but never bound or connected typically returns a sockaddr with the wildcard address and port `0`. It is not an error - the kernel simply reports "no local name assigned yet."
- **Not a socket**: Passing an open filehandle that is not a socket (a plain file, a pipe) returns `undef` with `$!` set to `ENOTSOCK`.
- **Closed filehandle**: Returns `undef` with `$!` set to `EBADF`. Under `use warnings` a `getsockname() on closed socket` warning is emitted.
- **Unix-domain sockets**: The packed form is an `AF_UNIX` sockaddr; unpack with `unpack_sockaddr_un`. The path may be shorter than what you passed to `bind`, and on Linux it may begin with a NUL byte for abstract-namespace sockets.
- **Wildcard address with IPv4**: A socket bound to `INADDR_ANY` reports the wildcard address (`0.0.0.0`) here, not the specific interface address the peer reached. To learn the peer's target address you must use `getsockname` *after* the kernel has routed the connection (post-`accept`), or consult `IP_PKTINFO` via `getsockopt`.
- **Call before `bind`/`connect`**: The behaviour is platform-specific but harmless. Linux returns a zeroed sockaddr of the socket's family; some BSDs return `undef` with `EINVAL`. Rely on it only after `bind`, `connect`, or `accept` have completed.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `getpeername` - the mirror call: returns the **remote** end's sockaddr for a connected socket
- `bind` - set the local address; `getsockname` reads back what the kernel actually bound, including ephemeral ports chosen for you
- `accept` - returns the remote sockaddr directly, so you rarely need `getsockname` on the accepted socket unless the listener bound to a wildcard and you want the concrete local address
- `socket` - creates the socket whose local name `getsockname` reports
- `Socket` - the module providing the unpackers (`sockaddr_in`, `sockaddr_in6`, `unpack_sockaddr_un`, `sockaddr_family`) you apply to the return value

- `getsockopt` - for per-packet destination-address information (`IP_PKTINFO`) when a wildcard-bound socket needs to know the concrete local address of an incoming datagram

Sockets

getsockopt

Read one socket option out of the kernel as an opaque packed string.

`getsockopt` asks the kernel for the current value of the option named `OPTNAME` at protocol level `LEVEL` on `SOCKET`. The kernel does not return a typed value - it returns a raw byte buffer whose layout is determined by the `(LEVEL, OPTNAME)` pair. You decode that buffer with `unpack`, usually with the `"i"` or `"I"` template, because most options are plain integers.

Synopsis

```
getsockopt SOCKET, LEVEL, OPTNAME
```

What you get back

A packed string holding the kernel's reply on success, or `undef` on failure with `$!` set to the system error. The string's contents are **opaque bytes** - its length and layout depend entirely on `(LEVEL, OPTNAME)`:

- Boolean and integer options (`SO_KEEPALIVE`, `SO_REUSEADDR`, `TCP_NODELAY`, ...) come back as a packed `int`. Decode with `unpack` using `"i"` or `"I"`.
- Timeouts (`SO_RCVTIMEO`, `SO_SNDTIMEO`) come back as a packed `struct timeval` - two `long`s on most platforms.
- Address-bound options (`SO_PEERCRED`, `IP_MULTICAST_IF`, linger structs, ...) have their own layouts; consult `getsockopt(2)` and the relevant protocol man pages for the exact struct.

Always check the return value before decoding - calling `unpack` on `undef` yields an empty result and hides the real error recorded in `$!`.

Global state it touches

- `$!` is set on failure to the `errno` value from the underlying `getsockopt(2)` call. It is not cleared on success, so check the return value, not `$!` directly.

Examples

Check whether Nagle's algorithm is currently disabled (`TCP_NODELAY` set) on a TCP socket:

```
use Socket qw(IPPROTO_TCP TCP_NODELAY);

my $packed = getsockopt($sock, IPPROTO_TCP, TCP_NODELAY)
```

(continues on next page)

(continued from previous page)

```
    or die "getsockopt TCP_NODELAY: $!";
my $nodelay = unpack("I", $packed);
print "Nagle is ", $nodelay ? "off" : "on", "\n";
```

Same idea, but obtaining the TCP protocol number at runtime from the system's protocol database rather than a `Socket` constant:

```
use Socket qw(TCP_NODELAY);

defined(my $tcp = getprotobyname("tcp"))
    or die "no protocol entry for tcp";
my $packed = getsockopt($sock, $tcp, TCP_NODELAY)
    or die "getsockopt: $!";
```

Read `SO_ERROR` to drain and clear a pending asynchronous error on a non-blocking socket after `connect` or after `select` flags it writable:

```
use Socket qw(SOL_SOCKET SO_ERROR);

my $packed = getsockopt($sock, SOL_SOCKET, SO_ERROR)
    or die "getsockopt SO_ERROR: $!";
my $err = unpack("i", $packed);
if ($err) {
    $! = $err;
    die "asynchronous socket error: $!";
}
```

Read the socket type (`SOCK_STREAM`, `SOCK_DGRAM`, ...) of an already opened handle:

```
use Socket qw(SOL_SOCKET SO_TYPE SOCK_STREAM);

my $packed = getsockopt($sock, SOL_SOCKET, SO_TYPE)
    or die "getsockopt SO_TYPE: $!";
my $type = unpack("i", $packed);
print "stream socket\n" if $type == SOCK_STREAM;
```

Read a `struct timeval` option. `SO_RCVTIMEO` returns two native `long`s - seconds and microseconds:

```
use Socket qw(SOL_SOCKET SO_RCVTIMEO);

my $packed = getsockopt($sock, SOL_SOCKET, SO_RCVTIMEO)
    or die "getsockopt SO_RCVTIMEO: $!";
my ($sec, $usec) = unpack("l!l!", $packed);
printf "receive timeout: %d.%06d s\n", $sec, $usec;
```

Edge cases

- **SOCKET is not an open socket**: the call fails, returns `undef`, and sets `$!` to `ENOTSOCK` (`"Socket operation on non-socket"`) or `EBADF` (`"Bad file descriptor"`) for a closed handle.
- **Unknown `LEVEL` / `OPTNAME`**: the kernel rejects the request with `ENOPROTOOPT` (`"Protocol not available"`). There is no way to tell this apart from the success case except by checking the return value.
- **Truthy-return idiom**: the common `getsockopt(...) or die` works because a successful call returns a non-empty packed string. A zero-length reply would make the idiom misfire, but no kernel option returns a zero-length buffer in practice - every defined option has at least one byte of payload.
- **Integer decode width**: use `"i"` or `"I"` for `int`-sized options (`SO_KEEPALIVE`, `TCP_NODELAY`, `SO_ERROR`, ...); use `"l!"` or `"L!"` only when the option's documented layout genuinely uses a native `long`. Getting the width wrong silently returns the wrong number on LP64 platforms.
- **Do not hand-pack the reply**: `getsockopt` returns whatever the kernel wrote, including any padding. Re-pack only on round-trips via `setsockopt`, and then pass the buffer back verbatim rather than reconstructing it.
- **No constants without `Socket`**: `SOL_SOCKET`, `IPPROTO_TCP`, `TCP_NODELAY`, `SO_ERROR`, and friends are not built-in names. Import them from `Socket` with `use Socket qw(...)` or `use Socket qw(:all)`.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `setsockopt` - the write half; sets the option whose value `getsockopt` reads back
- `Socket` - source of the `SOL_*`, `IPPROTO_*`, `SO_*`, `TCP_*`, `IP_*` constants you pass as `LEVEL` and `OPTNAME`
- `unpack` - required to decode the packed byte string the kernel hands back
- `getprotobyname` - run-time way to obtain the `LEVEL` protocol number when you do not want to pull in the `IPPROTO_*` constants from `Socket`
- `getsockname` - sibling introspection call; returns the local address bound to `SOCKET` rather than a single option
- `socket` - creates the socket that `getsockopt` then queries

Filehandles, files, directories

glob

Expand a shell-style filename pattern into the list of matching paths.

`glob` takes a pattern string such as `"*.c"` or `"src/**/*.pm"` and returns the filenames on disk that match, in the same style a Unix shell would produce. In list context it returns

every match at once; in scalar context it acts as a stateful iterator, returning one match per call and `undef` when the list is exhausted. With no argument, `glob` expands the pattern in `$_`.

Synopsis

```
glob EXPR
glob
<PATTERN>
```

What you get back

In list context, a (possibly empty) list of filenames. An empty list means the pattern did not match - it is **not** an error and does not set `$!`.

In scalar context, `glob` is an iterator: each call returns the next match, and after the last match one more call returns `undef`. The iterator state is per-call-site, so two `while (my $f = glob ...)` loops at different source locations do not interfere.

```
my @mp3  = glob "*.mp3";              # list context
while (my $f = glob "*.mp3") { }      # scalar context, iterator
```

Global state it touches

- `$_` - read when `glob` is called with no argument.
- `glob` itself does **not** set `$!` on "no matches". A non-match is a legitimate empty list, not a failure.
- When a `glob` expression is used as the condition of a `while` or `for` loop, Perl implicitly assigns the result to `$_` and tests for **definedness**, not truth - so a filename of `"0"` still keeps the loop running.

The <...> shortcut and how Perl picks it

`<*.c>` is an alternative syntax for `glob("*.c")`. The angle-bracket form shares a reader with `<FH>` (`readline`); the parser decides which one you meant at **compile time** based on what sits between the angle brackets:

- Bareword that could be a filehandle (`<STDIN>`, `<FH>`) → `readline`.
- A simple scalar (`<$fh>`) → `readline` on that handle.
- Anything that looks like a pattern (`<*.c>`, `<"*.txt">`, `<${dir}/*>`) → `glob`.

The named function `glob` is unambiguous and more searchable. Prefer it in new code; reach for `<...>` only for the shortest throwaway patterns.

```
my @txt = <"*.txt">;                  # same as glob '"*.txt"'
my @txt = glob "*.txt";               # recommended
```

Whitespace splits the pattern

`glob` splits its argument on whitespace and treats each segment as a separate pattern. The results are concatenated:

```
my @sources = glob "*.c *.h";       # all .c OR .h files
my @all     = glob ".* *";          # every entry in cwd
```

To glob a filename that itself contains spaces, wrap it in an inner pair of quotes so the split sees one segment:

```
my @spacey = glob '"*e f*"';        # files with "e f" in the name
my @spacey = glob q("*e f*");       # same, q() for clarity
my @spacey = <"*e f*">;             # same via the shortcut form
```

Interpolating a variable into a spacey pattern:

```
my @spacey = glob "'*${var}e f*'";
my @spacey = glob qq("*${var}e f*");
```

If the whitespace-splitting rule gets in the way, use `File::Glob`'s `bsd_glob` directly - it takes the pattern verbatim.

Brace expansion without a wildcard

If the only special characters are non-empty braces, `glob` does not touch the filesystem at all: it just enumerates the Cartesian product of the alternatives. This is useful for generating string combinations:

```
my @pairs = glob "{apple,tomato,cherry}={green,yellow,red}";
# apple=green, apple=yellow, apple=red,
# tomato=green, tomato=yellow, tomato=red,
# cherry=green, cherry=yellow, cherry=red
```

`File::Glob` - the engine, and its options

`glob` is implemented on top of `File::Glob`'s `bsd_glob`. Importing the module's tags tunes behaviour for the whole compilation unit:

- `:nocase` - case-insensitive matching.
- `:case` - force case-sensitive matching (the default on most Unix filesystems).
- `:globally` - replace the core `glob` operator with `bsd_glob` so every `glob(...)` and `<...>` in the file picks up your options.

```
use File::Glob qw(:globally :nocase);
my @txt = glob "readme*";           # README, readme.txt, Readme.md
```

Call `bsd_glob` directly when you need flags per call, or when you want to bypass the whitespace-splitting rule:

```
use File::Glob qw(bsd_glob GLOB_TILDE GLOB_BRACE);
my @files = bsd_glob("~/docs/*.{md,rst}", GLOB_TILDE | GLOB_BRACE);
```

Examples

List every Perl source file in the current directory:

```
my @perl_files = glob "*.pl *.pm";
```

Iterate MP3 files one at a time - useful when the list might be large and you do not want to materialise it:

```
while (my $song = glob "*.mp3") {
    process($song);                 # one filename per iteration
}
```

Recursive patterns are **not** built in; `*` does not cross `/`. Walk the tree with `File::Find` or combine `glob` with an explicit traversal:

```
my @all_c;
for my $dir (glob "src/*") {
    push @all_c, glob "$dir/*.c" if -d $dir;
}
```

Use the iterator form in a `while` condition - the result is assigned to `$_` and tested for definedness, so filenames like `"0"` do not terminate the loop:

```
while (<*.log>) {
    unlink $_ if -M $_ > 30;        # older than 30 days
}
```

Case-insensitive match for a whole file:

```
use File::Glob qw(:globally :nocase);
my @readmes = glob "readme*";
```

Edge cases

- **No matches**: returns the empty list. No warning, no `$!` set. If you need "pattern had to match", check `scalar @files` yourself.
- **`glob` with no argument**: uses `$_`. Bare `glob` inside `while (<>)` is rarely what you want - the pattern will be whatever line of input you just read.
- **`while (glob ...)` assigns to `$_` and tests definedness**: a filename `"0"` is false under normal boolean rules, but the loop still runs because the condition is `defined $_`. This matches the rule for `readline` in the same position.
- **Wildcards and whitespace**: the split-on-whitespace rule means `glob "a b"` searches for `a` and `b` as two patterns, not for the literal string `"a b"`. Use inner quotes or `bsd_glob` when the filename legitimately contains spaces.

- **Hidden files**: a leading `.` is not matched by `*` - you need an explicit `.*` segment (`glob ".* *"`) to include dotfiles. This matches shell glob semantics, not regex semantics.
- **Tilde expansion**: `glob "~/foo"` works because `File::Glob` expands `~` by default. Writing it through `bsd_glob` without `GLOB_TILDE` in the flags, you get a literal `~`.
- **Interpolation inside `<...>`**: `<$var/*.c>` interpolates `$var` at runtime. Using the angle-bracket form does not escape you from Perl's normal double-quote rules.
- **Angle-bracket ambiguity**: `<FOO>` is a handle read, `<$foo>` is a handle read, `<*.c>` is a glob. `<$foo/*.c>` - one scalar plus pattern characters - is a glob. The resolution is compile-time; runtime changes to `$foo` do not re-decide it.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `opendir` - open a directory handle for finer-grained traversal than pattern matching gives you
- `readdir` - read entries from a directory handle one at a time; pair with `opendir` when you need every entry (including ones `glob` hides, like dotfiles) or file-type info
- `readline` - the other meaning of `<...>`; shares the angle-bracket syntax but reads lines from a filehandle
- `File::Glob` - the underlying implementation; use `bsd_glob` directly for per-call flags and to bypass the whitespace-splitting rule
- `$_` - default pattern source for bare `glob`, and the variable implicitly assigned when `glob` is used as a `while`/`for` loop condition

Time

gmtime

Convert an epoch time to a broken-down UTC time, either as a 9-element list or as a `ctime(3)`-style string.

`gmtime` is `localtime` expressed in the Greenwich time zone. It takes an integer number of seconds since the epoch - the kind of value `time` returns - and hands back either the individual calendar fields (in list context) or a ready-to-print summary string (in scalar context). No time-zone lookup, no `TZ`, no DST: the result is always UTC.

Synopsis

```
gmtime EXPR
gmtime
```

`gmtime` without an argument uses the current `time`.

What you get back

In **list context**, a nine-element list drawn straight from the C `struct tm`:

```
#    0     1     2      3      4     5      6      7      8
my ($sec, $min, $hour, $mday, $mon, $year, $wday, $yday, $isdst)
    = gmtime(time);
```

Index	Field	Range	Notes
0	`$sec`	`0..60`	60 allows a positive leap second
1	`$min`	`0..59`	
2	`$hour`	`0..23`	
3	`$mday`	`1..31`	**1-based**
4	`$mon`	`0..11`	**0-based** - January is 0, December is 11
5	`$year`		**years since 1900** - add `1900` for AD
6	`$wday`	`0..6`	Sunday is 0
7	`$yday`	`0..365`	0 is January 1st; 365 only in leap years
8	`$isdst`	`0`	**always 0** for `gmtime` - UTC has no DST

In **scalar context**, a single `ctime(3)`-formatted English string:

```
my $s = gmtime;    # "Thu Oct 13 04:54:34 1994"
```

The scalar form is **always English and locale-independent**. For a locale-aware or differently-formatted string, build it from the list form with `POSIX::strftime`.

Global state it touches

None. `gmtime` does not read `$ENV{TZ}`, does not consult the interpreter's locale, and does not depend on or modify any of the Perl special variables. Given the same `EXPR`, it always returns the same result.

With no argument, it reads the current wall-clock time via `time`.

Examples

Current UTC broken down into its fields, with the month-0-index and year-since-1900 traps both handled:

```
my @abbr = qw(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec);
my ($sec, $min, $hour, $mday, $mon, $year) = gmtime;
printf "%s %d, %d  %02d:%02d:%02d UTC\n",
       $abbr[$mon], $mday, $year + 1900, $hour, $min, $sec;
# e.g. "Apr 23, 2026  14:07:09 UTC"
```

The scalar form - a single fixed-format string, useful for logs:

```
my $stamp = gmtime;
print "[$stamp] startup\n";
# "[Thu Apr 23 14:07:09 2026] startup"
```

Format the list with `POSIX::strftime` when you want ISO-8601 or a locale-aware rendering:

```
use POSIX qw(strftime);
my $iso = strftime "%Y-%m-%dT%H:%M:%SZ", gmtime;
# e.g. "2026-04-23T14:07:09Z"
```

Convert a stored epoch (not the current time) - any integer seconds since 1970-01-01 UTC works:

```
my $epoch = 1_700_000_000;
my @t = gmtime($epoch);
printf "%04d-%02d-%02d\n", $t[5] + 1900, $t[4] + 1, $t[3];
# "2023-11-14"
```

`$isdst` is always `0` under `gmtime` - do not use it to probe DST. Call `localtime` if you need that information:

```
my $dst_here = (localtime)[8];   # 0 or 1 depending on local zone
my $dst_utc  = (gmtime)[8];      # always 0
```

Edge cases

- **Bare call uses current time**: `gmtime` with no argument is `gmtime(time)`. Inside interpolation or a list that might swallow the call, parenthesise: `gmtime()`.
- **List vs scalar context is load-bearing**. In list context you get nine numbers; in scalar context you get a string. Forcing scalar on a bare call inside a list:

  ```
  print gmtime, "\n";                # nine digits, no spaces
  print scalar gmtime, "\n";         # "Thu Apr 23 14:07:09 2026"
  ```

- **`$mon` is 0-indexed, `$year` is since-1900**. Two of the most common Perl bugs. For a full calendar year do `$year + 1900`; for a 1-indexed month do `$mon + 1`.
- **`$mday` is 1-indexed, `$yday` is 0-indexed**. Mirrors C `struct tm` exactly - inconsistent, but stable.
- **`$sec == 60`** on a positive leap second. Code that validates with `$sec <= 59` will reject legitimate values.
- **Negative epoch values** represent dates before 1970-01-01 UTC and are accepted on platforms whose `gmtime(3)` accepts them - which Linux does. `gmtime(-1)` returns 1969-12-31 23:59:59 UTC.
- **Fractional seconds are discarded**. `EXPR` is truncated toward zero before the conversion; `gmtime(1.9)` is `gmtime(1)`. For sub-second precision, keep your own fractional part and format it separately.
- **No time-zone abbreviation in the scalar form**. Unlike `localtime`'s `ctime(3)` output on some platforms, `gmtime`'s scalar form never includes a zone name. It is always the five-field `"Wdy Mon DD HH:MM:SS YYYY"` shape.
- **`$isdst` is always `0`**, not `undef` and not `-1`. Do not test it for truth under `gmtime`; the answer is meaningless.

Alternatives and companions

- **Going the other way** - from broken-down fields back to an epoch - is the job of the `Time::Local` module. Use `timegm` (UTC) or `timelocal` (local zone) there:

```
use Time::Local qw(timegm);
my $epoch = timegm($sec, $min, $hour, $mday, $mon, $year);
```

- **Object-oriented access by field name** is offered by `Time::Piece`, which overloads `localtime` and `gmtime` to return objects with `->year`, `->mon`, `->mday` etc. The field conventions there are adjusted - `year` is already the full year, `mon` is already 1-indexed - so code written against `Time::Piece` cannot be mixed casually with code written against the built-in list form.
- **Human-readable formatting** belongs to `POSIX::strftime`; feed it the list returned by `gmtime` directly.
- **Comprehensive date arithmetic, time zones, durations** - reach for `DateTime` on CPAN. `gmtime` is the minimum viable primitive; `DateTime` is the full library.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `localtime` - same conversion, but in the local time zone; `$isdst` there is meaningful
- `time` - produces the epoch value `gmtime` consumes
- `POSIX::strftime` - format the 9-element list into an arbitrary date string (ISO-8601, locale-aware, etc.)
- `Time::Local` - inverse operation: build an epoch from broken-down UTC or local fields
- `Time::Piece` - object wrapper around `gmtime` / `localtime` with named accessors

Control flow

goto

Transfer execution elsewhere in the program without returning.

`goto` has three unrelated forms that share only a keyword. The useful one in modern Perl is `goto &NAME` - the **tail-call** form, which replaces the current subroutine frame with a call to another sub while preserving `@_`. The `goto LABEL` and `goto EXPR` forms exist, are constrained, and are almost never the right tool.

Synopsis

```
goto LABEL
goto EXPR
goto &NAME
```

What you get back

`goto` does not return. Control leaves the `goto` statement and does not come back. The return value of the construct the `goto` eventually lands in is the value of the enclosing expression.

For `goto &NAME`, the return value is whatever the target subroutine returns, propagated to the original caller as if the target had been called directly.

The tail-call form: `goto &NAME`

This is the form that matters. It exits the current subroutine - undoing any `local` bindings - and immediately calls `NAME` with the current `@_`. The current frame is **replaced**, not stacked on top of.

Consequences:

- `caller` inside the target sub sees the **original caller**, not the sub that issued the `goto`. The intermediate frame is gone.
- `@_` is passed through as-is. Modifications to `@_` before the `goto` are visible to the target.
- `local` changes made in the departing sub are undone before the target runs, exactly as they would be on a normal return.
- The target does not have to know it was reached via `goto &`; from its perspective it was called normally.

The canonical use is `AUTOLOAD` dispatch: resolve the real sub, then `goto` it so stack traces, `caller`, and `wantarray` all behave as though the real sub had been called in the first place.

`NAME` need not be a literal name. It can be any expression that yields a code reference - a scalar holding a coderef, a block returning one, or a dereferenced slot.

```
goto &$coderef;
goto &{ $dispatch{$op} };
```

For recursive tail calls, `goto __SUB__` jumps to the currently executing sub without re-resolving its name - safe under renaming and anonymous subs.

The `goto LABEL` form

Finds the statement labeled `LABEL` within the current dynamic scope and resumes execution there. In practice this means it can jump out of blocks and out of subroutines, but it **cannot jump into** a construct that requires initialization - a subroutine body, a `foreach` loop, a `given` block, the parameter of a binary or list operator, or code the optimizer removed. Jumping into such a construct is a fatal error as of Perl 5.44.

Most of what `goto LABEL` could do is better expressed with labeled loop control (`last LABEL`, `next LABEL`, `redo LABEL`) or with exceptions (`die` / `eval`). Reach for those first.

The `goto EXPR` form

Evaluates `EXPR`. If it yields a code reference, behaves like `goto &NAME`. If it yields a string, that string is used as a label name resolved at runtime - a computed goto. The same restrictions as `goto LABEL` apply to the label case.

Two parser quirks to remember:

- `goto EXPR` is exempt from the "looks like a function" rule. Parentheses after `goto` do not necessarily delimit its argument: `goto("NE")."XT"` is `goto NEXT`, not `goto "NE"` followed by a useless concatenation.
- `goto` has the same precedence as assignment, unlike most named operators.

Examples

Tail-call dispatch from `AUTOLOAD` - the target sub sees the original caller, not `AUTOLOAD`:

```
sub AUTOLOAD {
    our $AUTOLOAD;
    my ($method) = $AUTOLOAD =~ /::(\w+)\z/;
    my $code = $self->can_really($method)
        or die "no such method: $method";
    goto &$code;              # @_ forwarded, caller() unchanged
}
```

Tail recursion without growing the call stack. `__SUB__` resolves to the currently running sub:

```
sub fact {
    my ($n, $acc) = @_;
    $acc //= 1;
    return $acc if $n <= 1;
    @_ = ($n - 1, $acc * $n);
    goto __SUB__;
}
```

Computed `goto` via `goto EXPR`. The label is chosen at runtime from a list:

```
goto ("FOO", "BAR", "GLARCH")[$i];
FOO:    handle_foo();    return;
BAR:    handle_bar();    return;
GLARCH: handle_glarch(); return;
```

Delegating to a coderef held in a dispatch table. The caller never sees the intermediate frame:

```
my %ops = (
    add => sub { $_[0] + $_[1] },
```

(continues on next page)

(continued from previous page)

```
    mul => sub { $_[0] * $_[1] },
);

sub run_op {
    my $op = shift;
    goto &{ $ops{$op} // die "unknown op: $op" };
}
```

Modifying `@_` before a tail call - changes are visible to the target:

```
sub wrap {
    unshift @_, "prefix";
    goto &real_handler;       # real_handler sees ("prefix", @orig_
↪args)
}
```

Edge cases

- **`goto LABEL` cannot enter a sub, `foreach`, or `given`.** As of Perl 5.44 this is a fatal error, not a warning. It also cannot jump into optimized-away code or into most operator operands - the one exception is the first operand of a binary operator (for `=`, the right-hand side).
- **`goto LABEL` cannot get out of a `sort` comparison block.** The comparator runs in a context that does not permit the jump.
- **`goto &NAME` discards `local` bindings** from the departing sub before the target runs. Anything the sub localized is restored as though the sub had returned normally.
- **`caller` is invisible past a `goto &`.** The departing frame is gone. This is the whole point of the form for `AUTOLOAD` - do not rely on seeing the intermediate sub in a stack trace.
- **`@_` aliasing is preserved.** The target sub's `@_` aliases the same scalars the departing sub's `@_` did. Assigning to `$_[0]` in the target still modifies the caller's argument.
- **`goto &$coderef` requires a real coderef.** A string that looks like a sub name does not work in the `&` form - use `goto EXPR` (without the `&`) for that, or resolve the name to a coderef first with `\&{$name}`.
- **`goto` has assignment precedence.** `goto "A" . "B"` is `goto "AB"`, not `(goto "A") . "B"`.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `last` - exit the innermost or a named enclosing loop; the right tool for most "jump out of here" cases
- `next` - skip to the next iteration of a loop, with optional label
- `redo` - restart the current loop iteration without re-evaluating the condition
- `return` - leave the current sub with a value; compare with `goto &NAME` which leaves the sub by **replacing** it
- `caller` - inspect the call stack; note that `goto &` makes the departing frame invisible to it

Lists

grep

Filter a list to the elements where the block or expression is true.

`grep` walks `LIST` from left to right, evaluates `BLOCK` or `EXPR` once per element with `$_` aliased to that element, and returns the elements for which the test produced a true value. It is a list filter, not a loop - the list you get back is a subset of the list you passed in, in original order.

Synopsis

```
grep BLOCK LIST
grep EXPR, LIST
my @kept  = grep { COND } @list;
my @match = grep /pattern/, @list;
```

The two forms are interchangeable in power; `BLOCK` is the general case, and `EXPR` is a convenience for one-expression tests. When `EXPR` is a bare regex (`/.../`, `m/.../`, or `qr/.../`), it is implicitly matched against `$_` - this is the idiom behind `grep /^#/, @lines`.

What you get back

- **List context**: the filtered list, in original order, with each returned element an alias into the source list (see *Global state* and *Edge cases*).
- **Scalar context**: the count of elements for which the test was true. Not a boolean - it is the size of what the list context would have produced, so `0` is false, any positive count is true.

```
my @keep  = grep { $_ > 0 } @nums;   # list context: filtered list
my $count = grep { $_ > 0 } @nums;   # scalar context: count
```

Global state it touches

- `$_` is **aliased** to each element of `LIST` for the duration of one `BLOCK` or `EXPR` evaluation. On exit from `grep`, `$_` is restored to whatever it held before the call. Because the alias is to the real element - not a copy - assigning to `$_` inside the block mutates the source list in place. This is the same contract as `map` and as a `for` loop's index variable.
- The bare-regex form of `EXPR` reads and writes all the usual match-related magic variables (`$1`..`$9`, `$&`, `${^MATCH}`, `@+`, `@-`, `%+`, `%-`, `pos`) on each iteration, exactly as if you had written the match longhand.

Examples

Filter out undefined values:

```
my @vals    = (1, undef, 2, undef, 3);
my @defined = grep { defined $_ } @vals;
# @defined = (1, 2, 3)
```

Regex filter via the expression form - the bare regex matches against `$_`:

```
my @lines    = ("# comment", "code", "# another", "more code");
my @code     = grep !/^#/, @lines;       # drop comments
my @comments = grep  /^#/, @lines;       # keep comments
```

Custom test via the block form, when the condition is more than one expression:

```
my @users    = (
    { name => "alice", active => 1, age => 30 },
    { name => "bob",   active => 0, age => 25 },
    { name => "carol", active => 1, age => 17 },
);
my @adults_active = grep {
    $_->{active} && $_->{age} >= 18
} @users;
```

Count without building the list - use scalar context:

```
my @errors = ("ok", "fail", "ok", "fail", "fail");
my $nfail  = grep { $_ eq "fail" } @errors;   # 3
if (grep { $_ eq "fail" } @errors) {          # boolean use
    warn "had failures";
}
```

Remove blank and whitespace-only lines:

```
my @clean = grep { /\S/ } @lines;
```

Filter the keys of a hash by some property of the value:

```
my %scores  = (alice => 42, bob => 17, carol => 95);
my @passing = grep { $scores{$_} >= 50 } keys %scores;
```

Edge cases

- **`$_` is an alias, not a copy.** Modifying `$_` inside the block modifies the original list element. This is occasionally useful but more often a bug; write a read-only test:

```
my @bumped = grep { $_++; $_ > 10 } @nums;   # MUTATES @nums
```

  If the elements are themselves not variables (e.g. literals, constants, or the result of a function call), assigning to `$_` is a runtime error - "Modification of a read-only value".

- **Returned elements are aliases too.** Elements of the returned list share storage with the source list, just like the index variable of a `for` loop. Writing to them through the returned list propagates back:

```
my @arr  = (1, 2, 3, 4);
my @even = grep { $_ % 2 == 0 } @arr;
$even[0] = 99;      # @arr is now (1, 99, 3, 4)
```

  This is rarely wanted. Copy explicitly with `map { $_ } grep ...` or a fresh `my` when you need independent storage.

- **Expression form vs. block form - precedence.** `grep EXPR, LIST` uses a comma to separate the test from the list; `grep BLOCK LIST` does not. Mixing them trips you up:

```
grep /x/, @arr;          # EXPR form, correct
grep { /x/ } @arr;       # BLOCK form, correct
grep { /x/ }, @arr;      # WRONG: comma after block is a syntax
                         # error or silently wrong depending on
                         # context
```

- **grep in chained expressions.** `grep` takes a `LIST`, which means everything after the first argument is consumed as part of the list. Parenthesise when you want to follow `grep` with more terms:

```
my @out = (grep { /x/ } @a), @b;   # concatenates filtered @a
 ↪and @b
my @bad =  grep { /x/ } @a,  @b;   # filters BOTH @a and @b
 ↪together
```

- **Not a loop.** `grep` is an expression, not a control structure. `last`, `next`, and `redo` inside the block target the nearest enclosing *loop*, not the `grep` itself - they will exit or restart whatever `for`/`while` the `grep` sits inside, not the `grep`. Use an early `return` from a helper sub if you need to short-circuit, or switch to `List::Util` `first` when all you want is the first match.

- **Empty input is empty output.** `grep` over an empty list is an empty list in list context and `0` in scalar context. No special case needed.

- **Hash flattening.** `grep { ... } %h` walks the hash's alternating key/value flattening; the block sees keys and values interleaved in `$_`. This is almost never what you want; filter the keys and rebuild instead:

```
my %filtered = map  { $_ => $h{$_} }
               grep { some_test($_, $h{$_}) } keys %h;
```

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `map` - same iteration contract with the same `$_` alias semantics, but returns what the block *produced* for each element rather than the element itself
- `sort` - reorder a list; often chained with `grep` as `sort { ... } grep { ... } @list`
- `any` / `all` - short-circuiting boolean tests over a list from `List::Util`; use these instead of `scalar grep` when you only need a yes/no answer and the list is large or the test is expensive
- `first` (from `List::Util`) - return the first element matching a block and stop; the right tool when `grep` is being used only to fetch one match
- `for` / `foreach` - the looping construct with the same `$_` aliasing rule; reach for it when you want side effects, not a filtered list
- `$_` - the default scalar that `grep` aliases on every iteration

SCALARs and strings · Numeric functions

hex

Interpret a string as a hexadecimal numeral and return its numeric value.

`hex` reads `EXPR` as a sequence of hex digits - optionally introduced by a `0x` or `x` prefix - and returns the integer those digits represent. If `EXPR` is omitted, `$_` is used. The companion operation, turning a number back into a hex string, is `sprintf` with the `"%x"` or `"%X"` format.

Synopsis

```
hex EXPR
hex                 # operates on $_
```

What you get back

A non-negative number. `hex` itself never returns a negative value: it has no notion of sign, it only decodes digits.

- If the result fits in a native integer (typically 64-bit unsigned on a 64-bit build), you get an integer.

- If the decoded value exceeds that range, `hex` falls back to an IEEE 754 double. A warning is emitted under `use warnings` and precision may be lost for values above `2**53`.

To convert the other direction:

```
sprintf "%x", 255            # "ff"
sprintf "0x%08X", 4096       # "0x00001000"
```

Examples

Plain hex digits, with or without the `0x` prefix:

```
print hex '0xff';            # 255
print hex 'ff';              # 255
print hex 'DEADBEEF';        # 3735928559
```

Underscores between digits are permitted as visual separators, the same way numeric literals allow them:

```
print hex '1_0000';          # 65536
print hex 'cafe_f00d';       # 3405700109
```

Operating on `$_`:

```
for ('10', '20', 'ff') {
    print hex, "\n";         # 16, 32, 255
}
```

Round-tripping a value through hex:

```
my $n = 0xC0FFEE;
my $s = sprintf "%x", $n;    # "c0ffee"
print hex $s;                # 12648430
```

Values that overflow native integer width warn and return a float:

```
use warnings;
my $big = hex 'fffffffffffffffff';   # 19 'f's - too wide for u64
# "Integer overflow in hexadecimal number" warning
# $big is now an IEEE 754 double, with lost precision
```

Edge cases

- **Non-hex characters terminate parsing.** The first character that is not a hex digit, underscore, or recognised prefix ends the scan. Under `use warnings` this emits `Illegal hexadecimal digit '…' ignored`:

  ```
  use warnings;
  print hex '1f garbage';    # 31, with a warning
  print hex '0xAZ';          # 10, with a warning on 'Z'
  ```

- **Leading whitespace is not skipped.** Unlike `oct`, `hex` treats whitespace as a non-hex character, so a leading space stops parsing before any digit is seen:

```
print hex ' ff';          # 0
```

- **Empty string and `undef` return `0`.** An empty `EXPR` yields `0` with no warning. A bare `undef` yields `0` and emits the usual `Use of uninitialized value` warning under `use warnings`.
- **No sign handling.** A leading `-` or `+` is a non-hex character and terminates parsing before any digit, so `hex '-ff'` returns `0`. To parse a signed hex string, strip the sign yourself:

```
my $s = '-ff';
my $n = ($s =~ s/^-//) ? -hex($s) : hex($s);   # -255
```

- **Prefix forms.** The accepted prefixes are `0x`, `0X`, `x`, and `X`. A lone `0` is not a prefix - it's a digit. There is no `#` or `$` prefix.
- **Case is irrelevant** for the digits `a-f` and for the prefix letter `x`.
- **Underscore placement.** A single underscore may precede each digit. Two consecutive underscores, a leading underscore before the prefix, or a trailing underscore terminate parsing as non-hex characters.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `oct` - decodes strings with auto-detected base: `0x...` as hex, `0b...` as binary, `0...` as octal, anything else as decimal. Use it when the input's base is not known in advance
- `sprintf` - the reverse direction; `"%x"` / `"%X"` format a number into a lowercase or uppercase hex string, `"%#x"` adds the `0x` prefix, `"%08x"` zero-pads to a fixed width
- `printf` - same `"%x"` formatting, written straight to a filehandle
- `pack` - templates `H` (high nibble first) and `h` (low nibble first) convert between a hex-digit string and packed binary bytes; use these when moving hex data to and from a binary protocol
- `unpack` - the inverse of `pack`; with `"H*"` it turns a byte string into a hex digit string in one call

Scoping · Modules

import

Populate the caller's namespace with names a module chooses to export.

`import` is **not** a built-in function. It is a class method a module defines so that `use` has somewhere to hand off the argument list. When you write `use Module LIST`, Perl loads the module and then calls `Module->import(LIST)` at compile time, in the package that

contains the `use` statement. Whatever that method does - install subroutines into the caller's stash, flip feature bits, set pragma flags - is the module's entire `import` contract.

Most modules never write `import` by hand; they inherit it from `Exporter` and declare `@EXPORT` / `@EXPORT_OK` instead. Pragmas (`strict`, `warnings`, `feature`) write their own `import` because what they install is lexical state, not subroutines.

Synopsis

```
Module->import;
Module->import(LIST);
Module->import(@symbols);
```

Inside a module, the definition looks like:

```
package My::Module;
sub import {
    my ($class, @args) = @_;
    # install names into caller's package
}
```

What you get back

The return value of `import` is discarded by `use`. A module that is called directly as a method inherits whatever convention the author chose - by convention, return nothing useful.

The visible effect is in the caller's package: after `use Module qw(foo bar)` returns, `foo` and `bar` are callable without a package prefix from the code that ran the `use`.

How `use` calls `import`

`use Module LIST` is, in effect:

```
BEGIN {
    require Module;
    Module->import(LIST);
}
```

Three consequences fall out of that equivalence:

- **Compile time.** `import` runs while the enclosing file is still being parsed. Any subroutine prototypes or lexical pragmas it installs are visible to the rest of the file from that point on. Anything done at runtime - opening files, reading configuration - happens earlier than a naive reading of the source suggests.
- **`caller` reflects the `BEGIN` block**, not the surrounding subroutine. An `import` method that inspects `caller(0)` sees the package that wrote `use Module`, which is exactly what it wants; it does **not** see the subroutine that contains the `use` statement.
- **Absent list vs empty list.** `use Module` with no list calls `Module->import` with no arguments - the module installs its defaults. `use Module ()` with an **explicit**

empty list skips the `import` call entirely. This is the idiom for "load the file but do not touch my namespace."

`no Module LIST` is the mirror image: it calls `Module->unimport(LIST)` instead of `import`. Pragmas use this to turn features off for the enclosing lexical scope.

Defining an `import` method

The common path is inheritance from `Exporter`:

```
package My::Utils;
use Exporter 'import';                  # installs Exporter::import
our @EXPORT_OK = qw(slurp trim);
sub slurp { ... }
sub trim  { ... }
1;
```

Callers then write:

```
use My::Utils qw(slurp trim);
slurp($path);
```

A hand-rolled `import` has the same shape as any method - it receives the class name as its first argument and the `use` argument list as the rest:

```
package My::Feature;
sub import {
    my ($class, @flags) = @_;
    my $caller = caller;
    no strict 'refs';
    for my $name (@flags) {
        *{ "${caller}::${name}" } = \&{ "${class}::${name}" };
    }
}
```

The caller's package name comes from `caller`, not from `$class` - `$class` is whatever module the user named on the `use` line.

Examples

Default import - whatever the module's `@EXPORT` list contains:

```
use List::Util;                         # imports nothing by default
use File::Basename;                     # imports basename, dirname,␣
↪...
```

Explicit import list - only the names you ask for:

```
use List::Util qw(sum min max);
my $total = sum(1, 2, 3);               # 6
```

Empty list - load the file, touch nothing:

```
use POSIX ();                           # no POSIX::* names imported
my $pi = POSIX::acos(-1);               # fully qualified call
```

Importing from a pragma - lexical effect in the enclosing block:

```
use strict;
use warnings;
{
    no warnings 'uninitialized';        # unimport in this block only
    print "x=", $x, "\n";
}
```

Calling `import` directly after a runtime `require`:

```
require My::Plugin;
My::Plugin->import(@config);            # deferred to runtime on␣
↪purpose
```

This pattern loads the module only when a code path actually needs it, trading compile-time checking for startup speed.

Edge cases

- **No such thing as a built-in `import`.** `print import(@list)` calls the subroutine named `import` in the current package (if one exists); it is not a language-level operator. Parsing rules apply as for any user-defined subroutine.
- **Omitted list vs empty list** is the most common source of confusion. `use Mod` calls `import`; `use Mod ()` does not. The `()` is not a stylistic empty - it is a signal.
- **Version argument runs `VERSION`, not `import`.** `use Mod 1.23 LIST` calls `Mod->VERSION(1.23)` **before** `Mod->import(LIST)`. A bare `use Mod 1.23` is still a version check followed by a default `import`. To version-check and skip imports, write `use Mod 1.23 ()`.
- **No `import` defined at all.** If the module neither writes nor inherits an `import`, `use Module LIST` with a non-empty list does not error - the method call falls through silently as `UNIVERSAL::can('import')` returns false and `use` treats it as "nothing to do." Non-empty `LIST` is then effectively ignored, which is almost never what the caller wanted.
- **`import` runs once per `use` statement**, not once per process. `use Mod qw(foo); use Mod qw(bar);` calls `import` twice, even though the file is loaded only once - `@INC` / `%INC` gate the file load, not the method call.
- **Symbol installation is not automatic.** Writing an `import` method that merely returns does nothing; you must assign into the caller's stash (`*{"${caller}::name"} = \&sub`) or delegate to `Exporter`.
- **Pragmas typically install lexical state**, not package-level subroutines. Their `import` calls functions like `warnings::import_into` or manipulates `%^H`; this is why a pragma's effect is scoped to the enclosing block rather than persisting in the package.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `use` - the compile-time loader that invokes `import`; understanding one requires the other
- `no` - the mirror of `use`; calls `unimport` with the same argument-passing rules
- `require` - loads the file without calling `import`; pair with an explicit `Module->import(...)` when you want to defer or skip the compile-time dispatch
- `package` - defines the namespace that an `import` method populates from the caller's point of view
- `caller` - how an `import` method discovers which package to install names into
- `bless` - unrelated to `import`, but often invoked in the same module files; both shape how a package presents itself to its users

SCALARs and strings

index

Find the position of a substring inside a string.

`index` scans `STR` left-to-right looking for the first occurrence of `SUBSTR` and returns the zero-based position where it starts. No regular-expression metacharacters, no case folding, no wildcards - `SUBSTR` is matched literally, character for character. When the search fails, `index` returns `-1`.

Synopsis

```
index STR, SUBSTR
index STR, SUBSTR, POSITION
```

What you get back

An integer. On a match, the zero-based offset of the first character of `SUBSTR` inside `STR`. On no match, `-1`. The sentinel is the idiomatic way to test:

```
if (index($line, $needle) >= 0) { ... }   # found
if (index($line, $needle) == -1) { ... }  # not found
```

`POSITION` and the return value use the **same** zero-based scale, so you can feed one back into the other to walk every occurrence:

```
my $pos = -1;
while (($pos = index($text, $needle, $pos + 1)) != -1) {
    push @hits, $pos;
}
```

How POSITION is interpreted

`POSITION` is the earliest offset the match is allowed to start at. The search still proceeds to the end of `STR`; `POSITION` does not bound the search, it only shifts where it begins.

- `POSITION` omitted or `undef` - search from offset `0`.
- `POSITION` negative or otherwise before the start - treated as `0`.
- `POSITION` past the end of `STR` - treated as the end, so the only way to match is if `SUBSTR` is the empty string (which matches at any offset, including the end).

An empty `SUBSTR` always matches, and matches at `POSITION` (clamped into range). This follows from the "first position where `SUBSTR` occurs" rule: the empty string occurs everywhere.

```
index("hello", "");      # 0
index("hello", "", 3);   # 3
index("hello", "", 99);  # 5   (clamped to end of string)
```

Examples

Find a single character or a whole word:

```
index("Perl is great", "P");     # 0
index("Perl is great", "g");     # 8
index("Perl is great", "great"); # 8
```

Report a miss:

```
index("Perl is great", "Z");     # -1
```

Skip past an earlier match with `POSITION` to find the second occurrence:

```
index("Perl is great", "e", 5);  # 10
```

Walk every occurrence of a substring:

```
my $s = "abcabcabc";
my $p = -1;
while (($p = index($s, "bc", $p + 1)) != -1) {
    print "hit at $p\n";
}
# hit at 1
# hit at 4
# hit at 7
```

A common idiom - test for containment without building a regex:

```
if (index($path, "/tmp/") != -1) {
    warn "path touches /tmp";
}
```

Pairs naturally with `substr` to split on the first occurrence of a separator:

```
my $line = "key=value=with=equals";
my $eq   = index($line, "=");
my ($k, $v) = $eq >= 0
    ? (substr($line, 0, $eq), substr($line, $eq + 1))
    : ($line, undef);
```

Edge cases

- **Empty `SUBSTR`** matches at `POSITION` (clamped into `STR`). `index($s, "")` is `0`; `index($s, "", $n)` is `$n` capped at `length $s`. Never `-1`.
- **Empty `STR` with non-empty `SUBSTR`** returns `-1`. Empty `STR` with empty `SUBSTR` returns `0`.
- **Negative `POSITION`** is clamped to `0`. `index` does **not** interpret negative offsets as "from the end" - that is `rindex`'s job, and even there the semantics differ.
- **`POSITION` past the end of `STR`** is clamped to `length STR`, so only an empty `SUBSTR` can match.
- **Undef arguments** stringify to `""` and trigger an `uninitialized` warning under `use warnings`. `index(undef, "x")` is `-1`; `index("abc", undef)` is `0` (empty-substring rule).
- **Characters, not bytes.** `index` operates on the logical character sequence of the string. For a string of wide characters, the returned offset is a character offset, not a byte offset. If you need byte offsets, downgrade or encode the string first (`use bytes` for a lexical byte view, or `Encode::encode_utf8` to work on an octet string).
- **Case sensitivity**: `index` is case-sensitive. Lowercase both arguments first if you want a case-insensitive search, or use `=~ /\Q$needle\E/i` and `@-` / `$-[0]` to recover the position.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `rindex` - same matching rules, scans from the right and returns the offset of the **last** occurrence at or before `POSITION`
- `substr` - extract the matched region once `index` has located it, or replace it in place
- `length` - upper bound for a valid `POSITION`; returns character length on the same scale `index` uses
- `pos` - position tracking for regex-based scanning; use together with `m//g` when you need captures rather than a raw offset
- `sprintf` - build the search string when `SUBSTR` is assembled from parts; `index` takes a literal, so prepare it first

Numeric functions

int

Return the integer portion of a number, truncating toward zero.

`int` takes a single scalar, forces it to numeric context, and drops its fractional part. The discard is always **toward zero** - not toward negative infinity - so `int(3.9)` is `3` and `int(-3.9)` is `-3`, not `-4`. Non-numeric strings are coerced with the usual Perl numeric-conversion rules (leading whitespace, optional sign, digits; the rest is discarded, with an `Argument "..." isn't numeric` warning under `use warnings`). If `EXPR` is omitted, `int` operates on `$_`.

Synopsis

```
int EXPR
int
int($x)
```

What you get back

A number with no fractional part, same sign as the input. The result is an integer value if the truncated magnitude fits in the platform's native integer range (typically 64-bit); otherwise Perl keeps it as a floating-point value whose fractional part happens to be zero. Either way it compares and prints as an integer.

```
my $n = int(7.8);           # 7
my $m = int(-7.8);          # -7
```

`int` is not a rounding function. For rounding, see *See also*.

Examples

Positive truncation - the fractional part is discarded:

```
print int(3.9), "\n";               # 3
print int(3.1), "\n";               # 3
print int(3.0), "\n";               # 3
```

Negative truncation - the trap. `int` truncates toward zero, so `int(-3.9)` is `-3`, which is *larger* than `-3.9`. This is the opposite of a mathematical floor:

```
print int(-3.9), "\n";              # -3   (not -4)
print int(-3.1), "\n";              # -3
```

Works on `$_` when called with no arguments - handy inside `map` or `while (<>)`:

```
my @whole = map { int } @floats;    # truncate every element
```

Strings are coerced, keeping the leading numeric prefix:

```
print int("42.7 bottles"), "\n";    # 42
print int("  -5.5xyz"), "\n";       # -5
```

(continues on next page)

(continued from previous page)

```
print int("hello"), "\n";           # 0    (with a warning under␣
↪`use warnings`)
```

Division producing a fraction, then truncated - a common idiom for integer division of signed values, **with the caveat above about negatives**:

```
my $pages = int($total_items / $per_page);   # floor only for non-
↪negative $total_items
```

The classic floating-point surprise. `-6.725 / 0.025` is *mathematically* `-269`, but the IEEE-754 result is slightly greater than `-269` (around `-268.99999999999994`). `int` truncates toward zero, giving `-268`:

```
print int(-6.725 / 0.025), "\n";    # -268, not -269
```

This is not an `int` bug - every truncation/rounding function has to cope with inputs like this. See *Edge cases* below.

Edge cases

- **`int` on `$_`**: with no argument, operates on `$_`. Inside `map { int }` this is what you want; inside a larger expression it can surprise a reader who missed the implicit argument. Prefer `int($x)` when the source variable isn't obvious from context.
- **Already-integer input**: `int(5)` is 5. No conversion, no warning, no precision loss.
- **Very large floats lose precision before `int` sees them**. A double-precision float can represent every integer up to `2**53` exactly; beyond that, consecutive representable values are spaced 2, then 4, then 8 apart. `int(1e20)` returns a float whose integer value is close to but not equal to `10**20`. If you need exact large integers, keep them as integers throughout - don't route them through floating point.
- **Floating-point representation artefacts**: inputs like `-6.725 / 0.025` (see *Examples*) don't land on the mathematical integer they "should." Use `sprintf "%.0f"` or `POSIX::floor`/`POSIX::ceil` if you need rounding behaviour that tolerates this.
- **Infinities and NaN**: `int('Inf')` returns `Inf`; `int('-Inf')` returns `-Inf`; `int('NaN')` returns `NaN`. `int` has no integer to truncate to, so it returns the special value unchanged. This matches every alternative (`sprintf "%d"`, `POSIX::floor`, `POSIX::ceil`) - none of them give you a finite integer here.
- **`undef`**: `int(undef)` returns `0` and emits a `Use of uninitialized value` warning under `use warnings`.
- **Non-numeric strings**: `int("abc")` returns `0` and emits `Argument "abc" isn't numeric` under `use warnings`. A string that starts with a numeric prefix (`"42abc"`) yields the prefix's integer value (42) with the same warning - `int` inherits this from Perl's standard numeric coercion, not from any custom parsing.
- **Booleans and dualvars**: `int` reads the numeric half. `int(!!1)` is 1; `int(!!0)` is `0`. For a dualvar whose string and numeric values disagree, `int` uses the numeric side.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `abs` - magnitude without sign; pair with `int` when you want the truncated magnitude regardless of sign
- `sprintf` - `sprintf "%d", $x` also truncates toward zero, but additionally formats the result as a string; use when you want the truncated value *and* a specific textual form
- `POSIX::floor` - round *toward negative infinity*; `floor(-3.9)` is `-4`, which is what most mathematical contexts (graphics, bucketing) actually want
- `POSIX::ceil` - round *toward positive infinity*; `ceil(-3.1)` is `-3`, `ceil(3.1)` is `4`; the mirror of `floor`
- `POSIX::round` - round to the nearest integer, halves away from zero; use this when you want statistical rounding rather than truncation
- `sprintf` with `"%.0f"` - round to nearest integer, halves to even (banker's rounding) on most platforms; subtly different from `POSIX::round` at half-values

Filehandles, files, directories

ioctl

Perform a device-control `ioctl(2)` system call on a filehandle.

`ioctl` is the direct Perl binding for the POSIX `ioctl(2)` syscall - the catch-all kernel interface for device-specific operations that do not fit the regular read/write model: terminal window size, serial line parameters, socket device queries, tape positioning, and anything else a driver chooses to expose through a request code. It dispatches `FUNCTION` against `FILEHANDLE`, passing `SCALAR` as the third argument - either a small integer or a pointer to a byte buffer, depending on what the specific `FUNCTION` expects.

Unlike `fcntl`, whose request codes live in the `Fcntl` module, `ioctl` request codes are almost never exported as ready-made constants. The usual path is:

```
require "sys/ioctl.ph";
```

which pulls in the translated `<sys/ioctl.h>` header. If that file does not exist, or does not define the code you need, build it from the C headers with the `h2ph` tool shipped with Perl, or define the constant yourself from the kernel header.

Synopsis

```
require "sys/ioctl.ph";
ioctl FILEHANDLE, FUNCTION, SCALAR
ioctl($fh, $request, $buf)
my $rv = ioctl($fh, $request, $arg) || -1;
```

What you get back

On success, the integer the kernel returned. When that integer is `0`, Perl substitutes the dual-valued string `"0 but true"` - true in boolean context, numeric `0` in numeric context, and exempt from the `Argument "..." isn't numeric` warning. This means a plain boolean test reliably distinguishes success from failure even for requests whose success value is literally zero:

```
ioctl($fh, $request, $buf) or die "ioctl: $!";
```

On failure, `ioctl` returns `undef` and sets `$!` to the relevant `errno`. The full mapping:

OS returns	Perl returns
`-1`	`undef`
`0`	string `"0 but true"`
any other value	that integer

When you need the raw kernel return - for request codes that encode information in the return value itself - use the idiom from `perlfunc`:

```
my $retval = ioctl($fh, $request, $arg) || -1;
printf "System returned %d\n", $retval;
```

How SCALAR is used

`SCALAR` is handed to the kernel as the third argument of the C `ioctl` call, and its role depends on the request code:

- **Pointer to a buffer.** The usual case. Most `ioctl` requests read from or write into a C struct whose layout is defined by the driver. Build the buffer with `pack` before the call and, for requests that write back, decode with `unpack` afterward. Pre-size the scalar to at least the struct's length - Perl will grow an undersized scalar automatically, but pre-sizing makes the intent obvious and avoids surprises if the driver writes more than you expected.
- **Small integer.** A minority of requests take a bare integer rather than a pointer. If `SCALAR` has no string value but does have a numeric value, Perl passes the number instead of a pointer. To guarantee this dispatch - even for a scalar that might have been stringified earlier - add `0` to it first:

  ```
  my $n = 0 + $n;       # force numeric representation
  ioctl($fh, $request, $n);
  ```

- **Ignored.** For request codes that take no third argument, pass `0` as a placeholder. The kernel ignores it; the convention keeps the call readable.

Global state it touches

- `$!` - set on failure to the `errno` returned by `ioctl(2)`.
- Device or descriptor state on `FILEHANDLE` - `ioctl` typically mutates kernel state associated with the underlying file descriptor or its device (terminal attributes, socket options, tape position). That state is not Perl-visible through any other variable; read it back with another `ioctl` call using the matching `TIOCG...` / `SIOCG...` request.

Examples

Query the terminal window size on Linux. `TIOCGWINSZ` writes a `struct winsize` (two `unsigned short` rows/cols plus two ignored fields) into the buffer:

```
require "sys/ioctl.ph";

my $winsize = "\0" x 8;
ioctl(STDOUT, TIOCGWINSZ(), $winsize)
    or die "TIOCGWINSZ: $!";
my ($rows, $cols) = unpack('S S', $winsize);
print "terminal is $rows rows by $cols cols\n";
```

Count bytes available for reading on a socket without consuming them (`FIONREAD` is widely supported across Unixes):

```
require "sys/ioctl.ph";

my $pending = pack('L', 0);
ioctl($sock, FIONREAD(), $pending)
    or die "FIONREAD: $!";
my $n = unpack('L', $pending);
print "$n bytes pending\n";
```

Put a socket into non-blocking mode through the `FIONBIO` request - an alternative to the `fcntl` / `F_SETFL` approach, and the form historically portable to some older systems:

```
require "sys/ioctl.ph";

my $on = pack('L', 1);
ioctl($sock, FIONBIO(), $on)
    or die "FIONBIO: $!";
```

Distinguishing "syscall returned 0" from "syscall failed" using the `"0 but true"` convention:

```
my $rv = ioctl($fh, $request, $buf);
if (!defined $rv) {
    die "ioctl failed: $!";
} elsif ($rv eq "0 but true") {
    # kernel returned 0 - success with a zero value
} else {
```

(continues on next page)

(continued from previous page)

```
    # kernel returned $rv
}
```

Rolling your own request code when `sys/ioctl.ph` is missing or incomplete. The encoding (`_IOR` / `_IOW` / `_IOWR`) is kernel-specific; on Linux the macro expands to a 32-bit value that you can precompute and hardcode:

```
use constant TIOCGWINSZ => 0x5413;   # Linux <asm-generic/ioctls.h>

my $winsize = "\0" x 8;
ioctl(STDOUT, TIOCGWINSZ, $winsize) or die $!;
```

Edge cases

- **Forgot the `.ph` require**: constants like `TIOCGWINSZ` parse as barewords with no defined value, triggering an `EINVAL` from the kernel. Either `require "sys/ioctl.ph"` or define the constant yourself with `use constant`.
- **`"0 but true"` return**: do not compare the return with `== 0` as a failure test. Use a plain boolean (`or die`) or `defined`. Comparing for equality to the literal string also works (`$rv eq "0 but true"`) but is rarely necessary.
- **Scalar/integer dispatch surprise**: a scalar that was numeric but got stringified - for instance via interpolation into a message - flips from integer-argument to buffer-pointer semantics on the next call. Force numeric representation with `0 + $x` when the request code expects an integer.
- **Struct layout is platform-specific.** `struct winsize`, `struct termios`, `struct ifreq`, and friends differ between kernels and even between architectures on the same kernel. Do not hardcode `pack` templates from memory; verify against the target system's header.
- **Undersized output buffer.** If the driver writes more bytes than `SCALAR` currently holds, the scalar is grown, but you have no hint in Perl that this happened. Pre-size to at least the struct's length with `"\0" x N` so that reads past the end of the buffer are driver bugs, not Perl bugs.
- **Not every system implements every request.** `ioctl` itself is POSIX, but specific request codes are driver-defined. An unsupported request fails with `ENOTTY` or `EINVAL`; an `ioctl` call on a system without `ioctl(2)` at all raises an exception.
- **`ioctl` vs `fcntl`**: identical argument-processing and return-value convention, different kernel interface. Use `fcntl` for descriptor flags and POSIX locks; use `ioctl` for device control codes.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `fcntl` - same calling and return-value convention; used for descriptor-level flags and advisory locks rather than device-specific control codes
- `pack` / `unpack` - required for building and decoding the struct buffers most `ioctl` requests exchange with the kernel
- `open` / `sysopen` - open the device or special file whose driver the `ioctl` request will talk to
- `binmode` - some `ioctl` requests (e.g. terminal mode changes) interact with PerlIO buffering; switch to raw mode when the driver and PerlIO disagree about line discipline
- `select` / `sysread` - the normal I/O path; reach for `ioctl` only when the operation you need is not expressible as a read or write

Classes and OO

isa

Test whether an object is an instance of a class or of any subclass derived from it.

`isa` is a **binary operator**, not a function. The left operand is an expression that should yield an object reference; the right operand names a class. It returns true when the left operand is a blessed reference whose class is - or inherits from - the class on the right. It returns false for anything else, including `undef`, unblessed references, and plain strings. It never throws.

Unlike the old `UNIVERSAL::isa` method, `isa` is safe to apply to arbitrary values: calling `$x->isa("Foo")` on a plain string that happens to name a class (`$x = "Foo"`) invokes the class method and returns true, which is almost never what the caller meant. `isa` as an operator only reports true for **blessed instances**.

Synopsis

```
use feature 'isa';          # or: use v5.36;

$obj isa Some::Class
$obj isa "Different::Class"
$obj isa $name_of_class
```

What you get back

1 for a blessed object whose class is, or derives from, `CLASS`. The defined empty string `""` (a false value that is numerically zero and exempt from `uninitialized` warnings) in every other case.

The operator never raises an exception: undefined, unreferenced, and misbranded values simply test false.

Feature gate

`isa` is available from Perl 5.31.6 and became non-experimental in Perl 5.36. It is a **lexical feature**, enabled by either of:

```
use feature 'isa';
use v5.36;                  # enables the 5.36 feature bundle
```

Under `no feature 'isa'` (or in a scope that never enabled it), the bareword `isa` reverts to being a normal identifier, and the expression is a syntax error.

The right-hand side

The right operand of `isa` is resolved to a class name at compile time when it is a bareword:

```
$obj isa Some::Class        # bareword - resolved at compile time
```

Any other expression is evaluated at runtime and must yield a string naming the class:

```
$obj isa "Different::Class" # string literal
$obj isa $class_name        # scalar variable
$obj isa (ref $other)       # any expression yielding a string
```

Barewords on the right side do **not** need to be declared, imported, or loaded; `isa` only consults the symbol table of `CLASS` to walk `@ISA`. A class that was never loaded simply has no parents, so `$obj isa Unknown::Class` just tests false.

Examples

Basic instance check:

```
use feature 'isa';

package Animal { }
package Dog    { our @ISA = ('Animal'); }

my $rex = bless {}, 'Dog';

$rex isa Dog;               # true
$rex isa Animal;            # true - via @ISA
$rex isa Cat;               # false
```

Gate a method call on the actual class:

```
sub describe {
    my ($thing) = @_;
    return "unknown" unless $thing isa Animal;
    return $thing->sound;
}
```

Safe on arbitrary values - no exception on non-references:

```
undef      isa Animal;      # false
"Animal"   isa Animal;      # false - a string is not an instance
[]         isa Animal;      # false - unblessed ref
{}         isa Animal;      # false - unblessed ref
\&main::x  isa Animal;      # false - unblessed code ref
```

Dispatch on a class name captured at runtime:

```
for my $class (qw(Dog Cat Fish)) {
    if ($rex isa $class) {
        say "rex is a $class";
    }
}
```

Why `isa` is preferred over `UNIVERSAL::isa` as a method:

```
my $x = "Dog";
$x->isa('Dog');             # TRUE - string "Dog" is treated as a␣
↪class name
$x isa Dog;                 # false - $x is not a blessed instance
```

The first form silently does the wrong thing on any string that happens to match a loaded class name. The operator form is type-aware and reports what the caller actually wanted to know.

Edge cases

- **Left operand is undef**: the operator returns false. No `uninitialized` warning is emitted - this is a deliberately quiet test.
- **Left operand is an unblessed reference**: false. `isa` requires a blessed reference; array refs, hash refs, and code refs that have never been passed through `bless` fail the test.
- **Left operand is a plain string**: false, even when the string matches a loaded class name. This is the behavioural split from `UNIVERSAL::isa` called as a method.
- **Right operand names an unknown class**: the operator does not autoload or require the module. It walks the class's `@ISA`; an unknown class has no `@ISA`, so the test is false.
- **Multiple inheritance**: `isa` honours whatever method resolution order is active in the class (`mro`, `c3` or the default depth-first search). The lookup is the same walk that method dispatch uses.
- **Reblessed objects**: the test reflects the current blessing. Calling `bless` again on a reference changes what `isa` reports.
- **Parser precedence**: `isa` is a non-associative named operator at the same precedence as the relational operators (`lt`, `gt`, `le`, `ge`). Chaining (`$a isa B isa C`) is a syntax error, as with the other relationals.
- **Feature not enabled**: without `use feature 'isa'` (or `use v5.36`), the parser sees `isa` as an ordinary bareword and the expression fails to compile.

- **Right operand as a subroutine call**: write it in parens. `$obj isa some_class()` parses; `$obj isa some_class` treats `some_class` as the literal class name (bareword).

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `bless` - attach a class name to a reference so that `isa` (and method dispatch) has something to report
- `ref` - retrieve the class name of a blessed reference, or the underlying type for unblessed ones; complements `isa` when you need the name rather than a yes/no answer
- `class` - the native class declaration syntax (feature `class`, Perl 5.38+); objects built with `class` are ordinary blessed references and work with `isa` unchanged
- `UNIVERSAL::isa` - the historical method form. Works on every reference, but also returns true for plain strings that happen to name a class; use the operator in new code
- `mro` - controls the method resolution order that `isa` walks through `@ISA`
- `perlop` - full operator-precedence table; `isa` sits with the non-associative relational operators

Lists

join

Concatenate a list of strings with a separator between each adjacent pair, and return the single resulting string.

`join` takes a separator as its first argument and any number of values as the rest. It stringifies every value in `LIST`, glues them together with `EXPR` placed between each pair, and returns one scalar. The separator is inserted **between** items only - never at the start, never at the end. With zero or one items the separator never appears at all.

Synopsis

```
join $sep, @list
join "", @chars
join "\n", @lines
```

What you get back

A single string. The result is a fresh scalar; modifying it does not reach back into `LIST`. In scalar context you get that string; `join` is effectively always scalar because it only ever produces one value.

Examples

Comma-separated values, the canonical use:

```
my @cols = ("alice", 42, "ops");
my $row  = join ",", @cols;          # "alice,42,ops"
```

Build a file path with `/` as the separator:

```
my @parts = ("var", "log", "app.log");
my $path  = "/" . join("/", @parts);  # "/var/log/app.log"
```

Join lines with a newline between each - note that the trailing line does **not** get a newline, because `join` puts the separator *between* items:

```
my @lines = ("first", "second", "third");
my $text  = join "\n", @lines;       # "first\nsecond\nthird"
```

No-separator concatenation. Use the empty string `""`, not `undef` - `undef` triggers an uninitialized warning under `use warnings` and stringifies to `""` anyway:

```
my @chars = ("h", "e", "l", "l", "o");
my $word  = join "", @chars;         # "hello"
```

An empty `LIST` yields an empty string - this is safe to rely on and needs no guard:

```
my @empty;
my $s = join ",", @empty;            # ""
```

Edge cases

- **Empty `LIST`**: returns `""`. The separator is never used.
- **Single-item `LIST`**: returns that item stringified, with no separator anywhere: `join ",", ("only")` is `"only"`.
- **Separator is stringified**: pass a number, a reference, an object with overloaded `""` - `join` calls the same stringification every other Perl operator would. `join 0, 1, 2, 3` is `"10203"`.
- **Multi-character separator**: `EXPR` can be any string, not just a single character. `join ", ", @cols` gives `"a, b, c"`.
- **`undef` items in `LIST`**: each `undef` stringifies to `""` and emits an `uninitialized` warning under `use warnings`. Two adjacent `undef`s still get a separator between them: `join ",", (undef, undef)` is `","`.
- **`join` is not the inverse of `split` on a single character** when the separator you pass differs from the pattern used to split. `split` takes a *pattern*; `join` takes a literal string. Round-tripping `join "\t", split /\s+/, $s` loses the distinction between runs of whitespace and a single tab.
- **Not a pattern**: unlike `split`, the first argument to `join` is not a regex. `join /,/, @x` is a syntax surprise - the separator is always a string.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `split` - the inverse: take a string and a pattern, return a list of pieces
- `sprintf` - when you need per-item formatting, padding, or field widths that `join` can't express
- `pack` - when the output is a binary record, not a separator-delimited text string
- `map` - transform items before joining, e.g. `join ",", map { uc } @names`

Arrays · Hashes

keys

List all keys of a hash, or all indices of an array.

`keys` is the introspection primitive for associative and indexed containers. Given `%HASH` it returns the hash's keys; given `@ARRAY` (from Perl 5.12 onward) it returns the array's indices `0 .. $#ARRAY`. The list is what you iterate over, count, sort, or feed to another list-consuming operator. `keys` is also the idiomatic way to reset the iterator that `each` walks.

Synopsis

```
my @k = keys %h;
my @i = keys @a;
my $n = keys %h;        # scalar context: count
keys(%h) = 1000;        # preallocate buckets (hash only)
```

What you get back

In list context, a list of every key currently in `%HASH`, or every valid index of `@ARRAY`. In scalar context, the count - without building the list - so `scalar keys %h` is the cheap way to ask "how many entries?"

Hash keys are returned in an **arbitrary** order. The order is stable for a given unmodified hash - `keys`, `values`, and `each` all walk it in the same sequence - but it is not the insertion order, not sorted, and deliberately randomised per hash to defeat algorithmic complexity attacks (see `perlsec`). Any insertion or deletion may reshuffle. The one documented exception: deleting the key most recently returned by `each` or `keys` does not perturb the remaining order.

Array indices, by contrast, come back lowest-first: `(0, 1, 2, …, $#ARRAY)`.

The returned hash keys are **copies**. Modifying an element of the returned list does not modify the hash. Compare `values`, which returns aliases.

Global state it touches

- **Iterator of %HASH / @ARRAY**: `keys` resets the internal iterator shared with `each` and `values`. This is the idiomatic reset - call `keys %h` in void context specifically to discard any partial `each` walk:

```
keys %h;                # reset iterator, no list built
while (my ($k, $v) = each %h) { ... }
```

  Without this reset, a prior `each` that exited early leaves the iterator mid-hash and the next `each` call resumes there.

Examples

Iterate all keys in an arbitrary order:

```
my %h = (apple => 1, pear => 2, plum => 3);
for my $k (keys %h) {
    print "$k => $h{$k}\n";
}
```

Iterate in a deterministic order by running the keys through `sort`:

```
for my $k (sort keys %h) {
    print "$k => $h{$k}\n";     # apple, pear, plum
}
```

Iterate in **insertion order** by tracking keys separately - hashes themselves do not remember insertion order:

```
my %h;
my @order;
for my $pair (["apple", 1], ["pear", 2], ["plum", 3]) {
    my ($k, $v) = @$pair;
    push @order, $k unless exists $h{$k};
    $h{$k} = $v;
}
for my $k (@order) {
    print "$k => $h{$k}\n";     # apple, pear, plum (insertion)
}
```

Count entries without materialising the list:

```
my $n = keys %h;                # scalar context, O(1)
print "hash has $n entries\n";
```

Preallocate buckets when you know the hash will grow large. The assignment is a size hint, rounded up to the next power of two:

```
my %big;
keys(%big) = 10_000;            # allocates 16384 buckets
$big{$_} = 1 for 1 .. 10_000;   # no rehash churn during fill
```

Walk an array by index - useful when you need both the index and the element together:

```
my @a = ("zero", "one", "two");
for my $i (keys @a) {
    print "$i: $a[$i]\n";
}
```

Sort a hash by value, descending numeric:

```
for my $k (sort { $h{$b} <=> $h{$a} } keys %h) {
    printf "%4d %s\n", $h{$k}, $k;
}
```

Edge cases

- **Empty hash or array**: returns the empty list in list context, `0` in scalar context. No warning, no special case to code around.
- **`require v5.12` for arrays**: `keys @array` is a syntax error on perls older than 5.12. Put `use v5.12;` at the top of files that use the array form if portability to ancient perls matters.
- **Hash order is not reproducible across runs**: Perl randomises hash seeding at startup. Never build a test, cache key, or file format that assumes a particular `keys %h` order. Sort if you need determinism.
- **Hash order is not stable across Perl versions**: the randomised order within a run is stable; the algorithm itself is not guaranteed between releases. Do not pickle `keys` output and expect it to round-trip on a different Perl.
- **Deleting during iteration**: safe only for the key most recently returned by `each` or `keys`. Deleting a different key may cause other keys to be visited twice or skipped.
- **Lvalue keys on arrays is a syntax error**: `keys(@a) = 100` is not valid. The bucket-preallocation form works on hashes only.
- **Lvalue keys cannot shrink**: `keys(%h) = 10` on a hash that already has 1024 buckets has no effect. `%h = ()` clears the contents but preserves the bucket count; `undef %h` frees the buckets.
- **Memory cost of the list form**: `keys %h` in list context materialises every key as a fresh SV. For a million-entry hash that is a million allocations plus the returned list. Iterate with `each` or `while (my ($k, $v) = each %h)` when you want streaming access without the memory spike.
- **Tied hashes**: dispatch through `FIRSTKEY` / `NEXTKEY`. Order and side effects are whatever the tie class provides. In particular, tied hashes may reshuffle on insertion in ways ordinary hashes do not, and a tied `keys` may be expensive if the backing store is remote.
- **Scalar context on a tied hash**: may or may not be O(1). Ordinary hashes answer `scalar keys %h` in constant time; a tied class has to implement `SCALAR` explicitly to avoid a full walk.
- **Scalar keys on an array equals `scalar @array`**: both return the element count. Prefer `scalar @array` for clarity unless the surrounding code is already phrased

in keys/values/each terms.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- values - companion list of the values in the same order as keys; returns aliases, not copies
- each - streaming (key, value) pair walk sharing the same iterator that keys resets
- exists - test whether a specific key is present without building the full key list
- delete - remove a key; safe mid-iteration only for the most recently returned key
- sort - impose a deterministic order on the arbitrary sequence keys returns
- scalar - force scalar context to get the count without materialising the key list

Processes

kill

Send a signal to a list of processes.

kill is Perl's front-end to the kill(2) system call. It delivers SIGNAL to each process identified by LIST and returns the number of arguments that were successfully signalled. It is also the standard way to ask the kernel "does this PID still exist and may I signal it?" without actually disturbing the target, via the signal-zero form.

Synopsis

```
kill SIGNAL, LIST
kill SIGNAL            # tainting side-effect only; returns 0
```

What you get back

The number of arguments in LIST that were successfully passed to kill(2). This is **not** necessarily the number of processes that actually received the signal - when SIGNAL is negative, or when a PID identifies a process group, one argument can target many processes. On permission or "no such process" errors for an argument, that argument is not counted and $! is set to the corresponding errno (EPERM, ESRCH, EINVAL).

```
my $cnt = kill 'HUP', $child1, $child2;
kill 'KILL', @goners;
```

With an empty LIST, no signal is sent and the return value is 0. The bare kill SIGNAL form is sometimes written purely to trigger taint checks on SIGNAL; see perlsec.

SIGNAL: name or number

`SIGNAL` may be either a signal name (a string) or a signal number. The string form is recommended for portability because the same signal can have different numbers on different operating systems.

A signal name may carry the `SIG` prefix or not - `'TERM'` and `'SIGTERM'` name the same signal.

```
kill 'TERM', $pid;      # portable
kill 'SIGTERM', $pid;   # also fine
kill 15, $pid;          # SIGTERM on Linux; may differ elsewhere
kill 9, $pid;           # SIGKILL on Linux
```

The list of signal names available on the current platform lives in `$Config{sig_name}`, provided by the `Config` module.

SIGNAL 0: liveness check

If `SIGNAL` is the number `0` or the string `'ZERO'` (or `'SIGZERO'`), no signal is actually delivered. `kill(2)` still performs its permission check, so the call tells you whether it would be *possible* to signal the process - that is, the process exists and is owned by the same user, or the caller is the super-user.

```
if (kill 0, $pid) {
    # process exists and we may signal it
}
elsif ($!{ESRCH}) {
    # no such process - it has exited and been reaped
}
elsif ($!{EPERM}) {
    # it exists but we are not allowed to signal it
}
```

This is the canonical idiom for checking that a child is still alive, even as a zombie, and has not changed its UID. It does **not** reap the child - use `waitpid` for that.

Negative SIGNAL: signal a process group

A negative signal name or number signals a whole process group instead of a single process. `kill '-KILL', $pgrp` and `kill -9, $pgrp` both send `SIGKILL` to every process in the group identified by `$pgrp`.

```
kill '-TERM', $pgrp;    # polite shutdown of the whole group
kill -9, $pgrp;         # forced
```

Positive signals are almost always what you want. Reach for a negative signal only when the target is deliberately a process group leader (for example a child you placed in its own group with `setpgrp` so you can terminate the subtree at once).

PID 0, negative PIDs

The behaviour of `kill` when a *PID* is zero or negative depends on the operating system. On POSIX-conforming systems (Linux included):

- **PID `0`** - signal the sender's own process group.
- **PID `-1`** - signal every process the caller has permission to signal (super-user: the whole system, minus a few special PIDs).
- **Any other negative PID** - behaves like a negative `SIGNAL` against the absolute value: signal the process group whose PGID equals `abs(PID)`.

If both `SIGNAL` and the PID are negative, the result is undefined; a future Perl may warn. Do not combine the two.

Examples

Politely ask a child to shut down, then escalate:

```
kill 'TERM', $child;
sleep 2;
kill 'KILL', $child if kill 0, $child;
waitpid $child, 0;
```

Reap nothing, just check whether a PID file still names a live process:

```
open my $fh, "<", "/run/daemon.pid" or die $!;
chomp(my $pid = <$fh>);
close $fh;

if (kill 0, $pid) {
    print "daemon alive (pid $pid)\n";
} else {
    print "stale pid file: $!\n";
    unlink "/run/daemon.pid";
}
```

Signal a whole process group created by a shell pipeline child:

```
my $pgrp = getpgrp($child);
kill '-TERM', $pgrp;
```

Install a handler and send yourself a signal:

```
local $SIG{USR1} = sub { warn "got USR1\n" };
kill 'USR1', $$;        # $$ is our own PID
```

Broadcast to several workers and find out how many were reachable:

```
my @workers = (1234, 1235, 9999);   # 9999 has exited
my $hit = kill 'HUP', @workers;     # probably 2
warn "lost a worker: $!\n" if $hit < @workers;
```

Edge cases

- **Empty LIST**: `kill 'TERM'` sends nothing and returns `0`. The call still evaluates `SIGNAL`, which is the only reason to write it (taint propagation).
- **Unknown signal name**: raises an exception (`Unrecognized signal name "FOO"`). Validate against `$Config{sig_name}` if the name comes from user input.
- **`$!` after partial success**: `kill` walks `LIST` left to right. `$!` reflects the **last** failing argument, not all of them. If you need per-PID errors, call `kill` once per PID.
- **Zombies still count as alive**: `kill 0, $zombie_pid` returns truthy until the child has been reaped with `wait` or `waitpid`. This is usually what you want for a liveness check; occasionally it surprises.
- **Signal delivery is asynchronous**: a successful `kill` only means the signal was *queued*. The handler in `%SIG` on the receiving process may run later, and Perl's safe-signal machinery defers delivery to the next opcode boundary.
- **Sending to yourself**: `kill SIGNAL, $$` works and is the usual way to test your own `%SIG` handlers.
- **No LIST, with taint**: under `-T`, writing `kill 'TERM'` propagates taintedness of `'TERM'` for the side-effect of triggering the check - the form has no other use.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `fork` - create the child process you are later going to signal
- `wait` - reap any terminated child, clearing zombies that `kill 0` would still report as alive
- `waitpid` - reap a specific child, optionally non-blocking (`WNOHANG`) so it pairs cleanly with `kill 0`
- `getpgrp` / `setpgrp` - inspect and change process-group membership, which is what negative-SIGNAL `kill` operates on
- `%SIG` - install handlers for the signals you send with `kill`
- `perlipc` - full treatment of signals, handlers, and safe-signal semantics

Control flow

last

Exit a loop immediately, skipping the rest of the body and the `continue` block.

`last` is Perl's equivalent of C's `break`. It transfers control out of the enclosing loop to the statement that follows it. The current iteration's remaining statements do not run, and neither does the loop's `continue` block - that is the key difference from `next`. With no argument it targets the innermost enclosing `while`, `until`, `for`, `foreach`, or bare block; with a `LABEL` it targets a named loop further out.

Synopsis

```
last
last LABEL
last EXPR       # LABEL computed at runtime, since 5.18
```

What you get back

Nothing. `last` performs flow control; it does not produce a value. It cannot be used to return a result from a block that normally yields one, such as `eval { ... }`, `sub { ... }`, or `do { ... }`. In those contexts either the construct produces no value or the program dies with `Can't "last" outside a loop block` - see *Edge cases* below.

Targeting a loop by label

Without a label, `last` refers to the innermost enclosing loop. That is usually what you want:

```
while (my $line = <$fh>) {
    last if $line =~ /^__END__$/;
    process($line);
}
```

With a label you can break out of an outer loop from within a nested one. Labels are conventionally all-uppercase:

```
OUTER: for my $row (@grid) {
    for my $cell (@$row) {
        last OUTER if $cell eq 'STOP';
    }
}
# control resumes here
```

`last EXPR` (Perl 5.18+) computes the label at runtime. `EXPR` must evaluate to a string naming an enclosing label:

```
my $target = 'OUTER';
last $target;
```

Interaction with `continue` blocks

A loop's `continue` block normally runs at the end of every iteration, including the last. `last` **skips** it - that is its defining difference from `next`:

```
my $i = 0;
while ($i < 10) {
    last if $i == 3;
    print "body $i\n";
} continue {
    $i++;                   # runs after every iteration EXCEPT
```

(continues on next page)

(continued from previous page)

```
                            # the one terminated by last
}
# prints body 0, body 1, body 2; then exits the loop.
# $i stays at 3 because continue did not fire the final time.
```

If you want `continue` to run, use `next` with a guard instead, or advance the counter before the `last`.

Behaviour in nested loops

`last` only leaves one loop at a time - the innermost one, unless labelled. In this example the inner `last` exits the `for` but the `while` keeps running:

```
while (1) {
    for my $x (1 .. 5) {
        last if $x == 3;
    }
    # control lands here every iteration of the while
    last;                   # needed to exit the while as well
}
```

To unwind several levels at once, label the outer loop and target it:

```
SEARCH: for my $file (@files) {
    open my $fh, '<', $file or next;
    while (my $line = <$fh>) {
        if ($line =~ /$pattern/) {
            $found = $file;
            last SEARCH;  # exits both the while and the for
        }
    }
}
```

Bare blocks count as loops

A bare `{ ... }` block is, for flow-control purposes, a loop that executes exactly once. `last` can exit such a block early, which is the idiomatic way to write "try a sequence of checks, bail out on the first match":

```
{
    last if $cached;
    recompute();
    store_in_cache();
}
# control continues here
```

This is the only `last` form that does not involve an iterating construct.

Examples

Stop reading a file on a blank line - the classic header-terminator idiom:

```
LINE: while (<$fh>) {
    last LINE if /^$/;
    push @header, $_;
}
```

Find-and-report across a two-dimensional structure:

```
my $hit;
ROW: for my $r (0 .. $#grid) {
    for my $c (0 .. $#{ $grid[$r] }) {
        if ($grid[$r][$c] eq $needle) {
            $hit = [$r, $c];
            last ROW;
        }
    }
}
```

Early exit from a bare block used as a try-once sequence:

```
my $result;
{
    $result = $cache{$key}, last if exists $cache{$key};
    $result = compute($key);
    $cache{$key} = $result;
}
```

Computed label (Perl 5.18+). Rarely needed, but useful when the target loop is chosen dynamically:

```
my $level = $deep ? 'OUTER' : 'INNER';
OUTER: for (...) {
    INNER: for (...) {
        last $level if $done;
    }
}
```

Edge cases

- **Illegal in `grep` and `map`.** Both expect their block to produce a value for every input element; `last` precludes that. The parser accepts the code (the block is not lexically a loop it can see through), but executing it dies with `Can't "last" outside a loop block`. Filter with a conditional inside the block instead, or switch to an explicit `for` loop.
- **`do { ... }` is not a loop.** `last` inside `do { ... }` does not terminate the `do`; it unwinds to the nearest enclosing real loop. If there is none, the program dies with `Can't "last" outside a loop block`. For early exit from a `do` used as an expression, see the workarounds in *perlsyn* under “Statement Modifiers” - typically

a bare block plus `last`, or a one-shot `for` loop.

- **`sub { ... }` is not a loop.** `last` inside a sub body does **not** return from the sub; it unwinds past the sub to the enclosing loop in the caller. Use `return` to leave a sub.
- **`last` inside `eval` does not escape the `eval`.** `eval { ... }` is not itself a loop, so `last` looks outward past the `eval` for the nearest enclosing loop and unwinds to it. The `eval` is exited as a side effect of that unwind, but `$@` is left empty - no exception was raised. If the `eval` has no enclosing loop above it, the program dies with `Can't "last" outside a loop block`. To leave an `eval` normally, fall off the end or use `die` (caught by the `eval` itself) or `return` (if the `eval` is inside a sub and you want to leave the sub).
- **Precedence.** Unlike most named operators, `last` has the same precedence as assignment and is exempt from the looks-like-a-function rule. So `last ("foo") . "bar"` passes `"foo" . "bar"` as the label, not just `"foo"`. Use a space, or omit the parentheses, to avoid surprises:

```
last LABEL;
last "LABEL";
```

- **Unknown label.** `last NOPE` where no enclosing loop is tagged `NOPE` dies with `Label not found for "last NOPE"`. This is a runtime error, not a compile-time one - the label is resolved against the dynamic call stack.
- **Loop modifier form has no block to exit.** `EXPR if COND` and similar statement-modifier forms are not loops; `last` inside the expression unwinds past them.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `next` - skip to the next iteration, running the `continue` block first; the counterpart to `last` that keeps the loop alive
- `redo` - restart the current iteration without re-testing the condition and without running `continue`
- `return` - leave a subroutine with a value; the right tool when you reach for `last` inside `sub { ... }`
- `continue` - the loop-trailer block that `last` skips and `next` honours; read its page for the three-way illustration of `last` / `next` / `redo`

SCALARs and strings

lc

Return a lowercased copy of a string.

`lc` takes a string, converts every cased character to its lowercase equivalent, and returns the result. The original is never modified. If the argument is omitted, `lc` operates on `$_`. The exact set of characters that change depends on the string's encoding and the pragmas in scope at the call site - ASCII-only by default, full Unicode under `use feature 'unicode_strings'` or when the input is already a character string with the UTF-8 flag.

Synopsis

```
lc EXPR
lc
```

What you get back

A new string. The return value is always a fresh scalar - modifying it does not affect the argument, and modifying the argument afterward does not affect the returned string. Length in characters is preserved: `lc` never adds or removes characters, even in the rare Unicode cases where upper- and lower-case forms have different lengths in other directions (see `fc` for folding, which can change length).

```
my $str = lc("Perl is GREAT");   # "perl is great"
```

Global state it touches

- Reads `$_` when called without an argument.
- Observes `use locale` for `LC_CTYPE` - with locale in effect the current locale's lowercasing table applies to code points below 256.
- Observes `use bytes` - under `use bytes` only `A-Z` change, to `a-z`.
- Observes `use feature 'unicode_strings'` - forces Unicode rules regardless of the UTF-8 flag on the input.

Which casing rules apply

Perl picks one of four rulesets, in priority order. The first one whose condition holds wins:

1. **`use bytes` in effect** - ASCII rules. Only `A-Z` map to `a-z`; every other byte is left alone, including bytes in the 128-255 range that would otherwise be Latin-1 letters.
2. **`use locale for LC_CTYPE` in effect** - the current locale's tables apply to code points below 256; code points 256 and above (only reachable when the string already carries the UTF-8 flag) use Unicode rules. From Perl 5.20 onward, a UTF-8 locale uses full Unicode rules throughout.
3. **The argument has the UTF-8 flag set** - Unicode rules apply to every character.
4. **`use feature 'unicode_strings'` or `use locale ':not_characters'` in effect** - Unicode rules apply to every character, regardless of the UTF-8 flag.

5. **Otherwise** - ASCII rules. Characters outside `A-Z` are returned unchanged, including Latin-1 uppercase letters like À-Þ.

The upshot: if you want predictable Unicode behaviour on every string regardless of how it was constructed, enable `use feature 'unicode_strings'` (or `use v5.12` and above, which turns it on for you).

Examples

Basic ASCII lowercasing:

```
my $s = lc("Hello, World!");          # "hello, world!"
```

Default to `$_`:

```
for ("FOO", "Bar", "BAZ") {
    print lc, "\n";                   # foo / bar / baz
}
```

Unicode characters only lowercase when the rules allow it:

```
use feature 'unicode_strings';
my $s = lc("ÄÖÜ");                    # "äöü"
```

Without `unicode_strings` and without the UTF-8 flag on the input, non-ASCII characters pass through unchanged:

```
my $bytes = "\xC4\xD6\xDC";           # Ä Ö Ü as Latin-1 bytes
my $lower = lc $bytes;                # unchanged: "\xC4\xD6\xDC"
```

`lc` is what backs the `\L...\E` escape inside double-quoted strings:

```
my $str = "Perl is \LGREAT\E";        # "Perl is great"
```

Case-insensitive compare by lowercasing both sides:

```
sub ieq { lc($_[0]) eq lc($_[1]) }
ieq("Perl", "PERL");                  # true
```

For correct case-insensitive comparison across Unicode, prefer `fc` (the Unicode folding function) over `lc` - see *See also*.

Edge cases

- **`undef` argument** raises an `uninitialized` warning under `use warnings` and returns the empty string.
- **Empty string** returns the empty string.
- **Numbers are stringified first**: `lc(42)` returns `"42"`.
- **`LATIN CAPITAL LETTER SHARP S` (U+1E9E)** lowercases to U+00DF (ß) under Unicode rules, but only if the result can be represented without crossing the 255/256 boundary in a way the current ruleset accepts. Under `use locale` on a non-UTF-8

locale, Perl leaves the character unchanged rather than guessing - and from Perl 5.22 onward this raises a locale warning.

- **Locale and the UTF-8 flag interact**: under `use locale` on a non-UTF-8 locale, characters below 256 follow the locale while characters 256 and above follow Unicode. A string containing both ranges will get two different casing tables applied to it.
- **Tied variables** have their `FETCH` called once. `lc` does not modify the tied value; it returns a plain (non-tied) scalar.
- **`lc` is a unary named operator**, not a list operator. `lc $a, $b` parses as `(lc $a), $b` - only `$a` is lowercased, and the comma operator discards the result. Use parentheses or `map` when you mean to lowercase multiple strings:

```
my @lower = map { lc } @words;
```

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `uc` - the uppercasing counterpart, with the same pragma and locale rules
- `lcfirst` - lowercase only the first character; handy for de-capitalising sentences or identifiers
- `fc` - Unicode casefold, the correct choice for case-insensitive comparison when input may contain non-ASCII
- `$_` - the default subject `lc` reads when called with no argument
- `use locale` - controls whether locale tables or ASCII/Unicode rules apply to characters below 256
- `use feature 'unicode_strings'` - forces full Unicode casing regardless of the UTF-8 flag on the input

SCALARs and strings

lcfirst

Return a copy of a string with its first character lowercased.

`lcfirst` touches **only** the first character; the rest of the string is copied verbatim. The input is not modified - `lcfirst` returns a new string. It is the internal function implementing the `\l` escape in double-quoted strings, so `"\l$name"` and `lcfirst($name)` produce the same result.

If `EXPR` is omitted, `lcfirst` operates on `$_`.

Synopsis

```
lcfirst EXPR
lcfirst
```

What you get back

A new string equal to EXPR with its first character replaced by the lowercase form that character's case-folding rules produce. The rest of the string is untouched. If EXPR is the empty string, the empty string is returned. If EXPR is undef, the result is the empty string and an `uninitialized` warning is emitted under `use warnings`.

`lcfirst` never mutates its argument. To lowercase the first character in place, assign the result back:

```
$name = lcfirst $name;
```

For the all-characters variant, see `lc`.

Global state it touches

- `$_` - read when EXPR is omitted.
- `LC_CTYPE` locale - consulted when `use locale` is in effect (see *Pragmas and case rules* below).

Pragmas and case rules

`lcfirst` behaves the same way under various pragmas as `lc` does. The active rule set for the first character depends on which pragmas are in scope at the call site:

- **`use bytes`** - ASCII rules only. Only the bytes A-Z are folded to a-z; every other byte is returned unchanged.
- **`use locale`** (for `LC_CTYPE`) - the current locale decides the fold for code points below 256. Code points 256 and above fold by Unicode rules when the string has the UTF-8 flag set. A UTF-8 locale engages full Unicode rules from Perl 5.20 onward.
- **UTF-8-flagged string** - Unicode case-folding rules apply.
- **`use feature 'unicode_strings'`** or **`use locale ':not_characters'`** - Unicode rules apply regardless of the UTF-8 flag.
- **Otherwise** - ASCII rules. Characters outside ASCII are returned unchanged.

This means the same `lcfirst("É...")` can return `"é..."` under `use utf8` or `use feature 'unicode_strings'`, and `"É..."` (unchanged) under pure-ASCII defaults. When portability across pragmas matters, state the rule set explicitly:

```
use feature 'unicode_strings';
my $s = lcfirst $input;        # Unicode rules regardless of UTF-8␣
↪flag
```

Examples

Basic call, lowercasing the first letter of a word:

```
my $s = lcfirst "Hello, World";    # "hello, World"
```

Title-case normalisation for a list of names, where the caller wants the first letter lowercased for a URL slug:

```
my @slugs = map { lcfirst } @CamelCaseNames;
```

Default-argument form reads from `$_`:

```
for ("Apple", "Banana", "Cherry") {
    print lcfirst, "\n";           # "apple", "banana", "cherry"
}
```

Only the **first** character changes - later capitals are preserved. This matters for `camelCase` identifiers derived from `PascalCase`:

```
my $method = lcfirst "GetUserName";   # "getUserName"
```

Unicode rules must be requested explicitly, either via a UTF-8-flagged input string or via the `unicode_strings` feature:

```
use feature 'unicode_strings';
my $s = lcfirst "Éclair";          # "éclair"
```

```
my $s = lcfirst "Éclair";          # without unicode_strings and no
                                   # UTF-8 flag: "Éclair" unchanged
```

The `\l` escape in a double-quoted string is exactly `lcfirst` on the next character:

```
my $name = "Perl";
print "\l$name\n";                 # "perl\n"
```

Edge cases

- **Empty string**: `lcfirst("")` returns `""`. No warning.
- **`undef` input**: `lcfirst(undef)` returns `""`. Emits an `uninitialized` warning under `use warnings`.
- **First character already lowercase**: returned unchanged; no special case, no shortcut the user needs to know about.
- **First character has no lowercase mapping** (digit, punctuation, symbol, most CJK): returned unchanged.
- **Multi-codepoint case folds**: some Unicode characters lowercase to more than one code point (e.g. Turkish dotted/dotless I pairs in locale-tailored folds). `lcfirst` follows Unicode's *simple* lowercase mapping, which is always single-codepoint - it does not perform full case folding. Use `lc` combined with normalisation, or an explicit `Unicode::CaseFold`, when full-fold semantics matter.

- **Locale surprises around 255/256**: under `use locale` with a non-UTF-8 locale, a first character whose lowercase would cross the 255/256 boundary is returned unchanged, and from Perl 5.22 onward a `locale` warning is emitted.
- **Not list-aware**: `lcfirst` is a scalar operator. `lcfirst @list` takes `scalar(@list)` - the element count - and lowercases its first character, almost never what the caller meant. Use `map { lcfirst } @list` for per-element lowercasing.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `lc` - lowercase every character, not only the first; same pragma-sensitivity rules
- `ucfirst` - uppercase the first character; the exact inverse operation
- `uc` - uppercase every character
- `fc` - Unicode case-folding suitable for caseless comparison, not display
- `$_` - default argument when EXPR is omitted

SCALARs and strings

length

Return the number of characters in a string.

`length` stringifies its argument and returns how many **characters** that string contains. "Characters" means logical Perl characters, not bytes: a string holding ten codepoints has length `10` regardless of how many bytes those codepoints occupy when encoded as UTF-8 on disk or on the wire.

Synopsis

```
length EXPR
length
```

What you get back

A non-negative integer - the count of characters in the stringified value of EXPR. If EXPR is omitted, `length` operates on `$_`. If EXPR is `undef`, the result is `undef`, not `0`; this is the one case where `length` does not return a number, and it is deliberate - it lets you distinguish an undefined scalar from an empty string without a separate `defined` check:

```
length(undef)       # undef
length("")          # 0
length("abc")       # 3
```

Characters, not bytes

On a normal Perl string, `length` counts characters. The byte count of the same string encoded as UTF-8 is generally different and is what external systems (file sizes, network payloads, `Content-Length` headers) want. Get it by encoding first:

```
use Encode;
my $s     = "na\x{EF}ve";          # five characters
my $chars = length $s;              # 5
my $bytes = length(encode('UTF-8', $s));  # 6
```

See `Encode` and `perlunicode` for the full story on how Perl strings carry an internal character-vs-byte flag and when that flag matters.

Examples

Basic count:

```
length "hello"                      # 5
length ""                           # 0
```

Default argument is `$_`:

```
for ("a", "bb", "ccc") {
    print length, "\n";             # 1, 2, 3
}
```

Distinguishing undefined from empty:

```
my $x;
my $y = "";
print defined(length $x) ? "set" : "undef", "\n";   # undef
print defined(length $y) ? "set" : "undef", "\n";   # set
```

A Unicode string - character count vs byte count:

```
my $s = "caf\x{E9}";                # four characters, one of them
↪non-ASCII
length $s;                          # 4
length encode('UTF-8', $s);         # 5
```

Numbers are stringified before counting, which is occasionally useful and occasionally surprising:

```
length 12345                        # 5
length 3.14                         # 4  ("3.14")
length 1e6                          # 7  ("1000000")
```

Edge cases

- **`undef` in, `undef` out.** `length(undef)` returns `undef`, not `0`. Under `use warnings` reading an undefined scalar through `length $x` where `$x` is undef does **not** emit an `uninitialized` warning, unlike most other operators.
- **Whole array or hash**: `length` does not count elements. It stringifies its argument, so `length @arr` first evaluates `@arr` in scalar context (its element count) and then counts the digits of that number. For the actual size use `scalar @arr` or `scalar keys %hash`:

```
my @arr = (1) x 100;
length @arr;                    # 3  (digits of "100")
scalar @arr;                    # 100
```

- **Tied or magical scalars**: `length` triggers FETCH, so the counted value is whatever the tie layer returns at that moment.
- **Overloaded objects**: `length $obj` invokes the `""` stringification overload (or falls back to the default reference form) and counts characters in the resulting string.
- **Wide characters from byte input**: a string read from a handle without an encoding layer is a byte string. `length` counts bytes then - because each byte is one character to Perl. Decode first if you want character counts:

```
open my $fh, "<:raw", $path or die $!;
my $bytes = do { local $/; <$fh> };
my $text  = decode('UTF-8', $bytes);
length $bytes;                  # byte count
length $text;                   # character count
```

- **Locale and pragmas**: unlike `lc` / `uc` / `fc`, `length` does not vary by `use locale`. A character is a character regardless.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `scalar` - force scalar context; the correct way to get an element count from an array (`scalar @arr`) instead of the length of its stringified count
- `substr` - extract a substring by character offset; pairs naturally with `length` for slicing work
- `index` - find the position of a substring; returns a character offset measured the same way `length` measures
- `sprintf` - `%s` field widths are character counts, same notion as `length`
- `reverse` - in scalar context reverses a string character-by-character; same character model as `length`
- `pos` - regex-match position, also in characters

Filehandles, files, directories

link

Create a hard link from `NEWFILE` to the existing `OLDFILE`.

`link` adds a second directory entry that refers to the same underlying file as `OLDFILE`. After a successful call, the file has two names: `OLDFILE` and `NEWFILE`. Both names are equal citizens - the file itself is not deleted until every name is removed and no process holds the file open. The call is a thin wrapper around the `link(2)` system call and inherits its constraints: both paths must resolve on the **same filesystem**, and on POSIX systems only the superuser may hard-link a directory (most Linux kernels refuse even for `root`).

Synopsis

```
link OLDFILE, NEWFILE
```

What you get back

1 on success, `0` on failure. On failure, `$!` is set to the system error - typical values are `EXDEV` (cross-device link), `EPERM` (target is a directory, or filesystem forbids hard links), `EEXIST` (`NEWFILE` already exists), `EACCES` (insufficient permission), and `ENOENT` (`OLDFILE` missing).

```
link $src, $dst
    or die "link $src -> $dst failed: $!";
```

Global state it touches

- `$!` - set on failure to the system error code.

Examples

Create a second name for an existing file:

```
link "report.txt", "report.bak"
    or die "link failed: $!";
```

Use `stat` to observe the link count going up. The third element of the `stat` list is the number of hard links pointing at the inode:

```
link "data", "data.alt" or die $!;
my $nlink = (stat "data")[3];
print "data has $nlink names\n";    # "data has 2 names"
```

Atomically replace a file by building the new version under a temporary name, then swapping with `link` + `rename`:

```
open my $fh, ">", "config.new" or die $!;
print $fh $new_contents;
close $fh;
rename "config.new", "config" or die "rename: $!";
```

(For the genuinely atomic-under-crash variant you want `link` + `unlink` + `rename`; see the commentary in `perlport`.)

Detect a cross-device failure and fall back to copying:

```
use Errno qw(EXDEV);
unless (link $src, $dst) {
    if ($! == EXDEV) {
        # /tmp and $HOME are typically different filesystems
        copy_file($src, $dst);
    } else {
        die "link $src -> $dst: $!";
    }
}
```

Edge cases

- **Directories**: `link` on a directory fails with `EPERM` on effectively every modern system. POSIX forbids it for non-root, and Linux forbids it even for root. Use `symlink` when you want a second name for a directory.
- **Cross-filesystem**: the kernel returns `EXDEV` if `OLDFILE` and `NEWFILE` resolve to different filesystems. Hard links cannot span mount points because they share an inode number, which is only meaningful within one filesystem.
- **Existing target**: if `NEWFILE` already exists, the call fails with `EEXIST`. `link` never overwrites. Remove the target first with `unlink` if replacement is intended.
- **Symbolic link as `OLDFILE`**: the link is made to the symlink's **target**, not to the symlink itself, on most systems. If you need to link the symlink, dereference it explicitly or use a platform that exposes `linkat` with `AT_SYMLINK_FOLLOW` cleared (not available through Perl core).
- **Relative paths** are resolved against the current working directory at the moment of the call, for both operands.
- **Permissions**: `link` does not require write permission on `OLDFILE` - only search permission on its directory and write permission on the directory containing `NEWFILE`.
- **Same path both sides**: `link $f, $f` fails with `EEXIST`.
- **Unlinking**: removing `OLDFILE` with `unlink` after a successful `link` leaves the file fully reachable through `NEWFILE`. The data is freed only when the last name is unlinked and no open file descriptor refers to it.
- **Portability**: not every filesystem supports hard links. FAT, exFAT, and many network filesystems either refuse or silently behave as copies. See `perlport` for the platform matrix.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `symlink` - create a symbolic link instead of a hard link; works across filesystems and on directories
- `unlink` - remove a directory entry; the file itself survives as long as another link or open descriptor remains
- `rename` - move or rename a file within one filesystem; atomic where `link` + `unlink` is not
- `stat` - inspect the link count (field 3) to see how many names refer to the same inode
- `$!` - the system error set when `link` returns `0`

Sockets

listen

Mark a socket as passive so it can accept incoming connections.

`listen` is the third step in the classic server setup - after `socket` creates the endpoint and `bind` fixes it to a local address, `listen` switches it into the listening state and sizes the kernel's backlog queue of pending connections. Incoming connections are then drained one at a time with `accept`. `QUEUESIZE` is the maximum number of connections the kernel will hold for you before it starts refusing new ones.

Synopsis

```
listen SOCKET, QUEUESIZE
```

What you get back

True on success, false on failure (with `$!` set to the errno from the underlying `listen(2)`). Always check the return value - a server that silently fails to enter the listening state will accept no connections, and the cause (a stale socket file, a port collision, an unbound descriptor) is only visible through `$!`.

```
listen $srv, SOMAXCONN
    or die "listen on $path: $!";
```

Global state it touches

`listen` reads no special variables and writes only `$!` on failure. The call is a thin wrapper over the kernel's `listen(2)`; effects are on the socket descriptor, not on Perl state.

Examples

A minimal TCP server. `socket`, `bind`, `listen`, `accept` in order - miss any step and `accept` fails:

```
use Socket;

socket(my $srv, PF_INET, SOCK_STREAM, getprotobyname("tcp"))
    or die "socket: $!";
setsockopt($srv, SOL_SOCKET, SO_REUSEADDR, 1)
    or die "setsockopt: $!";
bind($srv, sockaddr_in(8080, INADDR_ANY))
    or die "bind: $!";
listen($srv, SOMAXCONN)
    or die "listen: $!";

while (accept(my $cli, $srv)) {
    # serve $cli ...
    close $cli;
}
```

A Unix-domain server. Same sequence, different address family:

```
use Socket;

my $path = "/tmp/myservice.sock";
unlink $path;                                # stale socket file would␣
↪cause EADDRINUSE
socket(my $srv, PF_UNIX, SOCK_STREAM, 0)  or die "socket: $!";
bind($srv, sockaddr_un($path))            or die "bind: $!";
listen($srv, 16)                          or die "listen: $!";
```

Using the POSIX-defined cap `SOMAXCONN` rather than hard-coding a number. The kernel silently clamps larger values to its own limit anyway; `SOMAXCONN` documents intent:

```
use Socket qw(SOMAXCONN);
listen($srv, SOMAXCONN) or die "listen: $!";
```

A small backlog for a service that processes connections faster than they arrive:

```
listen($srv, 5) or die "listen: $!";
```

Re-listening on an already-listening socket. A second call with a different `QUEUESIZE` is legal on Linux and resizes the backlog in place:

```
listen($srv, 128) or die "listen: $!";
# ... later, under load:
listen($srv, 1024) or die "listen: $!";   # grow the queue
```

Edge cases

- **Call order matters.** Calling `listen` on a socket that has not been `bind`-ed fails with `EDESTADDRREQ` (or on some kernels an autobound ephemeral port - do not rely on that). Calling `accept` before `listen` fails with `EINVAL`.
- **Only for connection-oriented sockets.** `listen` is meaningful on `SOCK_STREAM` and `SOCK_SEQPACKET`. Calling it on a `SOCK_DGRAM` socket fails with `EOPNOTSUPP`; UDP has no notion of pending connections.
- **`QUEUESIZE` is advisory.** The kernel clamps it to `/proc/sys/net/core/somaxconn` (Linux) regardless of what you ask for. A value of `0` is accepted but means "rely on kernel-default backlog"; it does not disable listening.
- **Negative values are silently treated as `0`** by the kernel. Perl does not diagnose this - check your input before passing it to `listen`.
- **`EADDRINUSE` at `listen` vs at `bind`.** Port conflicts for TCP/UDP surface at `bind`; `listen` rarely fails with this. For Unix-domain sockets a leftover socket file from a previous run causes `bind` to fail with `EADDRINUSE` - remove it with `unlink` before binding.
- **Stale connections in the queue.** When `accept` runs, the returned connection may already be closed by the peer (the client gave up while queued). Handle that like any other I/O error from the new socket, not as a `listen` problem.
- **After `fork`.** A listening socket inherited by a child remains listening; both parent and child can `accept` from the same queue. This is the standard pre-fork server pattern - do not call `listen` again in the child.
- **`SIGPIPE` is unrelated.** `listen` itself never raises `SIGPIPE`; that signal arises on writes to a closed peer after `accept`.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `socket` - creates the endpoint `listen` will mark as passive; always the first call in the sequence
- `bind` - assigns a local address to the socket; must run before `listen` or the call fails
- `accept` - pulls the next pending connection off the backlog queue that `listen` established
- `setsockopt` - set `SO_REUSEADDR` before `bind` to avoid `TIME_WAIT` collisions when restarting a server
- `shutdown` - half-close an accepted connection; not used on the listening socket itself
- `$!` - holds the errno after a failed `listen`; the only way to tell `EADDRINUSE` from `EACCES` from `EOPNOTSUPP`

Scoping

local

Save the current value of a package variable and restore it when the enclosing scope exits.

`local` is the dynamic-scope operator. It does **not** declare a new variable - it reaches into an existing package (global) variable, stashes the current value on a hidden save stack, and arranges for the original value to be put back when control leaves the current block, `eval`, or `do FILE`. Called subroutines see the temporary value. For most "I want a variable local to this block" needs, the right tool is `my`; reach for `local` specifically when the variable must be a package global - the punctuation variables, filehandles, or entries in the symbol table - and you want that global to have a different value for the duration of a block and every subroutine it calls.

Synopsis

```
local EXPR
local $var
local $var = EXPR
local (@list) = LIST
local $hash{key}
local $array[i]
local *GLOB
```

What you get back

The value of the localized expression (the new value, after any assignment in the same statement). The save-stack bookkeeping is a side effect; the return value matters only when `local` appears inside a larger expression:

```
our $x = 2;
foo($x, local $x = $x + 1, $x);   # foo() receives (2, 3, 3)
```

Dynamic scope, not lexical scope

`local` and `my` are different species of scoping:

- `my` creates a **new, lexically scoped** variable. Only code textually inside the enclosing block can see it. Called subroutines cannot.
- `local` does **not** create anything. It temporarily overrides the value of an **existing package variable**. Every piece of code - this block and every subroutine it calls, directly or transitively - sees the new value until the scope ends.

That is why `local` is the right tool for punctuation variables like `$/`, `$\`, `$,`, `$_`, and `$@`: those are package globals that the Perl runtime (and anything called from your code) reads at well-known names. A `my` shadowing would not affect them.

```
sub slurp {
    my ($path) = @_;
    open my $fh, "<", $path or die "open $path: $!";
```

(continues on next page)

(continued from previous page)

```
    local $/;                       # undef $/ for this sub only
    return <$fh>;                   # read to EOF in one go
}
```

The `local $/` line above is the single most common `local` idiom in real code: temporarily undefining the input record separator so that `readline` (`<$fh>`) returns the whole file. When `slurp` returns, `$/` is restored to whatever it was before - even if the caller had also changed it.

What you can and cannot localize

You can `local`ize:

- **Package scalars, arrays, hashes** - `local $Pkg::var`, `local @Pkg::arr`, `local %Pkg::hash`. Bareword `local $var` targets the variable in the current package.
- **Already-declared `our` variables** - `our` creates a lexical alias to a package variable, so `local` on it localizes the underlying global.
- **Single elements and slices of package arrays/hashes** - `local $hash{key}`, `local $array[3]`, `local @hash{qw(a b)}`, `local @array[0..2]`.
- **Globs** - `local *name` creates a fresh symbol-table entry so that `$name`, `@name`, `%name`, `&name`, and the `name` filehandle are all dynamically reset together.
- **Conditional lvalues** - `local ($cond ? $v1 : $v2)` when the chosen branch is itself localizable.

You **cannot** `local`ize:

- A `my` variable. `my` variables are lexicals living in a pad, not in the symbol table; they have no package name for `local` to save under. Trying it is a compile-time error: `Can't localize lexical variable $x`.
- A `state` variable. Same reason.
- Read-only magic globals. `local $1 = 2` fails with `Modification of a read-only value attempted` - the capture variables cannot be assigned to. (The one exception is `$_`: `local $_` has, since 5.14, explicitly stripped magic so it can be reused safely in a subroutine.)

Localizing composite-type elements

`local $hash{key}` and `local $array[i]` save the **named slot**, not the value that happens to be there. When the scope ends, the original value is restored to that same slot - even if the element was deleted or the array was shortened in the meantime. A deleted hash key springs back; a popped array element reappears, padding the array with `undef` if necessary.

```
our %hash = (a => "is");
{
    local $hash{a} = "drill";
    delete $hash{a};                # gone for now...
```

(continues on next page)

(continued from previous page)

```
}
print $hash{a}, "\n";              # ...but restored to "is"
```

Examples

Basic: give a global a temporary value for one block. Subroutines called from inside the block see the new value.

```
our $verbose = 0;
sub log_msg { print "[v=$verbose] $_[0]\n" }

{
    local $verbose = 1;
    log_msg("inside");             # prints "[v=1] inside"
}
log_msg("outside");                # prints "[v=0] outside"
```

Slurp-a-file idiom with `$/`:

```
my $content = do {
    open my $fh, "<", $path or die "open $path: $!";
    local $/;                      # list context for <> now reads␣
↪all
    <$fh>;
};
```

Temporarily trap `$@` so an inner `eval` does not clobber the caller's pending error:

```
sub try_cleanup {
    local $@;                      # caller's $@ preserved
    eval { risky_cleanup() };
    # even if eval failed, caller's $@ is untouched
}
```

Python-style `print` defaults, scoped to one block:

```
{
    local $, = " ";
    local $\ = "\n";
    print 1, 2, 3;                 # "1 2 3\n"
}
print 1, 2, 3;                     # "123" - defaults restored
```

List form needs parentheses. The parentheses also give the right-hand side list context - exactly like `my`:

```
our (@wid, %get);
local (@wid, %get) = (@defaults, %overrides);
```

Localize a single hash element - useful for threading overrides through a call without touching unrelated keys:

```
our %config = (timeout => 30, retries => 3);
{
    local $config{timeout} = 5;
    do_probe();                  # sees timeout => 5
}
# %config back to (timeout => 30, retries => 3)
```

Glob localization - whole symbol-table entry for LOG is replaced:

```
{
    local *LOG;
    open LOG, ">", "/tmp/trace.$$" or die $!;
    run_traced();                # sees our private LOG filehandle
}
# original LOG (if any) is back
```

Edge cases

- **Runtime operator, not a declaration.** `local` executes each time control reaches it. Inside a loop it saves and restores on every iteration - measurable overhead. Lift it out of the loop when the intent is "for the whole loop":

  ```
  { local $/ = "\n\n";            # once, not per iteration
      while (<$fh>) { ... }
  }
  ```

- **Assignment is evaluated before the save.** The right-hand side runs in the enclosing dynamic scope, so `local $x = $x + 1` reads the old `$x`, then saves, then assigns the new value.
- **Scope boundary is dynamic, not lexical.** The save is undone when the **current block exits at runtime**, not when the source block ends textually. Exiting via `return`, `die`, `last`, `next`, or a non-local goto still triggers the restore.
- **`local` on a `my` is a compile-time error.** The parser rejects `my $x; local $x;` with `Can't localize lexical variable $x`. Use `our` (or a fully qualified name) if you genuinely need dynamic scoping on that name.
- **Tied arrays and hashes** do not currently behave the way the POD describes - localizing the whole aggregate is buggy upstream. Localizing individual elements of a tied aggregate is safe. This caveat is inherited from upstream Perl.
- **Read-only magicals cannot be assigned.** `local $1 = "x"` dies at runtime. You can `local $_` (magic is stripped since 5.14) but not `local $1`, `local $&`, etc.
- **Negative array indices** in `local $array[-1]` are documented as "particularly surprising" upstream; treat the behaviour as unstable and avoid.
- **`delete local`** extends `local` to the removal of an element: `delete local $hash{key}` deletes the entry for the current block and restores it on exit. It returns the value that was there before localization.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `my` - lexical (compile-time, block-scoped) declaration; the right tool when you want a truly private variable
- `our` - lexical alias to a package variable; common partner for `local` when the global is in another package
- `state` - persistent lexical; cannot be `localized`
- `delete` - combined with `local` (`delete local $h{k}`) to scope an element deletion to the current block
- `eval` - common `local` target: `local $@` around an inner `eval` prevents it leaking errors to the caller
- `$/` - input record separator; canonical `local` target for slurp-mode file reads
- `$_` - default scalar; `local $_` strips magic (since 5.14) so a subroutine can reuse it safely
- `$@` - last `eval` error; `local $@` shields the caller from an inner `eval`'s error

Time

localtime

Convert an epoch time to a broken-down calendar time in the local time zone.

`localtime` takes a Unix epoch value - seconds since 1970-01-01 UTC, the shape returned by `time` - and decomposes it into year, month, day, hour, minute, second, weekday, yearday, and a daylight-saving flag, all expressed in the **local** time zone. The time zone is read from the `TZ` environment variable if set, otherwise from the system default (typically `/etc/localtime`). For the UTC companion that ignores `TZ`, see `gmtime`.

Synopsis

```
localtime EXPR
localtime
```

What you get back

The return shape depends on context, and `localtime` is one of the handful of built-ins where the **same expression in list context and scalar context produces completely different answers**.

List context - a 9-element list, all numeric, taken straight out of the C `struct tm`:

```
#     0     1     2     3     4     5     6     7     8
my ($sec, $min, $hour, $mday, $mon, $year, $wday, $yday, $isdst) =
                                                  localtime(time);
```

Index	Name	Range	Notes
0	`$sec`	`0..60`	`60` is possible for leap seconds
1	`$min`	`0..59`	
2	`$hour`	`0..23`	
3	`$mday`	`1..31`	1-based day of the month
4	`$mon`	`0..11`	**0-based**: `0` = January, 11 = December
5	`$year`	years - 1900	e.g. 129 for 2029
6	`$wday`	`0..6`	`0` = Sunday, 3 = Wednesday
7	`$yday`	`0..365`	0-based day of the year; 365 in leap year
8	`$isdst`	boolean	true when DST is in effect for the local TZ

The 0-based `$mon` and 1900-based `$year` are long-standing C inheritances; most bugs written against `localtime` come from forgetting exactly one of them. The fix for `$year` is `+= 1900`:

```
my @t = localtime;
my $full_year = $t[5] + 1900;
```

Scalar context - the familiar English `ctime(3)` string, always 24 characters, always in English regardless of locale:

```
my $now_string = localtime;      # "Thu Oct 13 04:54:34 1994"
```

For a locale-aware or custom-formatted string, do not try to pick apart the scalar form. Go through the list form and `POSIX::strftime`:

```
use POSIX qw(strftime);
my $now_string = strftime "%a %b %e %H:%M:%S %Y", localtime;
```

Global state it touches

- **`TZ`** in `%ENV` - selects the local time zone. A program that wants to interpret a timestamp in a specific zone sets `$ENV{TZ}` and calls `POSIX::tzset` before `localtime` (see *Edge cases* for the caveat).
- System zoneinfo (`/etc/localtime`, `/usr/share/zoneinfo/...`) when `TZ` is unset - read lazily by the C library, not by Perl directly.
- `%ENV` changes to `TZ` do **not** always take effect on the next call; see *Edge cases*.

`localtime` does not touch `$_` or any output-related globals.

Examples

Current wall-clock date in a human-readable string:

```
my $now = localtime;             # "Wed Apr 23 14:07:12 2025"
```

Current date as a 9-element list - note the two offsets:

```
my ($s,$m,$h,$md,$mo,$y,$wd,$yd,$dst) = localtime;
printf "%04d-%02d-%02d %02d:%02d:%02d\n",
       $y+1900, $mo+1, $md, $h, $m, $s;    # "2025-04-23 14:07:12"
```

Month name from `$mon` using a lookup table - this is the canonical way, because `$mon` is 0-based exactly so it indexes an array:

```
my @abbr = qw(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec);
my ($mday, $mon) = (localtime)[3,4];
print "$abbr[$mon] $mday";       # e.g. "Apr 23"
```

Formatting with `POSIX::strftime` - the right answer whenever you need anything other than the bare English `ctime` string:

```
use POSIX qw(strftime);
my @now = localtime;
my $stamp = strftime "%Y-%m-%d %H:%M:%S", @now;   # "2025-04-23␣
↪14:07:12"
```

Interpreting a stored epoch in a specific zone. Set `TZ`, call `POSIX::tzset`, then `localtime`:

```
use POSIX qw(tzset);
local $ENV{TZ} = "Europe/Berlin";
tzset();
my $wall = localtime(1_700_000_000);   # "Tue Nov 14 23:13:20 2023"
```

Context trap - `localtime` inside `print LIST` is in **list** context, so you get nine numbers concatenated with no spaces. Use `scalar` to force the string form:

```
print localtime, "\n";           # e.g. "12724232312505230"
print scalar localtime, "\n";    # "Wed Apr 23 14:07:12 2025"
```

Edge cases

- **No argument**: `localtime` with no argument uses `time` - the current wall-clock moment.
- **Negative epoch**: dates before 1970-01-01 UTC are accepted on systems with a signed `time_t`. Portability of very distant past or future times depends on the platform's C library.
- **`$year` pitfall**: `$year + 1900` gives the full year. Writing `"19$year"` was a Y2K bug and is still wrong; write `1900 + $year`.
- **`$mon` pitfall**: `$mon` is 0-based. `localtime` output feeding anything 1-based (including `POSIX::mktime`) does **not** need adjustment - `mktime` expects the same 0-based convention. External systems (databases, ISO-8601 strings) usually want `$mon + 1`.
- **`$isdst` is advisory**, not directly comparable across zones. `1` means "the local zone currently applies its DST offset"; `0` means "standard time"; `-1` occasionally

appears when the zone rules cannot decide (e.g. the hour that does not exist on a spring-forward day).

- **TZ reload caveat**: most C libraries (glibc included) cache the parsed zone rules on the first lookup. A later assignment to `$ENV{TZ}` is **not** picked up automatically - call `POSIX::tzset` after every change, or the old zone will keep being used.
- **Scalar string is always English** and is **not** affected by `LC_TIME` or any other locale setting. For a localized string, go through `POSIX::strftime` with `LC_TIME` set to the desired locale.
- **Thread safety**: on systems where the C `localtime(3)` call is not reentrant, concurrent threads mutating `TZ` can race. In practice pperl and modern glibc use the reentrant `localtime_r` internally, but `TZ` is still a process-wide global.
- **Leap seconds**: `$sec` can legally be `60` on systems with leap second support. Most systems smear or ignore them and you will never see it; code that must not break on `60` handles it already.
- **`%a` / `%b` width** under `POSIX::strftime`: the abbreviated weekday and month names are **not guaranteed three characters** in every locale. Code that assumes a fixed column width will misalign in German, French, and many others.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `gmtime` - same decomposition in UTC; does not consult `TZ` and is the right choice when you want a time zone–independent value
- `time` - the epoch source that `localtime` most often receives as its argument
- `POSIX::strftime` - format a `localtime` list into any string layout you need, locale-aware
- `POSIX::mktime` - inverse direction: take a 9-element list back to an epoch value
- `POSIX::tzset` - re-read `TZ` after changing it, mandatory if you want the change to take effect
- `$ENV{TZ}` - the environment variable `localtime` reads to pick the local zone

Misc

lock

Place an advisory lock on a shared variable, array, hash, or subroutine until the lock goes out of scope.

`lock` is the synchronisation primitive of Perl's interpreter-threads model (`use threads; use threads::shared`). When `THING` is a variable that has been declared `:shared` - or a reference to a shared aggregate - `lock` acquires an advisory mutex on that datum for the rest of the enclosing block. When the block exits, the lock is released. Any other thread calling `lock` on the same datum blocks until the holder releases it.

The lock is **advisory**: it stops other threads that also call `lock` on the same variable, nothing else. A thread that just reads or writes the shared variable without locking it is not blocked and not serialised. `lock` is a convention, not a barrier.

Synopsis

```
lock $shared_scalar
lock @shared_array
lock %shared_hash
lock &shared_sub
lock $ref_to_shared_aggregate
```

What you get back

The argument itself:

- For a scalar, the scalar value.
- For an array, hash, or subroutine, a reference to it.

The return value is rarely used. `lock` is called for its side effect (acquiring the mutex), not for its value.

Lock is scoped, not statement-scoped

The lock is released when the innermost enclosing block exits, not when the `lock` statement finishes. Two consequences:

- `lock` inside a `{ ... }` block serialises only that block's duration. Exiting the block - by fall-through, `return`, `last`, `die`, whatever - releases the lock.
- You cannot release a lock early. There is no `unlock`. If you need a shorter critical section, introduce a tighter `{ ... }` block:

```
{
    lock %shared_hash;
    $shared_hash{$key} = $value;
}   # lock released here
do_unlocked_work();
```

Weak keyword

`lock` is a **weak keyword**. If a subroutine named `lock` is in scope at the point of the call (declared or imported before the call site), that subroutine is called instead of the built-in. This makes `lock` safe to use as a method or function name in code that does not use threads. To force the built-in when a same-named sub is in scope, call it via its package: `CORE::lock($thing)`.

Examples

Serialise a hash update between threads:

```
use threads;
use threads::shared;

my %counts :shared;

sub bump {
    my $key = shift;
    lock %counts;
    $counts{$key}++;
}
```

Lock an aggregate through a reference. `lock` follows one level of reference to find the datum to lock:

```
my @queue :shared;
my $queue_ref = \@queue;

sub enqueue {
    my $item = shift;
    lock $queue_ref;        # locks @queue
    push @queue, $item;
}
```

Narrow the critical section with an inner block so unrelated work runs unlocked:

```
my $result;
{
    lock %cache;
    $result = $cache{$key};
}
expensive_post_processing($result);   # runs without the lock
```

Without `use threads::shared`, `lock` is a no-op. This makes it harmless to leave in code that may run single-threaded:

```
# no `use threads::shared` here
lock $x;                    # does nothing, does not warn
```

Edge cases

- **Unshared variable**: `lock` on a variable that was not declared `:shared` and is not a reference to shared data does nothing useful. Under upstream this may croak in some cases; under others it is silently a no-op. Do not rely on `lock` to catch the "forgot to share" bug.
- **Reference vs aggregate**: `lock \%h` and `lock %h` both lock `%h`. `lock` dereferences one level to find the shared datum.

- **Recursive locking within one thread**: the same thread may re-acquire a lock it already holds; the mutex is recursive per thread. Another thread still blocks.
- **No `unlock`**: releasing before block exit requires restructuring into a tighter scope. `undef $lock_guard` patterns from other languages do not apply - `lock` is not a value you store.
- **Weak-keyword ambiguity**: a `sub lock` declared or imported earlier wins over the built-in. Use `CORE::lock` to disambiguate.
- **Not cross-process**: `lock` coordinates threads within one interpreter process. For file-level locking across processes use `flock`; for advisory filesystem locks they are separate mechanisms with no interaction.

Differences from upstream

- pperl does not implement interpreter threads. The `lock` opcode is recognised by the parser but has no runtime effect - it behaves exactly as upstream `lock` does under a build without `threads::shared`: a silent no-op returning its argument. Code that uses `lock` defensively (as the upstream documentation recommends for single-threaded compatibility) runs unchanged. Code that depends on `lock` for actual synchronisation will not be synchronised under pperl.
- The companion distribution `threads::shared` is not available under pperl; `use threads::shared` fails at compile time. Parallelism in pperl is exposed through JIT auto-parallelisation (Rayon) on specific loop shapes, not through user-visible ithreads.

See also

- `flock` - advisory lock on a filehandle, coordinating between processes rather than threads
- `wait` - reap a child process; the process-level equivalent of waiting for another unit of execution
- `fork` - the other concurrency primitive in core Perl, with no shared memory by default
- `our` - declare a package-scoped variable; commonly paired with `:shared` in threaded code
- `local` - dynamic-scope binding, sometimes confused with `lock` because both are scope-bounded and both restore on block exit

Numeric functions

log

Return the natural logarithm (base *e*) of a number.

`log` is the inverse of `exp`. It takes a numeric `EXPR`, coerces it to a double-precision float if necessary, and returns its logarithm to the base *e* (Euler's number, approximately `2.718281828459045`). If `EXPR` is omitted, `log` operates on `$_`.

There is no built-in for logarithms to an arbitrary base - use basic algebra (see the examples below) or `POSIX::log10` / `POSIX::log2` for the two common fixed bases.

Synopsis

```
log EXPR
log
```

What you get back

A double-precision float: the natural logarithm of `EXPR`. Concretely:

- `log(1)` is `0`.
- `log(exp(1))` is `1` (the defining identity).
- `log(x)` is strictly increasing for `x > 0`.

Special inputs follow IEEE-754 rules as provided by the platform C library's `log(3)`:

- `log(0)` returns negative infinity (`-Inf`).
- `log(x)` for `x < 0` returns `NaN`.
- `log(Inf)` returns `Inf`; `log(NaN)` returns `NaN`.

The result is always a float, even when the input is an integer or a string.

Examples

The defining identity - the natural log of *e* is 1:

```
use POSIX ();
print log(exp(1)), "\n";            # 1
```

A base-10 logarithm, built from `log` by change of base. The base-*N* log of a number equals its natural log divided by the natural log of *N*:

```
sub log10 {
    my ($n) = @_;
    return log($n) / log(10);
}

print log10(1000), "\n";            # 3
print log10(0.01), "\n";            # -2
```

The same idiom generalised to any base - useful for bits (base 2), decibels (base 10), or anything else:

```
sub log_base {
    my ($base, $n) = @_;
    return log($n) / log($base);
}
```

(continues on next page)

(continued from previous page)

```
print log_base(2, 1024), "\n";      # 10
print log_base(16, 65536), "\n";    # 4
```

Operating on `$_` - handy inside `map` or a `while` line loop:

```
my @natural_logs = map { log } 1, 2, 3, 4;
```

Numeric coercion of a string argument - `log` evaluates its operand in numeric context just like the arithmetic operators:

```
print log("7.389056"), "\n";        # ~2 (log of e^2, give or take)
```

Edge cases

- **`log(0)` is `-Inf`**, not an error and not a division-by-zero exception. Perl does not die; you get a float infinity back and any further arithmetic propagates it:

  ```
  my $x = log(0);                   # -Inf
  print $x + 1, "\n";               # -Inf
  ```

- **log of a negative number is `NaN`**. Again, no exception - the caller is responsible for range-checking if that matters:

  ```
  my $x = log(-1);                  # NaN
  print $x == $x ? "num" : "nan", "\n";   # nan (NaN != NaN)
  ```

  Guard the input if you need a defined result:

  ```
  die "log domain error" unless $n > 0;
  my $l = log($n);
  ```

- **Non-numeric strings** coerce to `0` and therefore produce `-Inf`, and under `use warnings` trigger `Argument "..." isn't numeric`:

  ```
  use warnings;
  print log("hello"), "\n";         # -Inf, with warning
  ```

- **No argument form uses `$_`.** `log` with an empty argument list reads the current topic; inside `map { log }` or `while (<>) { print log; }` this is the usual shortcut.
- **Integer vs float input** makes no difference to the result type: the return value is always a float. `log(2)` and `log(2.0)` are the same value.
- **Precision** is whatever the platform C library's `log(3)` delivers - typically ~15–17 significant decimal digits on x86-64 Linux. Results are not bit-for-bit portable across architectures.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `exp` - the inverse: `exp(log($x))` is `$x` for `$x > 0`, and `log(exp($x))` is `$x` for all finite `$x`
- `**` - the exponentiation operator; `$x ** $y` is equivalent to `exp($y * log($x))` for positive `$x`
- `POSIX::log10` - base-10 logarithm, direct from the C library; faster and slightly more accurate than `log($x)/log(10)`
- `POSIX::log2` - base-2 logarithm, same story
- `POSIX::log1p` - `log(1 + $x)` computed accurately for small `$x`, where the naive form loses precision

Filehandles, files, directories

lstat

Return the 13-element status list for a path **without** following a symbolic link.

`lstat` does exactly what `stat` does - same 13-element result, same special _ filehandle cache, same failure semantics - with one difference: if the argument names a symbolic link, `lstat` returns information about the *link itself* (its own mode, owner, size, mtime), not the target the link points to. For anything that is not a symbolic link, `lstat` and `stat` are indistinguishable. On systems without symbolic links `lstat` silently degrades to `stat`.

Synopsis

```
lstat FILEHANDLE
lstat EXPR
lstat DIRHANDLE
lstat
```

What you get back

In list context, the same 13-element list as `stat`:

```
my ($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,$size,
    $atime,$mtime,$ctime,$blksize,$blocks)
       = lstat($path);
```

Field meanings are identical to `stat` - device, inode, mode (type + permissions), link count, owner uid/gid, rdev, size, atime, mtime, ctime, preferred I/O block size, and allocated blocks. The type bits in `$mode` are what distinguish a symbolic link: on the link itself they report `S_IFLNK`, whereas `stat` on the same name would report the type of the *target*.

In scalar context `lstat` returns a true value on success and the empty string on failure. On failure the list form returns the empty list and `$!` is set to the reason (`ENOENT`, `EACCES`, etc.).

If `EXPR` is omitted, `lstat` operates on `$_`.

The _ filehandle

Every successful `lstat` (and every successful `stat` or filetest) populates the interpreter's internal "last stat buffer". Passing the bareword `_` as the argument reuses that buffer instead of making another system call:

```
if (-l $path && (my @s = lstat(_))) {
    # $path was a symlink; @s describes the link, not the target
}
```

Because `-l` itself does an `lstat` under the hood, the `lstat(_)` above costs nothing extra. This is the canonical way to "ask several questions about one file without re-stat'ing it". Note that a *filetest* on a bareword filename (`-l $path`) leaves the buffer describing the link; a filetest on a filehandle follows the open descriptor and thus leaves a `stat`-shaped buffer. When you want symlink data, reach for `lstat` explicitly.

Global state it touches

- `$_` - the argument when `EXPR` is omitted.
- `$!` - set on failure to the `errno` from the underlying `lstat(2)` call.
- The special `_` filehandle - overwritten on every successful call.

Examples

Distinguish a symlink from its target. `stat` on a symlink silently follows it; `lstat` does not:

```
symlink "/etc/passwd", "link.tmp";

my $via_stat  = (stat  "link.tmp")[7];  # size of /etc/passwd
my $via_lstat = (lstat "link.tmp")[7];  # size of the link inode
                                        # (length of "/etc/passwd")
```

Walk a directory and separate links from regular files without a second syscall per entry, using the `_` buffer:

```
use Fcntl ':mode';

for my $name (readdir $dh) {
    next if $name eq '.' || $name eq '..';
    lstat "$dir/$name" or next;
    if (S_ISLNK((lstat(_))[2])) {
        print "link: $name\n";
    } elsif (-f _) {
```

(continues on next page)

(continued from previous page)

```
        print "file: $name\n";
    } elsif (-d _) {
        print "dir:  $name\n";
    }
}
```

Detect a dangling symlink - `lstat` succeeds on the link, `stat` fails because the target is gone:

```
if (lstat $path and not stat $path) {
    warn "$path is a dangling symlink: $!\n";
}
```

Follow the link manually when you want both pieces of information:

```
my @link   = lstat $path       or die "lstat $path: $!";
my $target = readlink $path    // die "readlink $path: $!";
my @target = stat $path        or die "stat $path: $!";
```

No-argument form operating on `$_`, idiomatic in `File::Find` preprocess blocks and `grep` filters:

```
my @symlinks = grep { lstat; -l _ } @paths;
```

Edge cases

- **Not a symlink**: `lstat` behaves exactly like `stat` - there is no penalty to preferring `lstat` when you are about to branch on `-l`.
- **Systems without symlinks**: `lstat` silently falls back to `stat`. Code written against `lstat` stays portable; it just loses the distinguishing behaviour.
- **Filehandle / dirhandle argument**: a passed FILEHANDLE or DIRHANDLE refers to an already-opened descriptor, which cannot itself be a symlink (the kernel resolved the link at `open` time). `lstat FILEHANDLE` therefore returns the same result as `stat` on that handle. Use the EXPR form on the *path* when you need the link metadata.
- **`lstat(_)` after a failed `lstat`**: the `_` buffer is left in whatever state the last *successful* stat put it in. A failed `lstat` does not clear it. Always check the return value before reading `_`.
- **Relative paths**: resolved against the process's current working directory, same as `stat`.
- **Permission to `lstat`**: requires search (x) permission on every directory component of the path, but - unlike `stat` - does **not** require any permission on the link target. `lstat` therefore succeeds on links into directories you cannot read.
- **Bareword vs expression**: `lstat $fh` where `$fh` holds a filehandle stats the handle; `lstat "$fh"` stringifies `$fh` and stats the resulting (usually meaningless) pathname. Quote intentionally.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `stat` - the link-following counterpart; identical return shape and _ semantics
- `readlink` - read the *target* string of a symbolic link without stat'ing it
- `symlink` - create a symbolic link
- `-l` - filetest operator for "is a symbolic link"; internally performs an `lstat` and leaves the result in _
- `$_` - default argument when EXPR is omitted
- `$!` - reason for failure when `lstat` returns the empty list

Regular expressions and pattern matching

m//

Search a string for a pattern and report whether - and what - it matched.

`m//` is the match operator. It compiles PATTERN as a regular expression (see the *regex guide*), runs it against a target string, and returns a value shaped by calling context and by the modifiers you apply. The target is whatever sits on the left of `=~` or `!~`; without a binding operator the target is `$_`. The leading `m` is optional when the delimiter is `/`, so `/PATTERN/` and `m/PATTERN/` mean the same thing.

Synopsis

```
$str =~ m/PATTERN/flags
$str =~ /PATTERN/flags
m/PATTERN/flags              # target is $_
/PATTERN/flags               # target is $_
$str =~ m{PATTERN}flags      # any paired non-word delimiters
```

What you get back

Context decides the shape of the return value.

- **Scalar context, no `/g`**: 1 on match, the empty string on no match. Both are usable as booleans; the empty string is a dual-value false that also equals `0` numerically.
- **List context, no `/g`**: the list of capture values `($1, $2, $3, ...)` on a successful match; if the pattern has no capture groups, the singleton `(1)`; on failure, the empty list. This is how `if (my ($x, $y) = $s =~ /(\w+)=(\w+)/)` works.
- **Scalar context, `/g`**: each call advances through the string, returning true for the next match and false once there are no more. `pos` on the target tracks where the next attempt will begin.
- **List context, `/g`**: every match in one shot. With capture groups, the flat list of all captures from every match. Without captures, the list of every full match.

Successful matches also populate the regex special variables (`$1` through `$9`, `$&`, `$``, `$'`, `$+`, `%+`, `%-`) for the enclosing dynamic scope. A failed match leaves them holding their previous values - always test the match itself, never a capture variable, to decide whether a match happened.

Global state it touches

- `$_` - the default target when no `=~` binding is given.
- `$1`, `$2`, ... - numbered captures, set on success, unchanged on failure.
- `$&`, `$``, `$'` - match, prematch, postmatch.
- `$+` - the highest-numbered capture that actually matched (useful with alternations).
- `%+`, `%-` - named-capture hashes.
- `${^LAST_SUCCESSFUL_PATTERN}` - the last pattern that matched in the current dynamic scope; also the pattern the empty form `m//` reuses (see *Edge cases*).
- `pos` on the target string - read and updated by `/g` matching; reset on failure unless `/c` is also set.
- Locale / Unicode rule sources when `/l`, `/u`, or `/d` are in effect.

Delimiters

With `m`, any pair of non-whitespace characters works as the delimiter, and bracketing pairs nest:

```
m/pattern/
m{pattern}
m[pattern]
m(pattern)
m<pattern>
m!pattern!
m#pattern#
m,pattern,
```

Picking a delimiter that does not appear in the pattern avoids backslash-clutter - known as LTS, *leaning toothpick syndrome*. A path-matching pattern reads cleanly with `m{...}` or `m!...!` and badly with `m/.../`.

Two delimiter choices change semantics:

- **`'` (single quote)** - no variable interpolation inside `PATTERN`. `m'$foo'` matches the literal four characters.
- **`?`** - `m?PATTERN?` matches only once between calls to `reset`. The leading `m` is mandatory; since Perl 5.22 the bare `?...?` form is a syntax error.

When the delimiter is a word character (a letter or digit), a space is required after `m`: `m q foo q` is legal, `mqfooq` is not.

Modifiers

Pattern-compile modifiers (also accepted by `qr`, `s`, and `split`):

- **`m`** - multi-line: `^` and `$` match at every embedded newline, not only at string ends.
- **`s`** - single-line: `.` matches every character including newline.
- **`i`** - case-insensitive matching.
- **`x`** - ignore whitespace and #-comments in the pattern; `xx` extends this into character classes.
- **`p`** - preserve copies of the matched string. Since 5.20 this is a no-op - `${^PREMATCH}`, `${^MATCH}`, `${^POSTMATCH}` are always available after a successful match.
- **`a`**, **`u`**, **`l`**, **`d`** - character-set rules for `\d`, `\s`, `\w`, and the POSIX classes. `/a` restricts them to ASCII; `/aa` additionally forbids ASCII/non-ASCII matching under `/i`.
- **`n`** - non-capturing: `(…)` behaves like `(?:…)` and does not populate `$1`, `$2`, ….
- **`o`** - compile the pattern exactly once even if interpolated variables change. Almost always the wrong tool; use `qr` to build a reusable compiled pattern instead.

Match-process modifiers (specific to `m//` and `s///`):

- **`g`** - global matching. Scalar-context behavior is iterative (advances `pos` on each call); list-context behavior returns every match at once.
- **`c`** - only meaningful with `/g`. A failed `/g` match keeps `pos` where it was instead of resetting to the start; required for `lex`-style scanners built around `\G`.

Examples

Test whether a string contains a pattern:

```
if ($line =~ /error/i) {
    warn "matched: $line";
}
```

Bind a capture in one go:

```
if (my ($key, $val) = $line =~ /^(\w+)\s*=\s*(.*)$/) {
    $config{$key} = $val;
}
```

Pull every number out of a string with list-context `/g`:

```
my @nums = "x=1 y=22 z=333" =~ /(\d+)/g;
# @nums = (1, 22, 333)
```

Iterate matches one at a time with scalar-context `/g`, using `pos` to see where the engine is:

```
my $s = "foo 1 bar 22 baz 333";
while ($s =~ /(\d+)/g) {
```

(continues on next page)

(continued from previous page)

```
    printf "matched %s at offset %d\n", $1, pos($s) - length($1);
}
```

Extended form with the x modifier and named captures:

```
if ($ts =~ m{
        ^ (?<year>\d{4}) -
          (?<mon> \d{2}) -
          (?<day> \d{2}) $
    }x) {
    printf "year=%s mon=%s day=%s\n", $+{year}, $+{mon}, $+{day};
}
```

Avoid LTS by picking a delimiter that does not appear in the pattern:

```
next if $path =~ m{^/usr/local/};
```

Use `\G` with `m//gc` to walk a string token-by-token without losing position on a failed arm:

```
while (1) {
    if    ($s =~ /\G(\d+)/gc)    { push @tok, ['num',  $1] }
    elsif ($s =~ /\G(\w+)/gc)    { push @tok, ['word', $1] }
    elsif ($s =~ /\G(\s+)/gc)    { next                    }
    else                         { last                    }
}
```

Edge cases

- **Empty pattern**: `//` and `m//` reuse the last successfully matched pattern in the current dynamic scope. If nothing has matched yet, an empty pattern matches everywhere. Passing user input straight into `m/$pat/` when `$pat` might be empty is a sharp edge - wrap it in a non-capturing group: `m/(?:$pat)/`. The last successful pattern is also readable as `${^LAST_SUCCESSFUL_PATTERN}`.
- **Defined-or ambiguity**: Perl resolves `$x // $y` as the defined-or operator, never as two empty matches. In pathological positions (`print $fh //`) Perl still assumes defined-or; force a match by writing `m//` explicitly or spacing out the delimiters.
- **Failed match leaves captures stale**: `$1` after a failing `/…/` still holds the capture from the *previous* successful match - always gate capture use on the match result.
- **`/o` with changing variables**: `m/$x/o` locks the first value of `$x` into the compiled pattern. Later changes to `$x` are silently ignored. Reach for `qr` instead when you want an explicit, reusable compilation.
- **Interpolation when the delimiter is `'`**: `m'$var'` is a literal-dollar-var match. This is rarely what you want, and the same effect is available with `\Q…\E` or `quotemeta` in any other delimiter.
- **`/g` plus target modification**: modifying the target between `/g` iterations resets `pos` to the start. Iterate over a copy if you need to mutate the original as you go.

- **`\G` outside `/g`**: without `/g`, `\G` anchors at the `pos` the target had at call time and matches at most once. On a string that has never had a `/g` applied, `\G` is equivalent to `\A`.
- **`m?…?` reset scope**: `reset` clears `m??` state only for the current package. A `m??` in one package is not affected by `reset` called from another.
- **Comparison operators near an empty regex**: `$x //= 1` is always the defined-or assignment; if you genuinely want the empty regex, write `m//` rather than `//`.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `qr` - compile a pattern once and reuse it; avoids `/o` and keeps the pattern a first-class value
- `s` - same pattern syntax, replaces what it matches
- `tr` - character-by-character translation; a different tool with a superficially similar shape
- `split` - when you want the pieces between matches rather than the matches themselves
- `pos` - read or set the position `/g` matching resumes from
- *Regular expressions guide* - the regex language itself: assertions, character classes, backreferences, named captures

Lists

map

Apply a block or expression to each element of a list and return the flattened results.

`map` evaluates `BLOCK` (or `EXPR`) once per element of `LIST`, with `$_` aliased to that element, and collects everything each evaluation returns into a single flat list. Because each evaluation may yield zero, one, or many values, the output length is independent of the input length - that flattening is the point of `map`. In list context `map` returns the generated list; in scalar context it returns the total element count.

Synopsis

```
my @out = map { BLOCK } LIST;        # block form
my @out = map EXPR, LIST;            # expression form
my @out = map { f($_) } grep { ... } sort LIST;   # chained
```

What you get back

A single flat list built by concatenating the return values of each evaluation, in input order:

- Return one value per input → transform (same length out).
- Return a list per input → 1-to-N expansion (longer out).
- Return the empty list `()` → skip this input (shorter out; `map`-as-filter idiom).

In scalar context the return value is the count of generated elements, not a reference and not a boolean. Assigning the result to a hash is well-defined: the flat list is consumed as alternating key/value pairs.

```
my @doubled = map { $_ * 2 }          1 .. 4;     # (2, 4, 6, 8)
my @words   = map { split /\s+/, $_ } @lines;     # 1-to-N
my @kept    = map { $_ > 0 ? $_ : () } @nums;     # filter
my $count   = map { $_ * $_ } @nums;              # scalar ctx
```

Global state

`$_` is **aliased** to each element of `LIST`, not copied. Modifying `$_` inside the block modifies the source element. This is occasionally useful but usually a footgun - reach for a plain `foreach` loop when you mean to mutate in place.

```
my @v = (1, 2, 3);
my @r = map { $_ *= 10; $_ } @v;
# @v is now (10, 20, 30) - the source was mutated
```

Aliasing to a non-variable (a literal, a function return value, a hash-slice element that doesn't exist yet) and then assigning to `$_` fails at runtime with `Modification of a read-only value`. See `$_`.

`map` also localises `$_` around the call, so an outer `$_` visible to the caller is restored when `map` returns.

Examples

Transform - uppercase every word:

```
my @upper = map { uc } @words;
```

1-to-N expansion - split each line on whitespace into its tokens:

```
my @tokens = map { split /\s+/ } @lines;
```

Filter idiom - keep only truthy, transformed values:

```
my @positives = map { $_ > 0 ? $_ : () } @numbers;
```

Build a hash from a list - each evaluation returns a two-element list:

```
my %seen = map { $_ => 1 } @keys;
my %idx  = map { ($keys[$_] => $_) } 0 .. $#keys;
```

Composed pipeline - sort, filter, transform. `map`, `grep`, and `sort` all take a `LIST` on the right, so they chain right-to-left:

```
my @top = map  { "$_->{name}: $_->{score}" }
          sort { $b->{score} <=> $a->{score} }
          grep { $_->{score} >= 50 }
          @records;
```

Flatten structure - each input contributes a variable number of outputs:

```
my @leaves = map { ref($_) eq 'ARRAY' ? @$_ : $_ } @mixed;
```

Pair with index using `List::Util::pairs` or an explicit index variable - `map` itself does not expose one:

```
my @numbered = do {
    my $i = 0;
    map { sprintf "%d. %s", ++$i, $_ } @items;
};
```

Edge cases

- **`$_` is aliased, not copied.** Assigning to `$_` writes back to the source. If the source element is read-only (a constant, the return value of a function call flattened into `LIST`) the assignment raises `Modification of a read-only value`. Treat `$_` as input-only unless you specifically want the side effect.
- **`map` is not a loop.** `last`, `next`, `redo`, and loop labels do not apply to `map`. They refer to the enclosing real loop and will behave accordingly - usually not what the author intended. Use `foreach` when you need loop-control flow.
- **Void context is deprecated.** `map { ... } LIST;` with the result discarded emits a `Useless use of map in void context` warning under `use warnings`. When you want the side effects, not the list, write a `foreach`.
- **`BLOCK` vs `EXPR` parser ambiguity.** `{` starts both a block and a hash-reference constructor. `map { ... } LIST` and `map { ... }, LIST` look identical to the parser at the opening brace; Perl guesses based on what it finds next. A block starting with what looks like a hash key plus `=>` is especially likely to be guessed wrong:

  ```
  my %h = map {  "\L$_" => 1  } @a;   # guessed EXPR, wrong
  my %h = map { +"\L$_" => 1  } @a;   # unary + forces BLOCK
  my %h = map {; "\L$_" => 1  } @a;   # leading ; forces BLOCK
  my %h = map { ($_, 1) }       @a;   # parens - unambiguous
  my %h = map { lc($_) => 1 }   @a;   # function call -
   ↪unambiguous
  ```

  To return an anonymous hash per element, force the hash-ref constructor with `+{`:

```
my @hashes = map +{ lc($_) => 1 }, @array;
```

- **Order is stable**. `map` evaluates left-to-right and concatenates results left-to-right. The generated list preserves input order, with each input's contribution appearing contiguously.
- **Empty `LIST`**. `map` evaluates the block zero times and returns the empty list. The block is not evaluated for its side effects.
- **Nested `map`**. Inner `map` sees its own `$_`; the outer `$_` is shadowed for the duration of the inner evaluation. Capture the outer value first if both are needed:

```
my @pairs = map {
    my $outer = $_;
    map { [$outer, $_] } @inner
} @outer;
```

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `grep` - same shape as `map`, but keeps elements for which the block is true rather than transforming them; chains naturally with `map` and `sort`
- `sort` - orders a list; commonly composed with `map` via the Schwartzian transform (`map` → `sort` → `map`)
- `for` / `foreach` - use when you want iteration for side effects, mutation of the source, or loop-control (`last`/`next`/`redo`)
- `x` - list repetition operator; orthogonal to `map` but often used to build the `LIST` that `map` consumes
- `List::Util::reduce` - fold a list to a single value; use when the result is not a list but an aggregate (sum, max, first match)
- `$_` - the element alias inside the block; read-only when the source element is a literal or function return value

Control flow · Classes and OO

method

Declare a named instance method inside a `class` block.

`method` defines a subroutine that expects to be invoked on an object of the enclosing class. It behaves like `sub`, with two additional guarantees that `sub` does not give you: the lexical `$self` is bound to the invocant automatically, and signatures are always on - you write parameters as if `use feature 'signatures'` were in effect, and `$self` does not appear in that signature.

`method` is only valid inside a `class` declaration. Outside a class block it is a compile-time error.

Experimental feature

The `class`, `method`, and `field` keywords are part of the experimental `class` feature (introduced in Perl 5.38). Enabling it with `use feature 'class';` emits a warning in the `experimental::class` category. The syntax and semantics may still change between Perl releases.

Synopsis

```
method NAME BLOCK
method NAME SIGNATURE BLOCK
method NAME : ATTRS BLOCK
method SIGNATURE BLOCK        # anonymous
method BLOCK                  # anonymous
```

What you get back

`method` is a declaration, not an expression, in its named form - it installs the method in the enclosing class's method table and evaluates to nothing useful at the statement position. The anonymous form returns a code reference that remembers the class context and can be called later with an object invocant.

The body of a `method` is called with the object as the invocant. The return value of the method is whatever the block returns, following the same context rules as a regular subroutine call.

What `method` gives you that `sub` does not

Four things differ from writing `sub` inside a `class` block:

- **`$self` is pre-declared.** A lexical `$self` is created in the method's scope and initialised to the invocant. You do not - and should not - shift it off `@_`.
- **Signatures are always enabled.** You can write `method greet($name) { ... }` without `use feature 'signatures'`. The `$self` invocant is consumed before the signature binds, so it never appears in the parameter list.
- **Fields are in scope.** Any `field` variables declared in the enclosing class are visible by their lexical names inside the method body. A plain `sub` inside a `class` block does not see fields.
- **It is registered as a method.** The method table of the class records the name, which is what the method-resolution machinery looks up for `$obj->name(...)` dispatch. A plain `sub` inside the class block is a helper, not a method.

Put the other way: a `sub` inside a `class` block is an ordinary package subroutine that happens to live in that package's namespace. It has no `$self`, no field visibility, and is not part of the class's method interface - call it via `ClassName::helper(...)` if you must, not via an arrow.

Examples

A minimal class with a method and a field:

```
use v5.38;
use feature 'class';
no warnings 'experimental::class';

class Greeter {
    field $greeting = "Hello";

    method greet($name) {
        return "$greeting, $name";
    }
}

my $g = Greeter->new;
say $g->greet("world");           # Hello, world
```

Signature defaults work exactly as they do for `sub`:

```
class Greeter {
    field $greeting = "Hello";

    method greet($name = "someone") {
        return "$greeting, $name";
    }
}

say Greeter->new->greet;          # Hello, someone
```

A method with no parameters still uses the signature form - the empty `()` is not required, but it is allowed and is the common style when the method takes arguments elsewhere in the class:

```
class Counter {
    field $n = 0;

    method inc        { $n++ }
    method value ()   { return $n }
}
```

An anonymous method - useful as a factory or as a callback that needs access to `$self` and fields:

```
class AnonMethodFactory {
    method make_printer {
        return method { say "called on $self" };
    }
}
```

A lexical (private) method declared with `my method`. Invoke it via the `->&` operator, which bypasses method lookup and calls the lexical directly as if it were a method:

```
class LexicalMethod {
    my method _check ($x, $y) {
        return $x > 0 && $y > 0;
    }

    method process ($x, $y) {
        return unless $self->&_check($x, $y);
        # ...
    }
}
```

Regular `sub` inside a class, contrasted with `method` - the `sub` is a helper, the `method` is the public interface:

```
class Temperature {
    field $celsius;

    sub _c_to_f ($c) { $c * 9 / 5 + 32 }   # helper, no $self, no
↪fields

    method fahrenheit {
        return _c_to_f($celsius);
    }
}
```

Edge cases

- **Outside a `class` block is a compile-time error.** `method foo { ... }` at the top level, or inside a `package` that is not a `class`, does not parse. Use `sub` there.
- **`$self` is read-only in the sense that you cannot declare another lexical with the same name** at the top of the method - it is already in scope. Assigning to `$self` (e.g. `$self = ...`) is permitted but rarely useful and does not change the caller's invocant.
- **`@_` is not the argument list you expect.** Under the signatures that `method` enables, `@_` inside the body is empty after the signature binds, just as in any signature-using sub. Read parameters from the signature, not `@_`.
- **Method-resolution order is set by the enclosing class.** `method` itself does not take an MRO option; inheritance and dispatch follow whatever the `class` declaration configured via `:isa` and related attributes.
- **`method` does not participate in prototypes.** Methods are invoked through dispatch, not direct symbol-table calls, so prototypes (the `:prototype(...)` attribute on a `sub`) have no effect here. Use the signature to constrain the call shape.
- **Attributes.** The `method NAME : ATTRS BLOCK` form accepts attributes the same way `sub NAME : ATTRS BLOCK` does. The set of useful attributes on methods is still settling with the `class` feature; consult upstream before relying on custom ones.
- **Anonymous-method return-from-method trap.** `return method { ... }` re-

turns a code reference. `method { ... }` as the last expression of a method without `return` is still a code reference, but readers often misread it as a block - prefer explicit `return` for clarity.

Differences from upstream

Fully compatible with upstream Perl 5.44.

The `class` feature - and therefore `method` - is marked experimental in upstream; pperl inherits that status and the `experimental::class` warning category.

See also

- `class` - declares the package that `method` lives in; `method` only parses inside a `class` block
- `field` - per-instance variables that are visible by name inside every `method` of the enclosing class
- `sub` - declare a regular subroutine; use this for helpers inside a class that do not need `$self` or field access
- `__CLASS__` - the name of the current class, useful inside methods that need to dispatch back to class-level code
- `return` - explicit return from a method body; recommended when the last expression is itself a `method { ... }` anonymous form
- `my` - prefix a `method` with `my` to declare a lexical (private) method invoked via `->&`

Filehandles, files, directories

mkdir

Create a single directory on the filesystem.

`mkdir` asks the operating system to create the directory named by `FILENAME`, with permission bits given by `MODE` (further restricted by the process `umask`). It creates exactly **one** directory - every parent component in `FILENAME` must already exist. For recursive creation, reach for `File::Path::make_path` instead.

Synopsis

```
mkdir FILENAME, MODE
mkdir FILENAME
mkdir
```

What you get back

1 on success, `0` on failure with `$!` set to the system errno. Unlike most I/O primitives, the failure value is a plain false `0`, not `undef` - check the return value, then consult `$!` for the reason:

```
mkdir $path, 0755
    or die "mkdir $path failed: $!";
```

Default arguments

Both arguments default:

- `MODE` defaults to `0777`. The kernel masks the supplied mode with the current process `umask`, so the directory's effective mode is `MODE & ~umask`. A typical `umask` of `022` turns the default `0777` into `0755` on disk.
- `FILENAME` defaults to `$_`. Bare `mkdir;` inside a loop over pathnames creates each in turn:

```
mkdir for qw(logs cache tmp);
```

Pass `MODE` explicitly whenever the directory should be private (mail spools, key stores, session directories). Otherwise prefer a permissive `MODE` and let the user's `umask` narrow it - the `umask` page discusses the trade-off in detail. Setting bits outside the permission range (e.g. setuid/setgid/sticky) yields implementation-defined behaviour per POSIX 1003.1-2008.

Global state it touches

- Sets `$!` (errno) on failure; leaves it untouched on success.
- Reads `FILENAME` from `$_` when called with no arguments.
- The effective permission bits depend on the process `umask`.

Examples

Create a directory with an explicit mode:

```
mkdir "build", 0755
    or die "mkdir build: $!";
```

Create it only if missing, without treating "already there" as an error:

```
use Errno qw(EEXIST);
unless (mkdir "cache", 0700) {
    die "mkdir cache: $!" unless $! == EEXIST;
}
```

Private directory - do not let `umask` widen it:

```
mkdir "$ENV{HOME}/.secrets", 0700
    or die "mkdir secrets: $!";
```

Bare form over a list, using `$_` as the filename:

```
for (qw(a b c)) {
    mkdir or warn "mkdir $_: $!";
}
```

Recursive creation is **not** `mkdir`'s job - use `File::Path`:

```
use File::Path qw(make_path);
make_path("var/log/app/2026", { mode => 0755 })
    or die "make_path failed: $!";
```

Edge cases

- **Parent must exist**. `mkdir "a/b/c"` fails with `ENOENT` if `a/b` is missing. Use `File::Path::make_path` for deep paths.
- **Target already exists**. Fails with `EEXIST`, regardless of whether the existing entry is a directory, file, or symlink. Check `$!` against `Errno::EEXIST` when "already there" is acceptable.
- **Trailing slashes**. POSIX 1003.1-1996 allows any number of trailing slashes; Perl strips them before the syscall so that filesystems which reject `"foo/"` still behave consistently.
- **Permission mismatch**. `MODE` is masked by `umask`. If you need an exact mode on disk, either save and restore the umask, or `chmod` the directory after creation:

  ```
  mkdir $dir, 0700 or die $!;
  chmod 0700, $dir;                    # defeat any umask surprises
  ```

- **`MODE` omitted entirely**. `mkdir $path` is `mkdir $path, 0777` - still masked by `umask`, so the on-disk mode is typically `0755`, not `0777`.
- **Non-permission bits in `MODE`**. Setuid / setgid / sticky bits on a `mkdir` call have implementation-defined results. Set them with a follow-up `chmod` if you need them.
- **Read-only filesystem or missing write permission on the parent** fails with `EROFS` or `EACCES` respectively. `$!` discriminates.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `rmdir` - remove an empty directory; the inverse operation
- `chmod` - adjust permission bits after creation when `umask` would otherwise narrow them
- `umask` - the mask that silently restricts every `mkdir MODE`
- `stat` - inspect an existing path's type and mode, e.g. to distinguish "already a directory" from "exists as a file"

- `File::Path::make_path` - recursive directory creation; the right tool when intermediate parents may not exist

SysV IPC

msgctl

Perform a control operation on a System V IPC message queue.

`msgctl` is a direct wrapper around the C `msgctl(2)` system call. It inspects, modifies, or removes the message queue identified by `ID`. The operation selected by `CMD` determines how `ARG` is used - either as an input descriptor, an output buffer, or ignored entirely. The symbolic constants needed for `CMD` (`IPC_STAT`, `IPC_SET`, `IPC_RMID`, and on Linux `IPC_INFO`, `MSG_STAT`, `MSG_INFO`) live in `IPC::SysV`, which you have to pull in before the names resolve.

Synopsis

```
use IPC::SysV;

msgctl $id, IPC_STAT, $buf       # read queue info into $buf
msgctl $id, IPC_SET,  $buf       # write queue info from $buf
msgctl $id, IPC_RMID, 0          # destroy the queue
```

What you get back

Return value follows the same convention as `ioctl`:

- `undef` on error, with `$!` set.
- The string `"0 but true"` when the underlying call returned `0`. This is the zero-valued success case - it tests true in boolean context and compares equal to `0` in numeric context, so one check handles both success and failure.
- The raw non-zero integer return otherwise (for example the index returned by Linux-specific `MSG_STAT`).

The `"0 but true"` convention exists precisely so that `msgctl(...) or die` does the right thing even when the syscall legitimately returns zero:

```
msgctl $id, IPC_RMID, 0
    or die "msgctl IPC_RMID failed: $!";
```

How ARG is used

`ARG` plays three different roles depending on `CMD`. The kernel only touches the memory in the directions the command requires:

- **`IPC_STAT`** - `ARG` must be a **writable scalar**. The kernel fills it with the packed `msqid_ds` structure describing the queue (permissions, owner, last-send/last-receive times, queue byte limits, counts). Unpack with `IPC::SysV` helpers rather than hand-rolling the layout; `msqid_ds` varies across kernels and libc versions.

- **IPC_SET** - ARG must contain a packed `msqid_ds` **produced by a prior IPC_STAT** on the same (or compatible) queue. Only a few fields are honoured: `msg_perm.uid`, `msg_perm.gid`, `msg_perm.mode` (low 9 bits), and `msg_qbytes`. The rest is ignored.
- **IPC_RMID** - ARG is unused. Pass `0`. The queue is marked for destruction; any blocked `msgsnd` / `msgrcv` on it fails with `EIDRM`.

On Linux only, `MSG_STAT` and `MSG_INFO` also accept a writable scalar for ARG and return extra data; these are non-portable and should be gated behind a runtime check.

Global state it touches

- `$!` - set on failure with the system errno (`EPERM`, `EACCES`, `EINVAL`, `EIDRM`, `EFAULT` are the common values).
- No other special variables are touched.

Examples

Read the queue header and inspect the current byte limit:

```
use IPC::SysV qw(IPC_STAT);

my $buf = "";
msgctl($id, IPC_STAT, $buf)
    or die "msgctl IPC_STAT failed: $!";

# Layout is platform-dependent; use IPC::SysV helpers to decode.
```

Shrink the queue's byte budget. Round-trip through `IPC_STAT` so the immutable fields keep their current kernel values:

```
use IPC::SysV qw(IPC_STAT IPC_SET);

my $buf = "";
msgctl($id, IPC_STAT, $buf) or die "stat: $!";
# ... modify msg_qbytes inside $buf via IPC::SysV ...
msgctl($id, IPC_SET,  $buf) or die "set: $!";
```

Destroy a queue when done. ARG is ignored, but it still has to be passed - `0` is the conventional placeholder:

```
use IPC::SysV qw(IPC_RMID);

msgctl($qid, IPC_RMID, 0)
    or warn "could not remove queue $qid: $!";
```

Idiomatic success test that works for the `"0 but true"` case:

```
my $rc = msgctl($id, IPC_RMID, 0);
die "msgctl failed: $!" unless defined $rc;   # undef = error
# $rc is now either "0 but true" or a positive integer
```

Edge cases

- **`IPC::SysV` not loaded**: bare constant names like `IPC_STAT` become barewords and, under `use strict`, a compile-time error. Always `use IPC::SysV` first.
- **Scalar passed to `IPC_STAT` is not writable**: croaks with `Modification of a read-only value attempted`. Use a plain `my $buf = ""` - not a constant, not a string literal.
- **Length mismatch on `IPC_SET`**: if `ARG` is shorter than the kernel's `msqid_ds`, the call fails with `EFAULT`. Always start from an `IPC_STAT` buffer rather than packing one from scratch.
- **Removed queue (`EIDRM`)**: once `IPC_RMID` has been issued, any further operation on the same ID fails with `$!` set to `EIDRM`, not `EINVAL`. Scripts that poll a queue must treat `EIDRM` as "queue is gone, stop."
- **Permission errors**: `IPC_SET` and `IPC_RMID` require effective UID match or `CAP_SYS_ADMIN`. Kernel returns `EPERM`, not `EACCES`, for ownership failures.
- **Return-value trap**: `if (msgctl(...) == 0)` is **wrong** - on success the value is the string `"0 but true"`, which stringifies as non-empty but compares numerically equal to `0`. Test with `defined` for the error case, or rely on `or die`.
- **Platform availability**: System V IPC is not universal. On systems where the kernel omits it, `msgctl` raises the runtime exception `msgctl not implemented`.

Differences from upstream

Fully compatible with upstream Perl 5.44 on Linux. System V IPC is the only supported IPC surface; pperl targets Linux, so the portability caveats that upstream `perlport` lists for non-Linux platforms do not apply.

See also

- `msgget` - obtain the queue ID that `msgctl` operates on
- `msgsnd` - send a message to the queue
- `msgrcv` - receive a message from the queue
- `semctl` - the semaphore-set analogue, same return convention and same `IPC_STAT` / `IPC_SET` / `IPC_RMID` pattern
- `shmctl` - the shared-memory-segment analogue
- `IPC::SysV` - the constants (`IPC_STAT`, `IPC_SET`, `IPC_RMID`) and the `msqid_ds` pack/unpack helpers this call requires

SysV IPC

msgget

Create or look up a System V IPC message queue and return its id.

`msgget` is the entry point to the System V message-queue family: every later `msgsnd`, `msgrcv`, or `msgctl` call needs an id, and this is where that id comes from. `KEY` names the

queue across the whole system (a `key_t` integer, typically built with `IPC::SysV::ftok` or the special value `IPC_PRIVATE`). `FLAGS` is the bitwise-or of permission bits (low nine, same layout as file modes) and optional `IPC_CREAT` / `IPC_EXCL` flags. The call is a direct wrapper around the `msgget(2)` syscall.

Synopsis

```
use IPC::SysV qw(IPC_CREAT IPC_EXCL IPC_PRIVATE S_IRUSR S_IWUSR);

my $id = msgget($key, IPC_CREAT | 0600);
my $id = msgget(IPC_PRIVATE, 0600);
my $id = msgget($key, IPC_CREAT | IPC_EXCL | 0600);
```

What you get back

A non-negative integer message-queue id on success, or `undef` on failure (with `$!` set to the `errno` value from the underlying syscall). The id is an opaque handle - do not do arithmetic on it, just pass it back to the other `msg*` built-ins.

```
my $id = msgget($key, IPC_CREAT | 0600)
    // die "msgget($key) failed: $!";
```

Global state it touches

- `$!` - set on failure to the `errno` from `msgget(2)` (commonly `EACCES`, `EEXIST`, `ENOENT`, `ENOSPC`, `ENOMEM`).

No other special variables are involved. The queue itself lives in the kernel, not in the Perl process, and persists across `exec` and after the creating process exits - it must be removed explicitly with `msgctl` and `IPC_RMID`.

Examples

Create a private queue that no other process can name - the usual choice when the queue is shared only with children of this process:

```
use IPC::SysV qw(IPC_PRIVATE);
my $id = msgget(IPC_PRIVATE, 0600) // die "msgget: $!";
```

Create or attach to a named queue using a `key_t` derived from a file path. `ftok` turns a path plus a single-character project id into a stable key so unrelated processes can agree on a queue:

```
use IPC::SysV qw(ftok IPC_CREAT);
my $key = ftok("/var/run/myapp", 'M');
my $id  = msgget($key, IPC_CREAT | 0600) // die "msgget: $!";
```

Create exclusively - fail if the queue already exists. This is how you detect that a previous instance of your program is still around:

```
use IPC::SysV qw(IPC_CREAT IPC_EXCL);
my $id = msgget($key, IPC_CREAT | IPC_EXCL | 0600);
if (!defined $id) {
    die "queue already exists" if $!{EEXIST};
    die "msgget: $!";
}
```

Attach to an existing queue without creating one - omit `IPC_CREAT`. Returns `undef` with `$!` set to `ENOENT` if no such queue exists:

```
my $id = msgget($key, 0) // die "no queue for key $key: $!";
```

Remove the queue when done. `msgget` itself has no teardown counterpart; removal goes through `msgctl`:

```
use IPC::SysV qw(IPC_RMID);
msgctl($id, IPC_RMID, 0) or warn "msgctl IPC_RMID: $!";
```

Edge cases

- **`KEY` must be an integer**, typically a `key_t`. Passing a string triggers the usual numeric-conversion rules and will almost certainly name the wrong queue. Build keys with `IPC::SysV::ftok` or use `IPC_PRIVATE`.
- **`IPC_PRIVATE` always creates a new queue**, regardless of whether `IPC_CREAT` is set. The key is not reusable - you cannot `msgget` the same queue later from an unrelated process using `IPC_PRIVATE`.
- **Permission bits apply at every call, not just creation**. Attaching to an existing queue without `IPC_CREAT` still checks the permission bits against the queue's owner/group and the current process's effective uid/gid. Mismatched bits give `EACCES`.
- **`IPC_EXCL` without `IPC_CREAT` is meaningless** - the kernel ignores it. If you want exclusive creation, always pair them.
- **System-wide limits**. Creation can fail with `ENOSPC` when the kernel's `MSGMNI` limit is hit, or `ENOMEM` under memory pressure. These are not bugs in your program; they are operational conditions.
- **Queue lifetime outlives processes**. A queue created here persists until explicitly removed with `msgctl` and `IPC_RMID`, or until the system reboots. Forgotten queues leak kernel resources that `ipcs` and `ipcrm` are used to find and clean up.
- **Not all systems have SysV message queues.** On platforms that omit them, the built-in croaks with `msgget not implemented on this architecture`. The portability warning in upstream POD applies.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `msgsnd` - send a message to the queue returned by `msgget`
- `msgrcv` - receive a message from that queue
- `msgctl` - query, set options on, or remove the queue; teardown after `msgget` goes through here
- `semget` - same API shape for System V semaphores when you need synchronization rather than data transfer
- `shmget` - same API shape for System V shared memory when you need direct memory sharing rather than message passing
- `IPC::SysV` - provides `ftok` and the `IPC_*` / `S_*` constants `msgget` expects in `FLAGS`

SysV IPC

msgrcv

Receive a message from a System V IPC message queue.

`msgrcv` is the Perl binding for the `msgrcv(2)` syscall. It pulls one message off the queue identified by `ID`, stores the message type and payload into `VAR`, and returns a truthy value on success or false on error. `SIZE` is the maximum number of payload bytes the caller is willing to accept; `TYPE` selects which message to receive; `FLAGS` is the bitmask passed to the kernel (`IPC_NOWAIT`, `MSG_NOERROR`, etc.).

The returned buffer is **not** just the message body - the first `sizeof(long)` bytes are the message type the sender passed to `msgsnd`, followed by the payload. Use `unpack` with the `l! a*` template to split them apart.

Synopsis

```
use IPC::SysV qw(IPC_NOWAIT MSG_NOERROR);

msgrcv $qid, my $buf, 1024, 0, 0            or die "msgrcv: $!";
msgrcv $qid, my $buf, 1024, 0, IPC_NOWAIT;  # non-blocking
my ($type, $payload) = unpack "l! a*", $buf;
```

What you get back

True on success, false on error with `$!` set. On success, `VAR` is overwritten with the received frame: a native `long` (the message type) immediately followed by up to `SIZE` bytes of payload. The variable is marked tainted under `-T`.

`VAR` is grown or shrunk to fit the actual frame length - you do not need to pre-size it. The content before the call is discarded.

To get the type and payload as separate values:

```
my ($type, $msg) = unpack "l! a*", $buf;
```

Arguments

- `ID` - the queue id previously returned by `msgget`.
- `VAR` - a lvalue scalar that receives the raw frame. Its previous value is discarded on success.
- `SIZE` - the maximum payload size in bytes. If the next matching message is larger than this, `msgrcv` fails with `E2BIG` and the message stays on the queue - **unless** `MSG_NOERROR` is set in `FLAGS`, in which case the message is truncated silently and the extra bytes are lost.
- `TYPE` - selects which message to receive:
  - `0` - the first message on the queue, whatever its type.
  - positive `N` - the first message whose type is exactly `N`.
  - negative `-N` - the first message of the lowest type that is `<= abs(N)`, useful for priority queues.
- `FLAGS` - bitmask of `IPC_NOWAIT` (return `ENOMSG` immediately instead of blocking when no matching message is available), `MSG_NOERROR` (truncate instead of failing on oversized messages), and the Linux-specific `MSG_EXCEPT` (with a positive `TYPE`, receive the first message whose type is *not* `TYPE`).

The constants live in `IPC::SysV`; the bare names do not become visible until you import them.

Global state it touches

- `$!` - set on error to the underlying `errno` (`ENOMSG`, `E2BIG`, `EIDRM`, `EINVAL`, `EACCES`, `EFAULT`, `EINTR`).
- Taint mode - the received `VAR` is marked tainted, since its contents came from another process.

Examples

Blocking receive of any message, then split type and payload:

```
use IPC::SysV qw(IPC_CREAT S_IRUSR S_IWUSR);

my $qid = msgget(IPC_PRIVATE, S_IRUSR | S_IWUSR) // die "msgget: $!
 ↪";
msgrcv $qid, my $buf, 4096, 0, 0            or die "msgrcv: $!";
my ($type, $payload) = unpack "l! a*", $buf;
print "got type=$type payload=$payload\n";
```

Receive only messages of type 42:

```
msgrcv $qid, my $buf, 4096, 42, 0           or die "msgrcv: $!";
```

Non-blocking poll - return immediately if the queue is empty:

```
use IPC::SysV qw(IPC_NOWAIT);
use Errno qw(ENOMSG);

if (msgrcv $qid, my $buf, 4096, 0, IPC_NOWAIT) {
    my ($type, $msg) = unpack "l! a*", $buf;
    handle($type, $msg);
} elsif ($! == ENOMSG) {
    # queue was empty; try again later
} else {
    die "msgrcv: $!";
}
```

Truncate oversized messages rather than failing:

```
use IPC::SysV qw(MSG_NOERROR);

msgrcv $qid, my $buf, 64, 0, MSG_NOERROR    or die "msgrcv: $!";
# payload is at most 64 bytes, excess discarded without error
```

Priority-style receive - take the lowest-numbered message whose type is <= 3:

```
msgrcv $qid, my $buf, 4096, -3, 0           or die "msgrcv: $!";
```

Edge cases

- **VAR must be an lvalue**. Passing a constant or a read-only scalar (for instance an element of `@_` that was passed by reference to a literal) raises `Modification of a read-only value attempted`.
- **Frame layout is native `long`, not four bytes**. On 64-bit Linux the type occupies 8 bytes; on 32-bit systems it occupies 4. The `l!` in `unpack "l! a*"` - the `!` - is mandatory; a plain `l` assumes 4 bytes and will corrupt payloads on 64-bit hosts.
- **`SIZE` excludes the type header for `MSG_NOERROR` purposes**. The kernel compares `SIZE` against the payload length, not the frame length. Size the buffer for the largest payload you expect; the type header is written beyond it.
- **Signal interruption**. A blocking `msgrcv` returns false with `$! == EINTR` when a signal arrives before a message is available. Retry the call unless the signal is meant to abort the receive loop.
- **Queue removed while blocked**. If another process calls `msgctl` with `IPC_RMID` while this process is asleep in `msgrcv`, the call returns false with `$! == EIDRM`.
- **Permissions**. Receiving requires read permission on the queue. `EACCES` means the queue exists but the current uid/gid doesn't match the queue's mode bits.
- **`TYPE = 0` sees FIFO order**. There is no "any" priority short of explicit negative `TYPE` - messages of all types are pulled in send order.
- **Kernel support**. Linux kernels built without `CONFIG_SYSVIPC`, and container runtimes that don't expose the IPC namespace, fail every `msgrcv` with `ENOSYS`. Check during startup if your deployment target is uncertain.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `msgsnd` - the counterpart that puts messages on the queue; its `pack "l! a*"` template is the inverse of what you `unpack` here
- `msgget` - obtain the queue id `ID` you pass in
- `msgctl` - inspect or delete the queue, including the `IPC_RMID` that causes a blocked `msgrcv` to return `EIDRM`
- `unpack` - split the returned frame into type and payload with `l! a*`
- `IPC::SysV` - source of the `IPC_NOWAIT`, `MSG_NOERROR`, and related flag constants
- `IPC::Msg` - object-oriented wrapper that hides the pack/unpack dance behind a `rcv` method

SysV IPC

msgsnd

Send a message to a System V IPC message queue.

`msgsnd` is a thin wrapper around the `msgsnd(2)` syscall. It appends `MSG` to the queue identified by `ID` (the queue id previously returned by `msgget`), and returns true on success or false on error with `$!` set. `MSG` is **not** free-form: it must start with a native `long` message type, followed by the payload bytes. `FLAGS` control blocking and error-on-overflow behaviour, typically `0` or `IPC_NOWAIT`.

Synopsis

```
msgsnd ID, MSG, FLAGS
```

What you get back

True on success, false on failure. On failure `$!` is set to the `errno` from `msgsnd(2)` - common values are `EAGAIN` (queue full, `IPC_NOWAIT` was set), `EIDRM` (queue was removed), `EINTR` (signal interrupted a blocking call), and `EACCES` (no write permission on the queue).

Because the return value is a plain boolean, always check it before assuming the message was enqueued:

```
msgsnd($qid, $msg, 0)
    or die "msgsnd($qid) failed: $!";
```

Message format

The kernel reserves the first `sizeof(long)` bytes of every queued message for the **message type**, a positive `long` integer that `msgrcv` callers use to select which messages to dequeue. The payload follows, uninterpreted.

Build the buffer with `pack` using the `l!` (native `long`) and `a*` (raw bytes) specifiers:

```
my $type    = 1;
my $payload = "hello queue";
my $msg     = pack("l! a*", $type, $payload);
msgsnd($qid, $msg, 0) or die "msgsnd: $!";
```

The message type **must be greater than zero**. Zero and negative values are reserved for the selector semantics of `msgrcv` and cause `msgsnd(2)` to fail with `EINVAL`.

Global state it touches

- `$!` - set to the `errno` from the failed syscall when `msgsnd` returns false. Untouched on success.

Examples

Send a simple typed message with the default blocking behaviour. If the queue is full, the call waits until space is available:

```
use IPC::SysV qw(IPC_PRIVATE IPC_CREAT S_IRUSR S_IWUSR);

my $qid = msgget(IPC_PRIVATE, IPC_CREAT | S_IRUSR | S_IWUSR)
    // die "msgget: $!";

my $msg = pack("l! a*", 1, "job-42");
msgsnd($qid, $msg, 0) or die "msgsnd: $!";
```

Non-blocking send - return immediately with `EAGAIN` in `$!` if the queue is full, rather than sleeping:

```
use IPC::SysV qw(IPC_NOWAIT);
use Errno qw(EAGAIN);

unless (msgsnd($qid, $msg, IPC_NOWAIT)) {
    if ($! == EAGAIN) {
        warn "queue full, dropping message\n";
    } else {
        die "msgsnd: $!";
    }
}
```

Use the type field as a priority / channel tag. Receivers pick by type via `msgrcv`:

```
my %CHANNEL = (log => 1, metric => 2, alert => 3);
```

(continues on next page)

(continued from previous page)

```
sub send_event {
    my ($kind, $body) = @_;
    my $msg = pack("l! a*", $CHANNEL{$kind}, $body);
    msgsnd($qid, $msg, 0) or die "msgsnd[$kind]: $!";
}

send_event(alert  => "disk > 90%");
send_event(metric => "rps=1240");
```

Send a structured payload - `pack` the header fields after the mandatory type prefix:

```
my $type = 1;
my $body = pack("N N a*", $job_id, $retry_count, $payload);
msgsnd($qid, pack("l! a*", $type, $body), 0)
    or die "msgsnd: $!";
```

Edge cases

- **Message type must be positive.** Passing `0` or a negative type fails with `EINVAL`. The type-selection semantics in `msgrcv` reserve those values for selector logic.
- **Message exceeds `msg_qbytes`.** If `MSG` is larger than the queue's configured per-message limit, `msgsnd` fails with `EINVAL` regardless of `FLAGS`. Check and raise the limit with `msgctl` `IPC_SET` if you control the queue.
- **Queue full without `IPC_NOWAIT`.** The call blocks until another process dequeues, the queue is removed (`EIDRM`), or a signal interrupts it (`EINTR`). Retry on `EINTR` if the signal is expected and benign.
- **Queue full with `IPC_NOWAIT`.** Fails immediately with `EAGAIN`. This is the only way to enforce bounded latency on the send side.
- **Missing type prefix.** Forgetting to `pack` the `l!` type prefix passes the raw payload as-is; the first `sizeof(long)` bytes are then interpreted as the type - typically a large positive or negative number - and the rest as the body. Receivers filtering by type see nothing.
- **`ID` is a queue id, not a key.** Pass the integer returned by `msgget`, not the `KEY` you gave to `msgget`. Confusing the two yields `EINVAL` or, worse, a successful send to an unrelated queue that happens to share the numeric value.
- **Permissions.** The process must have write permission on the queue (the `w` bit of the queue's mode for its effective uid/gid). Otherwise the call fails with `EACCES`.
- **Tainting.** `MSG` is read as a byte string; no taint checking is performed on it here. If `MSG` came from an untrusted source, the receiver side (`msgrcv`, which *does* taint its output) is where filtering belongs.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `msgget` - obtain the queue id passed as `ID`; call this first, or receive the id from whoever created the queue
- `msgrcv` - the matching receive primitive; its `TYPE` argument selects against the `l!` prefix written here
- `msgctl` - inspect the queue's `msg_qbytes` limit or remove the queue when done
- `pack` - build the `l! a*` buffer `msgsnd` requires; also the tool for any further structure inside the payload
- `$!` - carries the `errno` from a failed `msgsnd`; compare against `Errno` constants to branch on specific failures

Scoping

my

Declare one or more lexically scoped variables.

`my` introduces new variables whose visibility is bounded by the enclosing block, file, or `eval`. The compiler records the declaration at compile time; the storage is allocated and (re)initialised each time control enters the scope. Unlike `local`, `my` does not temporarily replace a package variable - it creates a genuinely new variable that exists only while the scope is live and is invisible outside it, including to any subroutines the scope calls.

Synopsis

```
my EXPR
my TYPE EXPR
my EXPR : ATTRS
my TYPE EXPR : ATTRS
```

A single variable may be written without parentheses; a list of two or more must be parenthesised:

```
my $x;
my ($a, $b, @rest);
my Foo $obj;
my $x : shared = 0;
```

What you get back

`my` is a declaration, not a value-producing operator in the ordinary sense. In rvalue position it yields its variable (or the list of variables) as an lvalue you can assign to. The common idioms are:

```
my $x = $expr;               # scalar context on RHS
my ($first, @rest) = @list; # list context on RHS; destructuring
my @copy = @src;             # list context; @copy gets the elements
my %h = @pairs;              # list context; pairs populate the hash
```

When no initialiser is given, each declared variable starts as `undef` (scalar), empty list (array), or empty hash.

Scope and lifetime

- **Block scope.** A `my` variable is visible from the point immediately after its declaration to the end of the innermost enclosing block, file, `eval`, `do`, `require`, or `use`'d file.
- **Control-expression scope.** The controlling expression of `if` / `unless` / `elsif` / `else`, `while` / `until`, `for` / `foreach` is part of the scope of the declaration it contains. In `while (my $line = <$fh>) { ... }` the variable `$line` is live throughout the loop body and its `continue` block, but not beyond.
- **Fresh storage on every entry.** Each time control re-enters the scope, the variable is created anew. This is the key difference from `state`, which preserves its value across entries.
- **Closures capture storage, not values.** An anonymous sub created inside the scope keeps that particular instance of the variable alive for as long as the sub is reachable.

```
sub make_counter {
    my $n = 0;
    return sub { ++$n };    # closes over this $n
}
my $c = make_counter();
$c->(); $c->();             # 1, 2
```

List form and destructuring

With parentheses, `my` supplies list context to the right-hand side and destructures the result:

```
my ($sec, $min, $hour) = localtime;
my ($head, @tail)      = @list;       # @tail absorbs the rest
my ($x, $y)            = ($y, $x);    # swap via list assignment
```

Use `undef` as a placeholder to skip a position:

```
my (undef, $min, $hour) = localtime;  # discard seconds
```

Without parentheses, `my $foo` is a single-variable declaration and the comma operator applies to whatever follows:

```
my $foo, $bar = 1;          # WRONG: declares $foo only;
                            # $bar is the package variable
my ($foo, $bar) = (0, 1);   # Right.
```

Context traps

`my` does not change how the right-hand side is evaluated - it only modifies the variable on the left. A scalar `my $foo` imposes scalar context; a parenthesised `my (...)` imposes list context:

```
my  $line  = <$fh>;         # scalar context: one line
my ($line) = <$fh>;         # list context: first of a list
my  @lines = <$fh>;         # list context: all remaining lines
```

The classic mistake is writing `my ($foo) = <$fh>` when a single line was wanted - the parentheses turn on list context and the filehandle returns a one-element list; the value happens to be the first line, but the surrounding context is wrong for anything that follows.

`my` vs `local` vs `our` vs `state`

Four declarations, four different meanings:

- `my` - introduces a **new lexical** variable. Not visible to called code. Fresh storage each entry.
- `local` - does **not** declare anything. It saves the current value of an **existing package** variable and restores it when the enclosing scope exits. Visible to called code through dynamic scope. Use it for the handful of magical globals (`$_`, `$/`, `$\`, `$,`, `$|`, `%ENV`, ...) that cannot be lexicalised.
- `our` - declares a **lexical alias** for a package (global) variable. The storage lives in the package's symbol table; `our` just lets you refer to it without the package prefix for the rest of the lexical scope. Satisfies `use strict 'vars'`.
- `state` - like `my` in visibility, but the storage **persists** across re-entries to the scope. Initialisation runs once; subsequent entries leave the value alone.

```
package Counter;
our   $total = 0;           # package global; $Counter::total
my    $cache;               # lexical; invisible outside this file
state $calls = 0;           # lexical; keeps its value across calls
sub tick {
    local $\ = "\n";        # temporarily override output separator
    ++$calls;
    print ++$total;
}
```

Critical point: `my $var` does **not** mask a package variable with the same name. The package variable remains accessible through its fully qualified form while the lexical is in scope:

```
package main;
our $x = 10;
my  $x = 20;
print "$x and $::x\n";      # "20 and 10\n"
```

TYPE and attributes

Both forms are part of the language but their semantics are still evolving (upstream wording).

- **TYPE** - a bareword class name, a constant declared via `use constant`, or `__PACKAGE__`. Historically bound to the `fields` pragma; for ordinary use it serves as a hint and does not enforce a type at runtime.

  ```
  my Foo $obj = Foo->new;
  ```

- **ATTRS** - a colon-introduced attribute list handled by the `attributes` pragma (and `Attribute::Handlers` since 5.8.0).

  ```
  my $x : shared;
  my @buf : Locked;
  ```

See `perlsub` upstream for the authoritative wording.

Declaration is not yet in scope on its own RHS

The newly declared variable is not visible until **after** the declaring statement finishes. That makes this idiom well-defined:

```
my $x = $x;                 # new $x initialised from the old $x
```

and this condition dependent on a pre-existing outer `$x`:

```
my $x = 123 and $x == 123;  # false unless the old $x was 123
```

Within a single statement, any further mentions of the name refer to the pre-declaration binding (or raise a `strict 'vars'` error if no such binding exists):

```
our $x = 2;
foo($x, my $x = $x + 1, $x); # foo() receives (2, 3, 2)
```

Examples

Typical subroutine entry - name the arguments:

```
sub distance {
    my ($x1, $y1, $x2, $y2) = @_;
    return sqrt( ($x2-$x1)**2 + ($y2-$y1)**2 );
}
```

Loop-local index with `for my`:

```
for my $i (1 .. 10) {
    # $i is fresh each iteration; invisible after the loop
}
```

File-scoped private state - the variable exists for the life of the interpreter but is unreachable from outside this file:

```
my $cache = {};
sub lookup { $cache->{ $_[0] } //= _compute($_[0]) }
```

Private subroutine via a lexical coderef:

```
my $helper = sub { ... };
$helper->($x);
```

Shadowing warning - redeclaring in the same scope emits a `shadow`-category warning under `use warnings`:

```
use warnings 'shadow';
my $x = 1;
my $x = 2;                  # warning: "my" variable $x masks ...
```

Edge cases

- **Only alphanumeric identifiers.** Magical built-ins such as `$/`, `$_`, `$\`, `%ENV`, `@ARGV` cannot be lexicalised. Use `local` to scope them dynamically.
- **No package qualification.** `my $Foo::bar` is a syntax error: `my` variables are not package-owned and cannot carry a `::` qualifier.
- **Missing parentheses around a list.** `my $a, $b` declares only `$a`; `$b` is the package variable. Always parenthesise two or more names.
- **List-context surprise.** `my ($line) = <$fh>` forces list context on the read; the RHS is a one-element list, not a scalar. Drop the parentheses if you want scalar semantics.
- **`foreach` without `my`.** `for $i (...)` localises `$i` dynamically in the manner of `local`. Prefer `for my $i (...)` to confine the index lexically.
- **Shadowing in loops.** A `my` inside a loop body declares a fresh variable every iteration. If that is unwanted (e.g. a counter that should survive iterations), declare it outside the loop or use `state`.
- **Closure capture.** A closure over a `my` variable keeps that particular instance alive. Different calls to an enclosing sub produce independent closures over independent variables.
- **`use strict 'vars'` interaction.** `my` satisfies strict-vars within its scope; variables without a declaration must be `our`, fully qualified, or predeclared with `use vars`.
- **Compile-time vs run-time.** The name binding happens at compile time; the initialiser runs at run time, and re-runs on each scope entry. Expensive initialisers inside hot loops pay that cost every iteration.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `our` - lexical alias for a package variable; satisfies `strict 'vars'` without creating new storage
- `local` - dynamically scoped save/restore of an existing package variable; the tool for magical globals
- `state` - lexical like `my`, but storage persists across scope re-entries
- `undef` - placeholder in a parenthesised `my` list, and the initial value of an uninitialised scalar
- `use strict` - makes `my` / `our` / fully-qualified names mandatory, catching accidental globals
- `attributes` - pragma that interprets the `: ATTRS` portion of a `my` declaration

Control flow

next

Start the next iteration of the enclosing loop immediately.

`next` is Perl's equivalent of C's `continue` statement: it abandons the rest of the current iteration's body and jumps to the loop's iteration step. If the loop has a `continue` block, that block runs first, then the controlling expression is re-tested (or the next element is taken for `foreach`), and execution resumes at the top of the body. With a label, `next LABEL` targets a specific enclosing loop rather than the innermost one.

Synopsis

```
next
next LABEL
next EXPR
```

What you get back

Nothing. `next` is a flow-control operator, not an expression - it unwinds out of the current iteration and therefore never yields a value to its caller. Attempting to use it in a value-producing position (for example as the right-hand side of an assignment inside `eval {}`, `sub {}`, or `do {}`) is a misuse: the surrounding construct never completes normally, so there is no return value to observe.

Loop targeting

Without a label, `next` refers to the **innermost enclosing loop** (`while`, `until`, `for`, `foreach`, or a bare block, which counts as a one-shot loop). With a label it refers to the named loop, no matter how deeply nested the `next` sits inside other loops:

```
OUTER: for my $row (@grid) {
    for my $cell (@$row) {
        next OUTER if $cell eq 'skip-row';
        process($cell);
    }
}
```

`next EXPR`, available since Perl 5.18, evaluates `EXPR` at run time and uses its string value as the label name. Otherwise it behaves exactly like `next LABEL`:

```
my $target = pick_label();
next $target;                    # same as: next FOO if $target eq
↪"FOO"
```

Use the runtime form sparingly - a static `next LABEL` is easier to read and to verify by eye.

Interaction with `continue` blocks

A loop's `continue` block runs at the end of every iteration, whether that iteration ended normally or via `next`. This is the whole point of having a `continue` block: it groups the bookkeeping that must happen on every pass.

```
my $i = 0;
LINE: while (my $line = <$fh>) {
    next LINE if $line =~ /^\s*#/;  # skip comments
    process($line);
}
continue {
    $i++;                           # runs even for skipped lines
}
```

`last` skips the `continue` block; `next` does not. This is the simplest distinguishing property of the two, and the reason `continue` exists as its own construct rather than being baked into the loop body.

Examples

Basic `while` loop - discard comment lines, process the rest:

```
while (my $line = <$fh>) {
    next if $line =~ /^\s*#/;
    next if $line =~ /^\s*$/;       # blank
    handle($line);
}
```

`foreach` with an accumulator - skip entries that fail a predicate:

```
my $total = 0;
for my $n (@values) {
```

(continues on next page)

(continued from previous page)

```
    next unless defined $n;
    next if $n < 0;
    $total += $n;
}
```

Nested loops with a label - move on to the next outer row as soon as we spot a sentinel in the inner scan:

```
ROW: for my $row (@grid) {
    for my $cell (@$row) {
        next ROW if $cell eq 'END';
        count($cell);
    }
}
```

`continue` block that always runs, even on `next`:

```
my @kept;
ITEM: for my $x (@input) {
    next ITEM if $x->{hidden};
    push @kept, $x;
}
continue {
    log_seen($_);                    # every $x is logged, kept or
↪not
}
```

Escaping a bare block - a standalone `{ ... }` is a one-shot loop, so `next` jumps past the remaining statements and out of it:

```
{
    last_chance() or next;           # bare block: next == fall out
    commit();
}
```

Edge cases

- **Illegal inside `grep` / `map` / a `do { }` value block.** These constructs are not loops - their block is evaluated for its value, not iterated in the loop-control sense. Using `next` to exit them is a misuse and is called out explicitly in upstream perlfunc. Use a conditional expression instead:

  ```
  my @kept = grep { wanted($_) } @xs;          # not: grep {
  ↪next unless ... }
  my @mapped = map { transform($_) // () } @ys; # empty list ==
  ↪skip
  ```

- **Not a `return`.** `next` cannot deliver a value out of a value-returning block such as `eval { }`, `sub { }`, or `do { }`. If you find yourself reaching for `next` to leave such a block, you want `return`, or a conditional that produces the value you need.

- **A bare block is a one-shot loop.** `{ ... }` with no `while` / `for` prefix is semantically a loop that runs exactly once, so `next` exits it. This is occasionally useful for early-out patterns, but `last` is the more common choice because it makes the intent clearer.
- **Unknown label is a fatal error.** `next NOSUCH` dies at run time with `Can't "next" outside a loop block` when no enclosing loop carries that label.
- **`continue` runs on `next`, not on `last`.** This is the usual cause of surprise when refactoring: moving bookkeeping out of the loop body into `continue` changes behaviour only for `last`, not for `next`.
- **Precedence and parsing quirks.** Unlike most named operators, `next` has the same precedence as assignment, and it is exempt from the looks-like-a-function rule. That means `next ("foo") . "bar"` passes `"foobar"` to `next`, not just `"foo"`. Parenthesise defensively when the label name comes from an expression:

```
next(($cond ? "A" : "B"));
```

- **Threads, signals, destructors.** `next` unwinds the current iteration's lexical scope: `my` variables go out of scope, `local` values are restored, and `DESTROY` fires on scope-exit objects, just as if the iteration had ended normally. Pending `$SIG{...}` handlers are not affected.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `last` - exit the enclosing loop entirely, skipping any `continue` block
- `redo` - restart the current iteration without re-testing the condition and without running `continue`
- `return` - the right tool for leaving `sub { }`, `eval { }`, or `do { }` with a value; `next` cannot do this
- `grep` - filter a list; use a conditional expression, not `next`, to drop elements
- `map` - transform a list; return the empty list `()` from the block to drop an element

Modules

no

The compile-time inverse of `use` - call a module's `unimport` method to turn off what `use` turned on.

`no MODULE LIST` is handled by the parser exactly like `use MODULE LIST`, except it calls `MODULE->unimport(LIST)` instead of `MODULE->import(LIST)`. Both run at compile time, both are lexically scoped, and both require the module to already be loaded. The common case is switching a pragma off inside a block - `no strict 'refs'`, `no warnings 'uninitialized'`, `no feature 'switch'` - without touching the enclosing file.

Synopsis

```
no MODULE LIST
no MODULE
no MODULE VERSION LIST
no MODULE VERSION
```

What you get back

no is a declaration, not an expression. It has no useful return value and cannot appear where a value is expected. The effect is a side effect on the compile-time state of the enclosing lexical scope.

How it expands

The parser rewrites

```
no MODULE LIST;
```

into the rough equivalent of

```
BEGIN {
    require MODULE;
    MODULE->unimport(LIST);
}
```

- the same shape as `use`, with `unimport` swapped in for `import`. Consequences that follow from this:
- **Compile time.** The `BEGIN` block runs as soon as the parser reaches the no statement, before any later code in the file is compiled. This is why `no strict 'refs'` takes effect on the statements that follow it in the same file, not at runtime.
- **Lexical scope.** Pragmas that honour lexical scope (`strict`, `warnings`, `feature`, `integer`, `utf8`, and the other core pragmas) see their "off" effect confined to the innermost enclosing block, `eval`, or file. Leaving the block restores the previous state.
- **`require` still happens.** `no MODULE` loads `MODULE` if it has not been loaded yet, then calls `unimport` on it. You do not need a matching `use` earlier in the file.
- **`unimport` is mandatory.** If the package has no `unimport` method, the call dies at compile time with `Can't locate object method "unimport" via package "MODULE"`. Most non-pragma modules do not define `unimport` - no is really only useful for modules that advertise it.

Examples

Turn off a single warning category inside a block:

```
use warnings;
```

(continues on next page)

(continued from previous page)

```
for my $row (@rows) {
    no warnings 'uninitialized';
    print join(",", @$row), "\n";   # empty fields render as ""
}
# 'uninitialized' warnings are back on here
```

Relax `strict 'refs'` for one statement that needs symbolic dereferencing:

```
use strict;

sub install {
    my ($name, $code) = @_;
    no strict 'refs';
    *{"main::$name"} = $code;       # symbolic typeglob assignment
}
```

Disable an experimental feature warning while using the feature:

```
use feature 'signatures';
no warnings 'experimental::signatures';

sub greet ($who) { "hello, $who\n" }
```

Roll back a default feature bundle to plain Perl semantics:

```
use v5.36;                          # enables strict, warnings, say,␣
↪signatures…
no feature 'switch';                # …but not given/when
```

Turn off a user pragma that provides its own `unimport`:

```
use My::Pragma qw(loud);
{
    no My::Pragma;                  # My::Pragma->unimport called␣
↪here
    quiet_code();
}
```

Edge cases

- **`no VERSION` is a compile error.** `use VERSION` exists as a compile-time version check against the running interpreter; there is no sensible inverse, so `no 5.36;` dies with `No such class 5.36`. The version check only goes one way.
- **`no MODULE VERSION` *does* parse.** It requires `MODULE` to be at least `VERSION` (same check as `use`), then calls `MODULE->unimport(LIST)`. The version check is about the loaded module, not the interpreter.
- **Module without `unimport`.** Dies at compile time. There is no fallback to a no-op; if you are writing a module and want `no` to be a silent no-op, define an empty `sub unimport {}`.

- **Bareword `MODULE` only.** Like `use`, the module name must be a bareword known at compile time. `no $name;` is a syntax error - the parser needs the literal token to decide which `unimport` to call.
- **Does not undo `use`.** `no Foo` calls `Foo->unimport`, which by convention reverses what `Foo->import` did to the current lexical scope - but that is a convention, not a language guarantee. A sloppy `unimport` can leave state behind, and side effects from `require Foo` (BEGIN blocks, `@INC` hooks, installed subs in other packages) are **not** rolled back.
- **Effect ends at the enclosing block.** For lexical pragmas, the "off" state stops at the next `}`, `eval` boundary, or end of file. Code in a different file that `requires` or `uses` this one is unaffected.
- **`import`/`unimport` are just method calls.** `MODULE->unimport(LIST)` runs exactly like any other class method - it sees `MODULE` as the first argument and `LIST` after. The method can inspect the caller via `caller` to find out which package to modify.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `use` - the positive form; no is literally `use` with `unimport` in place of `import`
- `require` - the runtime load step that no performs before calling `unimport`
- `package` - declares the current package, which is the scope a pragma's `unimport` typically modifies
- `import` - the conventional counterpart method; no calls `unimport`, `use` calls `import`
- `BEGIN` - the compile-time phase in which no runs
- `caller` - how an `unimport` method identifies the package it should affect

SCALARs and strings · Numeric functions

oct

Interpret a string as a number written in octal, hexadecimal, or binary, and return the resulting integer.

The name is misleading: `oct` is really a *radix-prefix* parser. It inspects the leading characters of `EXPR` and picks the base from them - `0x`/`x` for hexadecimal, `0b`/`b` for binary, `0o`/`o` (or no prefix at all) for octal. If `EXPR` is omitted, `oct` works on `$_`. Think of it as the inverse of `sprintf "%o"` / `"%x"` / `"%b"` for the common case where you have the textual form and need the number.

Synopsis

```
oct EXPR
oct                 # operates on $_
```

What you get back

A non-negative integer. The numeric value of EXPR interpreted in whichever base its prefix selected (octal by default). The result is always >= `0` - `oct` does not parse a leading sign and does not return floating-point values.

An empty string, a string with no recognised digits, or `undef` returns `0`.

Prefix rules

The leading characters of EXPR (after any leading whitespace, which is silently skipped) determine the base:

Prefix	Base	Example input	Result
`0x` or `x`	16 (hex)	`"0xff"`	255
`0b` or `b`	2 (binary)	`"0b1010"`	10
`0o` or `o`	8 (octal)	`"0o755"`	493
(none)	8 (octal)	`"755"`	493
(none, leading `0`)	8 (octal)	`"0755"`	493

The `0o`/`o` octal prefix was added in Perl 5.33.5 to match integer literal syntax; older forms (`0755` and a bare `755`) have always been octal.

Underscores between digits are ignored, so values copied straight from Perl integer literals parse cleanly:

```
oct("0xDEAD_BEEF")      # 3735928559
oct("0b1111_0000")      # 240
oct("07_55")            # 493
```

Examples

Decode a permission string typed by a user - the whole reason `oct` exists. The string `"755"` is **not** the number 755; it is the octal representation of the number 493. `chmod` expects the number:

```
my $mode = "0755";
chmod oct($mode), "script.sh";          # chmod 0755
```

```
my $mode = "755";                       # no leading zero - still␣
↪octal
chmod oct($mode), "script.sh";          # same result
```

Parse hex and binary literals read from a config file or a command line:

```
my $mask  = oct("0xff00");              # 65280
my $flags = oct("0b1010_0101");         # 165
```

Handle arbitrary radix-prefixed strings - decimal, octal, hex, or binary - in one expression. This is the classic `oct`-as-dispatcher idiom: a leading `0` (in any of its forms) means "not decimal", so route through `oct`; otherwise keep the value as-is:

```
my $n = $val =~ /^0/ ? oct($val) : $val;
```

Default-argument form operates on `$_`, handy inside `map`:

```
my @perms = map { oct } @mode_strings;
```

Round-trip with `sprintf` to go the other direction:

```
my $perm = (stat $file)[2] & 07777;
my $text = sprintf "%o", $perm;         # e.g. "644"
my $back = oct($text);                  # back to the same integer
```

Edge cases

- **Empty or undef input**: `oct("")` and `oct(undef)` both return `0`. `undef` triggers an `uninitialized` warning under `use warnings`.
- **No prefix defaults to octal**: `oct("12")` is `10`, not 12. This is a frequent surprise - if your strings might be decimal, test for a leading `0` first (see the idiom above) or do not use `oct` at all and parse explicitly.
- **Underscores are ignored** wherever they appear between digits. `oct("0x_de_ad")` is `57005`. Leading and trailing underscores are also silently accepted.
- **Invalid characters terminate parsing**: parsing stops at the first character that does not belong to the selected base; the rest of the string is discarded without a warning. `oct("0x1g2")` returns `1`, `oct("755.5")` returns `493`, `oct("12abc")` returns `10` (octal, stopped at `a`), `oct("0b102")` returns `2` (binary `10`, stopped at `2`).
- **Leading whitespace is skipped silently.** A leading `+` or `-` is **not** a sign - it is a non-digit, so parsing returns `0`. `oct` cannot produce a negative number.
- **Floating-point and exponent notation are not understood.** There is no `0x1.8p3` support; the `.` terminates parsing.
- **Case of prefix and digits is insensitive**: `0X`, `0B`, `0O`, and hex digits `a-f`/`A-F` are all accepted.
- **Overflow**: values larger than the platform integer silently promote to floating point, losing low-order precision for very large hex/binary inputs.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `hex` - forces base 16 regardless of prefix; use it when the input is *always* hex and you do not want `oct`'s autodetection
- `int` - truncate a numeric value toward zero; unrelated to radix parsing but often reached for by accident
- `chmod` - the canonical consumer of `oct`; takes a numeric mode, so string-form modes must go through `oct` first
- `sprintf` - the inverse direction: format %o, %x, %b turn an integer into its octal/hex/binary textual form

Filehandles, files, directories

open

Associate a filehandle with a file, a command, or an in-memory scalar.

open is the primary way a Perl program acquires a filehandle. The three-argument form - `open $fh, MODE, EXPR` - is the one to use in new code: the mode is stated explicitly, the filename cannot be reinterpreted as a shell pipeline, and leading or trailing whitespace in `EXPR` is preserved verbatim. The older two-argument form exists for compatibility and for the "magic" filename interpretation it enables; do not feed it untrusted input.

Synopsis

```
open my $fh, "<",  $path           or die "read $path: $!";
open my $fh, ">",  $path           or die "write $path: $!";
open my $fh, ">>", $path           or die "append $path: $!";
open my $fh, "+<", $path           or die "rw $path: $!";
open my $fh, "<:encoding(UTF-8)", $path or die $!;
open my $mem, ">",  \$buf          or die $!;   # in-memory
open my $pipe, "-|", "ls", "-la"   or die $!;   # read from cmd
open my $pipe, "|-", "gzip", ">$f" or die $!;   # write to cmd
```

What you get back

1 on success; `undef` on failure, with `$!` set to the OS error. For the pipe forms (`|-` / `-|`), the successful return value is the pid of the child process - useful when you need to `waitpid` on it later. After a one- or two-argument pipe open on the literal command `-`, open returns twice (parent gets the child pid, child gets `0`); use `defined($pid)` or `//` to distinguish the two.

Always check the return value. A failed open leaves the filehandle closed, and every subsequent read or write on it will set `$!` to `Bad file descriptor` and emit a warning under `use warnings`.

```
open my $fh, "<", $path
    or die "open $path: $!";
```

Global state it touches

- `$!` - set on failure to the OS-level error.
- `${^OPEN}` - default I/O layers installed by the open pragma or the `-CioD` switch. These apply only when `MODE` in the three-argument form contains no explicit layer; a bare `<` or `>` honours them, `<:raw` or any other explicit layer suppresses them.
- `$^F` - highest file descriptor not marked close-on-exec; the new fd inherits the close-on-exec flag based on this.
- Selected filehandle via `select` - unchanged by open, but becomes relevant the moment you `print` to the new handle without naming it.
- For pipe forms, `$?` and `${^CHILD_ERROR_NATIVE}` are set when the handle is later closed.

Modes

The second argument names an I/O role. The common cases:

Mode	Meaning	`fopen(3)` equivalent
`<`	read	`r`
`>`	write, truncate or create	`w`
`>>`	append, create if missing	`a`
`+<`	read and write, no truncation	`r+`
`+>`	read and write, truncate first	`w+`
`|-`	pipe to command (we write)	-
`-|`	pipe from command (we read)	-
`<&`, `>&`, `<&=`, `>&=`	dup or fdopen an existing handle	-

Newly created files get permissions `0666 & ~umask`. `+<` is almost always what you want for read-write; `+>` clobbers the file first.

I/O layers attach after the mode: `<:encoding(UTF-8)`, `>:raw`, `>>:crlf`, `<:encoding(UTF-8):crlf`. An explicit layer suppresses `${^OPEN}` defaults; a lone colon (`<:`) falls back to the OS default (`:raw` on Unix, `:crlf` on Windows).

Examples

Read a UTF-8 text file line by line:

```
open my $fh, "<:encoding(UTF-8)", "notes.txt"
    or die "open notes.txt: $!";
while (my $line = <$fh>) {
    chomp $line;
    # ... use $line ...
```

(continues on next page)

(continued from previous page)

```
}
close $fh or warn "close: $!";
```

Write a binary file with no encoding translation:

```
open my $out, ">:raw", "image.bin"
    or die "open image.bin: $!";
print $out $bytes;
close $out or die "close: $!";
```

In-memory filehandle - `print` into a scalar, useful for capturing output that expects a real handle:

```
my $buf = "";
open my $mem, ">", \$buf or die $!;
print $mem "hello\n";
close $mem;
# $buf now contains "hello\n"
```

Read from a command, list form so arguments are not subject to shell metacharacter interpretation:

```
open my $ls, "-|", "ls", "-la", $dir
    or die "ls: $!";
while (<$ls>) { print }
close $ls;                      # $? holds the exit status
```

Temporary anonymous file - pass `undef` as the third argument with a read-write mode, then `seek` back to the start to read what you wrote:

```
open my $tmp, "+>", undef or die $!;
print $tmp "scratch data\n";
seek $tmp, 0, 0;
my $line = <$tmp>;
```

Edge cases

- **Two-argument "magic" open**: `open $fh, $name` honours leading `<`, `>`, `>>`, a trailing `|`, and strips surrounding whitespace from `$name`. Fine for a literal filename you typed yourself; **never** for a filename that came from user input, the network, or an environment variable. Use the three-argument form.
- **Two-argument `-`**: `open $fh, "<-"` or `open $fh, "-"` opens `STDIN`; `open $fh, ">-"` opens `STDOUT`. Three-argument equivalents are `open $fh, "<", "-"` and `open $fh, ">", "-"`.
- **Reopening `STDOUT` / `STDERR`**: close them first, then reopen against the new target. `open STDOUT, ">", \$buf` redirects all subsequent `print STDOUT` output into `$buf`.
- **Bareword filehandles** (`open FH, ...`) are package globals and are disabled under `use v5.36` or later via the `bareword_filehandles` feature. Prefer a lexical

scalar.

- **Layers and `${^OPEN}`**: any explicit layer, even `:`, disables the defaults from the open pragma. If you rely on a pragma default, omit the colon entirely (`<`, not `<:`).
- **Pipe to a command with shell metacharacters**: the two-argument pipe form (`open $fh, "|cat -n | lpr"`) runs through `/bin/sh -c`. The three-argument list form (`open $fh, "|-", "cat", "-n"`) does not and is the safer default.
- **In-memory scalar strings**: treated as octet strings. Without truncation, the scalar may not contain code points above `0xFF`.
- **Automatic close on scope exit**: a lexical filehandle closes when its refcount hits zero, but the implicit close does not report errors. For files you wrote to, close explicitly and check the return value.
- **open vs sysopen**: open wraps `fopen(3)` semantics with PerlIO layers on top. For direct `open(2)` flags like `O_EXCL` or `O_NOFOLLOW`, use `sysopen`.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `close` - release a filehandle; check its return value for buffered writes and pipe commands
- `sysopen` - open with raw `open(2)` flags when you need `O_EXCL`, `O_NOFOLLOW`, or similar precision
- `binmode` - push or replace I/O layers on an already open handle
- `read` - fixed-size byte reads from the handle open returned
- `readline` - record-at-a-time reads (what `<$fh>` calls)
- `fileno` - the underlying OS file descriptor, for `flock`, `select`, or `fcntl`
- `$!` - OS error set when open returns `undef`

Filehandles, files, directories

opendir

Open a directory for reading.

`opendir` attaches `DIRHANDLE` to the directory named by `EXPR` so the directory's entries can be read with `readdir` and the position manipulated with `telldir`, `seekdir`, and `rewinddir`. When you're done, `closedir` releases the handle. `DIRHANDLE` lives in a namespace distinct from file handles - a bareword you have already used with `open` is free to reuse here, and vice versa.

Synopsis

```
opendir DIRHANDLE, EXPR
opendir my $dh, EXPR
```

What you get back

1 on success, `0` on failure. On failure `$!` holds the reason - the directory does not exist, is not a directory, is not readable, or the process has hit its open-file limit. Always check the return; a silent failure here turns into a mysteriously empty `readdir` later.

```
opendir my $dh, $path
    or die "opendir $path: $!";
```

Global state it touches

- `$!` - set to the system error string on failure, left untouched on success. Capture it before any other system call disturbs it.

Handle semantics

Three forms of `DIRHANDLE` are accepted, and the choice matters:

- **Bareword** - `opendir DIR, $path` creates a package-global dirhandle named `DIR` in the current package. It stays open until you `closedir` it explicitly or the program exits. Avoid in new code; the global namespace is shared across the whole package.
- **Undefined scalar** - `opendir my $dh, $path` autovivifies `$dh` into a reference to a fresh anonymous dirhandle. When `$dh` goes out of scope and its refcount drops to zero, the handle is closed automatically. This is the form to use.
- **Expression** - any expression whose value is usable as an indirect dirhandle (typically a real dirhandle name held in a variable). The handle identity comes from the expression's value, not its syntactic form.

Dirhandles and filehandles share the same underlying I/O object: a given object can be open as one or the other at a time, never both. Passing a dirhandle to `read` or a filehandle to `readdir` is a usage error, not a silent read of zero bytes.

Examples

The canonical `opendir` / `readdir` / `closedir` idiom:

```
opendir my $dh, $dir
    or die "opendir $dir: $!";
while (my $entry = readdir $dh) {
    next if $entry eq '.' or $entry eq '..';
    print "$dir/$entry\n";
}
closedir $dh;
```

Read every entry into a list in one go. `readdir` in list context returns all remaining entries:

```
opendir my $dh, '.' or die "opendir .: $!";
my @entries = grep { !/^\.\.?\z/ } readdir $dh;
closedir $dh;
```

Rely on lexical scope for cleanup - no explicit `closedir` needed:

```
sub list_dir {
    my ($path) = @_;
    opendir my $dh, $path or return;
    return grep { !/^\.\.?\z/ } readdir $dh;
    # $dh goes out of scope here, handle is closed
}
```

Handle a missing-or-unreadable directory without dying:

```
if (opendir my $dh, $path) {
    while (defined(my $e = readdir $dh)) { ... }
    closedir $dh;
} else {
    warn "skipping $path: $!";
}
```

Revisit the start of a directory with `rewinddir` rather than reopening:

```
opendir my $dh, $dir or die $!;
my @first_pass  = readdir $dh;
rewinddir $dh;
my @second_pass = readdir $dh;
closedir $dh;
```

Edge cases

- **Failure sets `$!`, not `$@`.** `opendir` reports through `$!` like the rest of the I/O family; `$@` is untouched.
- **Reusing an open dirhandle.** `opendir` on a handle that is already open silently closes the old directory before opening the new one. Prefer explicit `closedir` when the code is meant to be readable.
- **Relative paths.** `EXPR` is resolved against the current working directory at the moment of the call. If you `chdir` later, a relative path you captured earlier may no longer mean the same thing - the already-opened dirhandle keeps pointing at the right inode regardless.
- **Symlinks.** `opendir` follows symlinks by default: opening a link to a directory opens the target. The entries returned by `readdir` are the target's entries, not the link.
- **Bareword vs lexical lifetime.** A bareword dirhandle is global and outlives the enclosing block; a lexical `my $dh` is cleaned up when the scope ends. Mixing the

two by accident (`opendir DH, $path` inside a sub) leaks dirhandles on repeated calls.

- **File vs dir handle mix-up**. You cannot `print` to a dirhandle nor `readdir` from a filehandle. The error surfaces at the later call, not at `opendir`.
- **`glob` vs `opendir`**. `glob` expands shell-style patterns and returns paths; `opendir` enumerates raw directory entries (including `.` and `..`) without interpretation. Use `glob` when you want pattern matching; use `opendir` when you want every entry exactly as the filesystem stores it.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `readdir` - read the next entry (or all remaining entries) from a dirhandle opened with `opendir`
- `closedir` - release a dirhandle; implicit when a lexical `$dh` goes out of scope
- `rewinddir` - reset the read position to the start without reopening
- `seekdir` - reposition the dirhandle to a token previously returned by `telldir`
- `telldir` - current read position as an opaque token suitable for `seekdir`
- `glob` - pattern-based alternative when you want shell-style filename expansion instead of raw directory enumeration

SCALARs and strings

ord

Return the Unicode code point of the first character of a string.

`ord` takes a string, looks at its first character, and returns that character's numeric code point as an integer. Note *character*, not byte - if the string is a wide-character string containing the letter é (U+00E9), `ord` returns 233, regardless of how many bytes é occupies in memory. For the empty string, `ord` returns `0`. If `EXPR` is omitted, `ord` operates on `$_`.

For the reverse direction - turning a code point back into a character - see `chr`.

Synopsis

```
ord EXPR
ord
ord($str)
```

What you get back

A non-negative integer. For an ASCII input it is in the range `0..127`; for a Latin-1 input `0..255`; for a general Unicode string anything up to `0x10FFFF`. `ord` never returns a negative number and never returns a non-integer. Only the first character of the argument is consulted - trailing characters are ignored. To walk every code point of a string, combine with `split //` or `unpack "U*"`.

Global state it touches

With no argument, `ord` reads `$_`. It neither writes nor reads any other special variable. The interpretation of the input as characters (not bytes) depends on whether the scalar is internally flagged as UTF-8 - see *Edge cases* below.

Examples

Basic ASCII lookup:

```
print ord("A");             # 65
print ord("0");             # 48
print ord("\n");             # 10
```

Empty string returns zero, matching the “no first character” contract:

```
print ord("");              # 0
```

Default-argument form inside a `while` loop over `$_`:

```
while (<STDIN>) {
    last if ord == 4;       # stop on EOT (Ctrl-D) as first char
    print;
}
```

Decode a wide character. `ord` sees the character, not the bytes:

```
use utf8;
print ord("é");             # 233        (U+00E9)
print ord("€");             # 8364       (U+20AC)
print ord("😀");            # 128512     (U+1F600)
```

Walk every code point in a string - useful for debugging encoding surprises:

```
use utf8;
my $s = "héllo";
print join(",", map { ord } split //, $s), "\n";
# 104,233,108,108,111
```

Pair with `chr` for a no-op round-trip on any single character:

```
my $c = "Z";
print chr(ord($c)) eq $c ? "same\n" : "changed\n";   # same
```

Edge cases

- **Character vs byte semantics**: `ord` returns the code point of the first *character*, which for a UTF-8-flagged scalar is not the same as the first *byte*. `ord("\x{100}")` returns 256, even though that character occupies two bytes in memory. To inspect the first byte instead, force byte semantics:

  ```
  use bytes;
  print ord("\x{100}");       # 196   (first UTF-8 byte, 0xC4)
  no bytes;
  print ord("\x{100}");       # 256
  ```

  Reach for `use bytes` only when you genuinely need byte-level inspection; it is a local, surgical pragma.

- **Non-UTF-8 scalar containing high bytes**: if the scalar is a plain byte string (no UTF-8 flag) and its first byte is `0xC4`, `ord` returns 196, not 256. The byte is interpreted as Latin-1. Whether a given scalar is UTF-8-flagged depends on how it was built - `use utf8` in source, `decode_utf8`, a `:utf8` I/O layer, `chr` of a value above 255, and so on.

- **`undef`**: `ord(undef)` returns `0` and, under `use warnings`, emits `Use of uninitialized value in ord`.

- **Numeric argument**: `ord(65)` stringifies 65 to `"65"` first, then takes the first character - so it returns `ord("6") == 54`, not 65. This surprises people; if you have an integer and want to round-trip through a character, use `chr` first or skip `ord` entirely.

- **Multi-character argument**: only the first character matters. `ord("ABC")` returns 65; the `B` and `C` are never consulted.

- **Surrogate and non-character code points**: `ord` will happily return `0xD800..0xDFFF` or `0xFFFE`/`0xFFFF` if those appear in the input. Perl does not refuse to store them; validation, if needed, is the caller's job.

- **Maximum value**: on a standard build, `ord` can return up to `0x7FFFFFFF` (31-bit) for a string produced by `chr` of a value in that range. Real-world Unicode text tops out at `0x10FFFF`.

- **Precedence**: the unary form `ord $x` binds tighter than `,` but looser than most arithmetic operators, so `ord $x + 1` parses as `ord($x) + 1`, not `ord($x + 1)`. Parenthesise when in doubt.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `chr` - the inverse operation; turns a code point back into a one-character string
- `sprintf` - use `%c` to format an integer as its character, or `%x` / `%04x` to render an `ord` result in hex

- `unpack` - `unpack "U*"` for every code point of a string; `unpack "C*"` for every byte
- `lc` / `uc` - case-fold a character before taking its code point when you want case-insensitive comparisons
- `$_` - the default argument when `ord` is called without one

Scoping

our

Declare a lexically scoped alias to a package variable.

`our` does **not** create a new variable. It introduces a name into the current lexical scope that refers to the package variable of the same name in the current package. For the duration of that scope, the unqualified name is legal under `use strict 'vars'`, and every read or write through it hits the underlying package variable.

If the package variable did not exist yet, `our` does not conjure it either - it simply names it. Package variables spring into existence when first assigned to, and an `our` declaration makes that first assignment possible without a fully qualified name.

Synopsis

```
our $x;
our ($a, $b, $c);
our @EXPORT_OK = qw(foo bar);
our $VERSION : shared = '1.00';
```

What you get back

`our` returns its list of variables, so it can appear in any expression position where `my` can:

```
(our $x, our $y) = (1, 2);
our $count = 0;
```

As a declaration it has no separate return value beyond that - the useful effect is the scoping side effect, not the value.

Scope and what the alias actually is

The alias lives in the **enclosing lexical block**, just like `my` and `state`. It ends at the closing brace of that block (or the end of the file for a top-level declaration). Outside the scope, the unqualified name is no longer legal under `use strict 'vars'` - the package variable itself is untouched and still reachable by its fully qualified name.

```
package Foo;
use strict;

$Foo::foo = 23;
```

(continues on next page)

(continued from previous page)

```
{
    our $foo;         # alias to $Foo::foo
    print $foo;       # 23
}

print $Foo::foo;      # 23 - the package variable is unaffected
print $foo;           # error: requires explicit package name
```

Because `our` aliases a **package** variable, the package is fixed at the point of declaration, not at the point of use. Changing packages with `package` mid-scope does not rebind the alias:

```
package Foo;
our $bar;             # declares $Foo::bar for the rest of this scope
$bar = 20;

package Bar;
print $bar;           # 20 - still refers to $Foo::bar
```

our vs my

`our` and `my` look alike on the page and are constantly confused. They do opposite things.

- `my` creates a fresh **private** variable that lives only inside its lexical scope. It has no name in any package's symbol table and nothing outside the scope can reach it.
- `our` creates a lexically scoped **alias** to a **public** package variable. The variable lives in the symbol table; any other code that knows its fully qualified name can read or write it.

Use `my` by default. Reach for `our` when you deliberately want package-global state: module exporter lists (`our @EXPORT`, `our @EXPORT_OK`), `our $VERSION`, a package-wide configuration hash other code is expected to poke at, or the traditional `our @ISA` for inheritance.

```
package My::Module;

our $VERSION   = '1.23';
our @EXPORT_OK = qw(frobnicate);
our @ISA       = ('Exporter');
```

Each of those names is an alias to `$My::Module::VERSION`, `@My::Module::EXPORT_OK`, `@My::Module::ISA` - exactly what `Exporter` and `UNIVERSAL::isa` go looking for.

our vs local

`local` and `our` often appear together and are complementary, not alternatives.

- `our` makes the package variable reachable under its unqualified name inside a lexical scope. It does **not** save or restore the value.

- `local` saves a package variable's current value and arranges for it to be restored when the enclosing dynamic scope ends. It does **not** make the name legal under `use strict 'vars'`.

The idiomatic pairing - temporarily change a package global inside a subroutine - uses both:

```
sub with_verbose {
    our $verbose;        # make the unqualified name legal here
    local $verbose = 1;  # save old value, restore on return
    do_work();
}
```

our vs state

`state` is the other lexical declaration. It creates a private variable whose value persists across calls to the enclosing subroutine. `our` shares none of that machinery - its variable is public and has no per-call semantics. Pick `state` for per-subroutine memoisation, `our` for package-global state.

Scope of the declaration itself

Like `my`, `state`, and `local`, `our` can appear anywhere an expression can (outside string interpolation). The declaration takes effect **immediately** - even within the same statement - so the variable satisfies `use strict 'vars'` from the declaration onward. But the declaration does **not** retroactively cover earlier uses of the same name in the same statement:

```
package main;
my $x = 2;
foo($x, our $x = $x + 1, $x);   # foo() receives (2, 3, 2)
foo($x, our $z = 5, $z);        # foo() receives (3, 5, 5)
```

The rule: each occurrence of the name is resolved as of its own position in the statement. After the declaration, the name is the alias; before it, the name means whatever it meant before.

The declaration also lets you reference the package variable on the right-hand side of its own initialiser, which is something `my` does not allow:

```
my  $foo = $foo;   # error under strict: undeclared $foo on RHS
our $foo = $foo;   # fine: RHS $foo is the package variable
```

Multiple variables and placeholders

More than one variable requires parentheses:

```
our ($bar, $baz);
```

Inside a parenthesised list, `undef` is a valid placeholder that discards the corresponding value of a list assignment:

```
our ( undef, $min, $hour ) = localtime;
```

Repeated declarations

Multiple `our` declarations with the same name in the same lexical scope are allowed when they are in **different** packages - each rebinds the alias to its package's variable:

```
use warnings;

package Foo;
our $bar;            # declares $Foo::bar for the rest of this scope
$bar = 20;

package Bar;
our $bar = 30;       # declares $Bar::bar - aliases the Bar variable
print $bar;          # 30
```

A second `our` declaration for the same name **in the same package and scope** is merely redundant - it rebinds to the same variable. Under `use warnings` Perl reports it the way it reports a repeated `my`, but there is no fresh variable and nothing is shadowed:

```
use warnings;
package Bar;
our $bar = 30;
our $bar;            # warning, but still the same $Bar::bar
print $bar;          # 30
```

Attributes and TYPE

`our` accepts the full declaration syntax of `my`:

```
our $x : shared;
our MyType $obj;
our $VERSION : shared = '1.00';
```

`TYPE` is currently bound to the `fields` pragma. Attributes are processed by the `attributes` pragma (and, since 5.8.0, by `Attribute::Handlers`). See the `attributes` pragma for the set of standard attributes. Ordinary user-level code rarely uses either.

Examples

Module preamble - the overwhelmingly most common use:

```
package My::Thing;
use strict;
use warnings;

our $VERSION   = '0.01';
our @ISA       = ('Exporter');
our @EXPORT_OK = qw(thingify);
```

Declare and assign in one step:

```
package Cfg;
our $timeout = 30;   # same as $Cfg::timeout = 30
```

Temporarily override a package global inside a call:

```
sub quiet_run {
    our $verbose;
    local $verbose = 0;
    run_one_job();
}
```

Cross-package alias - declaration in `Foo`, use in `Bar`:

```
package Foo;
our $shared = 'hello';

package Bar;
# $shared here still refers to $Foo::shared, because the lexical
# alias was introduced while the current package was Foo.
print $shared;       # hello
```

Parenthesised list with placeholder:

```
our ( undef, $min, $hour, undef, $mon, $year ) = localtime;
```

Edge cases

- **`our` does not create the variable.** It declares an alias. The package variable exists the moment anything assigns to it (or takes a reference to it); `our $x;` alone does not allocate a new slot in the symbol table.
- **The alias is lexical, the variable is not.** Leaving the scope of the `our` ends the alias, not the variable. Another scope in another file that declares `our $x;` in the same package sees the same storage.
- **Same-statement resolution.** Only occurrences of the name **after** the `our` declaration in a statement use the alias. Earlier occurrences keep their prior binding (see the `foo()` example above).
- **Package, not current package at use.** The alias is bound to the package in force at the point of declaration. Switching package later does not re-target it.
- **Not the same as `use vars`.** `use vars '$x'` makes `$x` legal under `strict` for the rest of the **package**, regardless of scope. `our $x` is lexical. Prefer `our`; `use vars` is legacy.
- **Not a file-scoped pragma.** A top-level `our $x;` in a file covers until the end of the file, which is a lexical scope - but `require`-ing that file from elsewhere does not extend the alias into the requiring file's scope.
- **Attribute and TYPE semantics are still evolving.** Anything beyond `: shared` in user code is rare; read the `attributes` and `fields` pragma documentation before relying on them.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `my` - declare a private lexical variable; the default choice when you want a new variable
- `local` - dynamically scope a temporary value for a package variable; pairs with `our` when you need both strict legality and save/restore
- `state` - declare a private lexical variable that persists across calls to the enclosing subroutine
- `package` - set the current package, which is what `our` aliases into
- `use` - compile-time module loader; `use strict 'vars'` is the pragma that makes `our` necessary in the first place
- `undef` - valid placeholder inside a parenthesised `our` list to discard a value on assignment

SCALARs and strings · Fixed-length data

pack

Convert a list of Perl values into a binary string according to a template.

`pack` is the low-level serialiser: you describe the byte layout with a `TEMPLATE`, hand it a `LIST` of values, and it returns a scalar whose characters are the concatenated machine-level encoding of those values. It is the inverse of `unpack` and the standard way to build fixed-width records, wire protocols, C struct layouts, raw IP addresses for `sockaddr_in`, and any other byte-level payload that Perl does not model as a first-class type.

This page is a **directive reference**. For a narrative introduction - "I have this wire protocol, how do I parse it?" - start at the *pack/unpack tutorial*.

Synopsis

```
my $bytes = pack TEMPLATE, LIST;
my $ip    = pack "C4",     split /\./, "192.168.1.1";
my $rec   = pack "Z8 Z8 L", $user, $host, $ts;
```

What you get back

A plain Perl scalar holding the packed bytes. Its length is the sum of the widths of every template directive; its contents are binary and may contain embedded `NUL` bytes. Treat it as a byte string, not as text - writing it to a handle with a `:utf8` or `:encoding(…)` layer will re-encode it. Use `binmode $fh` or open with `">:raw"` before emitting.

By default the result is in **character mode** (`C0`). A template that starts with `U`, or switches to `U0` mid-template, produces a UTF-8-encoded Unicode string instead. Do not use this as a substitute for the `Encode` module.

Template syntax

A `TEMPLATE` is a sequence of **directives**. Each directive is an ASCII letter, optionally followed by:

- a **repeat count** - a decimal integer, `*`, or `[...]`
- one or more **modifiers** - `!`, `<`, `>`
- a **group** - `(` ... `)` gathers directives so a repeat count or endianness modifier applies to the whole

Whitespace between directives is ignored. A `#` introduces a comment running to end-of-line - the same convention as Perl source.

Directive table

Every pack directive in one place. **W** is the width in bytes per scalar consumed (a `C` consumes one value and produces one byte; an `a3` consumes one value and produces three bytes). **Endian** column: *native* means the CPU's byte order; *big* / *little* are fixed regardless of host.

Directive	Consumes	W	Signed	Endian	Modifiers	Notes
a	1 string	count	-	-	-	NUL-padded to width, truncates if too long
A	1 string	count	-	-	-	Space-padded to width
Z	1 string	count	-	-	-	NUL-terminated; `Z*` always appends a trailing `NUL`
b	1 string	count/8	-	-	-	Bit string, LSB first within each byte
B	1 string	count/8	-	-	-	Bit string, MSB first within each byte
h	1 string	count/2	-	-	-	Hex string, low nybble first

continues on next page

Table 1 – continued from previous page

Directive	Consumes	W	Signed	Endian	Modifiers	Notes
H	1 string	count/2	-	-	-	Hex string, high nybble first
c	1 integer	1	yes	-	-	Signed char
C	1 integer	1	no	-	-	Unsigned char (octet)
W	1 integer	1	no	-	-	Unsigned char; allows values above 255 in U0 mode
s	1 integer	2	yes	native	! < >	s! = native short
S	1 integer	2	no	native	! < >	S! = native unsigned short
l	1 integer	4	yes	native	! < >	l! = native long
L	1 integer	4	no	native	! < >	L! = native unsigned long
i	1 integer	native	yes	native	! < >	sizeof(int); at least 32 bits
I	1 integer	native	no	native	! < >	sizeof(unsigned int)
q	1 integer	8	yes	native	< >	Requires 64-bit-integer Perl
Q	1 integer	8	no	native	< >	Requires 64-bit-integer Perl
n	1 integer	2	no	big	!	Portable network order; n! = signed

continues on next page

Table 1 – continued from previous page

Directive	Consumes	W	Signed	Endian	Modifiers	Notes
N	1 integer	4	no	big	!	Portable network order; N! = signed
v	1 integer	2	no	little	!	"VAX" order; v! = signed
V	1 integer	4	no	little	!	"VAX" order; V! = signed
j	1 integer	IV size	yes	native	< >	Perl-internal signed integer
J	1 integer	UV size	no	native	< >	Perl-internal unsigned integer
f	1 number	4	-	native	< >	Single-precision IEEE 754
d	1 number	8	-	native	< >	Double-precision IEEE 754
F	1 number	NV size	-	native	< >	Perl-internal float (NV)
D	1 number	varies	-	native	< >	Long double; format varies by platform
p	1 string / undef	ptr	-	native	< >	Pointer to NUL-terminated string; undef → null pointer
P	1 string / undef	ptr	-	native	< >	Pointer to fixed-length buffer; count = buffer length
u	1 string	varies	-	-	-	Uuencoded; count = max bytes per output line (default 45)

continues on next page

Table 1 – continued from previous page

Directive	Consumes	W	Signed	Endian	Modifiers	Notes
`U`	1 code-point	varies	-	-	-	Unicode character number; encodes to UTF-8 in `U0` mode
`w`	1 integer ⩾ 0	varies	no	-	-	BER-compressed integer, big-endian base-128
`x`	nothing	1	-	-	!	Insert one `NUL`; `x!N` aligns to multiple of `N`
`X`	nothing	−1	-	-	!	Back up one byte; `X!N` aligns backward
`@`	nothing	absolute	-	-	!	Zero-fill or truncate to position N within group
`.`	1 integer	absolute	-	-	!	Zero-fill or truncate to position given by the value
`( … )`	-	-	-	-	`< > !`	Group: repeat count and endianness propagate inside
`/`	-	-	-	-	-	**Unpack-only.** In pack, use *length-item*`/`*item*

The modifiers:

Modifier	Effect
`!`	On `s` / `S` / `l` / `L` / `i` / `I`: use native `sizeof(…)` instead of fixed width. On `x` / `X`: turn into alignment commands. On `n` / `N` / `v` / `V`: interpret as signed. On `@` / `.`: count bytes of the internal representation, not characters.
`>`	Force **big-endian** byte order ("big end touches the construct").
`<`	Force **little-endian** byte order.

Applied to a group, `<` / `>` cascade into every byte-ordered directive inside the group and are silently ignored by directives that do not accept them.

Repeat counts

A directive letter may be followed by:

- a **number** `N` - apply the directive that many times, consuming `N` values from `LIST`
- `*` - consume all remaining values; for `x` / `X` / `@` this is equivalent to `0`; for `u` it selects the default of 45
- `[N]` - equivalent to a bare `N`
- `[template]` - the repeat count is the packed byte length of *template*. `x[L]` skips as many bytes as a packed long; `x![d]` aligns to a double boundary

String and bit/nybble directives treat the count as **width of a single value**, not count-of-values: `pack "A4", "abcdef"` produces `"abcd"`, not four copies of `"abcdef"`.

Grouping

Parentheses group directives. A group may take a repeat count or an endianness modifier:

```
pack "(sl)<", -42, 4711     # same as "s<l<", -42, 4711
pack "(CCS)*", @triplets    # repeat group for every triplet
```

Within each repetition of a group, `@` positioning starts over at 0 - `pack '@1A((@2A)@3A)', qw(X Y Z)` produces `"\0X\0\0YZ"`.

Length-prefixed payloads (`/`)

Write `length-item/sequence-item`. The length is computed from the sequence value and packed according to *length-item*; then the sequence itself is packed according to *sequence-item*:

```
my $msg = pack "n/a*", "hello, world";
# "\x00\x0chello, world" - 16-bit big-endian length, then the bytes
```

The length directive may be any integer directive (`n`, `N`, `w`, `C`, …) or even a string directive (`A4`, `Z*`) when you want the length written as ASCII.

Examples

Build a fixed-width record with two NUL-terminated strings and a 32-bit native timestamp:

```
my $rec = pack "Z8 Z8 L", "alice", "server01", $now;
# 8 + 8 + 4 = 20 bytes
```

Portable network byte order - prefer `n` / `N` / `v` / `V` over native-width directives when the bytes leave the machine:

```
my $be = pack "n N", 42, 4711;      # big-endian 16 + 32 bit
my $le = pack "v V", 42, 4711;      # little-endian
my $sx = pack "s>l>", -42, 4711;    # signed big-endian via modifier
my $gx = pack "(sl)<", -42, 4711;   # signed little-endian via group
```

Align a field inside a C struct. `x![d]` inserts just enough `NUL` bytes to reach the next multiple of a double's width:

```
# struct { char c; double d; char cc[2]; }
my $s = pack "c x![d] d c2", $c, $d, $c1, $c2;
```

Round-trip through a hex string:

```
my $raw = pack "H*", "deadbeef";    # "\xde\xad\xbe\xef"
my $hex = unpack "H*", $raw;        # "deadbeef"
```

Edge cases

- **Too few values**: missing values are treated as `""`. `pack "A4 A4", "hi"` yields `"hi \0\0\0\0"`, not a fatal error.
- **Too many values**: extras are silently ignored.
- **`a` vs `A` vs `Z`**: all three pad to an exact width. `a` pads with `NUL`, `A` pads with space, `Z` pads with `NUL` **and** guarantees a trailing `NUL` - so `Z8` encodes at most 7 data bytes plus the terminator.
- **Character width vs byte width** - the single most common trap. Counts for `a` / `A` / `Z` and offsets for `@` / `.` are in **characters of the packed string**, not bytes. In `C0` mode a character is one byte; in `U0` mode a character may span multiple UTF-8 bytes. Use the `!` modifier on `@` / `.` when you need byte offsets regardless of mode.
- **Endianness on `s` / `S` / `l` / `L` / `i` / `I` / `q` / `Q`**: native only, not portable. Use `n` / `N` / `v` / `V` or apply `>` / `<` explicitly. `f` / `d` are likewise native; IEEE 754 alone does not pin down endianness.
- **`Inf` / `NaN` packed as integers**: fatal error. No sensible mapping exists.
- **`q` / `Q` on non-64-bit Perl**: raises an exception.
- **`p` and `P` capture pointers** into the caller's memory. The referent must remain live until the packed string is consumed - a temporary string passed to `p` may be freed before you read it back. Avoid outside XS code.

- **Grouping and @**: positioning with `@` starts over at 0 inside every repetition of a group. `pack '@1A((@2A)@3A)', qw(X Y Z)` produces `"\0X\0\0YZ"`, not what a naive reading suggests.
- **Floating-point round-trip loss**: Perl stores numbers as doubles, so `unpack("f", pack("f", $x))` generally does not equal `$x` - packing through single precision truncates.
- **Writing packed bytes to a text handle**: a handle with `:utf8` or `:encoding(…)` will re-encode the bytes. Use `binmode $fh` or open with `">:raw"`.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `unpack` - the inverse operation; same template language
- `sprintf` - formatted text output rather than binary bytes
- `vec` - bit-level access to a string without a template
- `chr` / `ord` - single-character conversions
- `syswrite` - write a packed byte string without PerlIO surprises
- *pack/unpack tutorial* - task-oriented walkthrough with worked protocol and file-format examples

Scoping · Modules · Classes and OO

package

Declare the compile-time namespace for the declarations that follow.

`package` tells the compiler which symbol table to use when it resolves unqualified names in the code it is about to parse. It does **not** create a package in any run-time sense - packages spring into existence the first time something is stored in them. What a `package` statement really does is re-point the compiler: from here on, `$foo` means `$NAMESPACE::foo`, `&bar` means `&NAMESPACE::bar`, and so on, until the next `package` statement or the end of the enclosing scope.

Synopsis

```
package NAME;
package NAME VERSION;
package NAME      { ... }
package NAME VER { ... }
```

What you get back

`package` is a compile-time directive, not an expression. It has no useful runtime value - treat it as a statement. Its effect is on the parser: all subsequent unqualified identifiers resolve in `NAME`'s symbol table until scope ends.

The form with `VERSION` has one visible side effect: it sets `$NAME::VERSION` to a `version` object built from `VERSION`. This happens exactly once per `package NAME VERSION` statement, at compile time.

Global state it touches

- **The current compile-time package.** Every `package` statement replaces it for the rest of the scope. `__PACKAGE__` returns the current value at any point in the source.
- `$NAME::VERSION` - set when the `VERSION` argument is provided. Set it only once per package.
- The target symbol table (stash) is created on first use. A package's stash is itself a hash named `%NAME::` (so `main`'s is `%main::`), keyed by the unqualified symbol names it holds. Nothing is populated by the `package` statement itself; subsequent declarations (`sub`, `our`, assignments to `$NAME::foo`, etc.) fill it.

The four forms

`package NAMESPACE`

Switches the compile-time package to `NAMESPACE` from this point to the end of the current scope (enclosing block, file, or `eval`). This is the classic "one package per file" form:

```
package My::Thing;

sub new { ... }          # defines &My::Thing::new
our $count = 0;          # defines $My::Thing::count

1;
```

`package NAMESPACE VERSION`

Same scoping as the bare form, plus it sets `$NAMESPACE::VERSION` to a `version` object. `VERSION` must be a **strict** version literal: either a positive decimal (`1.23`, `0.001`) or a dotted-decimal v-string with a leading `v` and at least three components (`v1.2.3`). Exponent notation is not allowed.

```
package My::Thing 1.042;
package My::Thing v1.2.3;
```

Set `$VERSION` only once per package - a second `package NAME VERSION` statement in the same package overwrites the first, and `use Module VERSION` consumers see only the final value.

package NAMESPACE BLOCK

Lexically scopes the package switch to BLOCK. Inside the braces the compile-time package is NAMESPACE; **outside the closing brace the previous package is restored**. This is the only form that reverts automatically.

```
package Outer;
our $x = 1;              # $Outer::x

package Inner {
    our $y = 2;          # $Inner::y
}

our $z = 3;              # $Outer::z again
```

package NAMESPACE VERSION BLOCK

The block form with a version literal. Scope reverts at the closing brace; `$NAMESPACE::VERSION` stays set.

```
package My::Sub v0.9.1 {
    sub ping { "pong" }
}
```

What package does and does not scope

`package` affects **dynamic (package) variables** and unqualified name resolution. It does not affect lexicals:

- `my`, `state` - created in the lexical pad, invisible to package lookup. A `package` statement does nothing to them.
- `local` - dynamically scopes a package variable, which still belongs to whatever package its fully-qualified name names.
- `our` - declares a lexical alias to a package variable in the *current compile-time package*. This is the mechanism that lets `our $count` inside `package My::Thing` alias `$My::Thing::count`.

Switching packages after an `our` declaration does not retarget the alias: the alias was bound to the package in effect when `our` was compiled.

```
package Foo;
our $x;                  # aliases $Foo::x
package Bar;
$x = 1;                  # still writes $Foo::x
```

Interaction with use v5.36 and newer

`use v5.36` (and any `use v5.36` or later) enables `strict` and `warnings` implicitly. Once strict is on, unqualified package variables require an `our` declaration (or full qualification). Combined with a `package` statement, the idiomatic modern file skeleton is:

```
package My::Thing 1.00;

use v5.36;              # strict, warnings, say, signatures, ...

our @EXPORT_OK = qw(thingify);

sub thingify { ... }

1;
```

The `package NAME VERSION` form and `use VERSION` are independent: the first sets `$My::Thing::VERSION`; the second enables language features for the current file.

Special identifiers that always mean `main::`

A small set of identifiers are forced into the `main::` package regardless of the current package:

- `STDIN`, `STDOUT`, `STDERR`, `ARGV`, `ARGVOUT`
- `@ARGV`, `@INC`, `%ENV`, `%SIG`, `%INC`
- All punctuation variables: `$_`, `$!`, `$@`, `@_`, etc.
- The single-character identifier `_`, in all its sigils: the underscore filehandle used by `stat` and the `-X` file tests, the scalar `$_`, and the array `@_`.

Writing to `$!` inside `package Foo` still writes to `$main::!`. This is by design: these names would be useless if they moved with the package.

Examples

A minimal module file. The file name must match the package name (`require` translates `::` to the directory separator):

```
# lib/My/Greeter.pm
package My::Greeter 1.00;
use v5.36;

sub new  { my ($cls, %a) = @_; bless { %a }, $cls }
sub greet { my $self = shift; "hello, $self->{name}\n" }

1;
```

Two packages in one file using the block form. Scope reverts automatically, so the `1;` at the bottom belongs to no package in particular - it's evaluated in whichever package was active at the top of the file (`main` for a script, the file's own package for a module):

```
package My::Thing {
    sub do_stuff { ... }
}
```

(continues on next page)

(continued from previous page)

```
package My::Thing::Helper {
    sub helper { ... }
}
```

Switching back and forth in the statement form - legal but rarely what you want:

```
package A;
sub one { 1 }           # &A::one

package B;
sub two { 2 }           # &B::two

package A;
sub three { 3 }         # &A::three
```

Setting a version and reading it back. Note that `$VERSION` is a `version` object, not a plain string; stringify or compare with `version`:

```
package Foo 1.23;

say $Foo::VERSION;      # 1.23
say ref $Foo::VERSION;  # version
```

`our` inside a package gives you a lexically-scoped alias to a package variable - the canonical way to declare module-level globals under `strict`:

```
package Counter;
use v5.36;

our $count = 0;         # aliases $Counter::count
sub bump { $count++ }
```

Edge cases

- **`package NAMESPACE VERSION` accepts only strict version literals.** Exponent notation (`1e3`), non-numeric strings, and two-component v-strings (`v1.2`) are compile errors.
- **Setting `$VERSION` twice** in the same package (whether by two `package NAME VERSION` statements or by explicit assignment after a `package NAME VERSION`) is legal but confusing - `use Module VERSION` checks the final value, and CPAN tooling reads whichever one it finds first. Declare the version once.
- **The block form is a separate lexical scope.** `my` and `state` variables declared inside a `package NAME { ... }` are not visible outside, just like any other block.
- **`package NAMESPACE` without a trailing statement or block** affects the rest of the file. A `package` at the top of a script permanently switches the script out of `main`, which means `$main::ARGV[0]` is still your argument list but freshly declared globals belong to the new package. This is almost never what a script wants.

- **A file that contains only `package Foo;` and no other code** still compiles cleanly and creates the `Foo::` stash on first reference - but `require`ing it returns the value of the last statement, which is the `package` statement itself (false). Put `1;` at the end of every module file.
- **Nested block forms** restore the immediately enclosing package, not `main`:

```
package Outer;
package Inner {
    package Innermost {
        # __PACKAGE__ is Innermost
    }
    # __PACKAGE__ is Inner again
}
# __PACKAGE__ is Outer again
```

- **`package` inside `eval`.** The package switch is scoped to the `eval` body. Symbols created during the `eval` persist in their stashes (stashes are global), but the compile-time package reverts when the `eval` ends.
- **Apostrophe as package separator.** The apostrophe was Perl's original namespace separator: `$main'sail` meant `$main::sail`, long before `::` existed. It survives only for reading Perl 4 era code and is disabled under modern feature bundles, so a current program written with `use v5.36` (or any recent bundle) will not recognise it. Do not write it; the double-colon `::` is the only separator current code should use.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `use` - load a module at compile time and import symbols from its package; typically paired with a `package` declaration at the top of the loaded file
- `require` - load a module or file at runtime; the loaded file's `package` statement determines which stash its declarations populate
- `bless` - associate a reference with a package so method calls dispatch to that package's subs; the glue between `package` and OO
- `my` - declare a lexical variable that `package` does **not** affect; the counterpart to package globals
- `our` - declare a lexical alias to a package variable in the current compile-time package; the strict-safe way to use module-level globals
- `local` - dynamically scope a package variable's value without changing which package it belongs to

Processes

pipe

Open a pair of connected filehandles - one for reading, one for writing.

`pipe` wraps the `pipe(2)` system call. It creates an unnamed, unidirectional byte channel in the kernel and hands you two filehandles: `READHANDLE`, from which bytes can be read, and `WRITEHANDLE`, into which bytes can be written. Anything written to `WRITEHANDLE` becomes available on `READHANDLE` in the same order. The classic use is communication between a parent process and a child produced by `fork`: one side writes, the other reads.

Synopsis

```
pipe READHANDLE, WRITEHANDLE
```

What you get back

True on success, false on failure (with `$!` set to the reason - typically `EMFILE` or `ENFILE` when the process or system file-table is full). Both handles are created as side effects on the arguments; there is no "pipe object" return value.

On success you now hold **two** open file descriptors. Every descriptor you create is a descriptor you must `close` - especially the end each process does **not** use after a `fork` (see *Edge cases*).

Global state it touches

- `$!` - set on failure to the `errno` from the underlying `pipe(2)` call.
- `$|` - **autoflush** on the write-side handle. `print` buffers through PerlIO, so without autoflush the reader may see nothing until the writer exits or the buffer fills. Set `$|` on the write handle (typically via `select`) before writing, or your protocol will deadlock.
- `$^F` - the **maximum system filehandle**. New descriptors with `fileno` greater than `$^F` (default 2, i.e. above `STDERR`) have the close-on-exec flag set automatically, so they do not leak into programs started by `exec`. Lower `$^F` to keep the pipe open across `exec`; raise it, or dup onto a low-numbered fd, to force inheritance.

Examples

A parent writes a line; a forked child reads it:

```
pipe my $reader, my $writer or die "pipe: $!";

my $pid = fork // die "fork: $!";
if ($pid == 0) {
    # child: reads only
    close $writer;
    my $line = <$reader>;
    print "child saw: $line";
    close $reader;
```

(continues on next page)

(continued from previous page)

```
    exit 0;
}

# parent: writes only
close $reader;
$writer->autoflush(1);
print $writer "hello from parent\n";
close $writer;
waitpid $pid, 0;
```

The child reports the other direction - a child produces data, the parent consumes it:

```
pipe my $reader, my $writer or die "pipe: $!";

my $pid = fork // die "fork: $!";
if ($pid == 0) {
    close $reader;
    $writer->autoflush(1);
    print $writer "$_\n" for 1 .. 5;
    close $writer;
    exit 0;
}

close $writer;
while (my $line = <$reader>) {
    print "got: $line";
}
close $reader;
waitpid $pid, 0;
```

Autoflush from the selected-handle side, for code that predates `IO::Handle` method calls:

```
pipe my $reader, my $writer or die "pipe: $!";
my $old = select $writer; $| = 1; select $old;
```

For most uses, `open` with a pipe mode is simpler - it forks, pipes, and `exec`s for you:

```
open my $fh, '|-', 'gzip', '-c' or die $!;   # write to gzip's stdin
print $fh "data\n";
close $fh;

open my $fh, '-|', 'ls', '-l'   or die $!;   # read from ls's stdout
while (my $line = <$fh>) { print $line }
close $fh;
```

Reach for bare `pipe` when you need both directions, need to retain the unexec'd child, or are building something `open`'s two pipe-forms do not cover - then `IPC::Open2` / `IPC::Open3` wrap the `pipe` + `fork` + `exec` dance for you.

Edge cases

- **Close the end you do not use.** After `fork`, **both** processes hold **both** descriptors. The reader must close the write end, and the writer must close the read end. Skip this and you get the classic pipe bug: the reader's `<$reader>` never returns EOF, because the kernel still sees a writer open (the reader's own, idle, copy of `$writer`). The program hangs.
- **Writing to a reader-less pipe raises `SIGPIPE`.** Once every copy of the read end is closed, the next `print` on the write side delivers `SIGPIPE`, whose default action is to kill the process. Install `$SIG{PIPE} = 'IGNORE'` (or a handler) to turn it into a normal write failure with `$!` set to `EPIPE`, which you can then check.
- **Deadlock in pipe loops.** Two processes each writing to the other through pipes will block forever if both write buffers fill before either process reads. Kernel pipe buffers are small (typically 64 KiB). Drain one side, use non-blocking I/O, or use `select` / `IO::Select` to multiplex.
- **Buffering hides writes.** Without `$|` (or `$writer->autoflush(1)`) on the write side, data sits in PerlIO's buffer until the buffer fills or the handle is closed. A reader blocked on `<$reader>` will wait - often indefinitely - for data that has already been "written" from the writer's point of view.
- **Unidirectional only.** `pipe` produces a one-way channel: write to `WRITEHANDLE`, read from `READHANDLE`, not the reverse. For bidirectional communication use `socketpair` or two pipes.
- **Binary data.** The handles returned by `pipe` start in the default PerlIO layer stack. If you are moving raw bytes across the pipe, apply `binmode` to both ends; otherwise CRLF or encoding layers inherited from the process defaults can rewrite your data.
- **Close-on-exec and `fileno`.** On systems with `FD_CLOEXEC`, any new descriptor whose `fileno` exceeds `$^F` is created with the close-on-exec flag set. Pipe descriptors typically land well above 2, so by default they do **not** survive `exec` - which is normally what you want. Lower `$^F` *before* calling `pipe` if you intend to pass the descriptors to an `exec`'d program.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `fork` - the other half of the pipe idiom; `pipe` is almost always called just before a `fork`
- `open` - the `'|-'` and `'-|'` modes fork, pipe, and `exec` a child in one call; prefer them when you need exactly one direction to one external command
- `close` - closing the unused end in each process is mandatory, not optional
- `read` - byte-oriented reads from `READHANDLE` when the line-oriented `<$fh>` form is the wrong shape
- `print` - the normal way to push data into `WRITEHANDLE`; remember `$|`

- `fileno` - recover the underlying file descriptor, e.g. to hand to `select` / `IO::Select` or to a lower-level system call

Arrays

pop

Remove and return the last element of an array.

pop shortens the array by one and hands you the element that was removed. The array is modified in place. With no argument, pop operates on `@_` inside a subroutine and on `@ARGV` everywhere else - the same default target as `shift`.

Synopsis

```
pop @arr
pop @$aref
pop                 # default: @_ inside a sub, @ARGV outside
```

What you get back

The element that was removed, as a scalar. If the array was empty, the return value is `undef` and the array stays empty - this is not an error and emits no warning, so code that always calls pop in a loop needs its own termination check.

```
my @empty;
my $x = pop @empty;         # $x is undef, no warning
```

Note that pop also returns `undef` when the last element *was* `undef`. The two cases are indistinguishable from the return value alone; use `@arr` in boolean context, or `scalar @arr`, to tell them apart before popping.

```
my @arr = ('one', 'two', undef);
my $x = pop @arr;           # $x is undef - but so would an empty␣
↪pop be
```

Global state it touches

pop with no argument reads `@_` (inside a subroutine) or `@ARGV` (at file scope, in `BEGIN`/`INIT`/`CHECK`/`END` blocks, and inside `eval STRING`). It does not interact with `$_`, `$,`, `$\`, or any other punctuation variable.

Examples

Basic use - pull the last element off a plain array:

```
my @arr  = ('cat', 'dog', 'mouse');
my $item = pop @arr;        # $item is 'mouse'
# @arr is now ('cat', 'dog')
```

LIFO stack - `push` to add, pop to remove, both at the same end:

```
my @stack;
push @stack, 'a';
push @stack, 'b';
push @stack, 'c';
my $top = pop @stack;       # 'c'; @stack is ('a', 'b')
```

Discard the last element when you do not need its value. pop in void context is perfectly valid and is the idiomatic way to drop the tail:

```
pop @arr;                   # shorten @arr by one, value thrown away
```

Default target inside a subroutine - pop with no argument pops from @_, which is how a sub peels its last argument off the argument list:

```
sub last_arg {
    return pop;             # pops from @_
}
last_arg(1, 2, 3);          # returns 3
```

Pop through an array reference. Dereference with `@$aref` (or `@{ $aref }` for any non-trivial expression) and pop works exactly as on a named array:

```
my $aref = [10, 20, 30];
my $x    = pop @$aref;      # 30; $aref now points to [10, 20]
```

Edge cases

- **Empty array returns undef, not an error.** No warning is issued, even under `use warnings`. A loop of the form `while (defined(my $x = pop @arr)) { ... }` terminates on empty *or* on a genuine undef element - use `while (@arr)` instead if the array may legitimately contain undef.
- **Bareword expression in place of ARRAY.** pop requires an array, not an arbitrary expression. pop `@$aref` works because `@$aref` is array syntax; pop `$aref` was briefly allowed as an experiment between 5.14 and 5.22 and has been a compile error since 5.24. Always write the explicit @ sigil on the dereference.
- **Tied arrays.** pop `@tied` calls the tied class's POP method if defined; otherwise the default FETCHSIZE + FETCH + DELETE + STORESIZE fallback applies. Side effects of those methods fire in that order.
- **pop shortens the array.** After pop `@arr`, `$#arr` decreases by one and `scalar @arr` drops by one. If you need to inspect the tail without removing it, read `$arr[-1]` or `$arr[$#arr]` directly - see $#arr in perlvar.
- **No-argument form changes meaning by scope.** At file scope, pop; means pop @ARGV. Inside a subroutine it means pop `@_`. The same source line therefore behaves differently depending on where it appears - a common surprise when code is refactored into or out of a sub. Write the target explicitly (pop `@ARGV` or pop `@_`) whenever the intent is not obvious from context.
- **pop on an array aliased through @_.** Subroutine arguments are aliases to the caller's values. pop `@_` removes the alias entry from `@_` but does not shorten the

caller's array; to shrink the caller's array the sub must be passed the array *by reference* and pop `@$aref` used instead.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `push` - the counterpart; append to the same end of the array that pop removes from
- `shift` - remove and return the *first* element; same defaulting rules as pop (`@_` in a sub, `@ARGV` otherwise)
- `unshift` - prepend to the front; paired with `shift` the way `push` is paired with pop
- `splice` - remove or replace an arbitrary slice; `splice @arr, -1` is the long-hand form of `pop @arr`
- `last` - loop-control keyword, unrelated to array end; listed because the names are often confused
- `$#arr` - last index of `@arr`; changes when pop shortens the array

Regular expressions and pattern matching

pos

Report or set where the next `/g` regex match will resume in a string.

`pos` reads (or, as an lvalue, writes) the offset that the regex engine stores on a scalar after a global match (`m//g`, `s///g`). The offset counts characters, not bytes, and is the position **after** the last successful match - the place the next `/g` iteration will start scanning. With no argument `pos` operates on `$_`.

Synopsis

```
pos SCALAR
pos $str = N
pos
```

What you get back

An integer offset, or `undef` when no position is recorded. `0` is a valid offset and means "start of string"; it is not the same as `undef`, which means "no `/g` match has run, or the last one failed and reset the position." Always distinguish the two with `defined`:

```
if (defined pos $str) {
    # a /g scan is in progress
}
```

Used as an lvalue, `pos SCALAR` returns an assignable location:

```
pos($str) = 5;          # next /g match starts at char offset 5
```

Global state it touches

pos reads and writes the per-scalar regex position attached to its operand. With no argument it targets $_. The stored offset is what \G anchors against in the next match, so every call to pos potentially changes where \G binds.

Examples

Walk every word in a string with /g in scalar context, using pos to report progress:

```
my $s = "one two three";
while ($s =~ /(\w+)/g) {
    printf "%-5s ends at %d\n", $1, pos $s;
}
# one   ends at 3
# two   ends at 7
# three ends at 13
```

Skip ahead before starting the scan. The first match begins at offset 4, not 0:

```
my $s = "AAA BBB CCC";
pos($s) = 4;
$s =~ /(\w+)/g;
print $1, "\n";             # BBB
```

Anchor a follow-up match to the previous one with \G. Without \G the engine would scan forward past any gap; with it, the match must start exactly where the last one ended:

```
my $s = "12ab34cd";
while ($s =~ /\G(\d+)(\w+?)(?=\d|\z)/g) {
    print "num=$1 tail=$2\n";
}
# num=12 tail=ab
# num=34 tail=cd
```

Restart a scan by clearing the position:

```
pos($s) = undef;            # next /g starts from offset 0 again
```

Edge cases

- **Bare pos** targets $_. pos inside `while (<>) { ... }` therefore reports the position on the current input line.
- **Characters, not bytes.** For a string containing multi-byte characters, pos returns the character offset. The (deprecated) `use bytes` pragma switches to byte offsets; new code should not rely on it.

- **Failed `/g` match resets the position to `undef`** - the next `/g` starts over at offset `0`. Add the `/c` modifier (`m//gc`) to preserve the position on failure, which is the usual idiom when composing several alternative `\G`-anchored patterns against the same string.
- **Reads during a match are stale.** `pos` reflects the **previous** match's end. Expressions like `(?{ pos() = 5 })` or `s//pos() = 5/e` influence the next match, not the one currently running.
- **Zero-length match flag.** Setting `pos` also clears the internal *matched with zero-length* flag, so a subsequent zero-width match at the same position is allowed again. See the *performance chapter* of the regex guide under *zero-length-match termination*.
- **Non-lvalue operand.** `pos` requires a real scalar variable for the lvalue form; `pos("literal") = 3` is a compile-time error.
- **Offset `0` vs `undef`.** `pos` of `0` means the next `/g` starts at the beginning; `undef` means no position is set. They behave identically for the first match but differ after the zero-length-match flag state is considered.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `m` - the `/g` modifier is what creates and advances the position `pos` reads; the `/c` modifier on failure preserves it
- `qr` - precompile a pattern once, then reuse it in `\G`-anchored `/g` loops without reparsing
- `split` - the other everyday way to walk a string in pieces; no `pos` involved, but often a cleaner choice when the delimiters are simple
- `\G` *anchor* - zero-width anchor that binds to the current `pos`; the main reason to touch `pos` in the first place
- `$_` - default target of bare `pos`

I/O

print

Write a list of values to a filehandle.

`print` is the everyday output primitive. It stringifies each item in `LIST`, joins them with the current value of `$,` (the output field separator, usually empty), writes them to `FILEHANDLE`, and finally appends `$\` (the output record separator, usually empty). If `FILEHANDLE` is omitted, output goes to the currently selected handle (see `select`), which defaults to `STDOUT`. If `LIST` is omitted, `$_` is printed.

Synopsis

```
print LIST
print FILEHANDLE LIST
print {EXPR} LIST
print
```

What you get back

1 on success, `undef` on failure (with `$!` set). The return value is rarely checked - most `print` calls target `STDOUT` or `STDERR` and a write failure to either is already a program-ending condition. Check the return value when writing to a file, pipe, or socket where partial writes and broken pipes matter.

```
print $fh @records
    or die "write to $path failed: $!";
```

Default filehandle, default separators

Three pieces of global state shape every `print` call:

- The **currently selected filehandle**, changed with `select`. `print LIST` (no explicit filehandle) writes there. At program start this is `STDOUT`.
- `$,` - **output field separator**, inserted between list items. Default is the empty string, so `print 1, 2, 3` produces 123, not `1 2 3`. Also known as `$OFS` under `use English`.
- `$\` - **output record separator**, appended after the whole list. Default is the empty string, so `print` does **not** add a newline. Also known as `$ORS` under `use English`.

Setting both gives Python-like `print` semantics:

```
local $, = " ";
local $\ = "\n";
print 1, 2, 3;          # "1 2 3\n"
```

Use `local` when changing these inside a subroutine so the change does not leak to the caller.

`say` is exactly `print` with `$\` forced to `"\n"` for that one call - reach for it when you want a newline every time and do not want to touch `$\`.

Examples

Basic write to `STDOUT`:

```
print "hello, world\n";
```

Write to a filehandle with the indirect-object form. Note: **no comma** between the filehandle and the list:

```
open my $fh, ">", "out.txt" or die $!;
print $fh "line 1\n", "line 2\n";
close $fh;
```

Write to a filehandle held in an expression. Any filehandle more complex than a bareword or a plain scalar needs the `{EXPR}` block form:

```
my @logs = (\*STDOUT, \*STDERR);
print { $logs[$level] } "message\n";
```

Join list items with `$,` and terminate the record with `$\`:

```
local $, = ", ";
local $\ = ".\n";
print "apples", "pears", "plums";   # "apples, pears, plums.\n"
```

A list containing `undef` - each `undef` stringifies to the empty string and triggers an `uninitialized` warning under `use warnings`:

```
use warnings;
my $x;
print "value=", $x, "\n";           # "value=\n" plus a warning
```

List context vs scalar context. `@arr` in `LIST` position expands to its elements; wrap in `scalar` to print the count instead:

```
my @arr = (10, 20, 30);
print @arr, "\n";                   # "102030\n"
print scalar @arr, "\n";            # "3\n"
```

Suppressing the `undef` warning for a block where missing values are expected and intended to render as empty:

```
no warnings 'uninitialized';
print join(",", @row), "\n";        # empty fields render as ""
```

Edge cases

- **No arguments**: `print;` prints `$_` to the selected handle. Used idiomatically inside `while (<>) { print; }`.
- **The filehandle-or-first-arg ambiguity**: Perl decides whether the first token is a filehandle by parsing, not by runtime type. A bareword or a simple scalar variable immediately followed by a term (no comma) is taken as a filehandle:

  ```
  print STDERR "oops\n";            # STDERR is the filehandle
  print $fh "oops\n";               # $fh is the filehandle
  ```

  An expression that returns a filehandle does **not** work in that position - `print $handles[0] "x"` is a syntax error. Use the block form: `print { $handles[0] } "x"`.

- **List context applies to every argument**. A subroutine call in `LIST` runs in list context, not scalar:

```
print localtime(), "\n";          # prints nine numbers, no␣
 ↪spaces
print scalar localtime(), "\n";   # prints the familiar date␣
 ↪string
```

- **`print (…)` parses as a function call**. `print (2+3)*4` prints 5, then discards `*4`. Parenthesise the whole argument list, or prefix with `+`:

```
print((2+3)*4);                   # prints 20
print +(2+3)*4;                   # prints 20
```

- **Closed filehandle**: returns `undef` and sets `$!` to `"Bad file descriptor"`. Under `use warnings` a `print() on closed filehandle` warning is emitted.
- **Encoding layers**: `print` writes whatever the handle's PerlIO stack says to write. If the handle has a `:utf8` or `:encoding(...)` layer, strings are encoded on the way out. Bytes-mode handles expect byte strings; passing a wide-character string writes raw codepoints and emits a `Wide character in print` warning.
- **Autoflush**: `print` buffers through PerlIO. Set `$|` on the selected handle to flush after every write - commonly needed for progress output on pipes or when coordinating with another process:

```
$| = 1;                           # autoflush current handle
```

- **Printing to a closed pipe or socket** raises `SIGPIPE`. The default action is to terminate the process; install a handler (`$SIG{PIPE} = 'IGNORE'` or a sub) to turn it into a normal write failure you can check.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `say` - `print` with `$\` forced to `"\n"` for the call; use it when every record ends in a newline
- `printf` - same targeting rules as `print`, but the first argument is a `sprintf` format string and `$\` is **not** appended
- `sprintf` - build the formatted string without writing it; combine with `print` when you want to log it somewhere else first
- `select` - change which filehandle the no-argument form of `print` writes to
- `$,` - output field separator inserted between list items
- `$\` - output record separator appended after the list
- `$|` - autoflush flag for the currently selected handle

I/O

printf

Write a formatted string to a filehandle.

`printf` is `print FILEHANDLE sprintf(FORMAT, LIST)` with one deliberate exception: the output record separator `$\` is **not** appended. The targeting rules are otherwise the `print` rules - bareword or simple-scalar filehandle immediately before the argument list, or the `{EXPR}` block form for anything more complex. If `FILEHANDLE` is omitted, output goes to the currently selected handle (see `select`), which starts life as `STDOUT`.

See `sprintf` for the format-specifier reference - everything `%d`, `%s`, `%.3f`, `%*d`, `%1$s`, `%v`, vector flags, locale-sensitive `%f`, and so on. This page covers only the `printf`-specific surface: filehandle resolution, the format-is-first rule, and the quirks around the no-argument form.

Synopsis

```
printf FORMAT, LIST
printf FILEHANDLE FORMAT, LIST
printf {EXPR} FORMAT, LIST
printf FH                       # bareword filehandle, $_ as format
```

What you get back

1 on success, `undef` on failure (with `$!` set). As with `print`, the return value is rarely checked for writes to `STDOUT` / `STDERR`, and worth checking for writes to files, pipes, and sockets where partial writes and broken pipes matter.

```
printf $fh "%-20s %8d\n", $name, $count
    or die "write to $path failed: $!";
```

Global state it touches

- **Selected filehandle** (changed via `select`): the destination when `FILEHANDLE` is omitted. Starts as `STDOUT`.
- `$\` (output record separator): deliberately **not** appended - this is the one semantic difference between `printf FORMAT, LIST` and `print FILEHANDLE sprintf(FORMAT, LIST)`. Put the newline in the format string.
- `$,` (output field separator): **not** consulted. `printf` splices list elements into the format according to the conversions, not with a separator.
- `$_`: used as the **format** when both `FORMAT` and `LIST` are omitted, not appended to output. Almost never what you want; see *Edge cases*.
- `$!`: set on write failure.
- `$|`: autoflush flag for the selected handle. `printf` buffers through PerlIO like `print`.

- `LC_NUMERIC` locale: when `use locale` is active and `POSIX::setlocale` has been called, the decimal separator for floating-point conversions (`%f`, `%e`, `%g`) follows the locale.

Examples

Basic formatted write to `STDOUT`. Unlike `say` or `print`-with-`$\`, the trailing newline is on you:

```
printf "%s is %d years old\n", $name, $age;
```

Width and precision for a tidy column:

```
printf "%-20s %8.2f\n", $item, $price;
```

Write to a filehandle. Bareword and simple scalar work directly; no comma between the filehandle and the format:

```
open my $fh, ">", "report.txt" or die $!;
printf $fh "%5d  %s\n", $n, $msg;
close $fh;
```

Filehandle held in a more complex expression - use the `{EXPR}` block form (same rule as `print`):

```
my @logs = (\*STDOUT, \*STDERR);
printf { $logs[$severity] } "[%s] %s\n", $tag, $msg;
```

Repeated format with a list that spans multiple records. `printf` does **not** re-apply the format per record - conversions are consumed once, left to right:

```
printf "%s=%s\n", "user", "alice";        # "user=alice\n"
printf "%s=%s\n%s=%s\n",                  # two records, one call
       "user", "alice", "host", "db1";
```

Locale-sensitive decimal separator:

```
use POSIX qw(setlocale LC_NUMERIC);
use locale;
setlocale(LC_NUMERIC, "de_DE.UTF-8");
printf "%.2f\n", 1234.5;                  # "1234,50\n"
```

Edge cases

- **No LIST given, `$_` becomes the format.** `printf;` and `printf FH;` (bareword filehandle) format `$_`. If `$_` contains any % conversions they expand against an empty argument list - each `%s` / `%d` becomes the empty string and `use warnings` emits a `Missing argument in printf` warning. To write `$_` verbatim use `print`, not `printf`.
- **Indirect filehandle without a list is ambiguous** and not supported. `printf $fh;` parses `$fh` as the format, not the filehandle. If you need a no-argument form with a

scalar filehandle, go through `print` or pass an explicit format: `printf $fh "%s", $_`.

- **The first argument is always the format.** `printf(@_)` uses `$_[0]` as the format and the remaining elements as arguments - this is a parsing rule, not a convention. Build the format deliberately; don't splat user-supplied lists into `printf` unless the first element is known-safe.
- **No `$\` is appended.** Embedding `"\n"` in the format is the idiomatic terminator. Setting `local $\ = "\n"` has no effect on `printf` output.
- **`$,` is ignored.** The output field separator is a `print` concept; `printf` joins nothing - it substitutes.
- **`printf (…)` parses as a function call**, just like `print`. `printf ("%d", 1)*2` computes `printf(...)` then multiplies the return value by 2 and discards it. Wrap the whole argument list in parens or prefix with `+` if you need to disambiguate.
- **Closed filehandle**: returns `undef` and sets `$!` to `"Bad file descriptor"`. Under `use warnings` a `printf() on closed filehandle` warning is emitted.
- **Encoding layers**: `printf` writes through the handle's PerlIO stack. A `:utf8` or `:encoding(...)` layer encodes on the way out; a bytes-mode handle with a wide-character string emits `Wide character in printf` and writes raw code-points.
- **`SIGPIPE` on closed pipe/socket**: default action terminates the process. Install `$SIG{PIPE}` to turn it into a checkable write failure.
- **Don't use `printf` where `print` suffices.** `printf "%s", $msg` is slower and more error-prone than `print $msg` - the format string is parsed on every call, and a stray `%` in `$msg` used as a format becomes a bug waiting to happen.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `print` - unformatted write with the same filehandle rules; faster, and appends `$\` where `printf` does not
- `sprintf` - build the formatted string without writing it; the format-specifier reference lives here
- `say` - `print` with `$\` forced to `"\n"`; reach for it when every record ends in a newline
- `select` - change which filehandle the no-filehandle form of `printf` writes to
- `$\` - output record separator; deliberately not appended by `printf`
- `$_` - used as the format when both `FORMAT` and `LIST` are omitted

Misc

prototype

Return the prototype string of a subroutine, or `undef` if it has none.

`prototype` inspects a subroutine and gives back the prototype the parser will use when compiling calls to it. `FUNCTION` is either a name (bareword or string) or a code reference. With no argument, the subroutine named by `$_` is looked up. A `CORE::` prefix asks for the prototype-equivalent of a core built-in.

Prototypes are the old compile-time parsing hints declared with `sub foo ($$@) { ... }`. They control how the parser shapes calls - they are **not** subroutine signatures (`sub foo ($x, $y) { ... }`), which are a separate feature that `prototype` does not report on.

Synopsis

```
prototype FUNCTION
prototype
prototype \&foo
prototype 'Some::Package::foo'
prototype 'CORE::push'
```

What you get back

A string describing the prototype, or `undef` when:

- the subroutine exists but has no prototype declared, or
- the name resolves to a `CORE::` built-in whose argument shape the prototype mini-language cannot express (for example `system`, `print`, `sort`).

The string is exactly the text that appeared between the parentheses in the sub declaration, canonicalised: whitespace removed, each character one of the prototype letters below.

Character	Meaning in the prototype
`$`	one scalar argument
`@`	list - absorbs all remaining arguments
`%`	hash - absorbs all remaining arguments as a list
`&`	code block or code reference
`*`	typeglob or bareword treated as one
`\$ \@ \% \& \*`	reference to a scalar / array / hash / code / glob (caller writes `@ary`, callee receives `\@ary`)
`\[...]`	reference where any of the bracketed sigils is accepted
`+`	array or hash, passed by reference automatically
`_`	scalar that defaults to `$_` if omitted
`;`	subsequent arguments are optional

A leading `&` with no parentheses after it, as in `sub f (&@) { ... }`, is what lets `map`, `grep` and `sort`-lookalikes take a block without a comma.

Global state it touches

`prototype` with no argument reads `$_` as the subroutine name. No other globals are consulted or modified.

Examples

Read a user-defined prototype:

```
sub add ($$)   { $_[0] + $_[1] }
print prototype 'add';                  # $$
print prototype \&add;                  # $$
```

A sub with no prototype returns `undef`:

```
sub plain { "hi" }
print defined(prototype 'plain') ? "yes" : "no";   # no
```

A `CORE::` query for a built-in whose shape fits the prototype language:

```
print prototype 'CORE::push';           # \@@
print prototype 'CORE::keys';           # \[%@]
print prototype 'CORE::pos';            # ${\;}
```

A `CORE::` query for a built-in that cannot be expressed as a prototype:

```
print defined(prototype 'CORE::system') ? "yes" : "no";   # no
print defined(prototype 'CORE::print')  ? "yes" : "no";   # no
```

Use it to build a wrapper that inherits the callee's calling convention - useful when writing generic dispatchers:

```
sub wrap {
    my $name  = shift;
    my $proto = prototype $name;
    # install a new sub with the same prototype string
    eval "sub wrapped_$name ($proto) { goto &$name }";
}
```

Edge cases

- **No argument uses `$_`.** `prototype` with nothing between it and the next term consults `$_` as the sub name. A loop like `for (@names) { print prototype, "\n" }` works by accident of this rule.
- **Signatures are not prototypes.** `sub f ($x, $y) { ... }` declares a signature - `prototype 'f'` returns `undef`, because the sub has no prototype. To declare both, use the `:prototype(...)` attribute: `sub f :prototype($$) ($x, $y) { ... }`.
- **Unknown name returns `undef`.** If the sub has never been declared, `prototype` returns `undef` without raising an error. Combine with `defined` and `exists`

`&Some::Package::name` when you need to tell "no prototype" apart from "no such sub".

- **Auto-loaded subs**. If the sub is handled by `AUTOLOAD`, it has not been compiled yet, so `prototype` reports `undef` even when the eventual callable would have one. Force a stub first (`*foo = \&foo` after loading) if you must query it.
- **References versus names are interchangeable**. `prototype \&foo` and `prototype 'foo'` return the same string; pick whichever is cheaper at the call site. A reference skips package-name lookup.
- **Fully qualified names**. Unqualified names are looked up in the current package. For callers-of-callers, always qualify: `prototype 'Other::Pkg::fn'`.
- **Empty prototype is not `undef`**. `sub nullary () { 42 }` gives `prototype 'nullary'` the empty string `""`, which is defined and false. Use `defined`, not truthiness, to test for "has a prototype".

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `sub` - declare the subroutine whose prototype you are about to read
- `defined` - the correct test for "does this sub have a prototype," since an empty-string prototype is valid
- `ref` - companion inspector when `FUNCTION` is a code reference and you want to verify the ref type before calling `prototype`
- `caller` - neighbour introspector that answers "who called me," often used with `prototype` when building wrappers
- `wantarray` - runtime context query, the counterpart to compile-time prototype inspection
- `perlsub` - the prototype mini-language and how the parser applies it to call sites

Arrays

push

Append one or more values to the end of an array.

`push` adds every element of `LIST` to the end of `ARRAY`, in order. It is the paired counterpart of `pop` at the tail and of `unshift` / `shift` at the head. The array grows in place; no new array is returned.

Synopsis

```
push @arr, $val
push @arr, @more
```

(continues on next page)

(continued from previous page)

```
push @arr, list, of, values
push @$aref, $val
```

What you get back

The **new number of elements** in ARRAY after the append - i.e. `scalar @arr` as it stands once push returns. Not the pushed values, not the old length, not the array itself.

```
my @colors = ("red");
my $n = push @colors, "blue", "green";   # $n is 3
```

Useful when the push and the subsequent length check would otherwise be two statements:

```
if (push @queue, $job > $limit) { ... }  # one call, count + test
```

Examples

Basic append of a single value:

```
my @animals = ("cat");
push @animals, "mouse";                  # ("cat", "mouse")
```

Append several values in one call - each argument becomes its own element:

```
my @colors = ("red");
push @colors, "blue", "green";           # ("red", "blue", "green")
```

Append the contents of another array. The second array is flattened into the list, so its **elements** are appended, not the array as a single item:

```
my @a = (1, 2);
my @b = (3, 4);
push @a, @b;                             # (1, 2, 3, 4)
```

Queue discipline - push at the tail, `shift` at the head:

```
my @queue;
push @queue, $job;                       # enqueue
my $next = shift @queue;                 # dequeue
```

Stack discipline - push and pop at the same end:

```
my @stack;
push @stack, $frame;                     # enter
my $top = pop @stack;                    # leave
```

Push onto an array held by reference. The reference must be dereferenced with `@{ ... }` (or the short `@$ref` form) before push sees an array:

```
my $aref = [1, 2];
push @$aref, 3;                          # $aref now refers to [1,␣
 ↪2, 3]
push @{ $aref }, 4, 5;                   # same, explicit deref
```

Count elements after appending without a temporary:

```
my $size = push @log, $entry;
warn "log is large: $size entries" if $size > 1000;
```

Edge cases

- **Arrayref needs a deref.** `push $ref, $x` is a compile-time error in Perl 5.24+. Use `push @$ref, $x` or `push @{ $ref }, $x`. An experimental feature allowed a scalar first argument in 5.14–5.22 and was removed; do not rely on it.
- **Array arguments auto-flatten.** `push @a, @b` appends `@b`'s elements individually. To append `@b` as a single nested entry you need an arrayref: `push @a, \@b` or `push @a, [ @b ]`.
- **Hash arguments also flatten.** `push @a, %h` appends alternating keys and values in hash iteration order - almost never what you want. Use `push @a, \%h` to store the hash by reference instead.
- **List context but scalar return.** `push` is documented as taking a `LIST` and therefore evaluates its arguments in list context, but the value `push` itself produces is always the resulting count. Inside `print push @a, @b` the count is what gets printed, not the array.
- **Empty `LIST`.** `push @a;` (no second argument) is a no-op and returns the current `scalar @a`. Legal and occasionally useful for "give me the count without modifying anything" - though `scalar @a` is clearer.
- **Tied arrays.** `push` on a tied array calls `PUSH` on the tied object if defined; otherwise it falls back to repeated `STORE` calls at `FETCHSIZE`-derived indices. See `perltie`.
- **Autovivification.** Pushing through an undef-valued reference vivifies an anonymous array:

  ```
  my %index;
  push @{ $index{$key} }, $value;          # $index{$key} becomes␣
   ↪an arrayref
  ```

  After the first `push`, `$index{$key}` holds an arrayref containing `$value`. This is the standard idiom for building a hash-of-arrays incrementally.
- **Readonly array.** Pushing onto a readonly or fixed-size array (e.g. an element of `@_` bound to a caller's variable, or an array locked with `Hash::Util::lock_value`) raises `Modification of a read-only value attempted`.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `pop` - remove and return the last element; paired with `push` to form a stack
- `shift` - remove and return the first element; paired with `push` to form a queue
- `unshift` - `push`'s mirror image at the head of the array; returns the new length the same way
- `splice` - insert, remove, or replace any range; `push @a, @b` is equivalent to `splice @a, scalar @a, 0, @b`
- `scalar` - force scalar context on an array when you want the count without a `push`
- `@ARGV` - the canonical array most programs `push` onto or consume with `shift`

SCALARs and strings

q//

Single-quoted string literal with a choice of delimiter.

`q//` is the generic form of the customary `'...'` string literal. It produces a compile-time constant string, does **not** interpolate variables, and does **not** process the backslash escape sequences that `qq//` and regex literals recognise. Reach for `q//` whenever the literal you want to write would be awkward under `'...'` - most often because the string itself contains single quotes - or when a different delimiter simply reads better in context.

Synopsis

```
q/STRING/
q(STRING)
q[STRING]
q{STRING}
q<STRING>
q!STRING!
q#STRING#
'STRING'
```

Any non-alphanumeric character works as a delimiter. The four ASCII bracket pairs (`()`, `[]`, `{}`, `<>`) nest; all other delimiters do not. `'...'` is a shorthand for `q(...)` with identical semantics.

What you get back

A string in scalar context - the exact byte sequence between the opening and closing delimiter, with two rewrites:

- `\\` becomes a single backslash.

- `\<delim>` becomes a literal copy of the delimiter character (this is what lets you include the delimiter inside the body).

Every other character, including every other `\`, stays verbatim. There is no `\n`, no `\t`, no `$var` expansion, no `\x{...}`, no `\N{...}` - the inside of `q//` is as close to "what you typed is what you get" as Perl offers.

In list context `q//` still evaluates to the single string.

Delimiters and nesting

Non-bracketing delimiters use the same character fore and aft:

```
my $a = q!I said, "You said, 'She said it.'"!;
my $b = q/path/to/file/;   # WRONG - ends at the second "/"
my $b = q{path/to/file};   # RIGHT - braces nest, slashes are␣
↪literal
```

Bracket delimiters nest, so balanced brackets inside the body are fine and only an unbalanced closer ends the string:

```
my $c = q{foo{bar}baz};    # "foo{bar}baz"
my $d = q(f(o(o)b)ar);     # "f(o(o)b)ar"
```

Whitespace between `q` and the delimiter is allowed **except** when the delimiter is `#` - `q#foo#` is the string `"foo"`, but `q #foo#` is the operator `q` followed by a line comment, and the string body continues on the next line. When the delimiter is a word character (matches `/\w/`), whitespace between `q` and the delimiter is **required**: `q XfooX` is `"foo"`, `qXfooX` is a syntax error.

Escape rules

Inside `q//` the backslash has exactly two special jobs:

Sequence	Produces
`\\`	a single backslash
`\<delim>`	a literal copy of the delimiter

Anything else beginning with `\` stays as two characters:

```
print q(\n), "\n";       # prints "\n" then a real newline
print length q(\n);      # 2
print q(\t\r\0);         # prints "\t\r\0" - six characters
```

This is the single most frequent source of confusion for authors coming from double-quoted strings - in `q//` and `'...'`, `\n` is a backslash followed by the letter n, **not** a newline.

Examples

Embedding both quote marks without escaping:

```
my $msg = q{I said "hello", she said 'hi'.};
```

Windows-style path, where backslashes are plentiful and double-quote interpolation would be a hazard:

```
my $path = q(C:\Users\alice\Documents);
print $path;                  # C:\Users\alice\Documents
```

A literal $ without interpolation - qq// would require escaping every $ and @, q// leaves them alone:

```
my $expr = q($total = $x + $y);
print $expr;                  # $total = $x + $y
```

The one-and-only way to put the delimiter inside the body: escape it with a backslash. Note that the backslash itself is consumed.

```
my $s = q!He said "ouch\!" loudly!;
print $s;                     # He said "ouch!" loudly
```

A regex pattern stored as a plain string, deferred to qr// at use site - q// keeps backslashes literal so the pattern survives intact:

```
my $pat = q(\d{3}-\d{4});
if ($phone =~ /$pat/) { ... }
```

Multi-line literal with no escape processing - each line break in the source becomes a line break in the string:

```
my $sql = q{
    SELECT id, name
    FROM   users
    WHERE  active = 1
};
```

Edge cases

- **q// is compile-time.** The parser resolves the string when the source is read; there is no runtime cost beyond using the already-built constant. Same as '...'.
- **'...' and q(...) are indistinguishable** after parsing. The choice is purely stylistic - pick whichever reads better for the content.
- **Empty string.** q() and '' both yield the zero-length string.
- **No \n, no \t, no interpolation**, ever. If you need either, switch to qq// or "...". Mixing the two in one line is fine: q(literal) . qq(interpolated $x).
- **Backslash at end of body.** A single \ immediately before the closing delimiter escapes it, so q(foo\) is a syntax error - the \) becomes a literal) and the string is unterminated. Write q(foo\\) to end a string with one backslash.

- **Unbalanced brackets with bracket delimiters.** Because brackets nest, `q{foo{bar}` is a syntax error - the inner `{` opens a nested level and the outer `}` closes it, leaving the string unterminated. Use a non-bracket delimiter, or balance the brackets, or escape one of them: `q{foo\{bar}`.
- **# as delimiter needs no space from q**, but every other delimiter that is a word character (letters, digits, _) requires whitespace between q and the opening delimiter.
- **Quoting Perl code is fragile.** `q{ if ($x eq "}") { ... } }` is a syntax error: the `}` inside the embedded double-quoted string closes the outer brace before Perl sees the inner string. Tools like `Text::Balanced` exist because hand-writing a robust Perl source quoter is unpleasant.
- **Parser ambiguity with / as delimiter.** `q/.../` uses `/` the same way `m//` does. Inside the body no regex syntax applies, but a slash in the content will end the string prematurely - pick a different delimiter (`q{...}`, `q(...)`) when slashes appear in the text.
- **Heredocs with `<<'TAG'`** follow the same escape rules as `q//`: no interpolation, no escape processing beyond `\\` and `\TAG` not being special (the tag terminates on its own line). Use a single-quoted heredoc when you want `q//` semantics across many lines without a closing delimiter in the middle of the text.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `qq//` - double-quoted literal; same delimiter flexibility as `q//` plus variable interpolation and the full `\n`/`\t`/`\x{...}` escape vocabulary
- `qw//` - list of whitespace-separated words; what you want when the literal is a list of short tokens, not a single string
- `qr//` - compile a regex literal; the pattern-matching member of the quote-like family
- `qx//` - run a command in the shell and capture its output; same delimiter rules, semantics of backticks
- `quotemeta` - runtime equivalent of `\Q...\E`, useful when you need to feed a `q//` literal into a regex as a fixed string
- `$"` - list separator used when arrays interpolate inside `qq//`; irrelevant to `q//` itself but the usual reason someone reaches for `q//` is to **avoid** this machinery

SCALARs and strings

qq//

Build a double-quoted, interpolated string with a delimiter you pick.

`qq//` is the long form of the `"..."` string literal. It produces the same string `""` would, so `qq/hello $name\n/` and `"hello $name\n"` are interchangeable. The point of qq is

the delimiter: when the content contains " (or backslash sequences that would interfere with double quotes), you pick a different delimiter and keep the string readable instead of threading \" through it.

Synopsis

```
qq/STRING/
qq{STRING}
qq(STRING)
qq[STRING]
qq<STRING>
qqXSTRINGX        # any non-word, non-whitespace char works as␣
 ↪delimiter
```

What you get back

A scalar string. Variables referenced inside are interpolated at the point qq// is evaluated (not at parse time); backslash escape sequences are expanded. The result has no implicit newline - you have to write \n yourself.

Choosing a delimiter

Any character that is not a word character (/\w/) and not whitespace may follow qq. The four bracket pairs nest:

```
qq(foo (bar) baz)          # "foo (bar) baz" - inner () balanced
qq{ if ($x) { 1 } else { 0 } }
qq[a[b[c]d]e]
qq<<a<b>c>>
```

Non-bracketing delimiters use the same character at both ends and do not nest:

```
qq|one|two|                # SYNTAX ERROR - first | closes
qq/a\/b/                   # "a/b" - backslash escapes the delimiter
```

If the delimiter is a word character, whitespace between qq and the delimiter is required:

```
qq XfooX                   # OK - "foo"
qqXfooX                    # WRONG - parsed as identifier qqXfooX
```

as delimiter needs no space and is occasionally handy for strings containing every other punctuation mark:

```
qq#He said "hi" & waved.#  # "He said \"hi\" & waved."
```

The extra_paired_delimiters feature (5.36+) recognises a wide set of Unicode bracket pairs as nesting delimiters; enable it with use feature 'extra_paired_delimiters' when you want, say, guillemets or CJK brackets.

What gets interpolated

- **Scalars**: `$x`, `$hash{key}`, `$array[0]`, `$ref->{k}[2]`.
- **Arrays**: `@arr` expands in order, elements joined by `$"` (default `" "`). Equivalent to `join $", @arr`.
- **Slices**: `@h{qw(a b)}`, `@a[1..3]`.
- **Dereferences with arrows and subscripts**: `$ref->{key}[0]`.

What is **not** interpolated:

- **Method calls**: `$obj->meth` inside `qq//` interpolates `$obj` and leaves the literal text `->meth` in the string.
- **Function calls**: `@{[ func() ]}` is the usual workaround - array reference dereference runs arbitrary code and splices its result in.
- **Punctuation arrays** other than `@_`, `@+`, `@-`: only interpolate when braced (`@{*}`). Bare `@*` is a literal `@*`.

```
my $obj = Some::Class->new;
qq/$obj->name/;                          # interpolates $obj, leaves "-
↪>name"
qq/@{[ $obj->name ]}/;                   # calls the method,␣
↪interpolates result
```

Escape sequences

Available inside `qq//` (and anywhere else interpolation happens):

Sequence	Meaning
`\t`	tab (HT)
`\n`	newline (platform-virtual line terminator)
`\r`	carriage return (CR)
`\f`	form feed (FF)
`\b`	backspace (BS)
`\a`	alarm / bell (BEL)
`\e`	escape (ESC)
`\0`	NUL
`\\`	literal backslash
`\"`	literal double quote
`\x1b`	hex char, 0x00–0xFF
`\x{263A}`	hex char, any codepoint (braced form)
`\o{23072}`	octal char, any codepoint
`\033`	octal char, 000–777 (prefer `\o{}`)
`\cA`	control char - 64 xor'd with uppercased next char
`\N{GREEK SMALL LETTER ALPHA}`	named Unicode char (needs `use charnames`)
`\N{U+03B1}`	Unicode char by codepoint (always Unicode, even on EBCDIC)

Case and quoting modifiers apply until `\E` or end of string:

Sequence	Meaning
`\l`	lowercase the next character only
`\u`	titlecase (not uppercase!) the next character only
`\L`	lowercase until `\E`
`\U`	uppercase until `\E`
`\F`	foldcase until `\E`
`\Q`	quote regex metacharacters until `\E` (calls `quotemeta`)
`\E`	end the most recent `\L` / `\U` / `\F` / `\Q`

`\L`, `\U`, `\F`, `\Q` stack - each needs its own `\E`.

```
say "\u$name";                          # capitalise first letter
say "\U$name\E's account";              # uppercase $name, rest␣
 ↪untouched
say "match $file literally: \Q$file\E";
```

`\Q` inside `qq//` is applied **after** interpolation, so `qq/\Q$pat\E/` equals `quotemeta($pat)`. (In regex quote operators `qr//` and `m//`, `\Q` is applied after interpolation but before regex compilation - the difference matters when `$pat` itself contains regex metacharacters you want to keep literal.)

Perl has no `\v` escape in string literals - it is a regex metacharacter. Use `\x0b`, `\cK`, or `\N{VT}` for a vertical tab.

Global state it touches

- `$"` - list separator used when interpolating arrays and slices. Default `" "`. Change with `local $" = ","` for one scope.

No other interpreter globals participate in `qq//` evaluation itself. Interpolated variables are of course whatever their current values are at the point `qq//` runs.

Examples

Plain interpolation, identical in meaning to the `"..."` form:

```
my $name = "world";
my $s = qq/hello, $name!\n/;            # "hello, world!\n"
```

String that contains double quotes - qq with a different delimiter avoids backslash litter:

```
my $msg = qq{He said "yes" and then "maybe".};
# vs:   "He said \"yes\" and then \"maybe\"."
```

Multi-line HTML snippet with nested braces - the `()` pair nests:

```
my $user = "rj";
my $html = qq(
```

(continues on next page)

(continued from previous page)

```
  <p>Welcome, $user.</p>
  <p>You have @{[ scalar @inbox ]} messages.</p>
);
```

Array interpolation with a custom separator:

```
my @tags = qw(perl rust docs);
local $" = ", ";
say qq/tags: @tags/;                    # "tags: perl, rust, docs"
```

Method-call trap - `->` is literal inside `qq//`:

```
my $obj = File::Spec->new;
qq/path is $obj->rel2abs('.')/;         # "path is <stringified $obj>-
↪>rel2abs('.')"
qq/path is @{[ $obj->rel2abs('.') ]}/;
                                        # actually calls rel2abs()
```

Building an SQL fragment where `$ident` is untrusted - `\Q..\E` keeps embedded `%`, `_`, and backslashes literal under a `LIKE` clause only after you have handled proper parameterisation; for actual SQL escaping use the driver's quoting, not `\Q`:

```
my $needle = "100% sure";
my $pat = qq/\Q$needle\E/;              # the regex-safe form
# $pat eq "100\\%\\ sure"
```

Stacked modifiers:

```
say qq/\Qfoo \ubar \Ubaz\E qux\E done/;
# "foo\ Bar\ BAZ qux done"
#  \Q quotes spaces/?; \u titlecases b; \U uppercases baz until \E;
#  outer \E ends \Q.
```

Edge cases

- **Parse-time delimiter choice.** Perl picks the closing delimiter the moment it reads the opening one. `qq/.../` expects a `/`; there is no way to switch mid-string. If you need both kinds of braces, pick a non-bracket delimiter.
- **Backslash at end of string with non-bracket delimiter.** `qq/a\/` is `a/` (backslash escapes the delimiter, string continues); to get a literal trailing backslash, change delimiter: `qq{a\\}`.
- **No nesting for non-bracket delimiters.** `qq|a|b|` is the string `a` followed by the identifier `b` followed by `|` - a syntax error in most contexts. Pick bracket delimiters when the content contains the delimiter char.
- **`#` as delimiter** is a special case only because of how comments interact: `qq#...#` is a string, but `qq #...#` (with whitespace) is the operator `qq` followed by a comment, with the string taken from the next line. This mirrors the behaviour of all `q`-family operators.

- **Empty string**. qq// is the empty string, same as "".
- **Interpolated undef**. Stringifies to the empty string and, under use warnings, triggers an uninitialized warning at the point of interpolation, not at the point $var was assigned undef.
- **$ or @ followed by non-identifier**. Stays literal: qq/cost: \$5/ and qq/cost: $5/ both produce cost: $5 (there is no variable named $5 in the Perl 5 identifier sense - $5 is a regex capture, which does exist, so the second form interpolates capture group 5, whatever it currently is). For a literal $ or @, escape it: \$, \@.
- **\N{...} without charnames**. Named characters require use charnames ':full' (or similar) to be loaded in scope; without it you get a compile-time error. \N{U+HEX} and the single-name forms in the :loose set work without it.
- **Locale-sensitive case mapping**. With use locale including LC_CTYPE, \l, \u, \L, \U use the current locale's case tables. Without it, Unicode case tables are used for codepoints above 0xFF, and ASCII rules below. \F uses Unicode foldcase regardless of locale (except under a UTF-8 locale, where it matches \L).
- **One level only**. The result of interpolation is not re-scanned. my $v = '$x'; qq/$v/ is the two-character string $x, not the value of $x.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- q// - single-quoted literal, no interpolation, same delimiter rules
- qw// - word-list literal, produces a list of strings split on whitespace
- qx// - interpolated string executed as a shell command, captures stdout
- qr// - interpolated regex, compiled once and reusable
- sprintf - reach for this when the work is formatting numbers or padding fields, not splicing variables into text
- quotemeta - the function \Q..\E calls
- join - what array interpolation is secretly doing, with $" as the separator
- $" - list separator used when an array interpolates into a double-quoted string
- *Quote and Quote-like Operators* - the full table of q, qq, qx, qw, qr, m, s, tr and their shared delimiter and escape rules

Regular expressions and pattern matching

qr//

Compile a pattern once and hand back a reusable regex object.

qr// quotes its STRING as a regular expression, interpolates it the same way m does, and returns a Regexp object you can store, pass around, and splice into other patterns. The

compiled pattern carries its modifiers with it, so every later use sees the flags the pattern was built with.

Synopsis

```
my $re = qr/STRING/;
my $re = qr/STRING/msixpodualn;
my $re = qr{STRING};
my $re = qr'STRING';            # single-quote delimiter: no
 ↪interpolation
```

What you get back

A blessed Regexp object. `ref` returns the string `"Regexp"`. Stringifying the object yields a normalised form of the pattern with its modifiers encoded in a `(?flags:...)` wrapper, which is exactly the form the regex engine embeds when the object is interpolated into another pattern:

```
my $re = qr/my.STRING/is;
print $re;                      # (?^si:my.STRING)
```

The object is opaque - do not dereference it. The stringified form is intended for interpolation and debugging, not for manual parsing.

Global state it touches

- `$1`, `$2`, ..., `$+`, `$&`, `$`` `, `$'`, %+, %- - **not** set by `qr//` itself. Compilation does not match. These are populated only when the compiled pattern is later used by `m`, `s`, or `split`.
- `$@` - set if the pattern contains a syntax error and the error is trapped with `eval`. An untrapped compile error raises a fatal `Bareword "..." not allowed`-style exception at the point of the `qr//`.
- Locale and Unicode pragmas in effect at the `qr//` site (`use locale`, `use utf8`) are baked into the compiled object. Changing the pragma later does not change the object.

Modifiers

All modifiers that apply to `m` also apply to `qr//`. They are captured in the returned object and propagate when the object is interpolated into another pattern.

Flag	Meaning
`m`	Multi-line: `^` and `$` match at internal newlines.
`s`	Single-line: `.` matches a newline.
`i`	Case-insensitive match.
`x`	Extended: whitespace and # comments ignored in the pattern. `xx` also ignores whitespace inside `[...]`.
`p`	Preserve-match (no-op on Perl 5.20+; `${^PREMATCH}`, `${^MATCH}`, `${^POSTMATCH}` are always available).
`o`	Compile once; interpolated variables are frozen at first use. Rarely needed - `qr//` already caches.
`a`	ASCII-restrict `\d`, `\s`, `\w`, and POSIX classes. `aa` further forbids ASCII-to-non-ASCII matches under `i`.
`l`	Use the current locale's rules.
`u`	Use Unicode rules.
`d`	Default dual rules (legacy; usually picked up automatically).
`n`	Non-capture: unnamed `(...)` groups do **not** fill `$1`, `$2`, ...

When a compiled pattern is embedded in a larger pattern, the character-set and `msixn` flags it was built with stay in effect for that span only. The `o` modifier is the one exception - it is not propagated.

Examples

Compile once, use many times. Looping over patterns built ahead of time avoids recompiling on every iteration:

```
my @rules = map qr/$_/i, qw(error warn fail panic);
for my $line (@lines) {
    for my $rx (@rules) {
        print $line if $line =~ $rx;
    }
}
```

Interpolate a compiled pattern inside another pattern. The outer pattern sees the inner pattern's flags via the `(?flags:...)` wrapper:

```
my $word = qr/\w+/;
my $csv  = qr/^$word(?:,$word)*$/;
"alpha,beta,gamma" =~ $csv;          # matches
```

Single-quote delimiter for a literal pattern with no interpolation - useful when the pattern contains `$` or `@` that must stay literal:

```
my $price = qr'\$\d+\.\d{2}';        # matches "$1.99", not a sigil
"total: \$9.95" =~ $price;           # matches
```

Pattern re-use across calls. Storing the compiled form in a closure amortises compilation across every call to the returned sub:

```
sub matcher_for {
    my $pat = shift;
    my $rx  = qr/\Q$pat\E/i;        # \Q...\E quotes regex
↪metacharacters
    return sub { $_[0] =~ $rx };
}

my $is_error = matcher_for("ERROR:");
$is_error->($line);
```

Capture inside an interpolated `qr//` still populates `$1` at match time, not at `qr//` time:

```
my $num = qr/(\d+)/;
"port 8080" =~ /:$num$/;
print $1;                           # 8080
```

Using `/n` to suppress capture in `$1`, `$2`, ... while keeping named captures available:

```
my $rx = qr/(foo)(?<kw>bar)/n;
"foobar" =~ $rx;
print $+{kw};                       # "bar"
print defined $1 ? "yes" : "no";    # "no"
```

Edge cases

- **No match at compile time**. `qr//` only *compiles*. To test a string, use the object with `m` or `=~`: `$string =~ $rx`.
- **Interpolated variables are captured at `qr//` time**, not at use time. `my $rx = qr/$pat/; $pat = "other";` leaves `$rx` holding the pattern built from the original `$pat`. To re-capture, rebuild the `qr//`.
- **Compile-time pattern errors are fatal**. `qr//` raises an exception at the `qr//` site. Wrap in `eval` when the pattern comes from user input and an error should be recoverable:

  ```
  my $rx = eval { qr/$user_input/ };
  die "bad pattern: $@" if $@;
  ```

- **`ref($rx)` returns `"Regexp"`** - not `"SCALAR"`, not `"CODE"`. Use this to detect a compiled pattern in a polymorphic argument slot: `ref($arg) eq 'Regexp'`.
- **Stringification is lossy for re-parsing**. The normalised form is legal as an interpolation target but is not guaranteed to be a human-readable copy of the original source. Do not re-`qr//` the stringified output.
- **Locale and Unicode bake-in**. A `qr//` built inside `use locale` behaves with locale rules forever, even when used outside that scope. Rebuild under the desired pragma if the behaviour needs to change.

- **`\Q...\E` for literal text**. To compile a pattern that matches a user-provided string verbatim, wrap it with `\Q...\E`: `qr/\Q$literal\E/`. Without `\Q`, metacharacters in the string are interpreted as regex syntax.
- **Empty pattern**. `qr//` compiles to the empty pattern, which is treated specially at match time: `$str =~ $empty` reuses the *last successful* pattern in that scope. This is a match-time behaviour of `m`, not a `qr//`-specific trait, but it is a common trap when passing a default `qr//` around.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `m` - match a compiled pattern or a literal pattern against a string; the primary consumer of a `qr//` object
- `s` - substitute using a compiled pattern; accepts a `qr//` in the search slot
- `split` - split on a compiled pattern; the `qr//` form is the most efficient
- *Regular expressions guide* - full regex syntax reference; read this for the meaning of the modifiers and escape sequences
- `ref` - returns `"Regexp"` for a `qr//` object; the standard type check
- `eval` - wrap `qr//` when the pattern source is untrusted and a compile error should be recoverable

Regular expressions and pattern matching

quotemeta

Return a copy of a string with every regex-significant character backslash-escaped, so the result can be interpolated into a pattern and match its own literal content.

`quotemeta` exists to bridge string data and regex syntax. When user input, filenames, or configuration values end up on the right-hand side of `=~`, any character that happens to be a regex metacharacter (`.`, `*`, `+`, `?`, `(`, `[`, `\`, `|`, ...) would otherwise fire its special behaviour. `quotemeta` preempts that by inserting a backslash before every ASCII non-word character, leaving letters, digits, and underscore untouched.

Synopsis

```
quotemeta EXPR
quotemeta
```

What you get back

A new string. Every ASCII character that does **not** match `/[A-Za-z_0-9]/` is preceded by a backslash; word characters pass through unchanged. The result is always safe to paste into a regex as a literal pattern fragment.

quotemeta is a pure value producer - it does not modify its argument.

```
my $safe = quotemeta 'a.b*c';     # 'a\.b\*c'
```

Global state it touches

Reads $_ when called with no argument. Within the scope of use locale, additional non-ASCII Latin-1 characters are quoted to protect against locales that treat punctuation like | as a word character.

Examples

Bare form defaults to $_, like most list built-ins:

```
for ('a+b', 'c.d') {
    print quotemeta, "\n";     # a\+b
}                              # c\.d
```

Interpolating user input into a substitution. Without quotemeta, .*? in $substring would be regex-active and match far more than intended:

```
my $sentence  = 'The quick brown fox jumped over the lazy dog';
my $substring = 'quick.*?fox';
my $quoted    = quotemeta $substring;
$sentence =~ s{$quoted}{big bad wolf};
# $sentence unchanged - the literal text 'quick.*?fox' is not␣
 ↪present
```

The \Q...\E escape in a double-quoted string is exactly quotemeta applied to the enclosed region. The two forms below are equivalent:

```
my $pat1 = "\Q$substring\E";
my $pat2 = quotemeta($substring);
```

Building a regex that matches a literal filename anywhere in a path:

```
my $name = 'my.config[prod].ini';
if ($path =~ /\Q$name\E\z/) {
    # matches exactly the filename, no metachar surprises
}
```

Escaping a list of terms for an alternation:

```
my @terms = ('C++', 'C#', '.NET');
my $alt   = join '|', map { quotemeta } @terms;
# 'C\+\+|C\#|\.NET'
```

Edge cases

- **No argument**: `quotemeta` with no expression quotes `$_`. Inside a loop like `for (@input) { push @safe, quotemeta }` this is the idiomatic form.
- **Already-safe strings are untouched**: pure word-character input (`[A-Za-z0-9_]+`) round-trips identically.
- **Literal backslashes inside `\Q...\E`** interact with double-quotish backslash interpolation before `\Q` sees the text. A sequence like `"\Q\t\E"` quotes a literal tab character, not a backslash followed by `t`. When you need actual backslashes in the quoted region, keep the data in a variable and use `quotemeta $var` rather than embedding it in a double-quoted literal.
- **No way to inject a literal `$` or `@` inside `\Q...\E`**: a protected `\$` becomes the four-character sequence `\\$` in the output, and an unprotected `$` starts scalar interpolation before `\Q` runs. Drop out of the quoted region to splice in sigils.
- **Unicode-aware quoting** (Perl 5.16+): on UTF-8 strings - and on byte strings under `use feature 'unicode_strings'` or `use v5.12` or greater - non-ASCII characters are quoted only when they carry the Unicode properties `Pattern_Syntax`, `Pattern_White_Space`, `White_Space`, `Default_Ignorable_Code_Point`, or `General_Category=Control`. Identifier-class characters (letters, marks, digits) are left alone. This is the stable contract: Perl promises that any future regex metacharacter will have `Pattern_Syntax` set, so strings safe today stay safe.
- **Legacy non-UTF-8 strings** outside `unicode_strings` scope have every upper-Latin-1 code point (`\x80`–`\xFF`) quoted, for backwards compatibility with pre-5.16 behaviour.
- **Locale quoting**: within `use locale`, all non-ASCII Latin-1 code points are quoted whether the string is UTF-8 or not. ASCII-range quoting is unaffected by locale.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `m` - the match operator; `quotemeta` exists to feed safe literals into its pattern
- `qr` - compiled regex objects; `qr/\Q$var\E/` is the usual way to pin a pattern around a literal string
- `s` - substitution; the most common site of `quotemeta`-style escaping on the left-hand pattern
- `\Q` *and* `\E` - the double-quotish escape pair that is just `quotemeta` with different syntax
- `$_` - default input when `quotemeta` is called without an argument

Lists

qw//

Build a list of barewords by splitting a delimited string on whitespace, without quoting or comma-separating each word.

`qw//` is a compile-time quote-like operator, not a runtime function. The parser sees `qw(foo bar baz)` and emits the constant list `("foo", "bar", "baz")` - no splitting happens at run time, and the list is assembled once regardless of how often the code is executed. Reach for it wherever you would otherwise type a list of short, simple string literals separated by commas: import lists, `@ISA`, `@EXPORT`, hash-key sets, test-name tables.

Synopsis

```
qw(foo bar baz)
qw[foo bar baz]
qw{foo bar baz}
qw<foo bar baz>
qw!foo bar baz!
qw/foo bar baz/
```

Any non-alphanumeric character works as a delimiter. The four bracket pairs `()`, `[]`, `{}`, `<>`) nest; all other delimiters do not.

What you get back

A list of strings in list context - one element per run of non-whitespace characters in the body, with leading and trailing whitespace discarded and runs of internal whitespace treated as a single separator.

In scalar context `qw//` evaluates to the **last** element of the list, which is almost never what the author intended. `use warnings` flags the common scalar-context mistakes (see *Edge cases*).

Examples

Import list for a module - the canonical use:

```
use POSIX qw(setlocale localeconv strftime);
```

`@EXPORT` / `@EXPORT_OK` / `@ISA` tables:

```
our @ISA      = qw(Exporter);
our @EXPORT   = qw(sum min max);
```

Build a small hash where every key maps to 1 (a set):

```
my %is_keyword = map { $_ => 1 } qw(if unless while until for);
if ($is_keyword{$word}) { ... }
```

Compare the three ways of writing the same list - commas-and-quotes, fat-comma auto-quoting, and `qw//`:

```
my @a = ("foo", "bar", "baz");                  # explicit strings
my @b = (foo => "1", bar => "2", baz => "3");   # fat comma only
my @c = qw(foo bar baz);                        # qw//
```

Array initialisation with many short tokens:

```
my @months = qw(
    Jan Feb Mar Apr May Jun
    Jul Aug Sep Oct Nov Dec
);
```

Positional argument list to a function:

```
run_tests(qw(smoke integration regression));
```

Edge cases

- **No commas inside the body.** A comma is taken as part of a word: `qw(foo, bar)` yields the two-element list `("foo,", "bar")`, not three elements. `use warnings` emits `Possible attempt to separate words with commas`.
- **No comment syntax.** `#` is a literal character inside the body. `qw(foo # comment\n bar)` contains the word `#` and the word `comment`. `use warnings` emits `Possible attempt to put comments in qw() list`.
- **No backslash escapes.** `\` is literal - you cannot escape a whitespace character to include it in a word, and you cannot `\`-escape the delimiter. `use warnings` emits a warning for any `\` in the body.
- **No interpolation.** `qw($x @y)` produces the two literal strings `"$x"` and `"@y"`. Use `("$x", @y)` (or `q()`/`qq()`) when you want a variable's value.
- **Whitespace is the only separator.** Tabs and newlines split words exactly like spaces, which is why multi-line `qw(...)` lists are idiomatic for long tables.
- **Words with embedded whitespace are impossible.** If you need `"hello world"` as a single element, `qw//` is the wrong tool - use `("hello world", ...)` with real quotes.
- **Empty body.** `qw()` is the empty list, equivalent to `()`.
- **Scalar context** yields the last element, not the count. Wrap in `scalar(() = qw(...))` if you genuinely want the count, or assign to an array first:

  ```
  my @w = qw(foo bar baz);
  print scalar @w;                  # 3
  ```

- **Parser ambiguity with `//`.** `qw//` uses `/` as a delimiter just like `m//` and `s///`. Inside the body, no regex metacharacters apply - it is a flat string split on whitespace. The visual similarity to a regex is a reading hazard, not a semantic one.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `q//` - single-quote literal string; no interpolation, unlike `qq//`
- `qq//` - double-quote literal string with full interpolation
- `qr//` - compile a regex literal; the `//` family member for pattern matching
- `split` - the runtime equivalent when your word list is not known until run time: `split(" ", $str)` behaves almost exactly like a dynamic `qw//`
- `join` - the inverse operation, gluing a list back into a single delimited string

Processes

qx//

Run a shell command and capture its standard output.

`qx//` (and its twin, the backtick form `` `STRING` ``) interpolates `STRING` like a double-quoted string, hands the result to the system shell, waits for the child to finish, and returns whatever the child wrote to `STDOUT`. Standard error is untouched - it still goes to the parent's `STDERR` unless you redirect it inside the command. The child's exit status lands in `$?`.

Synopsis

```
my $out   = `date`;
my $out   = qx/date/;
my @lines = `ls -1 /etc`;
my $raw   = qx'echo $HOME';     # single-quote form: no Perl␣
↪interpolation
```

What you get back

Context-sensitive:

- **Scalar context** - one string containing the entire captured `STDOUT`, including any embedded newlines. `undef` if the shell or command could not be started.
- **List context** - a list of lines, split on the current value of `$/` (the input record separator, newline by default). Empty list if the shell or command could not be started.

The exit status of the child is always placed in `$?`, whether or not the captured output is used. A child that writes nothing and exits cleanly yields `""` in scalar context, `()` in list context, and `0` in `$?` - distinguishing "started but produced no output" from "failed to start" requires checking `$?` against `-1` or `$!`.

Global state it touches

- `$?` - set to the child's wait status (`waitpid`-style) on every call. `-1` means the child could not be started, in which case `$!` holds the reason.
- `$!` - set when the shell or command failed to start.
- `$/` - governs how list-context output is split into lines.
- `$|` - affects whether buffered Perl output reaches the child's environment before the child runs; set it on output handles before `qx//` if ordering with prior `print` matters.
- `%ENV` - exported to the child as its environment.
- The current working directory, umask, and open file descriptors are inherited by the child.

Examples

Capture a single line:

```
my $today = `date`;
chomp $today;
print "today is $today\n";
```

Capture line-by-line in list context:

```
my @users = `who`;
for my $line (@users) {
    my ($name) = split /\s+/, $line;
    print "logged in: $name\n";
}
```

Interpolation happens before the shell sees the string - Perl variables expand first, shell variables expand in the child:

```
my $dir = "/etc";
my $out = `ls $dir`;            # Perl interpolates $dir
my $sh  = qx'echo $HOME';       # single-quote form: $HOME is shell
 ↪'s
```

Redirect `STDERR` into the captured stream:

```
my $both = `gcc -v hello.c 2>&1`;
```

Discard `STDERR`, keep only `STDOUT`:

```
my $out = `find / -name core 2>/dev/null`;
```

Check the exit status:

```
my $out = `make test`;
if ($? == -1) {
```

(continues on next page)

(continued from previous page)

```
    die "failed to execute make: $!";
}
elsif ($? & 127) {
    die sprintf "make died with signal %d", $? & 127;
}
elsif ($? >> 8) {
    die sprintf "make exited with status %d", $? >> 8;
}
```

Edge cases

- **No shell metacharacters**: if `STRING` contains no shell metacharacters, Perl bypasses the shell and `execs` the command directly - same optimisation as `system`. This is faster and avoids one `sh -c` level of quoting.
- **Single-quote delimiter**: `qx'...'` suppresses Perl's double-quote interpolation. The literal `$var` is passed to the shell, which does its own expansion. Useful when the variable names belong to the shell, not to your Perl program.
- **Alternate delimiters**: `qx//` accepts any balanced or non-alphanumeric delimiter - `qx{ls -l}`, `qx(ls -l)`, `qx[ls -l]`, `qx!ls -l!`. Pick one that doesn't collide with shell syntax in your command. `qXfooX` is a syntax error - the delimiter must not be alphanumeric.
- **Backticks don't capture STDERR**: nothing written to the child's `STDERR` ends up in the returned string. Redirect with `2>&1` inside the command if you want it, or run the command through `open` with a pipe for finer control.
- **STDIN is inherited**: the child reads from whatever the Perl process had on `STDIN` at the time of the call. To feed the child specific input, use `open` in pipe mode instead.
- **Exit status vs `$!`**: a non-zero `$?` means the child ran and exited with an error; `$!` is only meaningful when `$?` is `-1` (the shell itself could not be launched).
- **Memory cost in scalar context**: the whole captured output is buffered in memory before being returned. For commands that produce very large or unbounded output, open a pipe with `open` and read it incrementally.
- **Function form**: `qx//` can also be spelled `readpipe` when you want a function-call syntax - useful when the command string is built dynamically and you'd rather pass it as a scalar than splice it into a quote-like.
- **Output buffering**: Perl flushes its buffered output handles before forking the child, but portability varies. When preceding `qx//` with `print` to a pipe or file, set `$|` on that handle to be sure your output reaches the destination before the child starts.
- **Shell selection**: the shell used is `/bin/sh` on Unix. Commands that rely on bash-isms need an explicit `bash -c '...'` wrapper.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `system` - run a shell command without capturing output; reach for this when you only care about the exit status
- `exec` - replace the current process with a command; no capture, no return
- `readpipe` - the named-function form of `qx//`; takes the command as a regular string argument
- `open` - pipe-mode (`open my $fh, "-|", CMD`) for streaming a child's output line by line instead of buffering the whole capture
- `$?` - the child wait status set by every `qx//` call; decode with `>> 8` for exit code and `& 127` for terminating signal
- `$/` - controls how list-context `qx//` splits output into lines

Numeric functions

rand

Return a pseudo-random floating-point number in the half-open range `[0, EXPR)`.

`rand` draws the next value from the interpreter's pseudo-random number stream and scales it to `0 <= n < EXPR`. The result is a regular Perl number (a double), never an integer on its own - wrap with `int` when you want a whole number. `EXPR` is optional and defaults to `1`, so a bare `rand` yields a float in `[0, 1)`.

Synopsis

```
rand EXPR
rand
```

What you get back

A double-precision float `n` with `0 <= n < EXPR`. The upper bound is **exclusive**: `rand(6)` never returns `6`, only values strictly below it. For an integer result, compose with `int`:

```
my $die = int(rand(6)) + 1;       # 1..6
```

The distribution is (approximately) uniform across the range. `rand` is a scalar operator - it returns one value per call, regardless of context.

Global state it touches

`rand` consumes the interpreter's internal PRNG state, shared with `srand`. Two rules follow from that:

- **Autoseeding**: the first call to `rand` in a program automatically calls `srand` if you haven't already. The seed is chosen from process-level entropy, so unseeded programs produce a different stream on every run.
- **Shared stream**: every subsequent `rand` call - anywhere in the program, in any module - draws from the same sequence. Calling `srand($seed)` with a fixed seed makes that sequence deterministic and reproducible, which is what you want for test fixtures and debugging. Calling `srand()` with no argument re-seeds from entropy.

There is no per-thread or per-scope isolation. If you need an independent stream for one piece of code, do not use `rand`; reach for a dedicated generator module.

Examples

Basic draw in `[0, 1)`:

```
my $u = rand();                    # e.g. 0.4173...
```

Dice roll - canonical `int(rand(N)) + 1` idiom:

```
my $roll = int(rand(6)) + 1;       # 1..6
```

Random element of an array:

```
my @colors = qw(red green blue yellow);
my $pick   = $colors[ rand @colors ];
```

`rand @colors` uses the array's length (scalar context forced by `rand`'s prototype), so this works for any array size.

Reproducible stream - fix the seed for tests:

```
srand(42);
my @sample = map { rand() } 1..5; # same five numbers every run
```

One-line shuffle idiom. Readable, but **biased** - `sort` compares pairs more than once, and a one-shot `rand()` per comparison does not produce a uniform permutation:

```
my @shuffled = sort { rand() <=> 0.5 } @list;   # biased; do not use
```

For a fair shuffle, use `List::Util::shuffle`:

```
use List::Util qw(shuffle);
my @shuffled = shuffle @list;
```

Simulate a weighted coin - note the exclusive upper bound means `rand() < 0.3` fires with probability exactly `0.3`:

```
my $heads = rand() < 0.3 ? "heads" : "tails";
```

Edge cases

- **EXPR is `0`**: treated as `1`, so `rand(0)` behaves like `rand(1)`. This is documented upstream behaviour and is preserved here, but it is a historical quirk - do not rely on it for new code.
- **EXPR is negative**: the result is undefined in practice - you may get `0`, you may get a negative float, and the behaviour is not portable across Perl versions. If you want a value in `[-N, 0)`, write `rand($n) - $n` explicitly.
- **EXPR is `undef`**: stringifies to `""`, numifies to `0`, and therefore behaves like `rand(0)` → `rand(1)`. Under `use warnings` you get an `uninitialized` warning.
- **EXPR is non-numeric**: the usual scalar-to-number coercion applies. `rand("3abc")` is `rand(3)` with a non-numeric warning.
- **Not cryptographically secure**: `rand` uses a fast PRNG suitable for simulation, shuffling, and test fixtures. It is **not** suitable for session tokens, password salts, key material, or any security-sensitive value. For those, read from `/dev/urandom` directly or use `Crypt::URandom` / `Crypt::PRNG`.
- **Thread/fork behaviour**: after `fork`, both parent and child inherit the same PRNG state and will produce the **same** stream unless one of them calls `srand()` to re-seed. A common footgun in pre-forking servers - seed in the child.
- **Large EXPR**: the generator produces 48 bits of randomness per draw; for `EXPR` close to or above `2**48`, the output is no longer finely distributed over the range. For cryptographic widths you shouldn't be using `rand` anyway (see above).

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `srand` - seed the PRNG explicitly for reproducibility, or re-seed in a forked child
- `int` - truncate a `rand` result toward zero to get a uniform integer in `[0, N)`
- `List::Util::shuffle` - fair Fisher-Yates shuffle; prefer this over the `sort { rand() <=> 0.5 }` idiom
- `Crypt::URandom` - cryptographically-secure random bytes from the OS entropy source; use this for tokens, salts, and key material
- `Crypt::PRNG` - cryptographically-secure PRNG with a rich API when you need more than raw bytes

I/O · Fixed-length data

read

Read a fixed amount of buffered input from a filehandle into a scalar.

`read` pulls up to `LENGTH` characters from `FILEHANDLE` and stores them in `SCALAR`, returning how many were actually read. It goes through the handle's PerlIO stack and is therefore buffered on top of the underlying OS read - contrast with `sysread`, which bypasses the buffer and calls `read(2)` directly. The optional `OFFSET` argument lets you splice the incoming data into the middle of `SCALAR` rather than overwriting it.

Synopsis

```
read FILEHANDLE, SCALAR, LENGTH
read FILEHANDLE, SCALAR, LENGTH, OFFSET
```

What you get back

- The **number of characters read**, which may be less than `LENGTH`.
- `0` at end of file.
- `undef` on error, with `$!` set.

`SCALAR` is grown or shrunk so that the last character actually read becomes the last character of the scalar - unless `OFFSET` is given, in which case only the slice at `OFFSET` is overwritten and anything beyond it is left alone (see *The OFFSET argument* below).

A short read is **not** an error. On a regular file it normally means you reached end of file; on a pipe, socket, or terminal it means no more data is available right now. Loop until you either have the bytes you need or `read` returns `0` / `undef`:

```
my $buf = "";
my $want = 4096;
while ($want > 0) {
    my $got = read($fh, $buf, $want, length $buf);
    die "read error: $!" unless defined $got;
    last if $got == 0;                  # EOF
    $want -= $got;
}
```

Global state it touches

- `$!` - set when `read` returns `undef`.
- `${^UTF8CACHE}` / the handle's PerlIO layers - determine whether `LENGTH` is counted in bytes or in characters (see *Character vs byte semantics* below).

`read` does not interact with `$_`, `$/`, `$\`, or `$,`. Unlike `readline`, it does not care about the input record separator.

The OFFSET argument

`OFFSET` controls **where in `SCALAR`** the incoming data lands. It does not seek the filehandle.

- **Omitted** - data replaces the entire contents of `SCALAR`.
- **Positive, within length** - data is written starting at position `OFFSET`. Characters before `OFFSET` are preserved; characters from `OFFSET` to the end of `SCALAR` are overwritten or extended.
- **Positive, beyond length** - `SCALAR` is first padded with `"\0"` bytes out to `OFFSET`, then the read is appended. Useful for reading into a fixed slot inside a larger buffer you are assembling.
- **Negative** - counts backwards from the end of `SCALAR`. `-1` means "overwrite the last character and append from there."

```
my $buf = "HEADER";
read($fh, $buf, 16, length $buf);      # append 16 chars after
 ↪"HEADER"

my $slab = "";
read($fh, $slab, 512, 1024);           # pad to 1024 "\0" bytes,␣
 ↪then
                                       # read 512 chars - $slab is
                                       # now 1536 chars long
```

Character vs byte semantics

`LENGTH` is measured in whatever unit the handle deals in:

- **Byte-mode handle** (the default, and every handle opened without an encoding layer): `LENGTH` is a byte count. `read($fh, $buf, 10)` pulls 10 bytes and `length $buf` is 10.
- **`:utf8` layer**: `LENGTH` is a **character** count. Perl decodes UTF-8 on the way in, and `$buf` holds decoded codepoints. The number of bytes consumed from the file can be anywhere from `LENGTH` to `4 * LENGTH`, depending on the text.
- **`:encoding(...)` layer**: same rule as `:utf8`, for any encoding the layer knows.

```
open my $fh, "<:utf8", "greek.txt" or die $!;
read($fh, my $buf, 5);                 # 5 characters, not 5 bytes
```

Mixing a byte-mode read with UTF-8 data produces mojibake and, under `use warnings`, a `Malformed UTF-8` warning if you later decode the result. Pick the layer at `open` time and stick with it.

Buffered vs unbuffered I/O

`read` is **stdio-buffered** through PerlIO - internally it calls `fread(3)` (or PerlIO's replacement) against the handle's buffer. That has two consequences worth remembering:

- You can mix `read`, `readline` / `<$fh>`, `getc`, and `seek` freely on the same handle. They all see the same buffer.
- You must **not** mix `read` with `sysread` on the same handle. `sysread` bypasses the buffer and goes straight to `read(2)`; any bytes already pulled into the buffer by a previous `read` become invisible to `sysread`, and vice versa. If you need raw syscall semantics, use `sysread` exclusively on that handle.

For byte-accurate, non-buffered input - for example on a non-blocking socket, or when implementing a protocol where a short read is meaningful rather than "try again" - reach for `sysread`.

Examples

Read a fixed-size header from a binary file:

```
open my $fh, "<", "packet.bin" or die "open: $!";
binmode $fh;
my $header;
my $n = read($fh, $header, 16);
die "short header: got $n bytes" unless $n == 16;
```

Append 16 bytes to the end of an existing buffer by using `OFFSET` equal to the current length:

```
my $buf = "PRELUDE:";
read($fh, $buf, 16, length $buf);      # $buf is now "PRELUDE:" . 16
 ↪new bytes
```

Read into position 1024 of a scalar, padding the gap with `"\0"`:

```
my $slot = "";
read($fh, $slot, 64, 1024);            # length($slot) == 1088
                                       # substr($slot, 0, 1024) is
 ↪"\0" x 1024
```

Loop until you have exactly `N` bytes or hit EOF - the correct pattern for pipes and sockets where a single `read` often returns fewer bytes than requested:

```
sub read_exact {
    my ($fh, $n) = @_;
    my $buf = "";
    while (length($buf) < $n) {
        my $got = read($fh, $buf, $n - length($buf), length $buf);
        return undef unless defined $got;
        return $buf if $got == 0;       # EOF; caller inspects
 ↪length
        # loop
    }
    return $buf;
}
```

Character-counted read through a UTF-8 layer:

```
open my $fh, "<:encoding(UTF-8)", "notes.txt" or die $!;
read($fh, my $chunk, 100);              # 100 characters
printf "chars=%d bytes=%d\n", length $chunk, do {
    use bytes; length $chunk;
};
```

Edge cases

- **Closed filehandle**: returns `undef` and sets `$!` to `"Bad file descriptor"`. Under `use warnings` a `read() on closed filehandle` warning is emitted.
- **Unopened filehandle**: same as closed - `undef` and `$!` set.
- **`LENGTH` of `0`**: `read` returns `0` immediately and does not touch `SCALAR`. It is **not** a reliable EOF probe; use `eof` for that.
- **Negative `LENGTH`**: a fatal runtime error (`Negative length at ...`). Validate `LENGTH` before calling.
- **Negative `OFFSET` whose magnitude exceeds the current length of `SCALAR`**: a fatal runtime error (`Offset outside string`). Clamp with `max($offset, -length $buf)` when the offset is computed.
- **Short read on a pipe or socket**: not an error. `read` returns fewer characters than requested whenever the PerlIO buffer empties before `LENGTH` is reached. Loop if you need the full count.
- **EOF mid-read**: returns the partial count. The next call returns `0`. After that, `$fh` stays at EOF until you `seek` or `clearerr`.
- **Reading from a tied handle**: `read` dispatches to the tie class's `READ` method, which is responsible for honouring `LENGTH` and `OFFSET`. Misbehaving tie classes can violate the "grow `SCALAR` so the last character read is the last character" contract.
- **Interaction with `sysread`**: do not mix them on one handle. `read` fills the PerlIO buffer in chunks of its own choosing; `sysread` ignores that buffer entirely.
- **Binary data on a text-mode handle**: on Unix-like systems there is no distinct text mode, but an encoding layer still transforms bytes. `binmode $fh` (or `open ..., "<:raw", ...`) before reading binary data.
- **`FILEHANDLE` as an expression**: a bareword or simple scalar is fine. Anything more complex must be parenthesised: `read(($handles[$i]), $buf, $len)`.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `open` - acquires the filehandle and decides whether subsequent `reads` are byte-counted or character-counted via the I/O layer
- `sysread` - the unbuffered counterpart, a direct `read(2)` system call; use it for non-blocking I/O or when a short read is meaningful

- `readline` / `<$fh>` - record-oriented input that respects `$/` instead of a byte/character count
- `getc` - read a single character; roughly `read($fh, $c, 1)` but with different EOF/undef reporting
- `binmode` - remove or add I/O layers so that `LENGTH` is unambiguously a byte count or a character count
- `eof` - the right way to test for end of file, rather than reading a zero-length chunk

I/O

readdir

Read the next entry, or all remaining entries, from a directory handle opened by `opendir`.

`readdir` is the directory counterpart of `readline`: it pulls names out of a directory handle one at a time in scalar context, or all at once in list context. The handle must already have been opened with `opendir`; iteration advances a position that `rewinddir`, `seekdir`, and `telldir` can manipulate.

Synopsis

```
readdir DIRHANDLE
my $name  = readdir $dh;
my @names = readdir $dh;
while (readdir $dh) { ... }     # sets $_
```

What you get back

Context-dependent:

- **Scalar context**: the next entry's name as a string, or `undef` when the directory is exhausted.
- **List context**: every remaining entry as a list of strings, or the empty list when the directory is already exhausted.

Entries are **bare filenames**, not paths. `readdir` returns `"foo.txt"`, never `"/some/dir/foo.txt"`. If you plan to hand the result to a file test, `open`, `stat`, or any other call that resolves paths against the current working directory, prepend the directory yourself - `readdir` does **not** `chdir` into the directory it is reading.

The list always includes the entries `.` (current directory) and `..` (parent directory) as reported by the underlying filesystem. Filter them out explicitly if you do not want them:

```
my @real = grep { $_ ne '.' && $_ ne '..' } readdir $dh;
```

Order is filesystem-dependent, not alphabetical. ext4, XFS, tmpfs, NFS, and FAT each return entries in their own internal order, which may be insertion order, hash order, or something else entirely. Never assume sorted output. If you need a predictable order, sort the result:

```
my @names = sort readdir $dh;
```

Examples

Read every name into a list, drop the dot entries, and sort:

```
opendir(my $dh, $some_dir) or die "opendir $some_dir: $!";
my @names = sort grep { !/^\.\.?\z/ } readdir $dh;
closedir $dh;
```

Iterate one entry at a time. A bare `readdir` in a `while` condition assigns to `$_` and tests the result for **definedness**, not truth - so an entry literally named `"0"` still keeps the loop running:

```
opendir(my $dh, $some_dir) or die "opendir $some_dir: $!";
while (readdir $dh) {
    print "$some_dir/$_\n";
}
closedir $dh;
```

Filetest with the directory prepended - forgetting the prefix is the classic `readdir` bug:

```
opendir(my $dh, $some_dir) or die "opendir $some_dir: $!";
my @dotfiles = grep { /^\./ && -f "$some_dir/$_" } readdir $dh;
closedir $dh;
```

Scalar context - one name per call, `undef` signals end of directory:

```
opendir(my $dh, $some_dir) or die "opendir $some_dir: $!";
while (defined(my $name = readdir $dh)) {
    next if $name eq '.' || $name eq '..';
    process("$some_dir/$name");
}
closedir $dh;
```

Rewind and read a second time - the handle stays open and usable:

```
my @first  = readdir $dh;
rewinddir $dh;
my @second = readdir $dh;   # same entries, possibly different order
```

Edge cases

- **Exhausted handle**: further calls return `undef` in scalar context, empty list in list context. Use `rewinddir` to start over.
- **Closed or invalid handle**: returns `undef` (scalar) or empty list (list) and sets `$!`. Under `use warnings` Perl emits `readdir() attempted on invalid dirhandle`.

- **Definedness test in `while`**: both `while (readdir $dh)` and `while (my $n = readdir $dh)` (and their `for` equivalents) are special-cased to test `defined`, not truth. An entry named `"0"` is returned and the loop continues.
- **`.` and `..` are always present** on POSIX filesystems, even in an otherwise empty directory. Filter them if you want only real content.
- **No path prefix**: entries are bare names. Any filetest or `open` on the raw return value tests relative to the current working directory, not `$some_dir`.
- **No sorting**: the order is whatever the kernel returns. Apply `sort` explicitly if stability matters.
- **Concurrent directory changes**: entries added or removed during iteration may or may not appear, per the platform. The only safe assumption is that entries present at `opendir` time and not removed will be reported.
- **Encoding**: `readdir` returns byte strings as the kernel stores them. On a filesystem using UTF-8 filenames, apply `Encode::decode_utf8` or configure the open pragma if you need character strings.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `opendir` - opens the directory handle that `readdir` reads from; always the first call in the sequence
- `closedir` - releases the handle once iteration is done
- `rewinddir` - resets the iteration position to the start of the directory
- `seekdir` - jumps to a previously recorded position
- `telldir` - records the current position for later `seekdir`
- `glob` - shell-style pattern matching over directory contents; higher-level when you want `*.txt` rather than every entry

I/O

readline

Read one or more records from a filehandle.

`readline` is the input primitive behind the `<FH>` diamond operator. It reads from the filehandle named by `EXPR` (a typeglob, a glob reference, or an `IO::Handle`), using the current value of `$/` to decide where one record ends. If `EXPR` is omitted, `*ARGV` is read. In scalar context it returns the next record; in list context it returns every remaining record. End-of-file is signalled by `undef` (scalar) or an empty list.

Synopsis

```
my $line  = readline $fh;       # one record, scalar context
my @all   = readline $fh;       # every remaining record
my $line  = <$fh>;              # same as scalar readline $fh
my @all   = <$fh>;              # same as list  readline $fh
readline;                       # reads from *ARGV
```

What you get back

- **Scalar context**: the next record as a string, or `undef` at end-of-file or on error. The record includes its terminator (the current `$/`) unless the last record in the file is missing one. After the returned value, `$.` is incremented - see the section on global state below.
- **List context**: a list of every remaining record until end-of-file. An empty list means the handle was already at EOF.
- **Slurp mode on an empty file**: when `$/` is `undef` and context is scalar, the first call on a zero-byte handle returns `''` (empty string, defined); the next call returns `undef`. This lets `defined($slurp = readline $fh)` distinguish an empty file from a missing one without a separate `-s` test.

Global state it touches

- `$/` - input record separator. Controls what counts as "one record". Default is `"\n"`. Special values are covered under *Record modes* below. Also known as `$INPUT_RECORD_SEPARATOR` or `$RS` under `use English`.
- `$.` - current input line number of the last handle read. `readline` increments `$.` once per record returned. `$.` is reset when the handle is explicitly `close`d, but **not** when `*ARGV` rolls from one file to the next - that continues counting across the whole input. Also `$INPUT_LINE_NUMBER` or `$NR` under `use English`.
- `$_` - the default scalar. When a `readline` call (or its `<FH>` equivalent) is used **as the condition of a `while` or `for` loop**, Perl implicitly assigns the result to `$_` and tests `defined`, not truth. `while (<$fh>) { ... }` is therefore safe on lines containing `"0"` or `""`.
- `$!` - errno. On a read error, `readline` returns `undef` (scalar) or a truncated list, and sets `$!` to the operating-system error.
- `@ARGV` - when `EXPR` is omitted (or the handle is `ARGV`), each end-of-file causes Perl to `open` the next filename from `@ARGV` and continue reading. An empty `@ARGV` makes `ARGV` read `STDIN`.

Record modes

`$/` selects one of four reading modes:

- **Line mode** (default) - `$/` is a non-empty string. A record ends at the next occurrence of that string. With the default `"\n"`, one call returns one line including its newline.

- **Paragraph mode** - `$/ = ""`. A record ends at the next run of two or more consecutive newlines. Leading blank lines before the first paragraph are skipped; the terminating blank line(s) are included in the returned string. Use `local $/ = ""` to confine the change to a block.
- **Slurp mode** - `$/ = undef`. The first call returns the entire remaining contents of the handle as one string; the next call returns `undef`. On an empty file, see the special case under *What you get back*.
- **Fixed-record mode** - `$/ = \N` (a reference to a positive integer) or `$/ = \"N"`. Each call reads exactly `N` bytes, or fewer at end-of-file. Useful for binary framing. The result is still counted as one record for `$.`.

Changes to `$/` take effect on the next `readline` call. Scope them with `local` so a subroutine does not silently change the caller's idea of "line".

Examples

Idiomatic line-by-line read. The `while` condition implicitly assigns to `$_` and tests `defined`:

```
open my $fh, "<", "input.txt" or die "open: $!";
while (<$fh>) {
    chomp;
    print "line $.: $_\n";
}
close $fh;
```

Explicit `readline` with error detection. `eof` must be checked separately from the return value, because a legitimate line can be the empty string:

```
while ( ! eof($fh) ) {
    defined( my $line = readline $fh )
        or die "readline failed: $!";
    # ... process $line ...
}
```

Slurp a whole file into a scalar:

```
my $contents = do {
    local $/;                    # undef - slurp mode
    open my $fh, "<", $path or die "open $path: $!";
    readline $fh;
};
```

Paragraph mode - read an RFC-822-style message header block:

```
open my $fh, "<", "message.eml" or die $!;
local $/ = "";
my $headers = <$fh>;             # up to the first blank line
my $body    = do { local $/; <$fh> };
```

Fixed-record mode - read 512-byte blocks from a device file:

```
open my $fh, "<:raw", "/dev/sda" or die $!;
local $/ = \512;
while (defined(my $block = <$fh>)) {
    last if length($block) < 512;   # short read at EOF
    # ... process $block ...
}
```

List context - pull every remaining line at once. Cheap for small files, ruinous for large ones:

```
open my $fh, "<", "hosts" or die $!;
my @lines = <$fh>;              # every line, including terminators
chomp @lines;
```

Reading from *ARGV with no explicit handle - the classic Perl filter idiom. @ARGV is consumed as a list of filenames, or STDIN is read if @ARGV is empty:

```
while (defined(my $line = readline)) {
    print $line;                # cat(1) in one line
}
```

Edge cases

- **EOF vs. error**: scalar `readline` returns `undef` for both. Call `eof($fh)` or check `$!` to distinguish them. Inside `while (<$fh>)` the loop exits on either; add an explicit `defined` plus `$!` check when you need to tell them apart.
- **A file with no trailing newline**: the last record is returned without a terminator in line mode. `chomp` is a no-op on such a value, which is the correct behaviour.
- **Paragraph mode on a file of only blank lines**: returns `undef` immediately - there is no non-empty paragraph to yield.
- **Slurp mode, empty file**: first call returns `''` (defined, zero-length); the second returns `undef`. Do not branch on truthiness of the first read.
- **Fixed-record mode with a short read at EOF**: returns the remaining bytes (possibly fewer than N), then `undef` on the next call. Check `length` if you need exactly N.
- **The ARGV magic handle**: when reading from *ARGV, `eof()` with no argument returns true only at the very end of **all** @ARGV files, while `eof(ARGV)` is true at the end of **each** file. You cannot use the `defined(readline) or die "...: $!"` idiom against *ARGV to distinguish EOF from error - open each filename yourself if you need that.
- **Already-closed handle**: `readline` returns `undef` and sets `$!` to `"Bad file descriptor"`. Under `use warnings` a `readline() on closed filehandle` warning is emitted.
- **Encoding layers**: `readline` returns whatever the handle's PerlIO stack produces. A `:utf8` or `:encoding(...)` layer decodes bytes into characters; a `:raw` handle returns bytes. Mixing a multi-byte `$/` with a byte handle can split characters.

- **Mid-record `$/` change**: only the next `readline` call observes the new separator. Any already-buffered data is re-scanned against it.
- **`<FH>` vs. `<$fh>` vs. `<>`**: these are all `readline` under the hood. `<FH>` reads from the typeglob `FH`; `<$fh>` reads from the handle in `$fh`; `<>` (or `<<>>` for safer filename handling) reads from `*ARGV`. The `<pattern>` form with any other content is `glob`, not `readline` - a different function entirely.
- **Loop condition assignment**: `while (my $line = <$fh>) { ... }` tests `defined`, not truth, only because the parser rewrites it. `while (($line = <$fh>)) { ... }` with the extra parentheses does **not** get the rewrite, and will loop forever on input containing `"0\n"`.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `open` - obtain the filehandle that `readline` consumes
- `read` - fixed-length byte read that ignores `$/`; use when you want a byte count, not a record
- `getc` - single-character read, likewise ignoring `$/`
- `eof` - test whether the next `readline` would return `undef`; behaves specially for `*ARGV`
- `$/` - input record separator; controls line, paragraph, slurp, and fixed-record modes
- `$.` - current input line number, incremented by every successful `readline`

Filehandles, files, directories

readlink

Return the target path a symbolic link points to.

`readlink` asks the kernel what string is stored inside a symbolic link, without following it and without touching the file the link eventually resolves to. The argument is the path of the link itself; the return value is the raw target string as recorded when the link was created. If `EXPR` is omitted, `$_` is used.

Synopsis

```
readlink EXPR
readlink
```

What you get back

On success, the link target as a string - exactly the bytes stored in the symlink, without any interpretation. On failure, `undef`, with `$!` set to the system error. Typical error codes:

- `EINVAL` - the path exists but is not a symbolic link
- `ENOENT` - the path does not exist
- `EACCES` - a directory component is not searchable
- `ELOOP`, `ENAMETOOLONG`, `ENOTDIR` - the usual path-resolution failures on components leading up to the link

The returned string is **not** canonicalised and **not** made absolute. A link created with `ln -s ../lib/libfoo.so libfoo.so` returns the literal `"../lib/libfoo.so"`, relative to the directory that contains the link itself - not to the current working directory. Resolving the target to an absolute, real path is the caller's job; see `Cwd::abs_path`.

Global state it touches

- `$_` - read as the default argument when `EXPR` is omitted
- `$!` - set on failure to the C `errno` from the underlying `readlink(2)` call

Examples

Read a link target directly:

```
my $target = readlink "/usr/bin/vi";
print defined $target ? "points to $target\n" : "not a symlink: $!\n
 ↪";
```

Default argument form inside a loop over filenames:

```
for (@paths) {
    next unless -l;                     # skip anything that isn't a␣
 ↪symlink
    my $t = readlink;                   # reads $_
    print "$_ -> $t\n" if defined $t;
}
```

Guard against non-symlinks explicitly. `readlink` on a regular file returns `undef` and sets `$!` to `EINVAL`:

```
my $t = readlink $path;
if (!defined $t) {
    die "readlink $path: $!" unless $!{EINVAL};
    # not a symlink - handle as a plain file
}
```

Resolve a relative link target against the link's own directory, which is the correct base - not the current working directory:

```
use File::Basename qw(dirname);
use File::Spec;

my $target = readlink $link // die "readlink $link: $!";
my $resolved = File::Spec->rel2abs($target, dirname($link));
```

Canonicalise fully (follow every link in the chain, collapse `.` and `..`, produce an absolute path) with `Cwd::abs_path`:

```
use Cwd qw(abs_path);
my $real = abs_path($link);              # undef if any component is␣
↪missing
```

Edge cases

- **Not a symlink**: `readlink` on a regular file, directory, or anything else that isn't a symbolic link returns `undef` and sets `$!` to `EINVAL`. Test with `-l` first if you want to distinguish "not a link" from other errors.
- **Missing path**: a component of `EXPR` that doesn't exist gives `ENOENT`, same as any other path-lookup failure.
- **Default argument**: bare `readlink` uses `$_`. No warning is emitted when `$_` is `undef`; the call simply fails with a path-lookup error.
- **Trailing components**: `readlink` operates on the final component only - intermediate symlinks in the path are followed normally. `readlink "/a/b/c"` where `/a` is itself a symlink reads the link at `c`, not at `a`.
- **Relative target, unchanged**: the returned string is stored verbatim. Never assume the string is absolute, and never join it onto the current working directory - join onto the directory that contains the link, or feed the link path (not the target) to `Cwd::abs_path`.
- **Dangling links**: a link whose target does not exist still reads fine. `readlink` does not stat the target. Use `lstat` to inspect the link itself, `stat` to inspect what it points to (and fail if the target is missing).
- **Platforms without symlinks**: on systems that do not implement symbolic links, `readlink` raises an exception rather than returning `undef`. Linux (the only supported pperl target) always has them, so this is relevant only for portable code shared with other targets.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `symlink` - create a symbolic link; inverse operation
- `link` - create a hard link (not readable with `readlink`; hard links are indistinguishable from the original name)
- `lstat` - `stat` the symlink itself rather than its target; use together with `readlink` to inspect a link fully
- `-l` - file test for "is a symbolic link"; check before calling `readlink` to disambiguate `EINVAL` from other errors
- `$!` - errno set on failure; `$!{EINVAL}` distinguishes "not a symlink" from other errors

Processes

readpipe

Run a shell command and return its standard output.

`readpipe` hands `EXPR` to the shell, waits for the child to finish, and returns whatever the child wrote to its standard output. It is the named form of the `qx` (backticks) operator - `` `cmd` `` and `qx/cmd/` both compile down to `readpipe`. Reach for the named form when you want the command in a variable, when you want to suppress the double-quote interpolation `qx` performs, or when you want to read a captured command out of `@ARGV`. If `EXPR` is omitted, `$_` is used.

Synopsis

```
readpipe EXPR
readpipe
$out  = readpipe $cmd;
@rows = readpipe $cmd;
```

What you get back

Context decides the shape:

- **Scalar context** - one string containing the entire captured output, newlines and all.
- **List context** - a list of records, split on `$/` (the input record separator, newline by default). The separator stays attached to each element unless you `chomp` it off.

`$?` is set to the child's wait status after the command finishes, exactly as with `system`. A non-zero `$?` is the usual signal that the command failed - the return value alone cannot tell you, since a failing command may still have written output (or nothing) before exiting.

```
my $out = readpipe "uname -a";
die "uname failed: \$? = $?" if $?;
```

Global state it touches

- `$_` - used as the command when `EXPR` is omitted.
- `$/` - controls how list-context output is split into records. Set it to `undef` to slurp the whole output into a single element, or to `""` for paragraph mode.
- `$?` - set to the child's wait status on return.
- `$!` - set if the shell itself could not be launched.
- `%ENV` - the shell inherits the current environment.

Examples

Capture one line of output:

```
my $host = readpipe "hostname";
chomp $host;
print "running on $host\n";
```

Command held in a variable - the reason `readpipe` exists as a named function. Backticks would still work here, but `readpipe` makes the intent explicit and avoids the visual noise of nested quoting:

```
my $cmd = "ls -1 /etc";
my @entries = readpipe $cmd;
chomp @entries;
print scalar(@entries), " entries under /etc\n";
```

Slurp everything as one string by disabling record separation:

```
local $/;                                  # slurp mode
my $log = readpipe "dmesg";
print length($log), " bytes of kernel log\n";
```

Paragraph mode - each element is one blank-line-separated block:

```
local $/ = "";
my @paragraphs = readpipe "cat README";
print scalar(@paragraphs), " paragraphs\n";
```

Check the exit status properly. The return value is output; success lives in `$?`:

```
my @files = readpipe "find . -name '*.tmp' 2>/dev/null";
if ($? == 0) {
    chomp @files;
    unlink @files;
} else {
    warn "find exited with status ", ($? >> 8), "\n";
}
```

Use `$_` as the command. Rarely the clearest form, but legal:

```
$_ = "date";
my $when = readpipe;                       # same as readpipe $_
```

Edge cases

- **No interpolation protection.** `readpipe` interpolates its argument exactly like a double-quoted string - the same as `qx`. `readpipe "rm $path"` with `$path` coming from user input is a shell-injection hole. Quote the value, or better, switch to a list-form `system` / piped `open` and skip the shell entirely.
- **Standard error is not captured.** Only the child's standard output comes back. Redirect stderr in the command itself if you want it merged: `readpipe "cmd`

2>&1".

- **Shell is mandatory.** Even a single-word `readpipe "true"` goes through `/bin/sh`, which means shell quoting, globbing, and environment-variable expansion all apply. No list form exists.
- **Large output buffers in memory.** The whole child output is read before `readpipe` returns. For a command that produces gigabytes, use a piped `open` and read incrementally instead.
- **Empty output, zero exit.** A successful command that writes nothing returns the empty string in scalar context and the empty list in list context. Distinguish "ran and said nothing" from "never ran" by checking `$?`.
- **Child died on signal.** `$?` encodes both exit code and signal - mask with `$? & 127` to get the signal number, `$? >> 8` for the exit code.
- **Windows quoting rules.** `readpipe` hands the string to the platform shell; on Windows that is `cmd.exe`, whose quoting and redirection rules differ from POSIX. pperl is Linux-only, so this is only relevant when porting scripts in.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `qx` - the operator form; `` `cmd` `` and `qx/cmd/` are shorthand for `readpipe` and carry identical semantics
- `system` - run a command without capturing output; use when you only care about exit status
- `exec` - replace the current process with the command; no return, no capture
- `open` - piped open (`open my $fh, "-|", @cmd`) streams the child's output line-by-line without buffering it all
- `$?` - child wait status set by every process-launch builtin
- `$/` - input record separator that splits list-context output

Sockets

recv

Read an incoming message from a socket into a scalar.

`recv` reads up to `LENGTH` bytes from `SOCKET` into `SCALAR`, replacing whatever `SCALAR` previously held. `SCALAR` is grown or shrunk to the number of bytes actually received. `FLAGS` is the same bit mask the underlying `recvfrom(2)` system call takes - `0` for ordinary reads; the common non-zero values are `MSG_PEEK` (look without consuming), `MSG_WAITALL` (block until `LENGTH` bytes arrive), and `MSG_DONTWAIT` (never block). For a connectionless socket `recv` returns the sender's packed address; for a connected socket it returns the empty string. Under the hood this is `recvfrom(2)`, not `recv(2)`.

Synopsis

```
recv SOCKET, SCALAR, LENGTH, FLAGS
```

What you get back

- On success from a connectionless socket (UDP, UNIX datagram): the sender's address as a packed sockaddr string. Unpack it with `Socket`'s `sockaddr_in`, `sockaddr_in6`, or `sockaddr_un` depending on the address family.
- On success from a connected socket (TCP, connected UDP): the empty string `""`. The empty string is **true** for the defined-vs-undef check but **false** in boolean context - test with `defined`, not plain truthiness.
- On error: `undef`, with `$!` set to the `errno` from the failing `recvfrom(2)` call. Typical errors: `EAGAIN` / `EWOULDBLOCK` on a non-blocking socket with no data ready, `ECONNRESET` on a TCP peer reset, `EBADF` on a closed handle.

`SCALAR` is always resized to the byte count read. A zero-byte read on a connected stream socket means the peer closed its write side cleanly - `recv` returns `""` and `SCALAR` becomes the empty string.

Global state it touches

- `$!` is set on error to the system `errno`.
- No other special variables are involved. `recv` does not honour `$/`, `$,`, or `$\` - those are line-oriented I/O concepts and `recv` is a raw datagram / byte-stream primitive.

Examples

Read one UDP datagram and identify the sender:

```
use Socket;
my $from = recv($sock, my $buf, 1500, 0)
    or die "recv: $!";
my ($port, $ip) = sockaddr_in($from);
print "got ", length($buf), " bytes from ",
      inet_ntoa($ip), ":$port\n";
```

Read from a connected TCP socket. The sender address is empty, so check with `defined` rather than boolean truth:

```
defined(recv($sock, my $buf, 4096, 0))
    or die "recv: $!";
# $buf now holds 0..4096 bytes; 0 means the peer closed.
```

Peek at the next datagram without consuming it. The second call returns the same bytes:

```
use Socket;
recv($sock, my $peek,    1500, MSG_PEEK) or die $!;
recv($sock, my $consume, 1500, 0)        or die $!;
# $peek and $consume hold the same datagram.
```

Non-blocking read loop. `EAGAIN` / `EWOULDBLOCK` is the normal "no data yet" signal, not a real error:

```
use Errno qw(EAGAIN EWOULDBLOCK);
while (1) {
    my $from = recv($sock, my $buf, 1500, 0);
    unless (defined $from) {
        last if $! == EAGAIN || $! == EWOULDBLOCK;
        die "recv: $!";
    }
    handle_datagram($from, $buf);
}
```

Force a full-length read on a stream socket that might fragment the message - `MSG_WAITALL` blocks until `LENGTH` bytes are in or the connection dies:

```
use Socket;
defined(recv($sock, my $frame, 64, MSG_WAITALL))
    or die "recv: $!";
# $frame is exactly 64 bytes, or the peer closed early.
```

Edge cases

- **UTF-8-layered socket**: if `SOCKET` has a `:utf8` layer (directly or via `:encoding(...)`), `recv` throws an exception. `recv` is byte-oriented and does not interact with PerlIO layers; use `binmode` to strip the layer before reading, or decode the resulting bytes yourself.
- **LENGTH vs datagram size**: for datagram sockets, if `LENGTH` is smaller than the incoming datagram, the excess is **discarded silently**. Size the buffer to the protocol's MTU (typically 1500 for UDP over Ethernet, 65507 for the IPv4 UDP maximum).
- **`""` return is not false-enough**: `recv` on a connected socket returns the empty string, which `!` treats as false. Code like `recv(...) or die $!` dies on every successful connected read. Use `defined`:

  ```
  defined(recv($sock, $buf, $len, 0)) or die "recv: $!";
  ```

- **Zero-byte result on a stream socket**: `recv` returns `""` and `SCALAR` becomes empty when the peer has performed an orderly shutdown. Distinguish end-of-stream from a real error by checking `length($buf) == 0` after a defined return.
- **Signals**: a blocking `recv` interrupted by a signal returns `undef` with `$!` == `EINTR`. Retry or install a proper `sigaction` with `SA_RESTART` if that's not what you want.
- **Closed socket**: `recv` on a closed or never-opened handle returns `undef`, sets `$!` to `EBADF`, and emits a `recv() on closed socket` warning under `use warnings`.
- **Pre-existing `SCALAR` contents are destroyed**. `recv` does not append; it overwrites. Use a fresh lexical or clear the scalar before each call.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `send` - the write side of the same socket; same `FLAGS` vocabulary, same connectionless-vs-connected distinction
- `socket` - create the `SOCKET` handle `recv` reads from
- `bind` - required before `recv` on a server-side datagram socket so the kernel knows which port to deliver to
- `connect` - when used on a datagram socket, makes `recv` return `""` like a stream socket and filters out datagrams from other peers
- `sysread` - byte-level read for stream sockets when you do not need the sender address; integrates with `select`-based loops
- `Socket` - `sockaddr_in`, `inet_ntoa`, `MSG_PEEK`, `MSG_WAITALL`, and the rest of the constants `recv` consumes

Control flow

redo

Restart the current iteration of a loop without re-testing the condition and without running the `continue` block.

`redo` jumps back to the top of the loop body and re-runs it on the *same* input - the conditional that guards the loop is not re-evaluated, `$_` is not advanced by `while (<FH>)`, the iterator of a `foreach` is not advanced, and any `continue { }` block attached to the loop is skipped. The current pass happens again from the beginning.

Synopsis

```
redo
redo LABEL
redo EXPR
```

What you get back

`redo` does not return a value. It is a flow-control operator: it transfers control and never falls through to the next statement. Do not try to use it as the value of an expression - in particular, it cannot be used to return a value from `eval { }`, `sub { }`, or `do { }`.

Which loop it targets

With no argument, `redo` refers to the innermost enclosing loop. With a `LABEL`, it restarts the loop tagged with that label, allowing you to reach out of nested loops:

```
OUTER: while (...) {
    while (...) {
        redo OUTER;      # restart the while (...) above
    }
}
```

`redo EXPR` (added in Perl 5.18) computes the label at runtime. The expression must evaluate to a string naming a label currently in scope:

```
my $target = "OUTER";
redo $target;
```

A bare block counts as a loop that runs once, so `redo` inside a block turns it into a looping construct:

```
{
    my $line = <STDIN>;
    redo if $line =~ /^\s*$/;   # skip blank lines, re-read
    print $line;
}
```

`redo` vs `next` vs `last`

All three take an optional `LABEL` and all three affect loop control, but they differ in what they skip:

- `last` - exit the loop entirely. `continue` block is **not** run.
- `next` - go to the next iteration. `continue` block **is** run, then the loop condition is re-tested.
- `redo` - re-run the *same* iteration. `continue` block is **not** run, and the loop condition is **not** re-tested.

The `continue`-block difference is the one that catches people out. If a loop uses `continue { }` to increment a counter or advance a cursor, `redo` will not advance it - that's the whole point, but it is also the whole foot-gun.

Examples

Re-prompt until the user gives valid input. The `continue` block is skipped, so `$attempts` only counts *completed* iterations, which is usually what you want:

```
my $attempts = 0;
while (1) {
    print "Enter a positive integer: ";
    my $n = <STDIN>;
    chomp $n;
    redo unless $n =~ /^[0-9]+$/ && $n > 0;
    print "Got $n after $attempts rejected tries.\n";
    last;
} continue {
```

(continues on next page)

(continued from previous page)

```
    $attempts++;
}
```

Restart processing of the current line after patching it - the classic "lie to yourself about what was just read" pattern, here stripping Pascal-style block comments that may span multiple input lines:

```
LINE: while (<STDIN>) {
    while (s|({.*}.*){.*}|$1 |) {}
    s|{.*}| |;
    if (s|{.*| |) {
        my $front = $_;
        while (<STDIN>) {
            if (/}/) {          # end of comment?
                s|^|$front\{|;
                redo LINE;      # re-process the patched line
            }
        }
    }
    print;
}
```

Target an outer loop by label. `redo INNER` would restart the inner loop on the current `$x`; `redo OUTER` restarts the outer loop on the current `$x` *without* advancing the `foreach` iterator:

```
OUTER: foreach my $x (@xs) {
    INNER: foreach my $y (@ys) {
        redo OUTER if should_restart($x, $y);
    }
}
```

Turn a bare block into a retry loop. Without `redo` this block runs exactly once; with `redo` it runs until the fetch succeeds:

```
my $tries = 0;
{
    my $ok = try_fetch();
    if (!$ok && ++$tries < 3) {
        sleep 1;
        redo;
    }
}
```

Edge cases

- **`continue` is not run.** This is the defining difference from `next`. If your loop relies on `continue { }` for counters, cursors, or cleanup, `redo` will bypass all of it. Either move the work into the body or use `next` instead.

- **Loop condition is not re-tested.** `while (cond) { ... redo; }` will keep looping even if cond would now be false. You are responsible for ensuring the `redo` path eventually exits via `last`, `return`, or a conditional fall-through.
- **Easy infinite loop.** `redo` with no state change is a tight infinite loop - typically not what you want. Always pair `redo` with a condition that eventually becomes false, or with a retry counter that triggers `last`.
- **`foreach` does not advance.** `foreach my $x (@xs) { redo }` keeps `$x` pinned at the current element and loops on it forever. The iterator only advances on fall-through or `next`.
- **`while (<FH>)` does not read the next line.** `$_` retains the current line across the `redo`, which is precisely why the Pascal comment stripper above works.
- **`redo` out of `grep` / `map` is not supported.** Upstream documents this as undefined; do not use `redo` to exit or restart a `grep BLOCK LIST` or `map BLOCK LIST`.
- **`redo` out of `eval`, `sub`, or `do` is not a return.** It performs flow control; there is no value to return. A `redo` that would have to cross such a boundary to reach a loop is a runtime error ("Can't `redo` outside a loop block").
- **Precedence quirk.** Unlike most named operators, `redo` has the precedence of assignment and is exempt from the looks-like-a-function rule. `redo ("foo")."bar"` passes the whole concatenation as the argument to `redo` - the parentheses do not isolate `"foo"`.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `last` - exit the loop entirely; `continue` block is not run, same as `redo`
- `next` - advance to the next iteration; `continue` block **is** run and the loop condition is re-tested
- `return` - return from the enclosing subroutine, not just the loop; unlike `redo`, it yields a value
- `continue` - the block that `redo` skips but `next` runs; the core of the three-way distinction

Classes and OO

ref

Return a string describing what a reference points to.

`ref` examines its operand and returns a string identifying the kind of referent. For a blessed reference it returns the class name; for an unblessed reference it returns one of the built-in type strings (`SCALAR`, `ARRAY`, `HASH`, `CODE`, `GLOB`, `LVALUE`, `IO`, `FORMAT`, `REF`, `VSTRING`, `REGEXP`). For anything that is not a reference it returns the empty string. With no argument, `ref` operates on `$_`.

Synopsis

```
ref EXPR
ref
```

What you get back

A string. Three mutually exclusive categories:

- **Blessed reference** - the name of the class the referent is blessed into. For example, a `bless {}, 'Dog'` reference returns `"Dog"`. Subclasses return their own name, not the parent's: a reference blessed into Puppy (an `ISA` of Dog) returns `"Puppy"`. See `bless`.
- **Unblessed reference** - one of the built-in type strings listed below, identifying the physical type of the referent.
- **Non-reference** - the empty string `""`.

The empty string is the **only** false value `ref` ever returns, so `if (ref $x)` is a safe "is this a reference?" test. See the edge cases section for why `if (ref $x eq 'HASH')` is *not* a safe "is this a hash reference?" test.

Built-in type strings

Returned for unblessed references according to the physical referent:

String	Referent
`SCALAR`	reference to an ordinary scalar
`ARRAY`	reference to an array
`HASH`	reference to a hash
`CODE`	reference to a subroutine
`GLOB`	reference to a typeglob
`LVALUE`	reference to an assignable expression
`IO`	reference to an I/O handle
`FORMAT`	reference to a format
`REF`	reference to another reference
`VSTRING`	reference to a v-string scalar
`REGEXP`	reference to a raw (unblessed) regexp

A compiled regex from `qr//` is *blessed* into the class `Regexp` at creation time, so `ref qr/.../` normally returns `"Regexp"`, not `"REGEXP"`. The bare `REGEXP` string appears only for unblessed regex referents, which are rare outside of internals.

Examples

Basic type probes:

```
ref \1              # "SCALAR"
ref []              # "ARRAY"
```

(continues on next page)

(continued from previous page)

```
ref {}               # "HASH"
ref sub { }          # "CODE"
ref \*STDOUT         # "GLOB"
ref \\1              # "REF"    (reference to a reference)
ref qr/foo/          # "Regexp" (blessed)
ref "plain string"   # ""       (not a reference)
ref 42               # ""
ref undef            # ""
```

Blessed references return the class name:

```
my $obj = bless {}, 'My::Widget';
ref $obj;                        # "My::Widget"
```

Guard before dereferencing:

```
sub stringify {
    my ($v) = @_;
    return "[undef]"           if !defined $v;
    return "[ref: " . ref($v) . "]" if ref $v;
    return $v;
}
```

No-argument form reads `$_`:

```
for my $item (@things) {
    print "skipping non-ref\n" and next unless ref;
    ...
}
```

Picking the right test

`ref` is the oldest introspection primitive in the language and it conflates three different questions. Modern code should pick the operator that matches the question being asked.

Is this a reference at all?

```
if (ref $x) { ... }             # empty string is the only false␣
↪return
```

Is this an object of (or derived from) a given class? Use the `isa` operator (or the `UNIVERSAL::isa` method) - it walks `@ISA` and honours subclassing, which a literal `ref` comparison does not:

```
if ($obj isa 'My::Widget') { ... }      # true for subclasses too
if ($obj->isa('My::Widget')) { ... }    # method form

if (ref($obj) eq 'My::Widget') { ... }  # FRAGILE: fails for␣
↪subclasses
```

What is the underlying physical type, regardless of blessing? Use `Scalar::Util::reftype` - it ignores the class and returns the built-in type string even for blessed references:

```
use Scalar::Util qw(reftype blessed);
reftype($obj);                  # "HASH" if $obj is blessed hash ref
blessed($obj);                  # class name, or undef if not␣
 ↪blessed
```

Edge cases

- **Never use `ref` as a truth value of the referent.** A reference blessed into a class literally named `0` makes `ref $obj` return the string `"0"`, which is false. This is rare by construction but the foot-gun is old and well known. Compare to the empty string instead: `if (ref $x ne "")`.
- **Class name clashes with built-in type strings.** Nothing stops you from writing `package HASH; sub new { bless {}, shift }`. Then `ref(HASH->new)` returns `"HASH"` - indistinguishable from an unblessed hash reference by `ref` alone. Use `Scalar::Util::blessed` to disambiguate, or `reftype` to ask the physical-type question directly.
- **`ref` on `qr//` returns `"Regexp"`, not `"REGEXP"`.** `qr//` objects are blessed at construction; the uppercase form is only produced for the (unusual) unblessed regex reference.
- **Tied variables**: `ref` sees through the tie to the underlying reference. Use `tied` to retrieve the object backing the tie.
- **Weak references** are still references: `ref` behaves identically for weak and strong refs. Use `Scalar::Util::isweak` to detect weakness.
- **Overloaded stringification does not affect `ref`.** The return value of `ref` is produced by the runtime, not by the object's `""` overload.
- **`ref EXPR` has very high precedence** - it is parsed as a named unary operator. Parenthesise when the argument is anything more complex than a simple variable:

```
ref $a || $b;           # parsed as (ref $a) || $b
ref($a || $b);          # what you probably meant
```

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `bless` - install a class name on a reference; the name installed here is exactly what `ref` returns afterwards
- `isa` - class-membership test that honours `@ISA` inheritance; prefer this over `ref(...) eq 'ClassName'` for OO dispatch

- `tied` - retrieve the object implementing a tied variable; `ref` on a tied scalar follows the tie to the referent
- `Scalar::Util::reftype` - physical-type string for a reference, ignoring blessing
- `Scalar::Util::blessed` - class name if blessed, `undef` otherwise; the unambiguous counterpart to `ref` for OO code
- `perlref`, `perlobj` - reference and object model reference manuals

Filehandles, files, directories

rename

Change the name of a file.

`rename` asks the operating system to rebind the path `OLDNAME` to `NEWNAME`. On a POSIX filesystem this is a single atomic directory operation: after the call returns successfully, `NEWNAME` refers to the file that used to be reachable as `OLDNAME`, and `OLDNAME` no longer exists. It is the same primitive the shell `mv` command reaches for when source and destination live on the same filesystem.

Synopsis

```
rename OLDNAME, NEWNAME
```

What you get back

`1` on success, `0` on failure with `$!` set to the `errno` the underlying `rename(2)` syscall returned. Always check the return value - a silent failure here leaves the filesystem in a state the rest of the program does not expect:

```
rename $tmp, $final
    or die "rename $tmp -> $final failed: $!";
```

Atomicity and the same-filesystem rule

`rename` is atomic **only when `OLDNAME` and `NEWNAME` resolve to the same filesystem**. Within that filesystem, no intermediate state is ever visible: another process reading the directory sees either the old name or the new name, never both and never neither. This is why the canonical "write a file safely" idiom writes to a temporary file in the same directory and then renames it into place:

```
open my $fh, ">", "$path.tmp" or die "open: $!";
print $fh $data;
close $fh or die "close: $!";
rename "$path.tmp", $path
    or die "rename: $!";
```

Across filesystems the call fails with `EXDEV` (`"Invalid cross-device link"`) and nothing is moved. The system `mv` command papers over this by falling back to copy-then-unlink; `rename` does not. For a portable cross-device move, use `move` from `File::Copy`, which performs the copy-and-unlink fallback itself.

Examples

Rename within a directory:

```
rename "draft.txt", "final.txt"
    or die "rename: $!";
```

Atomic file replacement - write to a sibling tempfile, then rename:

```
my $tmp = "$path.$$.tmp";
open my $fh, ">", $tmp or die "open $tmp: $!";
print $fh $content;
close  $fh             or die "close $tmp: $!";
rename $tmp, $path     or die "rename $tmp -> $path: $!";
```

Detect the cross-device case and fall back to `File::Copy::move`:

```
use Errno qw(EXDEV);
use File::Copy qw(move);

unless (rename $src, $dst) {
    if ($! == EXDEV) {
        move($src, $dst) or die "move: $!";
    }
    else {
        die "rename: $!";
    }
}
```

Rename every `.log` in a directory to `.log.old`:

```
for my $f (glob "*.log") {
    rename $f, "$f.old"
        or warn "rename $f: $!";
}
```

Edge cases

- **`NEWNAME` already exists**: silently clobbered. `rename` does **not** ask, warn, or refuse. If the target is a regular file it is unlinked as part of the operation; if it is a directory the call fails with `ENOTDIR` or `EISDIR` depending on whether `OLDNAME` is a directory. To get "refuse to overwrite" semantics, test with `-e` first (and accept that the check is racy - there is no portable `renameat2(RENAME_NOREPLACE)` exposed through `rename` itself).
- **Cross-filesystem (`EXDEV`)**: fails, leaves both paths intact. Use `File::Copy::move` when the destination may be on another mount.
- **Open filehandles**: on Linux (and POSIX generally) renaming a file that is currently open is fine. The open filehandle keeps pointing at the same inode; subsequent reads and writes through it continue to work. New opens use the new name.
- **Directories**: renaming a directory is allowed, but only when `NEWNAME` does not

exist or is an empty directory. Moving a directory into a subdirectory of itself fails with `EINVAL`.

- **Permissions**: `rename` needs write + execute permission on both the source and destination parent directories, not on the file itself. Failure shows up as `EACCES` or `EPERM`.
- **Sticky-bit directories** (`/tmp`): if the destination parent has the sticky bit set, you must own either the source file or the destination parent; otherwise the call fails with `EPERM` even when the directory is world-writable.
- **Symlinks**: `rename` operates on the link itself, never on its target. `rename "link", "other"` renames the symlink; the file it points at is untouched.
- **Trailing slash on `OLDNAME`**: many kernels reject this with `ENOTDIR` when the source is not a directory. Do not rely on trailing-slash behaviour being portable.
- **Same-path rename**: `rename $p, $p` succeeds and does nothing (POSIX-required behaviour) as long as `$p` exists.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `unlink` - remove a file by name; the tear-down counterpart to creating one with `rename` into place
- `link` - make an additional hard-link name for an existing file without removing the original
- `symlink` - create a symbolic link; note that `rename` on a symlink moves the link, not its target
- `File::Copy` - portable `move` and `copy` that handle the cross-filesystem (`EXDEV`) case `rename` cannot
- `$!` - `errno` after a failed `rename`; compare against constants from `Errno` (`EXDEV`, `EACCES`, `ENOENT`, ...)

Modules

require

Load a Perl source file at runtime, or demand a minimum Perl version.

`require` has three forms that do quite different things. With a version number it asserts the running interpreter is new enough. With a bareword it loads a module by transforming `Foo::Bar` into `Foo/Bar.pm` and searching `@INC`. With a runtime string or `$_` it loads a filename verbatim. All three are **runtime** operations - contrast with `use`, which is `require` + `import` wrapped in a compile-time `BEGIN` block.

Synopsis

```
require VERSION
require EXPR
require
```

What you get back

For the version-check form: 1 if the running Perl is new enough; otherwise an exception is thrown (`Perl vX.Y.Z required--this is only ...`). Nothing to check.

For the file-loading forms: the true value returned by the last statement of the loaded file, or 1 if the file enables the `module_true` feature (default under `use v5.38` and later). A second `require` of the same path returns 1 without re-executing the file. Failure throws an exception - either the compile/runtime error from the file, `Can't locate Foo/Bar.pm in @INC ...`, or `Foo/Bar.pm did not return true value`.

Wrap in `eval` when you want to test whether a module is loadable:

```
eval { require Some::Optional::Module };
if ($@) { ... }                    # module missing or broken
```

Global state it touches

- `@INC` - the search path for bareword and relative-filename forms. Entries may be directory strings, coderefs, array refs, or blessed objects (see `@INC` *hooks* below).
- `%INC` - maps the requested filename (`Foo/Bar.pm`) to the absolute path that was loaded. `require` consults it to avoid double-loading and sets it on success. An entry of `undef` marks a file whose compilation failed.
- `$_` - used as the filename when `require` is called with no argument.
- `${^HOOK}{require__before}` and `${^HOOK}{require__after}` - optional coderefs run around every `require`, available for tracing and instrumentation wrappers.

The three forms

`require VERSION` - minimum Perl check

`VERSION` is either a v-string like `v5.38.0` (compared against `$^V`) or a numeric literal like `5.038000` (compared against `$]`). If the running interpreter is older, `require` throws.

```
require v5.38;        # preferred
require 5.038;        # numeric; same effect
require 5.038_000;    # same; pre-5.6 syntax
```

This is a **runtime** check. If you want the check to happen at compile time (so the script never starts up on an old Perl), use `use VERSION` instead:

```
use v5.38;            # compile-time equivalent
```

`require` EXPR where EXPR is a bareword - module by name

A bareword is interpreted as a module name. `require` replaces every `::` with `/` and appends `.pm`, then searches `@INC`:

```
require Foo::Bar;     # looks for Foo/Bar.pm in @INC
```

This form also **autovivifies the stash** at compile time - the `Foo::Bar::` package hash springs into existence even if the load later fails. No symbols are imported; `require` only loads code.

Before trying `Foo/Bar.pm`, `require` looks for `Foo/Bar.pmc` in the same directory and prefers it if found. The `.pmc` hook lets build tools ship precompiled companion files alongside sources.

`require` EXPR where EXPR is a string - file by path

Any non-bareword expression is treated as a literal filename. No `::` translation, no `.pm` appending, no stash autovivification:

```
require "Foo/Bar.pm";           # explicit .pm path
require "$config_dir/init.pl";  # arbitrary filename

my $class = 'Foo::Bar';
require $class;                 # ERROR: looks for file "Foo::Bar"
```

The last line is the classic trap: once the module name is in a scalar, it is a string, not a bareword. To load a module whose name is computed, either do the translation by hand or go through `eval`:

```
(my $file = "$class.pm") =~ s{::}{/}g;
require $file;

eval "require $class; 1" or die $@;
```

If EXPR is omitted entirely, `$_` is used.

The `%INC` guard - loaded files stay loaded

Before searching `@INC`, `require` consults `%INC`. If the requested filename is present and has a true value, `require` returns 1 immediately without re-reading the file. If the value is `undef`, a previous load failed and `require` throws `Compilation failed in require` - Perl refuses to retry a file that has already blown up once in this interpreter.

This means a module's top-level code runs **exactly once** per interpreter, no matter how many places `require` it. Setup in a module's file body is a one-shot operation; do not rely on it running for every `require` call in every caller.

A hook is free to set `%INC` itself; if it does not, `require` records the hook as the "source" of the load.

The 1; convention

Historically every loadable file had to return a **true** value as its last expression, or `require` threw `filename did not return true value`. The canonical way to guarantee this is a trailing `1;`:

```
package My::Thing;
# ... module code ...
1;                                  # mandatory, pre-5.38
```

As of Perl 5.38 the `module_true` feature - enabled by default under `use v5.38` or later - makes every `required` file implicitly return true on successful compilation. Files that opt in no longer need `1;`. Files that don't, still do.

This only affects the compilation unit that turned the feature on; legacy modules loaded by a modern caller still need their own `1;`.

When the loaded file does return a value and `module_true` is off, that value is what `require` returns - but **only on the first load**. Subsequent `require` calls of the same file return 1 from the `%INC` cache. To capture a file's return value reliably, use `do`:

```
my $config = do "config.pl"
    or die "config.pl: ", $@ || $!;
```

@INC hooks - coderefs, array refs, and objects

Entries in `@INC` do not have to be directory strings. A reference stored in `@INC` is called a **hook** and gets a chance to satisfy the `require`.

Subroutine reference

Called with two arguments: the coderef itself and the filename (`Foo/Bar.pm`). It should return either nothing (meaning "I can't handle this; try the next entry") or up to four values:

1. A scalar reference to source text prepended to the file.
2. A filehandle the rest of the source is read from.
3. A generator or filter subroutine (see `perldoc perlfunc` for the exact protocol).
4. Optional state passed back to the generator.

```
push @INC, sub {
    my ($self, $filename) = @_;
    return unless $filename eq 'Virtual/Module.pm';
    open my $fh, '<', \"package Virtual::Module; 1;";
    return $fh;
};
```

Array reference

The first element is a coderef or object as above; the rest is state the hook can read on each call. When the first element is an object with an `INC` or `INCDIR` method, the method receives the array ref as its third argument.

```
push @INC, [\&my_loader, $db_handle, $cache];
```

Blessed object

If the object has an `INC` or `INCDIR` method, `require` calls that method; otherwise, if the object is also a coderef, it's invoked like a plain subroutine reference. `INC` methods must be declared with their fully qualified name (`sub Foo::INC { ... }`), because the unqualified symbol `INC` is hard-wired to package `main`.

A hook may freely update `%INC` to record the path it supplied; if it doesn't, `require` stores the hook reference itself.

`require` vs `use`

```
use Foo::Bar qw(x y);

# is almost exactly:

BEGIN {
    require Foo::Bar;
    Foo::Bar->import(qw(x y));
}
```

Aspect	`use`	`require`
When it runs	Compile time (inside `BEGIN`)	Runtime
Calls `import`	Yes	No
Bareword `::` → `/`	Yes	Yes, when called on a bareword
Version check form	`use v5.38`	`require v5.38`
Good for	Fixed dependencies	Optional / late-bound modules

Reach for `require` when the module is optional, loaded from a computed name, or loaded only inside a branch that may never run. Reach for `use` for everything else.

Examples

Minimum Perl version at runtime:

```
require v5.38;
```

Bareword module load - searches `@INC`, autovivifies `Data::Dumper::` at compile time:

```
require Data::Dumper;
Data::Dumper->import;
print Data::Dumper::Dumper(\%ENV);
```

Optional module - fall back gracefully when missing:

```
my $have_json = eval { require JSON::PP; 1 };
if ($have_json) {
    print JSON::PP->new->encode($data);
}
```

Load a module whose name is computed at runtime. Convert to a filename path; `require $class` would search for a literal file named `Foo::Bar`:

```
my $driver = $ENV{DB_DRIVER} // 'SQLite';
my $class  = "DBD::$driver";
(my $file  = "$class.pm") =~ s{::}{/}g;
require $file;
```

Load a configuration fragment by pathname, keeping its return value - note the do, not `require`, because we want the value every call:

```
my $config = do '/etc/myapp/config.pl'
    or die "config load failed: ", $@ || $!;
```

Install an `@INC` hook that serves a single virtual module from an in-memory string:

```
unshift @INC, sub {
    my ($self, $filename) = @_;
    return unless $filename eq 'MyApp/Version.pm';
    my $src = "package MyApp::Version; our \$VERSION = '1.23'; 1;";
    open my $fh, '<', \$src;
    $INC{$filename} = '(generated)';
    return $fh;
};
require MyApp::Version;
```

Edge cases

- **Bareword vs string**: `require Foo::Bar` and `require "Foo::Bar"` are **not** the same. The bareword form translates to `Foo/Bar.pm`; the string form looks for a file literally named `Foo::Bar`.
- **Computed class names**: `require $class` where `$class` holds `'Foo::Bar'` searches for a file named `Foo::Bar`, not `Foo/Bar.pm`. Translate the name yourself, or use `eval "require $class"` when the name is trusted.
- **No argument**: `require;` uses `$_` as the filename. Rare in modern code; almost always an accident.
- **Second-chance retries don't work**: If a `require` fails partway through compilation, `%INC` records `undef` for that filename. A later `require` of the same name sees

the undef and throws `Compilation failed in require` without re-running the file. Delete the `%INC` entry if you genuinely want to retry (this is a sharp edge; prefer fixing the underlying failure).

- **Relative `.` removal**: `.` was removed from the default `@INC` in Perl 5.26. A script that used to find `Foo/Bar.pm` next to itself now has to add its own directory explicitly (`use FindBin; use lib $FindBin::Bin;`).
- **`.pmc` shadowing**: If `Foo/Bar.pmc` exists in the same directory as `Foo/Bar.pm`, the `.pmc` wins. Stale `.pmc` files are a classic "why is my edit not taking effect" trap.
- **Parsing `require(EXPR)`**: `require` is parsed as a named unary operator; `require EXPR + 1` means `require(EXPR + 1)`. Use explicit parentheses when the argument is a non-trivial expression.
- **Stash autovivification**: `require Foo::Bar` creates the `Foo::Bar::` stash even if the file is missing. Downstream `defined &Foo::Bar::some_sub` checks may pass the "package exists" stage but fail the "sub exists" stage.
- **Filehandle from a hook must be real**: Tied filehandles are silently ignored by the hook protocol; processing falls off the end and `require` reports "not found". Use a real typeglob.
- **`INC` must be fully qualified in hook classes**: `sub Foo::INC { ... }` works; `sub INC { ... }` inside `package Foo` installs `main::INC` because of a hard-coded package rule.
- **`require()` is hard to wrap**: Many modules inspect the call stack to find their caller. A wrapper that adds a frame often breaks them. Prefer the `${^HOOK}{require__before}` / `${^HOOK}{require__after}` hooks (added in 5.38) for tracing.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `use` - compile-time `require` + `import`; the normal way to pull in a module you depend on unconditionally
- `do` - load a file and return its value on every call, with no `%INC` caching and no "must return true" rule
- `import` - the method `use` calls after `require`; `require` never calls it for you
- `@INC` - search path consulted for bareword and relative-filename loads; accepts directory strings, coderefs, array refs, and objects as hooks
- `%INC` - one-entry-per-loaded-file cache that makes `require` idempotent and records where each module came from
- `eval` - wrap `require` when the load is allowed to fail

Misc

reset

Clear every package variable whose name begins with one of a set of letters, and re-arm one-shot `m?pattern?` matches.

`reset` is a holdover from Perl 4's globals-only world. In modern code it survives in two narrow roles: re-arming `m?...?` one-match regexes so they fire again, and - at the end of a loop's `continue` block - wiping a batch of package variables before the next iteration. Its reach is limited to the current package and to **package** variables; anything declared with `my` is untouched.

Synopsis

```
reset EXPR
reset
```

EXPR is a string interpreted as a set of single characters, with hyphens allowed as ranges: `'X'`, `'a-z'`, `'abc'`, `'A-Za-z'`. Every scalar, array, and hash in the **current package** whose name starts with one of those letters is reset to its pristine, undefined state.

With no argument, only `m?pattern?` one-match searches in the current package are re-armed - variables are not touched.

What you get back

Always 1. The return value carries no success signal; there is nothing for `reset` to fail at.

Global state it touches

- **Every package variable in the current package** whose name starts with a listed letter: scalars, arrays, and hashes all together. Scalars become `undef`, arrays and hashes become empty.
- **One-match regexes** (`m?pattern?`) compiled in the current package - their "already matched" flag is cleared so the next evaluation matches again.
- Does **not** cross package boundaries. `reset 'X'` inside `package Foo` leaves `$Bar::X` alone.
- Does **not** touch lexicals. `my $x` and `our $x` behave differently here: `our` is an alias to the package variable and is reset; `my` is not.

Examples

Re-arm all one-shot matches in the current package - the typical modern use of `reset`:

```
reset;
```

Clear every package variable starting with `X` in the current package:

```
package Stats;
our $Xtotal = 0;
our @Xsamples = (1, 2, 3);
our %Xindex = (a => 1);
reset 'X';
# $Xtotal, @Xsamples, %Xindex are all emptied
```

Reset at the bottom of a loop so the next iteration starts clean:

```
while (more_work()) {
    process();
} continue {
    reset 'abc';   # wipe $a*, @b*, %c* package vars for next pass
}
```

A character range - lower-case variables only:

```
reset 'a-z';
```

Combining individual letters and ranges:

```
reset 'a-cXY';    # a, b, c, X, Y
```

Edge cases

- **`reset` with no argument does not touch variables at all.** It re-arms one-shot `m?...?` matches in the current package and nothing else. If you want to clear variables you must pass an argument.
- **Lexicals are invisible to `reset`.** A `my $Xtotal` is completely unaffected by `reset 'X'`; only the package variable `$Xtotal` is. Code that migrated from `our`/package globals to `my` will find `reset` has silently become a no-op for those variables.
- **`reset 'A-Z'` is a foot-gun.** In the `main` package that wipes `@ARGV`, `@INC`, `%ENV` and every other upper-case global, almost certainly not what was intended.
- **Scalars, arrays, and hashes share one namespace here.** `reset 'X'` clears every variable of every sigil whose name starts with `X`, not just scalars. There is no way to restrict the reset to one sigil.
- **Current package is decided at compile time** via the enclosing `package` declaration, not by the caller. A `reset` call inside a module clears that module's variables, not the caller's.
- **No regex argument form.** `reset` takes a string of letters, not a regex or a list of names. You cannot write `reset qr/^X/` or `reset '$Xtotal'` - the latter tries to reset variables starting with `$`, `X`, `t`, `o`, `a`, `l`, which is almost never what the author meant.

Differences from upstream

- pperl's `reset` is a no-op - the call parses, returns 1, and has no side effects. Package variables are not cleared and `m?pattern?` one-shot match state is not tracked. Code that relies on `reset` to wipe a batch of globals at the end of a loop, or to re-fire a one-match regex, will not see that behaviour under pperl. Prefer `my` and ordinary `m//` regexes - both work identically on pperl and upstream, and are the recommended idiom upstream as well.

See also

- `my` - lexical variables; preferred in all new code and unaffected by `reset`
- `local` - dynamically scoped save/restore of a package variable, the usual answer when you want a temporary change without a manual cleanup
- `m` - the regular `m//` form; `m?...?` is the one-shot variant that `reset` re-arms
- `continue` - loop-end block where batch-reset of iteration state is conventionally placed
- `package` - declares which package's variables `reset` will operate on
- `undef` - clear a single named variable, the per-variable alternative to a broad `reset`

Control flow

return

Leave the current subroutine, `eval`, `do FILE`, `sort` block, or regex eval block, yielding a value to the caller.

`return` is the explicit exit from a named or anonymous sub. It stops evaluation of the sub body at the point of call and hands `EXPR` back to whoever invoked the sub. The value `EXPR` produces depends on the caller's context - list, scalar, or void - which the body can inspect with `wantarray` before deciding what to return.

Synopsis

```
return EXPR
return
```

What you get back

`return` itself does not have a return value - it does not continue in the current frame. It terminates the frame and the caller receives `EXPR`, evaluated in the caller's context:

- **List context** - `EXPR` is evaluated in list context; its list value is what the caller sees.
- **Scalar context** - `EXPR` is evaluated in scalar context; its scalar value is what the caller sees.
- **Void context** - `EXPR` is still evaluated (for side effects), but the value is discarded.

With no `EXPR` the caller receives:

- **list context** - the empty list `()`
- **scalar context** - `undef`
- **void context** - nothing

This asymmetry matters: `return;` (bare) is **not** the same as `return undef;`. A bare `return` in list context yields `()`, which is false and has zero elements; `return undef;` in list context yields a one-element list containing `undef`, which is **true** (because the list has one element). Error-signalling subs should almost always use bare `return`.

```
sub find_it {
    my ($key) = @_;
    return unless exists $cache{$key};  # bare: () in list, undef in
↪scalar
    return $cache{$key};
}

if (my @hits = find_it($k)) { ... }     # () is false; undef-in-list
↪is true
```

Where return is legal

`return` unwinds these frames:

- a named or anonymous **subroutine**
- a string or block **`eval`** - `return` from inside `eval { ... }` leaves the `eval`, not the enclosing sub
- a **`do FILE`** - `return EXPR` is the file's value
- a **`sort`** **comparator block** - returns the comparison result
- a **regex eval block** - `/(?{ ... })/` and `/(?&NAME)/` code blocks

`return` is **not** legal inside these blocks - it is a compile-time or run-time error depending on the context:

- **`grep BLOCK LIST`** and **`map BLOCK LIST`** - the block is not a sub frame; use the block's last expression instead
- **`do BLOCK`** - the block is an expression, not a call; its value is already the block's last expression
- bare blocks, `if` / `while` / `for` bodies - same reason; these do not form a call frame

If you need "return a value from this block", make it the block's last evaluated expression:

```
my @evens = grep { $_ % 2 == 0 } @nums;   # not: grep { return ... }
```

Context propagation with wantarray

The caller's context reaches the callee through `wantarray`:

- `wantarray` is **true** in list context
- `wantarray` is **defined and false** (`""`) in scalar context
- `wantarray` is **`undef`** in void context

A sub that wants to return shape-appropriate values inspects it:

```
sub stats {
    my @xs = @_;
    return unless @xs;                    # bare: () / undef as
 ↪above
    my $sum = 0; $sum += $_ for @xs;
    my $avg = $sum / @xs;
    return wantarray ? ($sum, $avg) : $avg;
}

my ($s, $a) = stats(@data);               # list context  → two
 ↪values
my $a       = stats(@data);               # scalar context →
 ↪average only
stats(@data);                             # void context  →
 ↪computed, discarded
```

The same call site can be evaluated in different contexts across different invocations; `return` re-resolves `EXPR` each time.

Implicit return: last expression wins

If control reaches the end of a sub (or `eval`, or `do FILE`) without hitting `return`, the value of the last evaluated expression is returned, in the caller's context. This is the idiomatic "trailing expression" style:

```
sub square { my ($x) = @_; $x * $x }      # implicit return of $x *
 ↪$x

sub classify {
    my ($n) = @_;
    $n < 0   ? "negative"
  : $n == 0  ? "zero"
  : "positive"                            # trailing ternary is the
 ↪value
}
```

Use explicit `return` when you exit early, or when the end of the sub is a statement (e.g. a loop, an assignment) whose value is not what you want the caller to see.

Examples

Early exit on a guard:

```
sub lookup {
    my ($id) = @_;
    return unless defined $id;
    return $table{$id};
}
```

Return from an `eval` block without unwinding the enclosing sub:

```
sub try_parse {
    my ($text) = @_;
    my $parsed = eval {
        return {} if $text eq "";          # leaves eval, not try_
↪parse
        parse($text);
    };
    return $@ ? undef : $parsed;
}
```

Context-sensitive return:

```
sub users {
    return wantarray ? @all_users : scalar @all_users;
}

my @u = users();                           # the users themselves
my $n = users();                           # how many there are
```

Parser exemption - `return` is not a looks-like-a-function operator:

```
sub greet { return ("hello") . " world" } # returns "hello world"
                                           # the . " world" is part␣
↪of EXPR
greet();                                   # "hello world"
```

This is why `return (EXPR)` does **not** insulate `EXPR` from a trailing operator; anything after the closing paren is still argument to `return`. Same rule as `print`.

Illegal: `return` inside `grep` / `map`:

```
my @odd = grep { return $_ % 2 } @n;       # wrong; leaves the sub
my @odd = grep { $_ % 2 } @n;              # right
```

Inside a `grep` / `map` the block's last expression is already the value; a `return` there exits the **enclosing sub**, which is almost never what you mean.

Edge cases

- **Bare `return` vs `return undef`**: as above, bare `return` yields `()` in list context, which lets `if (my @x = f())` work as a "no results" test. `return undef` puts one element in the list and flips that test to true. Prefer bare `return` unless you specifically want a defined-undef scalar.
- **`return` from a `sort` comparator**: legitimate and common in complex comparators:

```
sort { return $a->[0] <=> $b->[0] || $a->[1] cmp $b->[1] }␣
↪@rows;
```

- **`return` from `do FILE`**: becomes the file's value, which is what `require` and `use` inspect to decide module-load success. A `.pm` ending in `1;` is the common idiom; `return 1;` near the top of the file is equivalent.
- **`return` from a regex `(?{ ... })` block**: ends the code block only, not the regex match. Rarely useful; normal block flow is almost always clearer.
- **Void-context side effects still happen**: `return expensive();` in void context still calls `expensive()`. `return` is not an optimisation hint.
- **Not a function, no parentheses needed**: `return` is an operator, not a function. `return(1, 2, 3)` and `return 1, 2, 3` are the same. The parser exemption above means parens around EXPR do not delimit it from the rest of the statement - parens are purely for grouping inside EXPR.
- **Return in a block form of `eval`**: `eval { return $x }` leaves the `eval` with `$x`; `$@` is cleared (no exception). To return from the enclosing sub via an exception, `die` instead.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `wantarray` - inspect the caller's context so `return` can yield the right shape
- `caller` - the other half of caller introspection; useful when a sub decides what to return based on who called it
- `eval` - `return` inside an `eval` block leaves the `eval`, not the enclosing sub
- `sub` - how subroutines are defined and how implicit return interacts with the last expression
- `die` - the non-local exit; use when you want to unwind past intermediate frames instead of returning through each one
- `last` - exits a loop, not a sub; do not confuse the two

SCALARs and strings · Lists

reverse

Reverse a list - or, in scalar context, reverse the characters of a string.

`reverse` is two operators wearing the same name. Context decides which one you get. In list context it returns `LIST` with its elements in the opposite order; element values are untouched. In scalar context it concatenates `LIST` into a single string and returns that string with its **characters** reversed. The single biggest trap is context: `print reverse $str` puts `reverse` in list context, where it reverses a one-element list and prints `$str` unchanged; `scalar reverse $str` is what reverses the characters - see *Edge cases*.

Synopsis

```
my @r    = reverse @arr          # list context: elements reversed
my $s    = reverse $str          # scalar context: characters
 ↪reversed
my %inv  = reverse %h            # hash invert (value → key)
reverse                          # in scalar context, reverses $_
```

What you get back

Context-dependent. The operator inspects the context of its caller and branches:

- **List context** → a new list with the elements of `LIST` in reverse order. Element values are not stringified, not modified, not copied at the character level. A list of refs stays a list of the same refs.
- **Scalar context** → a single string: the concatenation of the stringified elements of `LIST`, with all characters in reverse order. Under `use utf8` / wide strings, reversal is character-wise, not byte-wise.

This means the *same* call site returns radically different shapes depending on how you use it:

```
my @back  = reverse "abc", "def";    # ("def", "abc")
my $back  = reverse "abc", "def";    # "fedcba"
```

Used with no arguments in scalar context, `reverse` reverses `$_`. Used with no arguments in list context, `reverse` returns the empty list - **not** a reversed `$_`. `print reverse;` produces no output.

Global state it touches

Reads `$_` when called with no arguments in scalar context. That is the only global it touches. It does not modify `$_`; it returns the reversed string.

Examples

Reverse a list. The elements themselves are unchanged:

```
my @a = (1, 2, 3, 4, 5);
my @r = reverse @a;              # (5, 4, 3, 2, 1)
```

Reverse a string. Force scalar context with `scalar` when the surrounding expression would otherwise be list:

```
my $s = reverse "Hello, world";  # "dlrow ,olleH"
print scalar reverse "Hello";    # "olleH"
```

Iterate an array backward without building a second array. `reverse` in the loop header is a list-context call; Perl consumes the reversed list lazily from the loop machinery:

```
my @stack = ("first", "second", "third");
for my $elem (reverse @stack) {
    print "$elem\n";             # third / second / first
}
```

Descending sort. `reverse sort` is the idiom when your comparison is ascending and you want the opposite. For anything more complex than default string compare, swap the operands in the comparator instead:

```
my @asc  = sort { $a <=> $b } @nums;
my @desc = reverse sort { $a <=> $b } @nums;     # works
my @desc2 = sort { $b <=> $a } @nums;            # preferred
```

Invert a hash. Values become keys, keys become values:

```
my %by_name    = (alice => 1, bob => 2, carol => 3);
my %by_id      = reverse %by_name;               # (1 => "alice", ..
 ↪.)
print $by_id{2};                                 # "bob"
```

The `"dlrow ,olleH"` trick - using `reverse` in scalar context as a one-shot character flip on a string variable:

```
my $word = "stressed";
my $rev  = scalar reverse $word;                  # "desserts"
```

Edge cases

- **The #1 trap: context.** `my $s = reverse @arr` does **not** give you a reversed array; it gives you the reversed *string* formed by concatenating the array's elements and flipping every character. If you want a reversed array stored via a scalar, use a reference: `my $ref = [ reverse @arr ]`.
- **Single scalar, wrong context.** `my @r = reverse $str` returns `($str)` - a one-element list containing `$str` unchanged. There is nothing to reorder. To reverse the characters, force scalar context: `my $r = reverse $str` or `my @r = (scalar reverse $str)`.
- **Hash invert loses data on duplicate values.** `reverse %h` walks the key/value pairs as a flat list and rebuilds a hash. If two keys share a value, one of them wins and the other is silently dropped; which one wins is not defined. Check for uniqueness first if it matters:

```
my %seen;
$seen{$_}++ for values %h;
die "non-unique values" if grep { $_ > 1 } values %seen;
my %inv = reverse %h;
```

- **No arguments in list context does nothing useful.** `reverse` with no args returns `()` in list context. `$_` is only consulted in scalar context. `print reverse;` prints nothing; `print scalar reverse;` prints the reverse of `$_`.
- **Character-wise, not byte-wise.** For wide-character strings, `reverse` reverses characters, not bytes. A `:utf8` string of three code points stays three code points after reversal, regardless of their byte widths. Reversing a raw byte string with multi-byte characters you never decoded will corrupt it - decode first.
- **Self-assignment preserves holes.** `@a = reverse @a` on a non-magical or well-behaved tied array preserves non-existent elements in their reversed positions; a sparse array stays sparse.
- **`reverse sort` is not the same as `sort { $b cmp $a }`.** Both give descending order for a stable default compare, but `reverse sort` does two passes over the data and is measurably slower on large lists. Prefer swapping the comparator operands.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `sort` - pair with `reverse` for descending order, or swap `$a`/`$b` in the comparator to avoid the second pass
- `pop` - remove from the end of an array when you just want the last element, not the whole list reversed
- `shift` - remove from the front; combined with a `reversed` copy you get stack-like access to the original tail
- `split` - produce a list of characters (with `//`) or fields that you can then `reverse`
- `join` - the natural partner when you reverse a list of strings and want a single string back
- `$_` - the default target when `reverse` is called with no arguments in scalar context

I/O

rewinddir

Reset a directory handle to the start of its listing.

`rewinddir` sets the current position of `DIRHANDLE` back to the beginning of the directory, so that the next `readdir` call returns the first entry again. It is the directory-handle counterpart to `seek` on a file handle: no re-open, no re-scan of the parent directory - just a position reset on the open handle.

Synopsis

```
rewinddir DIRHANDLE
```

What you get back

1 on success, `0` on failure (with `$!` set). Failure is rare in practice - the usual cause is a handle that was never opened or has already been closed. Check the return value if you care about either case:

```
rewinddir $dh
    or die "rewinddir failed: $!";
```

Typical use

You have already walked a directory with `readdir` and want to walk it again without paying for another `opendir`:

```
opendir my $dh, $path or die "opendir $path: $!";

my @all = readdir $dh;          # first pass: collect everything

rewinddir $dh;                  # back to the top

while (my $entry = readdir $dh) {   # second pass: process one by␣
↪one
    next if $entry =~ /^\.\.?\z/;
    process("$path/$entry");
}

closedir $dh;
```

A second common pattern is reusing one handle across several scans with different filters, where re-opening would be wasteful or would race against concurrent directory changes:

```
opendir my $dh, $path or die $!;

my @perl = grep { /\.p[lm]\z/ } readdir $dh;
rewinddir $dh;
my @text = grep { /\.txt\z/  } readdir $dh;

closedir $dh;
```

Edge cases

- **Unopened or closed handle**: returns `0` and sets `$!`. Under `use warnings` you also get a `rewinddir() attempted on invalid dirhandle` warning.
- **Mid-scan directory changes**: `rewinddir` is **not** guaranteed to surface entries created (or hide entries removed) after `opendir` returned. The underlying `rewind-`

dir(3) operates on the kernel's cached directory stream; whether a rewound scan observes concurrent mkdir, unlink, or rename in the same directory is filesystem- and OS-dependent. If you need a fresh snapshot, closedir and opendir again.

- **Not a seek to an arbitrary position**: rewinddir resets to the start and nothing else. To return to a remembered mid-directory position use telldir / seekdir instead.
- **Entry order is unspecified**: rewinding replays the same stream the kernel gave you, but the order of entries is whatever the filesystem produces - not sorted, not creation order, not the order you saw last time on a different handle. Sort explicitly if order matters.
- **Portability**: rewinddir is a POSIX call. On platforms without a working rewinddir(3) it may be implemented by closing and re-opening the directory; see perlport for the historical list. On Linux (the platform pperl targets) this is not a concern.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- opendir - open a directory and obtain the handle that rewinddir operates on
- readdir - read the next entry; resumes from wherever rewinddir left the stream (the beginning)
- closedir - release the handle; pair with opendir when a rewound scan is not enough and you need a fresh snapshot of the directory
- telldir - record the current position so you can return to it later
- seekdir - restore a position previously captured by telldir; rewinddir $dh is equivalent to seekdir $dh, 0 on conforming systems, but rewinddir is the portable spelling

SCALARs and strings

rindex

Find the position of the **last** occurrence of a substring inside a string.

rindex is the right-to-left counterpart of index. It scans STR for SUBSTR and returns the zero-based offset of the rightmost match, or -1 if no match exists. When POSITION is given, the search is constrained to matches whose starting offset is less than or equal to POSITION - that is, the last occurrence at or before that position.

Synopsis

```
rindex $str, $substr
rindex $str, $substr, $position
```

What you get back

An integer. Zero or positive means the zero-based offset of the rightmost match; `-1` means `SUBSTR` does not occur in `STR` (subject to the `POSITION` constraint, if any).

Offsets are measured in **characters** when `STR` holds a character string, and in **bytes** when `STR` is a byte string. This matches `index` and the rest of Perl's string built-ins - the unit is whatever the string itself is using.

How POSITION shapes the search

`POSITION` is an **upper bound on the match's starting offset**, not on where scanning begins. The result satisfies:

- `result <= POSITION`
- `result + length(SUBSTR) <= length(STR)` (as always)

Concretely:

- `POSITION` omitted or `undef` - search the whole string; return the rightmost match.
- `POSITION` greater than or equal to `length(STR)` - behaves as if `POSITION` were `length(STR)`; the entire string is considered.
- `POSITION` less than `0` - behaves as if `POSITION` were `0`; only a match starting at offset `0` can satisfy the constraint, so the return value is `0` when `STR` begins with `SUBSTR` and `-1` otherwise.
- Empty `SUBSTR` - every offset from `0` to `length(STR)` is a valid match position, so `rindex` returns `min(POSITION, length(STR))` (or `length(STR)` when `POSITION` is omitted).

Think of `POSITION` as "find the last match at or before here," not as "start scanning here."

Examples

Basic right-to-left search:

```
rindex("banana", "a");          # 5  (the final 'a')
rindex("banana", "na");         # 4  (last "na")
rindex("banana", "z");          # -1 (not found)
```

Stepping backwards through every occurrence by using each result, minus one, as the next `POSITION`:

```
my $str = "banana";
my $pos = length($str);
while ((my $hit = rindex($str, "a", $pos - 1)) >= 0) {
    print "match at $hit\n";   # 5, 3, 1
    $pos = $hit;
}
```

`POSITION` as an upper bound - the match must **start** at or before it:

```
rindex("banana", "na", 3);      # 4? No: 4 > 3, so the match at 4
                                #   is excluded; returns 2
rindex("banana", "na", 4);      # 4  (match starting at 4 is allowed)
```

Empty SUBSTR matches at every offset, so the result is the capped position:

```
rindex("banana", "", 4);        # 4
rindex("banana", "");           # 6  (length of "banana")
```

Pairing with index to locate the last segment of a delimited string - a common idiom for splitting off a final extension or path component:

```
my $path = "/usr/local/bin/perl";
my $slash = rindex($path, "/");
my $file  = substr($path, $slash + 1);   # "perl"
```

Edge cases

- **Substring longer than string**: `rindex("abc", "abcd")` returns `-1`. No match can fit.
- **Substring equals string**: `rindex("abc", "abc")` returns `0`. The last (and only) match starts at offset `0`.
- **POSITION past the end**: `rindex($str, $sub, 9999)` behaves as if POSITION were `length($str)` - the entire string is scanned.
- **POSITION is negative**: treated as `0`. `rindex("abc", "b", -5)` returns `-1` (no match can start at offset `0` or earlier); `rindex("abc", "a", -5)` returns `0`.
- **POSITION is undef**: same as omitting it. Under `use warnings` this emits an `uninitialized` warning.
- **Empty STR, non-empty SUBSTR**: returns `-1`.
- **Empty STR, empty SUBSTR**: returns `0`.
- **Unicode**: for character strings (those the interpreter knows are text), offsets count characters, not bytes. A string flipped to byte semantics via `use bytes` or `encode_utf8` will report byte offsets instead. The rule mirrors index and substr.
- **Tainted arguments**: the result inherits taint from STR and SUBSTR under `-T`.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- index - the left-to-right counterpart; identical argument conventions, returns the **first** occurrence
- substr - typically the next call after a successful `rindex`, to slice the matched region or what follows it

- `length` - useful when computing the trailing slice: `substr($s, $hit + length($sub))`
- `pos` - for regex-driven scanning where you need the engine's match state rather than a single substring offset
- `split` - when you want every segment delimited by SUBSTR, not just the last one
- `reverse` - combined with `index` it is sometimes mistaken for a `rindex` substitute; `rindex` is both clearer and faster

Filehandles, files, directories

rmdir

Remove an empty directory.

`rmdir` asks the operating system to delete the directory named by `FILENAME`. The directory must be empty - it must contain no entries other than `.` and `..`. If `FILENAME` is omitted, `rmdir` operates on `$_`. This is a thin wrapper over the `rmdir(2)` system call and carries all of its restrictions: you cannot remove a non-empty directory, you cannot remove a directory you lack write permission on the parent of, and on most systems you cannot remove your current working directory.

Synopsis

```
rmdir FILENAME
rmdir
```

What you get back

1 on success, `0` on failure (with `$!` set to the `errno` from the failed system call). Unlike most I/O primitives `rmdir` does not return `undef` on failure - it returns the integer `0`, so the common test `rmdir $dir or die $!` still works because `0` is false.

Always check the return value. A failed `rmdir` is silent otherwise; the next line of code will happily proceed as if the directory had been removed.

```
rmdir $dir
    or die "rmdir $dir failed: $!";
```

Global state it touches

- `$_` - used as `FILENAME` when no argument is given.
- `$!` - set to the `errno` of the failed `rmdir(2)` call on failure; left untouched on success. Read it immediately; any intervening system call may overwrite it.

Examples

Plain call, explicit directory:

```
rmdir "build/tmp"
    or die "rmdir build/tmp: $!";
```

Default target. Inside `while (readdir ...)` or a loop that leaves the name in `$_`:

```
for (@empty_dirs) {
    rmdir or warn "rmdir $_: $!";
}
```

Distinguish "already gone" from a real error. `ENOENT` means the directory does not exist, which is often fine for cleanup code:

```
use Errno qw(ENOENT);
unless (rmdir $dir) {
    die "rmdir $dir: $!" unless $! == ENOENT;
}
```

Remove a directory only after emptying it first:

```
opendir my $dh, $dir or die "opendir $dir: $!";
while (my $entry = readdir $dh) {
    next if $entry eq '.' or $entry eq '..';
    unlink "$dir/$entry" or die "unlink $dir/$entry: $!";
}
closedir $dh;
rmdir $dir or die "rmdir $dir: $!";
```

Recursive removal is **not** `rmdir`'s job. Use `File::Path`'s `remove_tree` - it walks the tree and deletes entries bottom-up:

```
use File::Path qw(remove_tree);
remove_tree("build", { error => \my $err });
die "remove_tree: @$err" if @$err;
```

Edge cases

- **Directory not empty**: `$!` is set to `ENOTEMPTY` on most Unix systems (some BSDs use `EEXIST`). This is the single most common `rmdir` failure. List the directory contents or call `remove_tree` if recursive deletion is what you actually want.
- **No argument**: `rmdir` with no argument uses `$_`, not `@ARGV` and not the current directory. Writing `rmdir()` with empty parentheses is the same as `rmdir` with no argument - it still reads `$_`.
- **Symlink to a directory**: `rmdir` refuses to remove symlinks; it operates on directories only. Use `unlink` on the symlink itself. `$!` is typically set to `ENOTDIR`.
- **Current working directory**: On Linux, `rmdir "."` usually fails with `EBUSY` or `EINVAL` depending on the filesystem. Do not rely on being able to remove the directory you are sitting in; `chdir` elsewhere first.

- **Permissions**: you need write + execute permission on the **parent** directory, not on the directory being removed. Removing `/a/b/c` checks `/a/b`'s mode, not `/a/b/c`'s.
- **Sticky-bit parent** (e.g. `/tmp`): you also need to own the directory being removed or own the parent; otherwise `EPERM`.
- **Trailing slash**: `rmdir "foo/"` works and means the same as `rmdir "foo"`.
- **Race with another process**: between your `opendir`/`readdir` loop and the final `rmdir`, another process may create a file in the directory. You will then see `ENOTEMPTY` despite having just emptied it. Retry or fail loudly - do not silently ignore.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `mkdir` - create a directory; the inverse operation, with the same `FILENAME`-or-`$_` argument convention
- `unlink` - remove **files** (and symlinks); `rmdir` removes only empty directories
- `chdir` - change the current working directory, often used before `rmdir` to avoid trying to remove the directory you are in
- `opendir` / `readdir` - list a directory's contents before emptying and removing it
- `$!` - holds the `errno` of a failed `rmdir`; check for `ENOENT` / `ENOTEMPTY` / `EPERM` to branch on the reason

Regular expressions and pattern matching

s///

Search a string for a pattern and replace every match with a replacement.

`s///` is Perl's substitution operator. It finds text matching `PATTERN` in the target string, replaces each match with `REPLACEMENT`, and by default returns the number of substitutions made. With no explicit target (no =~ or !~), it operates on `$_`. The target must be a scalar lvalue - a variable, an array or hash element, or a `substr` / assignment producing one - unless the `/r` flag is used, which works on any value and returns the modified copy.

Synopsis

```
s/PATTERN/REPLACEMENT/FLAGS
$str =~ s/PATTERN/REPLACEMENT/FLAGS
$str !~ s/PATTERN/REPLACEMENT/FLAGS
my $copy = $str =~ s/PATTERN/REPLACEMENT/r
```

What you get back

By default, the count of substitutions performed as an integer: `0` when nothing matched (which is false in boolean context), a positive integer on success. The target scalar is modified in place.

With the `/r` flag, the semantics flip: the target is **not** modified, and the operator returns the (possibly changed) string. If nothing matched, `/r` still returns a copy of the original. The result of `/r` is always a plain string, even when the source is a tied variable or a blessed object with overloading.

`!~` inverts the boolean sense of the return value: it yields true when nothing was substituted, false when at least one substitution happened. The target is still modified either way.

```
my $n = ($text =~ s/foo/bar/g);      # count of replacements
my $out = $text =~ s/foo/bar/r;      # copy-and-modify, $text␣
↪untouched
```

Global state it touches

- `$_` - the default target when no `=~` or `!~` is given. `s/old/new/` with no binding reads and writes `$_`.
- `$&`, `$``, `$'`, `$1` and friends - set by each successful match, the same way `m//` sets them. Inside `REPLACEMENT` they refer to the current match, not any earlier one.
- `pos` - cleared on the target scalar by a successful non-`/g` substitution; under `/g` it advances with each match and can be inspected between iterations.
- The **last successful pattern** - used when `PATTERN` evaluates to the empty string (`s//new/` re-uses the last successful regex from this scope).
- `$/`, `$\` and output separators are not touched - `s///` produces a string, it does not write one.

Flags

The full set is `msixpodualngcer`. Match-side flags behave exactly as in `m//` and `qr`; see the *modifiers chapter* of the regex guide for the match-side details. Replacement-specific flags:

- `e` - evaluate `REPLACEMENT` as a block of Perl code. The block's value is the replacement string. Equivalent to wrapping it in `do { ... }` and stringifying the result.
- `ee` - evaluate as code, then `eval` the resulting string again. Equivalent to `eval(do { REPLACEMENT })`. Additional `e` modifiers add more `eval` layers.
- `r` - non-destructive: leave the target unchanged and return the modified copy (or an unchanged copy on no match). Cannot be combined with a non-scalar-lvalue target restriction - any expression is a valid target under `/r`.
- `g` - global: replace every non-overlapping match, not just the first.
- `c` - accepted for syntactic symmetry with `m//` but has no behavioural effect on `s///`; triggers a warning under `use warnings`.

Match-side flags in one line: `m` (multiline ^/$), `s` (`.` matches newline), `i` (case-insensitive), `x` / `xx` (extended whitespace, comments), `p` (preserve match variables; obsolete since 5.20), `o` (compile pattern once), `d` / `u` / `a` / `l` (character-set semantics), `n` (non-capturing by default). See the *modifiers chapter*.

Delimiters

Any non-whitespace character may replace `/`. If the chosen delimiter is one of `(`, `[`, `{`, or `<`, the operator takes **two** bracketed groups and a delimiter between them is optional:

```
s(foo)(bar)
s{foo}{bar}
s<foo>/bar/
s[foo]{bar}g
```

Single-quote delimiters turn off interpolation in both `PATTERN` and `REPLACEMENT` (unless `/e` is in force, which always parses `REPLACEMENT` as code):

```
s'$name'$value';    # literal $name → literal $value, no
 ↪interpolation
```

Backticks are ordinary delimiters; `s`x`y`` does not run a shell command.

When the delimiter is a character that is also an identifier character, a space after the `s` is required so the parser does not read it as part of the operator name.

Examples

Basic in-place substitution on `$_`:

```
$_ = 'the quick brown fox';
s/quick/slow/;                  # $_ is now 'the slow brown fox'
```

Explicit target with =~:

```
my $path = '/usr/bin/perl';
$path =~ s|/usr/bin|/usr/local/bin|;
# $path is '/usr/local/bin/perl'
```

Global replacement, capturing the count:

```
my $text = 'one two two three two';
my $n = ($text =~ s/two/2/g);   # $text: 'one 2 2 three 2'; $n == 3
```

Non-destructive `/r` - leaves the source alone, returns the copy. Idiomatic in `map`:

```
my @clean = map { s/\s+\z//r } @lines;   # @lines untouched
```

Code evaluation with `/e`. The right-hand side is a Perl expression whose value becomes the replacement:

```
my $s = 'abc123xyz';
$s =~ s/\d+/$& * 2/e;           # $s is 'abc246xyz'

my %percent = (n => "\n", t => "\t");
$s =~ s/%(.)/$percent{$1} || $&/ge;     # expand %-escapes
```

Chained `/r` - pure-functional transforms with no intermediate variable:

```
my $out = $text =~ s/foo/bar/r
                =~ s/baz/qux/r;
```

Capture groups and backreferences in the replacement - note the `$1` form, not `\1`:

```
# swap the first two whitespace-separated fields
s/(\S+)\s+(\S+)/$2 $1/;
```

The `1 while` idiom for substitutions that create new matches:

```
# insert commas into an integer from the right
my $n = '1234567890';
1 while $n =~ s/(\d)(\d\d\d)(?!\d)/$1,$2/;
# $n is '1,234,567,890'
```

Edge cases

- **Target must be a scalar lvalue** (unless `/r`): a variable, element, or `substr` lvalue. `s/x/y/` against a string literal or the result of a function call is a compile-time error. Under `/r` any expression is fair game since nothing is modified.
- **Empty pattern re-uses the last successful match.** `s//new/` substitutes against whatever `PATTERN` most recently matched in the enclosing scope. Convenient inside `while (m/.../g) { s//x/ }`, surprising when unrelated code has run in between.
- **Interpolation happens at runtime.** `s/$foo/$bar/` recompiles the pattern every call if `$foo` changes. Use `/o` to compile once and freeze the captured value - or precompile with `qr`, which is the modern alternative.
- **`/e` parses the replacement at compile time** even though it runs at match time. Syntax errors surface when the script is loaded, not when the substitution fires. Each extra `e` wraps one more `eval` around the value.
- **`/g` with an empty match advances by one character** to avoid an infinite loop: `s/\b/-/g` inserts between every token, it does not hang at the first position.
- **`!~` flips the return**, not the work. `$x !~ s/A/a/g` still modifies `$x`; it just returns true when no `A` was found.
- **Delimiter matching for brackets** uses the mirrored closer: `s{pat}{rep}`, `s<pat><rep>`. A delimiter other than a bracket must appear three times total: `s/pat/rep/`.
- **`/c` is a no-op on `s///`.** Accepted for parser symmetry with `m//`; using it produces a warning under `use warnings` and nothing more.

- **Assignment-as-target idiom.** `($copy = $src) =~ s/x/y/g` populates `$copy` with the modified value and leaves `$src` alone. Modern code prefers `my $copy = $src =~ s/x/y/gr`.
- **Tied or overloaded targets.** Without `/r`, `STORE` is invoked after the substitution to write the new value back; the fetched value is copied into a plain buffer before the pattern is applied. With `/r` the result is always a plain string - tie / overloading is not preserved in the return.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `m//` - the matching operator; shares match-side flag semantics and return-vs-context rules
- `tr///` - character-by-character transliteration; much faster when the job is literal character translation, not a regex
- `qr//` - precompile a pattern once and interpolate it into many `s///` or `m//` calls; the modern alternative to `/o`
- `split` - when the goal is to break a string on a pattern rather than replace matches, reach for `split` instead
- *Substitution chapter* - the chapter dedicated to `s///`; covers `/e`, `/r`, `/g`, `\K`, and zero-length-match handling
- *Regular expressions guide* - the full pattern-language reference; every match-side flag on `s///` is defined there
- `$&` - the matched text, available inside the `REPLACEMENT` of every `s///`

I/O

say

Print a list of values followed by a newline.

`say` is `print` with the output record separator forced to `"\n"` for that one call. It stringifies each item in `LIST`, joins them with `$,` (output field separator, usually empty), writes them to `FILEHANDLE`, and appends a literal `"\n"` - **not** the value of `$\`. If `FILEHANDLE` is omitted, output goes to the currently selected handle (see `select`), which defaults to `STDOUT`. If `LIST` is omitted, `$_` is printed.

`say` is a feature-gated built-in. It is available only when the `"say"` feature is enabled - automatic under `use v5.10` or higher, or explicit with `use feature 'say'` - or when called as `CORE::say`.

Synopsis

```
say LIST
say FILEHANDLE LIST
say {EXPR} LIST
say
```

What you get back

1 on success, `undef` on failure (with `$!` set). As with `print`, the return value is rarely checked for `STDOUT` or `STDERR`. Check it when writing to a file, pipe, or socket where a write failure is a recoverable condition:

```
say $fh $record
    or die "write to $path failed: $!";
```

Default filehandle, default separators

`say` reads the same global state as `print`, with one deliberate exception:

- The **currently selected filehandle**, changed with `select`. `say LIST` writes there when no filehandle is given. Defaults to `STDOUT`.
- `$,` - **output field separator**, inserted between list items. Defaults to the empty string, so `say 1, 2, 3` produces `"123\n"`, not `"1 2 3\n"`.
- `$\` - **output record separator**. `say` **ignores `$\` entirely** and appends a literal `"\n"` instead. Setting `$\` has no effect on `say`.

That last point is the whole reason `say` exists: one record, one newline, no fiddling with `$\`.

Examples

Basic write to `STDOUT`:

```
use feature 'say';
say "hello, world";                 # "hello, world\n"
```

Write to a filehandle. Same indirect-object form as `print` - **no comma** between the filehandle and the list:

```
open my $fh, ">", "out.txt" or die $!;
say $fh "line 1";
say $fh "line 2";
close $fh;
```

Write to a filehandle held in an expression. Any filehandle more complex than a bare-word or a plain scalar needs the `{EXPR}` block form:

```
my @logs = (\*STDOUT, \*STDERR);
say { $logs[$level] } "message";
```

`say` joins list items with `$,` and then appends `"\n"` regardless of `$\`:

```
local $, = ", ";
local $\ = ".\n";                    # ignored by say
say "apples", "pears", "plums";      # "apples, pears, plums\n"
```

No arguments - print `$_` plus a newline. The filehandle must be a bareword in this form; `say $fh;` would parse `$fh` as the content, not as the target:

```
for (qw(alpha beta gamma)) {
    say;                             # "alpha\n", "beta\n", "gamma\n"
}
```

List context applies to every argument:

```
my @arr = (10, 20, 30);
say @arr;                            # "102030\n"
say scalar @arr;                     # "3\n"
```

Edge cases

- **Feature gate**: without `use feature 'say'` or `use v5.10`, `say` is parsed as an ordinary subroutine call and will fail at compile time if no such sub is in scope. Use `CORE::say(...)` to call it unconditionally.
- **No arguments with an explicit filehandle**: `say FH;` prints `$_` to `FH`, but only for **bareword** filehandles. `say $fh;` parses `$fh` as the content, not the target - write `say $fh $_;` if you mean that. This matches `print`.
- **`$\` is ignored, not overridden**: `say` does not set or `local`ise `$\`. Other code that reads `$\` (including nested `print` calls) sees whatever value was already there.
- **Filehandle-or-first-arg ambiguity**: same rule as `print`. A bareword or a simple scalar immediately followed by a term (no comma) is taken as the filehandle. An expression returning a filehandle needs the block form: `say { $handles[0] } "x";`.
- **`say (…)` parses as a function call**. `say (2+3)*4` prints 5 then discards `*4`. Parenthesise the whole argument list, or prefix with `+`:

  ```
  say((2+3)*4);                      # "20\n"
  say +(2+3)*4;                      # "20\n"
  ```

- **Closed filehandle**: returns `undef` and sets `$!` to `"Bad file descriptor"`. Under `use warnings` a `say() on closed filehandle` warning is emitted.
- **Encoding layers**: `say` writes through the handle's PerlIO stack exactly like `print`. A handle with a `:utf8` or `:encoding(...)` layer encodes on the way out; the appended newline is a single `"\n"` byte and is encoded by the same layer. A wide-character string written to a bytes-mode handle triggers a `Wide character in say` warning.
- **Autoflush**: `say` buffers through PerlIO like `print`. Set `$|` on the selected handle for line-buffered output.

- **SIGPIPE**: writing to a closed pipe or socket raises SIGPIPE. The default action terminates the process; install a handler to convert it into a normal write failure.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `print` - same targeting and separator rules, but appends `$\` (empty by default) instead of a forced newline
- `printf` - formatted output; does not append a newline, you put one in the format string
- `select` - change which filehandle the no-argument form of `say` writes to
- `$,` - output field separator inserted between list items; `say` honours it
- `$\` - output record separator; `say` **ignores it** and appends `"\n"` instead
- `$|` - autoflush flag for the currently selected handle

Misc

scalar

Force EXPR to be evaluated in scalar context and return its value.

`scalar` is not really an operator in its own right - it is a context-coercion marker the parser recognises in front of a unary expression. Perl evaluates the expression in scalar context and yields whatever that expression produces there. The function exists because most positions in a Perl program already impose a context (list or scalar) on their operands, and occasionally you need to override list context with scalar context explicitly.

Synopsis

```
scalar EXPR
scalar @array
scalar %hash
scalar localtime
```

What you get back

The value of EXPR as evaluated in scalar context. The exact shape depends entirely on what EXPR is:

- For an array, the element count.
- For a hash, a true value (non-empty) or `0` (empty); in older Perls this was a bucket-usage string, but from 5.26 onward it is the key count.

- For a list-context-sensitive function, whatever that function returns when called in scalar context (for example, `localtime` returns its ctime-style string instead of the nine-element list).
- For an already-scalar expression, the same value - `scalar` is a no-op there.

Global state it touches

None. `scalar` is a pure context cast; it does not read or write any special variables of its own. The *expression* you pass in may, of course, touch whatever globals it normally touches.

Examples

Count the elements of an array inside a list-context position:

```
my @items = (10, 20, 30);
my @counts = (scalar @items);        # (3)
print scalar @items, "\n";           # 3
```

Without `scalar`, `@items` would expand to its elements and `@counts` would be `(10, 20, 30)`.

Print the human-readable date from `localtime`. In list context `localtime` returns nine fields; in scalar context it returns the familiar ctime string:

```
print localtime, "\n";               # nine numbers run together
print scalar localtime, "\n";        # "Thu Apr 23 14:02:11 2026"
```

The same trap appears everywhere a builtin is list-context-sensitive:

```
my @parts = localtime;               # (sec, min, hour, mday, ...)
my $now   = scalar localtime;        # ctime string
```

Test whether a hash is non-empty without allocating its key list:

```
if (scalar %config) {
    # %config has at least one key
}
```

`if (%config)` already imposes scalar (boolean) context, so the `scalar` here is redundant but explicit; writers sometimes include it to make intent visible to readers.

Build a list of counts from several arrays in one expression:

```
my @counts = (scalar @a, scalar @b, scalar @c);
```

Coerce a function call that would otherwise splat into a surrounding list:

```
my @row = ($id, scalar get_tags($id), $mtime);
```

Without `scalar`, `get_tags` would run in list context and its returned list would be flattened into `@row` alongside `$id` and `$mtime`.

Edge cases

- **No list-context counterpart.** There is no `list EXPR` operator; list context is the default in almost every position where you might want to impose it. If you genuinely need to coerce a scalar expression into list context, the idiom is `@{[ EXPR ]}`, though in most cases plain `(EXPR)` is enough.
- **Parenthesised list trap.** `scalar` is a unary operator. Passing it a parenthesised comma expression does **not** evaluate that expression as a list; it evaluates it as a *scalar comma operator*, which runs every sub-expression in void context except the last and returns the last in scalar context:

```
print uc(scalar(foo(), $bar)), $baz;
# equivalent to:
foo();                                  # void context
print(uc($bar), $baz);                  # $bar in scalar context
```

  If you wanted `foo()`'s return values included, `scalar (list)` is the wrong tool.
- **Hash in scalar context.** From Perl 5.26 onward, `scalar %h` returns the number of keys, the same as `scalar keys %h`. On older Perls it returned a string like `"4/8"` reporting bucket usage; pperl targets 5.44 and so always returns the key count.
- **Subroutine prototypes.** A subroutine declared with a scalar prototype (`($)`) already imposes scalar context on its argument, so `scalar` is redundant in front of the call. A `(@)` or `(\@)` prototype does not, so `scalar` still matters:

```
sub count ($) { $_[0] }
count @items;                           # @items in scalar context:␣
↪3
```

- **Redundant on already-scalar expressions.** `scalar $x`, `scalar 42`, `scalar "text"` are no-ops. Perl does not warn about them; they are occasionally used for emphasis but add nothing.
- **Not a function call.** Because `scalar` is a parser-level context marker, you cannot take `\&scalar`, cannot override it with a `sub scalar { ... }`, and cannot pass it as a code reference.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `wantarray` - inspect the context you were called in, rather than impose one on a sub-expression
- `defined` - similarly a unary operator applied to an expression; useful alongside `scalar` when testing function results
- `localtime` - canonical example of a builtin whose return shape depends on context

- `reverse` - also context-sensitive; `scalar reverse $s` reverses characters, `reverse @a` reverses a list
- `keys` - `scalar keys %h` is the long form of `scalar %h` with the same result on 5.26+

I/O

seek

Reposition a filehandle for random-access reads or writes.

`seek` moves `FILEHANDLE`'s read/write pointer to a new byte offset, mirroring the C `fseek(3)` call. After a successful `seek`, the next `read`, `readline`, or `print` on the handle starts from the new position. `POSITION` is a signed byte offset; `WHENCE` selects the anchor it is measured from.

Synopsis

```
seek FILEHANDLE, POSITION, WHENCE
seek $fh, 0, 0             # rewind to start
seek $fh, 0, 1             # no-op: clear EOF, keep position
seek $fh, -1024, 2         # 1024 bytes before EOF
```

What you get back

1 on success, a false value on failure (with `$!` set). Always check the return value - seeking past a short file, on an unseekable handle (pipe, socket, `TTY`), or after a write error all fail here rather than at the next read.

```
seek $fh, $offset, 0
    or die "seek to $offset failed: $!";
```

WHENCE values

`WHENCE` is an integer with three meaningful values. Use the symbolic constants from `Fcntl` for readability:

- `0` / `SEEK_SET` - `POSITION` is measured from the **start of the file**. `POSITION` must be non-negative.
- `1` / `SEEK_CUR` - `POSITION` is added to the **current position**. Negative values move backward, positive values forward. `seek $fh, 0, 1` is the canonical "move nowhere, but clear EOF" idiom.
- `2` / `SEEK_END` - `POSITION` is added to the **end-of-file** offset. `POSITION` is typically zero or negative.

```
use Fcntl qw(SEEK_SET SEEK_CUR SEEK_END);

seek $fh, 0,     SEEK_SET;   # rewind
```

(continues on next page)

(continued from previous page)

```
seek $fh, 0,      SEEK_END;   # go to EOF (e.g. to append)
seek $fh, -$n,    SEEK_CUR;   # $n bytes back from here
```

Bytes, not characters

Even when the handle has a character-oriented layer such as `:encoding(UTF-8)`, `seek`, `tell`, and the `sysseek` family operate on **byte offsets**. A `seek` to a byte offset that lands in the middle of a multi-byte sequence will produce decoding errors or replacement characters on the next read.

If you need to position by characters, read forward from a known byte boundary rather than trying to translate character counts to byte offsets. `tell` values are safe to feed back into `seek` because they were produced at byte boundaries the I/O layer already crossed cleanly.

Global state it touches

- `$!` - set to the system error message on failure.
- The filehandle's PerlIO buffer is discarded on success, so any data read ahead by buffering is dropped and the next read pulls fresh bytes from the underlying file.
- The end-of-file flag on `FILEHANDLE` is cleared on success, even when `POSITION` and `WHENCE` leave the position unchanged.

Examples

Rewind to the start of a file before re-reading it:

```
seek $fh, 0, 0 or die "rewind failed: $!";
while (my $line = <$fh>) { ... }
```

Append by positioning at end-of-file, then writing. For append-only use, prefer opening with >> - this form is for handles already open for read-write:

```
seek $fh, 0, 2 or die $!;           # SEEK_END
print $fh "appended line\n";
```

Read a fixed-width record by index, where each record is 128 bytes:

```
my $record = 42;
seek $fh, $record * 128, 0 or die $!;
read $fh, my $buf, 128;
```

Emulate `tail -f` - the `seek $fh, 0, 1` resets the EOF condition so the next `readline` retries the file for new data:

```
while (1) {
    while (my $line = <$fh>) { print $line }
    sleep 1;
```

(continues on next page)

(continued from previous page)

```
    seek $fh, 0, 1;                 # clear EOF, keep position
}
```

Save a position with `tell`, read ahead, then restore:

```
my $mark = tell $fh;
my $peek = <$fh>;
seek $fh, $mark, 0 or die $!;       # back to where we were
```

Edge cases

- **Unseekable handles**: pipes, sockets, TTYs, and most special devices return false and set `$!` to `ESPIPE` (`Illegal seek`). Check the return value; do not assume every filehandle is seekable.
- **Mixing reads and writes on the same handle** on a file opened `+<` or `+>` requires a `seek` (or `tell`, or an explicit flush) when switching direction. A WHENCE of 1 with `POSITION` of `0` is the usual no-op separator:

  ```
  seek $fh, 0, 1;                   # allowed to switch read <->␣
  ↪write
  ```

- **Seeking past EOF** on a file opened for writing is legal and creates a sparse hole up to `POSITION` on filesystems that support holes; the next write fills part of the hole and the bytes in between read back as `"\0"`.
- **`sysread` / `syswrite` users**: do **not** mix `seek` with `sysread` or `syswrite`. `seek` operates on the PerlIO buffer layer; the unbuffered sys-family bypasses it, so their effective positions drift apart. Use `sysseek` for handles you access through the sys-family.
- **Character offsets via arithmetic**: multiplying a character count by an assumed encoding width is wrong for `UTF-8` and other variable-width encodings. Use byte offsets obtained from `tell`, or read forward from a known boundary.
- **Directory handles**: `seek` does not work on directory handles. Use `seekdir` with a position from `telldir`.
- **Closed filehandle**: returns a false value and sets `$!`; under `use warnings` a `seek() on closed filehandle` warning is emitted.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `tell` - read the current byte offset; the value round-trips safely through `seek` with `WHENCE 0`
- `sysseek` - unbuffered seek for handles accessed via `sysread` / `syswrite`; use this instead of `seek` when bypassing PerlIO

- `read` - buffered fixed-length read; the natural partner for positioning by byte offset
- `readline` - line-oriented read; `seek $fh, 0, 1` before a retry is the `tail -f` idiom
- `eof` - test for end-of-file; `seek` clears the EOF flag a prior read may have set
- `Fcntl` - source of the `SEEK_SET`, `SEEK_CUR`, `SEEK_END` constants

I/O

seekdir

Restore a directory handle to a position previously captured by `telldir`.

`seekdir` moves the read cursor of `DIRHANDLE` to `POS`, so that the next `readdir` call returns the entry that would have been returned when `POS` was first obtained from `telldir` on the same handle. It is the directory counterpart of `seek` on a file handle - with the crucial restriction that `POS` is not a byte offset you can compute, it is an opaque cookie you must have taken from `telldir` earlier.

Synopsis

```
seekdir DIRHANDLE, POS
```

What you get back

No meaningful value. `seekdir` returns what the underlying `seekdir(3)` returns, which on Linux is nothing useful - do **not** branch on the return value. To detect trouble, check `$!` after the call, or verify that the handle is still open before calling.

Typical use

You are walking a directory, notice an entry you want to revisit later, and remember where you were so you can return to it after reading on:

```
opendir my $dh, $path or die "opendir $path: $!";

my %bookmarks;
while (my $entry = readdir $dh) {
    $bookmarks{$entry} = telldir $dh  # position AFTER reading
 ↪$entry
        if $entry =~ /\.conf\z/;
}

# later: re-read the entry that follows the first .conf file
seekdir $dh, $bookmarks{(sort keys %bookmarks)[0]};
my $next = readdir $dh;

closedir $dh;
```

Returning to the start is the common special case - a `seekdir $dh, 0` call is equivalent to `rewinddir` on conforming systems, but `rewinddir` is the idiomatic spelling for that intent:

```
seekdir $dh, 0;      # works, but say what you mean:
rewinddir $dh;       # clearer
```

Edge cases

- **POS must come from `telldir` on the same handle.** Passing an integer you made up, a value from a different handle, or a stale value from a handle that has since been closed and re-opened is undefined behaviour. The kernel may accept it silently and hand you garbage on the next `readdir`, skip entries, or return duplicates.
- **Directory compaction between `telldir` and `seekdir`**: if the filesystem has compacted or reorganised the directory between the two calls (files created, deleted, or renamed in the same directory by any process), the POS cookie may no longer refer to the same entry - or to any entry at all. This is a property of the underlying `seekdir(3)` and is not a pperl limitation. For stable iteration over a changing directory, snapshot the list with `readdir` in list context and work off the copy.
- **Unopened or closed handle**: sets `$!`. Under `use warnings` you get a `seekdir() attempted on invalid dirhandle` warning. Nothing in the return value tells you this happened - check `$!` or use `defined fileno($dh)` / explicit bookkeeping if you need to guard.
- **Not a byte offset**: POS is an opaque token. Arithmetic on it (`seekdir $dh, $pos + 1`) is meaningless; the only safe values are `0` (start of directory, and even then prefer `rewinddir`) and values returned by `telldir`.
- **Entry order is unspecified**: `seekdir` replays whatever order the kernel originally returned. That order is not sorted, not alphabetical, and not guaranteed to match a fresh `opendir` of the same path later. If your code sorts, sort explicitly each time.
- **SEEK_CUR-style relative seeks do not exist** for directory handles. There is no equivalent of `seek`'s whence argument.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `telldir` - produce the POS cookie that `seekdir` consumes; the two are only useful as a pair
- `readdir` - read the next entry; resumes from the position `seekdir` just set
- `rewinddir` - the idiomatic way to return to the start of a directory; prefer it over `seekdir $dh, 0`
- `opendir` - open the directory and obtain the handle `seekdir` operates on

- `seek` - the file-handle analogue, but with a true byte offset and a `whence` argument; directory handles have neither

I/O · Filehandles, files, directories

select

Either set the default output filehandle, or call the `select(2)` syscall for I/O multiplexing. Same name, two unrelated jobs - disambiguated by argument count.

The one-argument form (or no-argument form) is about where `print`, `printf`, `say`, and `write` send their output when you don't name a filehandle. The four-argument form is a direct wrapper around the BSD `select(2)` syscall for waiting on sets of file descriptors.

Synopsis

```
select FILEHANDLE              # set default output handle, return␣
 ↪old
select                         # return current default output␣
 ↪handle
select RBITS, WBITS, EBITS, TIMEOUT   # select(2) syscall
```

What you get back

`select FILEHANDLE` returns the *previously* selected filehandle (as a typeglob), so the idiomatic save/restore pattern works:

```
my $old = select $fh;
$| = 1;
select $old;
```

`select` (no arguments) returns the currently selected filehandle. At program start this is `STDOUT`.

`select RBITS, WBITS, EBITS, TIMEOUT` returns different things in different contexts:

- In scalar context: the number of filehandles ready, or `-1` on error with `$!` set.
- In list context: `($nfound, $timeleft)` - the same `$nfound` plus the remaining time from `TIMEOUT`. Many kernels do not update `$timeleft`; on those systems it is always equal to the input `TIMEOUT`.

On timeout expiry `$nfound` is `0`. The three bit masks (`RBITS`, `WBITS`, `EBITS`) are updated in place to indicate which descriptors are ready - pass copies if you want to preserve the original masks.

Global state it touches

The one-argument form changes one piece of interpreter-global state: the currently selected output filehandle. While that handle is selected, these variables refer to **it**, not to `STDOUT`:

- `$|` - autoflush flag

- `$,` - output field separator
- `$\` - output record separator
- `$#`, `$%`, `$=`, `$-`, `$~`, `$^` - format-related variables

That is why `my $old = select($fh); $| = 1; select($old);` works: `$|` is a *property of the selected handle*, not a global scalar.

The four-argument form touches no per-interpreter state. It may leave `$!` set on error.

Examples

Enable autoflush on a handle without disturbing the default:

```
my $old = select $fh; $| = 1; select $old;
```

Redirect `print` temporarily and restore on block exit:

```
{
    my $old = select $log_fh;
    print "entry: $msg\n";       # goes to $log_fh
    select $old;
}
```

Sleep for 250 milliseconds using the four-argument form:

```
select undef, undef, undef, 0.25;
```

Wait up to 5 seconds for `STDIN` to become readable:

```
my $rin = '';
vec($rin, fileno(STDIN), 1) = 1;
my $nfound = select my $rout = $rin, undef, undef, 5;
if ($nfound) {
    my $line = <STDIN>;
}
```

Build a read mask over multiple handles:

```
sub fhbits {
    my $bits = '';
    vec($bits, fileno($_), 1) = 1 for @_;
    return $bits;
}
my $rin = fhbits(\*STDIN, $sock);
my ($nfound, $timeleft) =
    select my $rout = $rin, undef, undef, 10;
```

Edge cases

- **Method-style alternative.** `$fh->autoflush(1)` sets the autoflush flag on `$fh` without changing the default handle. Prefer it when you only need to tweak one property of one handle - no global state involved.
- **Typeglob localization is cleaner for scoped redirects.** `local *STDOUT = $fh` rebinds `STDOUT` itself for the dynamic scope; on `die` the original is restored automatically. `select` does **not** restore on `die`. The two are not equivalent: with `select` an explicit `print STDOUT ...` still reaches the *real* `STDOUT`; with the `local *STDOUT` form, even `print STDOUT ...` goes to the rebound handle.
- **Bit-mask form requires bytes, not a list of fileno integers.** A bit mask is a packed bit string built with `vec`; passing a list of integers or a reference silently produces wrong results (or an empty mask).
- **`undef` for a mask means "don't wait on this set."** Pass `undef` for masks you don't care about. An empty string `''` technically also works but is less idiomatic.
- **Fractional timeouts work.** `TIMEOUT` is a float in seconds. `0.25` is 250 ms; `0` returns immediately (polling).
- **`TIMEOUT` of `undef`** means block indefinitely until at least one descriptor is ready or a signal arrives.
- **Masks are modified in place.** On return, each mask holds only the bits for descriptors that are ready. If you need the originals afterwards, pass copies - the idiom `my $rout = $rin` on the call line does exactly that.
- **Do not mix buffered I/O with the four-argument form.** `select` reports readiness at the kernel level; `readline`, `read`, and the diamond operator `<>` read through PerlIO buffers that may already hold unread data the kernel doesn't know about. Use `sysread` for data once `select` says a descriptor is ready.
- **Spurious readiness on sockets.** On some Unixes `select(2)` may report a socket as "ready for reading" when a following `read` would still block. Set `O_NONBLOCK` on the socket if this matters.
- **Restart-after-signal behaviour is system-dependent.** A signal (e.g. `SIGALRM`) may cause `select` to return early with `$! == EINTR`, or the kernel may restart the call transparently. Portable code checks for `EINTR` explicitly.
- **Argument count disambiguates the two forms.** One argument (or none) is the default-handle form; exactly four is the syscall form. Other counts are a syntax error.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `print` - reads the selected handle when no filehandle is given; `select` is how you change which handle that is
- `printf` - same default-handle rule as `print`

- `write` - format-based output; the top-of-form and page-length variables follow the selected handle, not `STDOUT`
- `fileno` - maps a filehandle to the integer descriptor number you need when building a mask
- `vec` - packs descriptor numbers into the bit string the four-argument form expects
- `sysread` - the unbuffered read you use after the four-argument `select` says a descriptor is ready
- `IO::Select` - object wrapper over the four-argument form that hides the bit-mask arithmetic

SysV IPC

semctl

Perform a control operation on a System V semaphore set.

`semctl` is the Perl-level binding for the `semctl(2)` system call. It reads or modifies the state of the semaphore set identified by `ID` - getting or setting a single semaphore's value, fetching or installing the full value vector, reading the associated `semid_ds` metadata, or removing the set entirely. `CMD` selects the operation; `SEMNUM` selects which semaphore within the set (ignored for set-wide commands like `IPC_STAT`, `IPC_RMID`, `GETALL`, `SETALL`); `ARG` is the command's input or output buffer.

`semctl` is a thin wrapper - the semantics, the command constants, and the shape of `ARG` all come straight from the C-level call. You will almost always want:

```
use IPC::SysV;
```

first, to pull in the `IPC_STAT`, `IPC_RMID`, `GETVAL`, `SETVAL`, `GETALL`, `SETALL`, `GETPID`, `GETNCNT`, `GETZCNT` constants.

Synopsis

```
semctl $id, $semnum, $cmd, $arg
semctl $id, 0,       IPC_RMID, 0           # destroy set
semctl $id, $n,      SETVAL,   $value      # set one
semctl $id, 0,       SETALL,   $packed     # set all
```

What you get back

Return follows the same convention as `ioctl`:

- `undef` on error (with `$!` set).
- The string `"0 but true"` when the underlying call returned `0` - this string is false numerically (so it prints `0` in numeric context) but true in boolean context, letting you distinguish success-with-zero from failure in one test.
- The actual integer return value otherwise.

The idiomatic test is therefore:

```
my $rc = semctl($id, $n, GETVAL, 0);
defined $rc or die "semctl: $!";
```

Check `defined`, not truth - a legitimately zero semaphore value would fail a plain `if ($rc)`.

For `CMD` values that read data into `ARG` (`IPC_STAT`, `GETALL`), `ARG` must be a **modifiable lvalue** (a plain variable, not a literal or expression); on success the buffer is overwritten in place with the returned bytes and you unpack it yourself.

Global state it touches

Sets `$!` on failure. No other interpreter globals involved.

Examples

Read a single semaphore's current value:

```
use IPC::SysV qw(GETVAL);
my $value = semctl($id, $semnum, GETVAL, 0);
defined $value or die "GETVAL: $!";
```

Set a single semaphore:

```
use IPC::SysV qw(SETVAL);
semctl($id, $semnum, SETVAL, 1)
    or die "SETVAL: $!";
```

Read all semaphore values at once. The buffer has to be pre-allocated to the right size - `nsems` native shorts - before the call. `pack` with an unused count is the usual way:

```
use IPC::SysV qw(GETALL);
my $nsems = 4;
my $buf   = pack("s!*", (0) x $nsems);      # allocate room
semctl($id, 0, GETALL, $buf) or die $!;
my @vals = unpack("s!*", $buf);
```

Install a full set of values:

```
use IPC::SysV qw(SETALL);
my $buf = pack("s!*", 1, 0, 0, 1);
semctl($id, 0, SETALL, $buf)
    or die "SETALL: $!";
```

Fetch the `semid_ds` structure for the set. `ARG` is written in place with the packed struct; its layout is platform-specific, so unpack with the help of `IPC::Semaphore` rather than by hand unless you have to:

```
use IPC::SysV qw(IPC_STAT);
my $ds = "";
semctl($id, 0, IPC_STAT, $ds)
    or die "IPC_STAT: $!";
```

Destroy the semaphore set. SEMNUM and ARG are unused; pass 0:

```
use IPC::SysV qw(IPC_RMID);
semctl($id, 0, IPC_RMID, 0)
    or die "IPC_RMID: $!";
```

Edge cases

- **ARG must be a real variable for read commands.** GETALL and IPC_STAT write into ARG; passing a literal (`semctl($id, 0, GETALL, "")`) or a read-only expression is a runtime error. The variable's existing contents are overwritten, not appended to.
- **ARG must be pre-sized.** The kernel writes exactly `nsems * sizeof(short)` bytes for GETALL, or `sizeof(struct semid_ds)` for IPC_STAT. Passing a shorter buffer is undefined behaviour in the underlying C call; `pack("s!*", (0) x $nsems)` is the standard idiom for GETALL.
- **Native short, not portable short.** Semaphore values are C `short` (signed, typically 16-bit). Use `s!` in pack / unpack - the `!` picks the platform-native type. Plain `s` forces little-endian 16-bit and will produce wrong results on big-endian or LP64-short systems.
- **"0 but true" returns.** GETVAL on a semaphore whose value is 0, GETPID/GETNCNT/GETZCNT returning zero, and other zero-valued successes all return the literal string `"0 but true"`. Arithmetic contexts see 0; boolean contexts see true. Use `defined` to check for failure.
- **SEMNUM is ignored for set-wide commands.** IPC_STAT, IPC_RMID, SETALL, GETALL act on the whole set. Pass 0 by convention; passing any other value is harmless but obscures intent.
- **After IPC_RMID the id is invalid.** Every subsequent `semctl` or semop on `$id` fails with EINVAL / EIDRM. Do not cache the id past removal.
- **The set is kernel-global and persists across process exits.** A set created by a crashed script remains until someone calls IPC_RMID (or the system reboots). Use `ipcs -s` / `ipcrm -s` from the shell to inspect or clean up leaked sets.
- **Permissions come from semget.** The `IPC_CREAT | mode` passed at creation time controls who may issue which `semctl` commands. EACCES from `semctl` means the mode bits disallow the requested operation for the current effective uid.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- semget - create or look up the semaphore set whose id you then pass to `semctl`
- semop - the actual wait / signal operations; `semctl` is for control and inspection, semop does the work

- `msgctl` - the analogous control call for System V message queues; same return-value convention
- `shmctl` - the analogous control call for System V shared memory; same return-value convention
- `pack` - how to build the ARG buffer with `s!` for semaphore values and the `semid_ds` struct
- `IPC::Semaphore` - object wrapper that hides `semctl` behind named methods; reach for it when a script does more than one or two calls

SysV IPC

semget

Create or look up a System V semaphore set and return its identifier.

`semget` is the entry point into the SysV semaphore API. It wraps the `semget(2)` system call: given a numeric KEY that names the set, the number of semaphores NSEMS in the set, and a FLAGS word combining permission bits with creation flags, the kernel either returns the id of a matching existing set or, if `IPC_CREAT` is in FLAGS, creates a new set and returns its id. The returned id is the handle every other semaphore call (`semop`, `semctl`) expects.

Synopsis

```
semget KEY, NSEMS, FLAGS
```

What you get back

The semaphore set identifier (a non-negative integer) on success, or `undef` on failure with `$!` set to the `errno` from the underlying `semget(2)` call. The id is **not** a file descriptor - it is a kernel-wide SysV IPC handle that outlives your process and persists until explicitly removed with `semctl($id, 0, IPC_RMID, 0)`.

```
use IPC::SysV qw(IPC_CREAT IPC_PRIVATE);

my $semid = semget(IPC_PRIVATE, 3, IPC_CREAT | 0600)
    // die "semget: $!";
```

Global state it touches

- `$!` - set to the `errno` value from `semget(2)` on failure (EACCES, EEXIST, ENOENT, ENOSPC, EINVAL, ENOMEM).

The KEY argument

KEY is an integer that names the semaphore set system-wide. Three conventions matter:

- **`IPC_PRIVATE`** (imported from `IPC::SysV`) - the kernel allocates a fresh key unused by any other set. The resulting id is known only to this process and anyone it passes

the id to, typically children after `fork`. Use this when the set is an implementation detail of one process tree.

- **A fixed integer literal** - every cooperating process hard-codes the same number. Fragile across unrelated programs that might pick the same integer.
- **A key derived from `ftok(3)`** - `IPC::SysV::ftok($path, $proj_id)` turns a filesystem path plus a project byte into a numeric key. The standard recipe for unrelated processes that need to agree on a key without hard-coding a magic number.

The FLAGS argument

`FLAGS` combines **permission bits** (low 9 bits, like a `chmod` mode: `0600`, `0660`, `0666`) with **creation flags** from `IPC::SysV`:

- `IPC_CREAT` - create the set if it does not already exist. Without this flag, `semget` only looks up an existing set with the given key.
- `IPC_CREAT | IPC_EXCL` - create the set, but fail with `EEXIST` if one already exists for `KEY`. The SysV analogue of `O_CREAT | O_EXCL` in `open`.

Permission bits are checked by the kernel on every subsequent `semop` and `semctl` call against the invoking process's uid/gid.

Examples

Create a private set of three semaphores usable by this process and its children:

```
use IPC::SysV qw(IPC_PRIVATE IPC_CREAT);
my $semid = semget(IPC_PRIVATE, 3, IPC_CREAT | 0600)
    // die "semget: $!";
```

Look up an existing, named set without creating it. `NSEMS` must match the size the set was created with, or can be `0` if you do not care:

```
use IPC::SysV qw(ftok);
my $key   = ftok("/var/run/myapp.pid", ord('M'));
my $semid = semget($key, 0, 0) // die "no set for key $key: $!";
```

Create-or-open idiom - try exclusive create, fall back to lookup. The first process through initialises the set; later ones just attach:

```
use IPC::SysV qw(ftok IPC_CREAT IPC_EXCL);

my $key = ftok("/var/run/myapp.pid", ord('M'));
my $semid = semget($key, 4, IPC_CREAT | IPC_EXCL | 0660);
if (defined $semid) {
    # We created it - initialise every semaphore to 1.
    semctl($semid, 0, SETALL, pack("s!*", (1) x 4))
        or die "SETALL: $!";
} else {
    $semid = semget($key, 4, 0) // die "semget lookup: $!";
}
```

Remove a set when you are done with it. SysV sets leak across process exit - nothing cleans them up for you:

```
use IPC::SysV qw(IPC_RMID);
semctl($semid, 0, IPC_RMID, 0) or warn "semctl RMID: $!";
```

Edge cases

- **`NSEMS` on lookup**: when looking up an existing set (no `IPC_CREAT`), `NSEMS` of `0` means "any size"; a positive value must be less than or equal to the set's actual size or `semget` fails with `EINVAL`.
- **`IPC_PRIVATE` still obeys `FLAGS`**: permission bits apply. A set created with `0400` is read-only even to the creating process for `semop` calls that modify it.
- **Kernel limits**: `semget` can fail with `ENOSPC` when the system-wide semaphore-set limit is reached, or `EINVAL` when `NSEMS` exceeds the per-set maximum. See `ipcs -sl` for the current limits on Linux.
- **Leaked sets**: a set persists until `IPC_RMID` or reboot. Long-running daemons that forget to remove their sets accumulate them; `ipcs -s` lists the survivors, `ipcrm -s $id` removes one.
- **Key collisions**: two unrelated programs picking the same hard-coded `KEY` will see each other's set - hence the `ftok(3)` idiom.
- **Not a filehandle**: `$semid` is an integer, not a handle. Do not pass it to `close`, `fcntl`, or any filehandle operation.
- **Forks inherit ids, not ownership**: a child sees the same `$semid` value as the parent and can use it, but the set itself is owned by the uid that created it, not by any particular process.
- **Numeric context**: the id is a plain integer, safe to print, compare, and store in a file for another process to pick up.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `semop` - perform wait/signal operations on the set returned by `semget`
- `semctl` - control operations on the set: initialise values (`SETALL`, `SETVAL`), read them (`GETALL`, `GETVAL`), and remove the set (`IPC_RMID`)
- `msgget` - the message-queue counterpart, same `KEY`, `FLAGS` conventions
- `shmget` - the shared-memory counterpart, same `KEY`, `FLAGS` conventions
- `IPC::SysV` - imports `IPC_CREAT`, `IPC_EXCL`, `IPC_PRIVATE`, `IPC_RMID`, `SETALL`, `GETALL`, and the `ftok` helper
- `IPC::Semaphore` - object wrapper around the raw `sem*` built-ins when you would rather not pack structs by hand

SysV IPC

semop

Perform one or more System V semaphore operations atomically.

`semop` is the workhorse of SysV semaphore use: it takes a semaphore set identifier (from `semget`) and a packed list of operations, then asks the kernel to apply them all or none. Each operation names a single semaphore within the set, an amount to add to it (typically `-1` to wait, `+1` to signal), and a flag word. The kernel blocks the caller until every requested operation can proceed, then commits them as one unit. This is what makes `semop` usable as a mutual-exclusion or counting-resource primitive between unrelated processes.

Synopsis

```
semop $semid, $opstring
```

What you get back

True on success, false on error with `$!` set. The truth value is the only result - `semop` communicates outcome by the side effect on the semaphore set, not through its return. Check it on every call:

```
semop($semid, $ops)
    or die "semop failed: $!";
```

A false return means no operation in `OPSTRING` was applied; SysV semantics guarantee all-or-nothing.

Global state it touches

Sets `$!` on failure, with the usual errno values from `semop(2)`: `EAGAIN` when `IPC_NOWAIT` was set and the operation would block, `EIDRM` when the semaphore set was removed while waiting, `EINTR` when a signal interrupted the wait, `EINVAL` for a bad semaphore number or identifier, `EACCES` for a permission mismatch against the set's mode.

The operation structure

`OPSTRING` is a concatenation of fixed-width records. Each record is three native-width `short` integers and is produced with `pack`:

```
pack("s!3", $semnum, $semop, $semflag)
```

- `$semnum` - the index of the target semaphore within the set (`0` through `nsems-1`, where `nsems` is the size passed to `semget`).
- `$semop` - the amount to add to the semaphore's value. Negative values wait until the semaphore is at least `|$semop|` before subtracting; positive values add unconditionally and wake waiters; zero waits until the semaphore reaches zero.

- `$semflag` - bitwise OR of `IPC_NOWAIT` (fail with `EAGAIN` instead of blocking) and `SEM_UNDO` (have the kernel reverse the operation if the process exits without undoing it - essential for lock-like uses where a crash must not leave the semaphore held).

The length of `OPSTRING` implies the number of operations: the kernel processes `length($opstring) / sizeof(struct sembuf)` records. Passing two records packed end-to-end asks for two operations applied atomically.

The `s!` in the template is the **native** `short`, which matches the `sembuf` layout on Linux. Plain `s` is a fixed 16-bit value - also in native byte order, but always exactly two bytes - so it will silently mis-pack on any platform where `sizeof(short) != 2`.

Examples

Wait on (decrement by 1) semaphore 0 of the set in `$semid`:

```
my $op = pack("s!3", 0, -1, 0);
semop($semid, $op)
    or die "wait on semaphore failed: $!";
```

Signal (increment by 1) the same semaphore:

```
my $op = pack("s!3", 0, 1, 0);
semop($semid, $op)
    or die "signal failed: $!";
```

Try to acquire without blocking - fail fast if someone else holds it:

```
use POSIX qw(EAGAIN);
my $op = pack("s!3", 0, -1, IPC_NOWAIT);
unless (semop($semid, $op)) {
    die "semop: $!" unless $! == EAGAIN;
    # contended; caller decides what to do
}
```

Acquire semaphore 0 with automatic release on process exit - the pattern for crash-safe mutual exclusion:

```
my $op = pack("s!3", 0, -1, SEM_UNDO);
semop($semid, $op) or die "lock: $!";
# critical section
my $release = pack("s!3", 0, 1, SEM_UNDO);
semop($semid, $release) or die "unlock: $!";
```

Two operations applied atomically - wait for semaphore 0 and semaphore 1 together, releasing neither if either would block:

```
my $ops = pack("s!3", 0, -1, 0)
        . pack("s!3", 1, -1, 0);
semop($semid, $ops) or die "paired wait: $!";
```

Wait for semaphore 0 to reach zero - the “barrier” form, where `$semop` is zero rather than negative:

```
my $op = pack("s!3", 0, 0, 0);
semop($semid, $op) or die "barrier: $!";
```

Edge cases

- **Short `OPSTRING`**: the length must be an exact multiple of the `sembuf` record size. A truncated or padded string gives `EINVAL`. Always build the string from `pack` calls, never by hand.
- **Native width matters**: `s!3` not `s3`. A script written with plain `s` works on a 16-bit-short host by accident and breaks elsewhere.
- **Zero-length `OPSTRING`**: asks for zero operations. The kernel accepts this and returns success without side effects - rarely what you want. Guard the call with `return unless length $ops`.
- **Signal interruption**: a blocked `semop` returning with `$!` set to `EINTR` is not a bug - retry unless you installed a handler that wants to abort. Perl's default `$SIG{*}` handling does not automatically restart the syscall.
- **`SEM_UNDO` with matched pairs**: the kernel tracks the cumulative adjustment per semaphore per process. Acquiring twice with `SEM_UNDO` then releasing once leaves an outstanding `-1` to be undone on exit. Pair every `-1` with a matching `+1` in normal flow; `SEM_UNDO` is insurance against crashes, not a substitute for releasing.
- **Removed set while waiting**: if another process calls `semctl` with `IPC_RMID`, every blocked `semop` returns false with `$!` set to `EIDRM`. The semaphore identifier is invalid from that point on.
- **Permissions**: `semop` checks alter-permission on the set. A process that can `semget` a read-only handle still cannot `semop` it.
- **Large operation counts**: the kernel caps the number of operations per call at `SEMOPM` (typically 32). Building an `OPSTRING` with more records fails with `E2BIG`.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `semget` - obtain the identifier you pass as the first argument; the two are always used together
- `semctl` - set or query semaphore values, and remove the set when you're done with it
- `pack` - build each `sembuf` record; `s!3` is the template you want
- `msgsnd` - the message-queue counterpart when you need to pass data rather than signal availability
- `shmget` - the shared-memory counterpart; semaphores are typically used to guard access to a SysV shared-memory segment

- `IPC::SysV` - exports the `IPC_NOWAIT`, `SEM_UNDO`, and `IPC_RMID` constants used in `$semflag` and in `semctl` commands
- `IPC::Semaphore` - object wrapper over `semget` / `semop` / `semctl` when a procedural interface feels too bare

Sockets

send

Send a message on a socket.

`send` writes the scalar `MSG` to `SOCKET` through the operating system's socket layer, not through PerlIO. On a connected socket (`SOCK_STREAM` after `connect`, or an accepted peer) the three-argument form issues a `send(2)` - use the four-argument form on an unconnected datagram socket (`SOCK_DGRAM`) to route each message to `TO`, which triggers `sendto(2)` under the hood. `FLAGS` is the bitmask the system call accepts (commonly `0`; see your platform's `send(2)` man page for `MSG_DONTWAIT`, `MSG_OOB`, `MSG_NOSIGNAL`, and friends).

Synopsis

```
send SOCKET, MSG, FLAGS, TO
send SOCKET, MSG, FLAGS
```

What you get back

The **number of bytes** accepted by the kernel, or `undef` on error with `$!` set. Always check the return value:

```
my $n = send $sock, $buf, 0
    or die "send failed: $!";
```

Two properties bite people:

- The count is **bytes**, not characters. `length $buf` on a wide string and the returned count will not agree.
- On `SOCK_STREAM` sockets the kernel may accept **fewer bytes than you gave it** - a short write. Loop until the whole buffer is drained, or use `syswrite` which has the same short-write semantics but works on any filehandle.

Global state it touches

- `$!` - set to the system error on failure.
- The socket's PerlIO layer stack is **not** consulted. Output layers (`:utf8`, `:encoding(...)`, `:crlf`) have no effect on what `send` writes. In particular, a `:utf8` layer causes `send` to **throw an exception** rather than silently encode - the socket layer and the PerlIO layer cannot both own byte framing.

Examples

Connected TCP - three-argument form:

```
use Socket;
socket my $sock, PF_INET, SOCK_STREAM, getprotobyname("tcp")
    or die "socket: $!";
connect $sock, sockaddr_in(80, inet_aton("example.com"))
    or die "connect: $!";
send $sock, "GET / HTTP/1.0\r\n\r\n", 0
    or die "send: $!";
```

Unconnected UDP - four-argument form with an explicit destination:

```
use Socket;
socket my $sock, PF_INET, SOCK_DGRAM, getprotobyname("udp")
    or die "socket: $!";
my $to = sockaddr_in(514, inet_aton("10.0.0.1"));
send $sock, "<14>hello syslog\n", 0, $to
    or die "send: $!";
```

Handle short writes on a stream socket:

```
my $offset = 0;
while ($offset < length $buf) {
    my $n = send $sock, substr($buf, $offset), 0;
    defined $n or die "send: $!";
    $offset += $n;
}
```

Send with a flag - `MSG_DONTWAIT` makes a single non-blocking attempt and returns `undef` with `$!` set to `EAGAIN`/`EWOULDBLOCK` if the kernel buffer is full:

```
use Socket qw(MSG_DONTWAIT);
my $n = send $sock, $buf, MSG_DONTWAIT;
if (!defined $n) {
    if ($!{EAGAIN} || $!{EWOULDBLOCK}) {
        # back off, retry later
    } else {
        die "send: $!";
    }
}
```

Edge cases

- **Wide characters**: passing a string with codepoints above `0xFF` on a byte socket writes the UTF-8 encoding of each character and emits a `Wide character in send` warning under `use warnings`. Encode explicitly with `Encode::encode` before calling `send`.
- **`:utf8` layer on the socket**: `send` **croaks**. Either open the socket without the layer, or call `binmode` on `$sock` to strip it before the first `send`. An `:encoding(...)`

layer implicitly adds `:utf8`, so it's banned too.

- **Four-arg form on a connected socket**: passing `TO` to a connected `SOCK_STREAM` socket is an error - the kernel returns `EISCONN`. Use the three-argument form once connected.
- **Three-arg form on an unconnected datagram socket**: fails with `ENOTCONN` (or `EDESTADDRREQ` on some platforms). Either `connect` the datagram socket first (which pins a default peer), or supply `TO`.
- **Empty `MSG` on UDP**: valid - sends a zero-length datagram the peer can read with `recv`. On TCP it is a no-op that returns `0`.
- **`SIGPIPE` on a closed stream peer**: writing to a `SOCK_STREAM` whose remote end has closed raises `SIGPIPE`, which by default terminates the process. Pass `MSG_NOSIGNAL` in `FLAGS` (Linux) or install `$SIG{PIPE} = 'IGNORE'` to get an `EPIPE` return instead.
- **`sendmsg(2)`**: Perl exposes `send`/`sendto`, not `sendmsg`. Scatter/gather writes and ancillary data (credentials, file descriptors over `AF_UNIX`) require an XS module such as `Socket::MsgHdr`.
- **Non-socket filehandle**: `send` on a non-socket returns `undef` with `$!` set to `ENOTSOCK`. Use `syswrite` for pipes and regular files.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `recv` - the inverse; reads bytes from a socket with the same layer restrictions and the same `FLAGS` argument
- `syswrite` - raw write that bypasses PerlIO buffering on any filehandle; use it for pipes and files where `send` refuses to work
- `connect` - establishes the peer address that lets you use the three-argument form of `send`
- `socket` - creates the filehandle `send` writes to; the socket type (`SOCK_STREAM` vs `SOCK_DGRAM`) decides which form of `send` is legal
- `bind` - assigns a local address; needed before `send` on a datagram socket if the peer expects replies to a known port
- `binmode` - strips a stray `:utf8` layer that would otherwise make `send` throw

User and group info

setgrent

Rewind the group-database iterator back to the first entry.

`setgrent` tells the system's group-database iterator to start over. The next call to `getgrent` returns the first group record again, regardless of how far a previous iteration walked. It takes no arguments, returns no useful value, and exists so that a program can

re-scan the group database without having to close and reopen it by hand. Pair it with `endgrent` at the end of a scan to release the underlying file descriptor.

Synopsis

```
setgrent;
setgrent();
```

What you get back

The return value is unspecified and conventionally ignored. Call `setgrent` for its side effect - positioning the iterator - not for its value.

Global state it touches

- The **per-process group-database iterator** maintained by the C library (`getgrent(3)`). It is global to the process, not scoped to a package or a lexical: any other code walking `getgrent` in the same process, including inside threads that share the iterator, sees the rewind.
- `$!` - on systems whose `setgrent(3)` reports errors (rare; most implementations return `void`), a failure leaves `errno` set. Portable code does not rely on this.

`setgrent` does not touch `$_`, `@ARGV`, or any filehandle.

Examples

Rewind between two passes over the group database - first to collect all group names, then to collect the members of a specific group:

```
my @names;
while (my @g = getgrent) {
    push @names, $g[0];
}

setgrent;                               # start over
while (my @g = getgrent) {
    next unless $g[0] eq 'wheel';
    print "wheel members: $g[3]\n";
    last;
}
endgrent;
```

Use `setgrent` at the top of a sub that is called repeatedly and wants a fresh scan every time:

```
sub groups_containing {
    my ($user) = @_;
    setgrent;
    my @groups;
```

(continues on next page)

(continued from previous page)

```
    while (my ($name, undef, undef, $members) = getgrent) {
        push @groups, $name if grep { $_ eq $user } split / /,
↪$members;
    }
    endgrent;
    return @groups;
}
```

Pair `setgrent` / `endgrent` around a scan so an unrelated caller that was mid-iteration is not surprised when control returns:

```
setgrent;
while (my @g = getgrent) {
    # ... process ...
}
endgrent;
```

Edge cases

- **No argument form only.** Unlike `sethostent`, `setnetent`, `setprotoent`, and `setservent`, which take a `STAYOPEN` flag, `setgrent` takes nothing. Passing an argument is a compile-time error.
- **Calling `setgrent` without an intervening `getgrent` is a no-op on most systems** - the iterator is simply positioned at the first entry, and the first `getgrent` after it reads that entry. Don't rely on `setgrent` as an "open the group file" primitive; its only defined behaviour is "rewind".
- **Process-global state.** `setgrent` inside one module rewinds the iterator for every other module in the same process. If a library walks `getgrent` to completion and a caller then calls `setgrent`, the library's cursor is gone. Treat the iterator as a shared resource: own it for the duration of a scan, bracketed by `setgrent` and `endgrent`.
- **Threads share the iterator.** Two threads calling `getgrent` concurrently will each see a subset of the records; `setgrent` from one thread disturbs the other. For per-thread iteration, read `/etc/group` directly or use a thread-safe wrapper; the built-in is not reentrant.
- **Not all platforms implement it identically.** On systems without `setgrent(3)` the call is a no-op and silently succeeds. On systems that do implement it, the exact cost (reopening `/etc/group`, restarting an NSS query, re-querying LDAP) depends on the backing name-service and can be non-trivial - avoid calling it in a tight loop.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `getgrent` - read the next group record; consumes the iterator that `setgrent` rewinds
- `endgrent` - close the iterator when the scan is finished; the symmetric bookend to `setgrent`
- `getgrnam` - lookup a single group by name without touching the iterator at all; prefer this for one-shot queries
- `getgrgid` - lookup a single group by GID; same reasoning as `getgrnam`
- `setpwent` - the equivalent rewind for the password database

Network info

sethostent

Open or rewind the hosts database for iteration.

`sethostent` prepares the system hosts database (typically `/etc/hosts` plus any name-service sources configured by the platform) for a sequential walk with `gethostent`, and rewinds the cursor if iteration is already in progress. `STAYOPEN` is a flag passed through to the underlying C `sethostent(3)` call: a true value asks the resolver to keep the database open across subsequent `gethostby*` lookups, which on some systems avoids reopening the file or reconnecting to the name service for every query.

Synopsis

```
sethostent STAYOPEN
sethostent 0
sethostent 1
```

What you get back

Returns nothing useful. The call is invoked for its side effect of priming the iterator; ignore the return value.

Global state it touches

Mutates **process-global** resolver state maintained by the C library: the hosts-database file descriptor or name-service handle, and the per-process iteration cursor read by `gethostent`. Two pieces of code in the same process walking the hosts database simultaneously will step on each other - there is one cursor, not one per caller. On failure `$!` is set by the underlying system call; there is no `h_errno`-style reporting for this function.

Examples

Walk every entry in the hosts database:

```
sethostent(0);
while (my ($name, $aliases, $addrtype, $length, @addrs)
        = gethostent()) {
    print "$name\n";
}
endhostent();
```

Ask the resolver to keep the database open while a batch of name-to-address lookups runs, then close it:

```
sethostent(1);
for my $host (@hosts) {
    my $packed = gethostbyname($host);
    # ...
}
endhostent();
```

Rewind an in-progress iteration to start over from the first entry:

```
sethostent(0);
my $first = (gethostent())[0];
sethostent(0);                          # back to the top
my $first_again = (gethostent())[0];
```

Edge cases

- **`STAYOPEN` is mandatory syntax, not mandatory semantics.** The argument must be present - `sethostent;` is a compile error - but many platforms ignore the flag and always keep the database open for the duration of the iteration. Pass `0` when you have no preference.
- **Not every platform has a meaningful hosts iterator.** On systems where the hosts database is served exclusively by DNS or a name service that does not expose enumeration, `gethostent` returns the empty list on the first call regardless of `sethostent`. The call itself still succeeds.
- **Thread safety.** The resolver cursor is per-process, not per-thread. Two threads iterating concurrently will interleave records and neither will see the full list. Upstream Perl substitutes `*_r` variants for point lookups like `gethostbyname` where available; there is no thread-safe variant of the iterator.
- **Interaction with `gethostby*` caches.** On some libc implementations a `sethostent(1)` issued before a burst of `gethostbyname` / `gethostbyaddr` calls keeps the backing file open between queries; on others it has no effect. Measure before relying on it as an optimisation.
- **Pair with `endhostent`.** Leaving the database open after iteration can leak a file descriptor or a resolver connection. Always close the walk with `endhostent`.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `gethostent` - read the next entry from the hosts database after `sethostent` has primed the cursor
- `endhostent` - close the database and release resolver state when iteration is done
- `gethostbyname` - point lookup by name; benefits from `sethostent(1)` on platforms that honour it
- `gethostbyaddr` - point lookup by address; same caveat as `gethostbyname`
- `setnetent` - the networks-database counterpart, same iteration pattern
- `setservent` - the services-database counterpart

Network info

setnetent

Open or rewind the networks database for iteration.

`setnetent` prepares the system networks database so that subsequent calls to `getnetent` return entries starting from the first one. It is a direct wrapper around the C library's `setnetent(3)` and reads the same underlying source as the C call - typically `/etc/networks` on Linux, or whatever `nsswitch.conf` / NSS backends have been configured to provide.

The `STAYOPEN` argument is a hint to the C library: a true value asks it to keep the database file open across related calls (`getnetbyname`, `getnetbyaddr`, `getnetent`) so individual lookups do not each pay the cost of opening and closing the file. On modern glibc this flag is effectively ignored - the database is handled efficiently either way - but passing it costs nothing and keeps the code portable to systems where it still matters.

Synopsis

```
setnetent STAYOPEN
setnetent 0
setnetent 1
```

What you get back

`setnetent` returns nothing meaningful. In list context it returns an empty list, in scalar context `undef`. Do not rely on the return value; the call's effect is the side effect of positioning the database iterator at the start.

If the underlying C call fails (for example the database file is missing or unreadable) the error is silent from Perl's perspective - `getnetent` will simply return an empty list on the first call.

Global state it touches

- The process-wide **networks-database cursor** maintained by the C library. Every call to `getnetent` advances this cursor; `setnetent` rewinds it. `endnetent` closes the database and releases any resources held by a `STAYOPEN` hint.
- No Perl special variables are read or written.

Because the cursor is per-process, not per-thread, interleaving networks-database iteration from multiple threads in the same interpreter will corrupt each iterator's position. Perform the iteration from a single thread, or serialise access.

Examples

Rewind the database and iterate from the first entry:

```
setnetent(1);
while (my ($name, $aliases, $addrtype, $net) = getnetent()) {
    printf "%-20s %d\n", $name, $net;
}
endnetent();
```

Re-run the iteration from the top without closing in between. The second `setnetent` rewinds, so the second loop sees every entry again:

```
setnetent(1);
my @first_pass  = collect_nets();
setnetent(1);
my @second_pass = collect_nets();
endnetent();

sub collect_nets {
    my @out;
    while (my @ent = getnetent()) { push @out, $ent[0] }
    return @out;
}
```

Pairing with `Net::netent` for a named-field interface over the same data source - `setnetent` still controls the cursor:

```
use Net::netent;
setnetent(1);
while (my $n = getnetent()) {
    printf "%s -> %s\n", $n->name, $n->net;
}
endnetent();
```

Edge cases

- **`STAYOPEN` is a hint, not a guarantee.** On systems where the C library ignores it, passing 1 and passing `0` produce identical observable behaviour.

- **No implicit `setnetent` before `getnetent`.** A fresh process that calls `getnetent` without first calling `setnetent` still starts from the first entry on most systems, but this is a libc convention rather than a guarantee. Call `setnetent` explicitly when you want to be sure.
- **Calling `setnetent` mid-iteration rewinds the cursor.** Any in-progress `while (getnetent)` loop restarts from the top - useful for a deliberate second pass, a bug if it happens by accident from a helper sub that also iterates.
- **Interaction with `getnetbyname` / `getnetbyaddr`.** On systems where the C library honours `STAYOPEN`, a by-name or by-address lookup during iteration will *not* disturb the iterator. On systems that ignore the hint, each lookup may reset the cursor. Do not mix by-name lookups with `getnetent` iteration if you need stable positioning.
- **Permissions.** The networks database is typically world-readable, but NSS backends (LDAP, `sssd`) may impose their own authorisation. A lookup that worked as `root` can yield nothing as a normal user without Perl seeing an error.
- **Empty or missing `/etc/networks`.** On most Linux systems this file is empty or absent by default. `setnetent` still succeeds; the following `getnetent` returns an empty list on the first call. This is not an error condition.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `getnetent` - the iterator primed by `setnetent`; reaches for the next row on each call
- `endnetent` - close the database and release any resources held open by `STAYOPEN`; call it when iteration is done
- `getnetbyname` - lookup a single entry by name without running the iterator
- `getnetbyaddr` - lookup a single entry by packed address without running the iterator
- `sethostent` - the same idiom for the hosts database
- `setservent` - the same idiom for the services database
- `Net::netent` - object-oriented wrapper returning named fields instead of a positional list

Processes

setpgrp

Set the process group of a process.

`setpgrp` moves the process identified by `PID` into the process group identified by `PGRP`. A `PID` of `0` means the current process; a `PGRP` of `0` means “use `PID` as the new group id” - that is, make the process a group leader. With no arguments it defaults to `setpgrp(0,`

`0`), which makes the current process the leader of a new process group whose id equals its own pid.

This is the Perl interface to the underlying `setpgid(2)` / `setpgrp(2)` system call. It is the building block for detaching a process from its controlling terminal and for job-control schemes that want to send signals to a whole group at once with `kill`.

Synopsis

```
setpgrp PID, PGRP
setpgrp            # same as setpgrp(0, 0)
```

What you get back

1 on success, `undef` on failure with `$!` set to the reason (commonly `EPERM` when the target process is not the caller or a child of the caller, or `ESRCH` when no such process exists). Check the return value - silently proceeding with a process still in the old group is almost never what you want.

```
setpgrp(0, 0)
    or die "setpgrp failed: $!";
```

Global state it touches

- `$!` - set to the `errno` value on failure. Not touched on success.

Examples

Detach the current process into a new process group of its own. This is the classic first step of a daemonisation sequence, run after `fork` and before closing the standard handles:

```
setpgrp(0, 0)
    or die "setpgrp failed: $!";
```

Move a just-spawned child into its own group so that a `SIGTERM` sent to the group kills the child and any grandchildren it spawns, without touching the parent:

```
my $pid = fork // die "fork: $!";
if ($pid == 0) {
    setpgrp(0, 0) or die "setpgrp: $!";
    exec @cmd or die "exec: $!";
}
# parent: signal the whole group with a negative pid
kill 'TERM', -$pid;
```

Read the group id back after setting it, using `getpgrp`:

```
setpgrp(0, 0) or die $!;
my $pgid = getpgrp(0);
print "now leader of group $pgid\n";
```

Handle the portability failure mode - on a platform without the underlying system call the function raises an exception rather than returning `undef`:

```
eval { setpgrp(0, 0) };
if ($@) {
    warn "no process-group support on this platform: $@";
}
```

Edge cases

- **No arguments defaults to `(0, 0)`.** Writing `setpgrp;` is the same as `setpgrp(0, 0)` - make the current process a new group leader. The no-argument form is the portable form (see below).
- **BSD 4.2 compatibility.** The historical BSD 4.2 `setpgrp` took no arguments. Scripts that must run on such systems can only rely on the `setpgrp(0, 0)` / `setpgrp()` forms; any other argument combination is non-portable.
- **Raises, not returns `undef`, on unsupported platforms.** On a machine that implements neither POSIX `setpgid(2)` nor BSD `setpgrp(2)`, `setpgrp` raises an exception. This is the one failure mode that does **not** go through `$!` - wrap in `eval` if you need to degrade gracefully.
- **`EPERM` on cross-session moves.** The kernel rejects attempts to move a process into a group in a different session, or to move a process that has already called `exec`. Catch `$! == EPERM` and fall back rather than retrying.
- **Signalling the whole group.** Once a group is established, `kill SIG, -$pgid` delivers `SIG` to every process in the group. The negative-pid convention is a property of `kill`, not of `setpgrp` - but it's the reason you usually call `setpgrp` in the first place.
- **Not a session leader.** `setpgrp` changes process group, not session. To detach from the controlling terminal entirely you need `POSIX::setsid`, which creates a new session and a new group in one step.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `getpgrp` - read the current process group id; the obvious companion for verifying a `setpgrp` call took effect
- `POSIX::setpgid` - the POSIX-named wrapper around the same system call, preferred in new code because its argument order and semantics are unambiguous across platforms
- `POSIX::setsid` - create a new session *and* a new process group in one call; use this, not `setpgrp`, for full terminal detachment
- `fork` - the call that almost always precedes `setpgrp`; a child that wants its own group calls `setpgrp(0, 0)` right after `fork` returns `0`

- `kill` - how you actually use a process group once you've built one: `kill SIG, -$pgid` signals every member

Processes

setpriority

Set the scheduling priority (nice value) of a process, process group, or user.

`setpriority` is the Perl-level wrapper around the `setpriority(2)` system call. `WHICH` selects the target class (`PRIO_PROCESS`, `PRIO_PGRP`, or `PRIO_USER`), `WHO` names the specific target within that class (`0` means "self" or "current" depending on `WHICH`), and `PRIORITY` is the new nice value. Higher values mean *lower* scheduling priority; on Linux the useful range is `-20` (most favourable) through `19` (least favourable).

Synopsis

```
setpriority WHICH, WHO, PRIORITY
```

What you get back

Returns true on success, `undef` on failure (with `$!` set to the underlying `errno`). The most common failures are `EACCES` or `EPERM` - lowering the nice value below the current one, or touching another user's processes, requires privilege. Always check the return value when the priority change is not just cosmetic:

```
setpriority(PRIO_PROCESS, $$, 10)
    or die "cannot renice self: $!";
```

Global state it touches

- `$!` - set on failure to the `errno` from the underlying `setpriority(2)` call.

The three `WHICH` classes

`WHICH` is one of three integer constants, each selecting a different interpretation of `WHO`:

- `PRIO_PROCESS` - `WHO` is a process ID. `0` means the current process.
- `PRIO_PGRP` - `WHO` is a process group ID. `0` means the process group of the current process.
- `PRIO_USER` - `WHO` is a numeric user ID (UID). `0` means the real UID of the current process. Setting by user changes the priority of *every* process owned by that user.

These constants are not built in - import them from `POSIX`:

```
use POSIX qw(PRIO_PROCESS PRIO_PGRP PRIO_USER);
```

Examples

Renice the current process to `10` (friendlier to other work):

```
use POSIX qw(PRIO_PROCESS);
setpriority(PRIO_PROCESS, 0, 10)
    or die "renice failed: $!";
```

Renice a child process after `fork`:

```
use POSIX qw(PRIO_PROCESS);
my $pid = fork // die "fork: $!";
if ($pid == 0) {
    setpriority(PRIO_PROCESS, $$, 15);  # child runs at low priority
    exec @cmd;
}
```

Renice every process in the current process group - useful in a session leader that wants its whole pipeline to yield:

```
use POSIX qw(PRIO_PGRP);
setpriority(PRIO_PGRP, 0, 5)
    or warn "could not renice process group: $!";
```

Pair with `getpriority` to adjust relative to the current value rather than setting an absolute one:

```
use POSIX qw(PRIO_PROCESS);
my $cur = getpriority(PRIO_PROCESS, 0);
setpriority(PRIO_PROCESS, 0, $cur + 5)
    or die "renice failed: $!";
```

Edge cases

- **Negative priorities require privilege.** Lowering the nice value (more favourable scheduling) below the current one needs `CAP_SYS_NICE` on Linux, or root. An unprivileged attempt fails with `EACCES` or `EPERM` depending on kernel version.
- **Nice values are clamped.** The kernel silently clamps `PRIORITY` to the supported range (`-20` . . `19` on Linux). Passing `100` does *not* fail; it ends up as 19.
- **`WHO = 0` is context-sensitive.** Its meaning depends on `WHICH`: current PID for `PRIO_PROCESS`, current process group for `PRIO_PGRP`, real UID for `PRIO_USER`. It is not a universal "self" token.
- **Non-existent target.** A PID, PGID, or UID that does not match any running process fails with `ESRCH`.
- **Platform without `setpriority(2)`.** Perl raises a fatal exception (`setpriority() not implemented`). pperl targets Linux only, so this case does not arise in practice, but portable code should still guard with `eval`.
- **Signed return from `getpriority` vs. success of `setpriority`.** `getpriority` can legitimately return `-1` as a nice value, which is why it sets `$!` to distinguish

error from result; `setpriority` has no such ambiguity - treat its return purely as a boolean.

- **Nice value is per-process, not per-thread.** On Linux this differs from POSIX: the kernel applies the nice value to the calling task (thread). Perl is single-threaded at the interpreter level, so this only matters if XS code has spawned kernel threads.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `getpriority` - read the current nice value; the natural counterpart when adjusting relative to the existing priority
- `POSIX` - source of the `PRIO_PROCESS`, `PRIO_PGRP`, and `PRIO_USER` constants
- `fork` - typical pairing: spawn a child, then renice it before `exec`
- `$!` - holds the `errno` describing why a failed `setpriority` call returned `undef`
- `kill` - other per-process control operation with the same "target by PID or PGID" convention

Network info

setprotoent

Open the protocols database and prepare it for sequential reads.

`setprotoent` rewinds (or opens) the system protocols database - on most Unix-like systems that's `/etc/protocols` - so that a subsequent sequence of `getprotoent` calls starts at the first entry. It is the first leg of the `setprotoent` / `getprotoent` / `endprotoent` iteration triad. The single `STAYOPEN` argument is a hint to the C library about whether the file handle backing the iteration should stay open between calls: a true value asks the library to keep it open (faster for long walks), a false value lets it close and reopen per call.

Calling `getprotobyname` or `getprotobynumber` anywhere mid-iteration may reset the walk position on some platforms; see *Edge cases*.

Synopsis

```
setprotoent STAYOPEN
setprotoent(1)
```

What you get back

No meaningful return value. Like its `endprotoent` and `sethostent` siblings, `setprotoent` is called for its side effect: rewinding the iteration and, depending on `STAYOPEN`, pinning the descriptor open. There is nothing worth checking in its return value.

Global state it touches

`setprotoent` manipulates a **per-process**, C-library-owned cursor into the protocols database. That state is shared across every subsequent `getprotoent` call in the same process - including calls from other modules you did not write. Code that walks the database must either fully drain it with `endprotoent` or accept that a later caller may see an iteration already in progress. No Perl-level special variables are read or written.

Examples

Open the database and iterate every entry, keeping the underlying descriptor alive for the full walk:

```
setprotoent(1);                          # 1 = keep file open between
↪calls
while (my @p = getprotoent()) {
    my ($name, $aliases, $proto) = @p;
    print "$proto\t$name\n";
}
endprotoent();
```

Restart an in-progress iteration from the beginning:

```
setprotoent(1);
my @first  = getprotoent();
my @second = getprotoent();

setprotoent(1);                          # rewind
my @again  = getprotoent();              # same as @first
endprotoent();
```

Let the C library close and reopen `/etc/protocols` on every `getprotoent` - cheaper in memory, more expensive in syscalls, useful only when you read a handful of entries:

```
setprotoent(0);
my @tcp = getprotoent();
endprotoent();
```

Pair with a one-shot lookup - `getprotobyname` and `getprotobynumber` do not participate in the iteration and do not need `setprotoent`:

```
my @tcp = getprotobyname('tcp');         # no setprotoent needed
```

Edge cases

- **`STAYOPEN` is a hint, not a contract**: many C libraries ignore it entirely and behave as if the descriptor were always kept open. Do not rely on the argument to control resource usage at a fine grain.
- **Mixing with single-shot lookups**: calling `getprotobyname` or `getprotobynumber` during a walk may reset the iteration cursor on some platforms. Portable code drains the walk first, or re-calls `setprotoent` afterwards to restart from the top.

- **Not reentrant**: the protocols database cursor is a single global resource inside the C library. Two concurrent iterations in the same process (threads, signal handlers, anything) corrupt each other. Use one walk at a time.
- **Argument is mandatory at the language level**: `setprotoent` without an argument is a syntax error - the built-in has arity one. Pass `0` if you have no preference.
- **Platforms without a protocols database**: on systems where the C library has no `/etc/protocols` equivalent, the call is a no-op rather than an error. See `perlport` for the full list of network-database quirks.
- **Untainted result of the iteration**: the records returned by `getprotoent` after `setprotoent` come from a file the kernel reads on your behalf and are not marked tainted. Treat them as untrusted anyway if the database can be user-edited.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `getprotoent` - read the next entry; the body of the loop that `setprotoent` sets up
- `endprotoent` - close the database after iteration; the terminator for the triad `setprotoent` starts
- `getprotobyname` - single-shot lookup by name; does not participate in the iteration
- `getprotobynumber` - single-shot lookup by protocol number; likewise independent
- `sethostent` - analogous opener for the hosts database; same idiom, different table
- `setservent` - analogous opener for the services database

User and group info

setpwent

Rewind the password-database iterator so the next `getpwent` call returns the first entry again.

`setpwent` wraps the C library's `setpwent(3)` call. It (re)opens the system password database and resets the iteration cursor used by `getpwent`, so a subsequent loop starts from the top. Use it when you want to iterate the password database more than once in the same process, or when you want to restart iteration after a partial walk. Pair it with `endpwent` to release the database when done.

Synopsis

```
setpwent;
setpwent();
```

What you get back

Return value is **not useful** and must not be relied on. On most systems the underlying `setpwent(3)` has a `void` return; Perl yields a single meaningless true value. Call `setpwent` for its side effect, never for its value.

Global state it touches

- The per-process cursor shared by `getpwent`, `setpwent`, and `endpwent`. This cursor is owned by the C library, not by Perl - forking a child inherits an open database handle, and two threads iterating concurrently will interleave and lose records.
- `$!` is **not** set by `setpwent`; errors from the underlying `setpwent(3)` are swallowed by the platform.

Examples

Iterate the password database twice in the same program:

```
while (my @pw = getpwent) {
    print "$pw[0]\n";           # first pass: every login name
}
setpwent;
while (my @pw = getpwent) {
    print "uid=$pw[2]\n";       # second pass: every uid
}
endpwent;
```

Restart iteration after an early exit:

```
while (my @pw = getpwent) {
    last if $pw[0] eq 'root';
}
setpwent;                       # rewind so the next walk sees root␣
↪again
```

Typical cleanup pattern - always call `endpwent` when done, whether or not you called `setpwent`:

```
setpwent;
while (my ($name, undef, $uid) = getpwent) {
    print "$name => $uid\n" if $uid >= 1000;
}
endpwent;
```

Edge cases

- **No arguments.** `setpwent` takes no arguments. Writing `setpwent $x` is a syntax trap: like other list operators, it will try to consume `$x` as the first element of a list.
- **Before any `getpwent` call.** Safe. `setpwent` opens the database if it is not already open, so it doubles as an explicit "open" call.

- **After endpwent.** Safe. The database is reopened.
- **Forked children.** Each process has its own iteration cursor after the fork, but the underlying file descriptor may be shared; interleaved reads from parent and child produce missing or duplicated records. Call `setpwent` in the child to get a clean iterator.
- **Threads.** `getpwent` iteration is per-process, not per-thread. Concurrent iteration from multiple threads loses records. Upstream Perl substitutes thread-safe `_r` variants for `getpwnam` and `getpwuid` when the platform provides them; there is no `setpwent_r`, and this function remains process-global.
- **NSS-backed databases.** On systems where `/etc/nsswitch.conf` routes `passwd` through LDAP, SSSD, or similar, `setpwent` may block on a network round-trip. Do not call it from signal handlers or latency-critical code paths.
- **Shadow passwords.** `setpwent` does not affect the shadow-password iterator (if any); it only resets the regular `passwd` walk.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `getpwent` - fetch the next entry; the function `setpwent` exists to rewind
- `endpwent` - release the password database when you are finished iterating
- `getpwnam` - direct lookup by login name; does not interact with the iteration cursor
- `getpwuid` - direct lookup by uid; does not interact with the iteration cursor
- `setgrent` - the same rewind operation for the group database
- `User::pwent` - object-oriented wrapper that returns named-field objects instead of 10-element lists

Network info

setservent

Rewind the services database and optionally keep it open across lookups.

`setservent` is the services-database counterpart of the C library's `setservent(3)`. It resets the internal cursor used by `getservent` so the next call starts again at the first entry of `/etc/services` (or whatever source the system resolver is configured to use), and it controls whether the underlying file stays open between by-name and by-port lookups.

Synopsis

```
setservent STAYOPEN
setservent 0                # close file between lookups (default)
setservent 1                # keep file open across lookups
```

What you get back

The return value is whatever the underlying C `setservent(3)` returns - on most systems that's nothing useful, and scripts do not check it. Treat `setservent` as called for its side effect on the iterator, not for a value.

Global state it touches

- The per-process services-database cursor shared with `getservent`, `getservbyname`, `getservbyport` and `endservent`. All four read and mutate the same state; iteration is not thread-safe and not reentrant.
- The open/closed status of the services database file. `STAYOPEN` true leaves the file open until `endservent` is called; `STAYOPEN` false (the default) causes each subsequent by-name or by-port lookup to open and close the file again.

The STAYOPEN argument

Pass a true value to hold the file open across a run of individual `getservbyname` or `getservbyport` lookups. This is an optimisation for batch resolution, not a semantic change: results are identical either way.

```
setservent 1;
for my $name (@service_names) {
    my @svc = getservbyname $name, "tcp";
    # ... process @svc ...
}
endservent;
```

Pass a false value (or `0`) when you want each lookup to reopen the file. This is the default behaviour and the one most programs want.

Examples

Walk every entry in the services database and print name/port pairs:

```
setservent 0;
while (my ($name, undef, $port, $proto) = getservent) {
    print "$name $port/$proto\n";
}
endservent;
```

Rewind mid-iteration to start over:

```
setservent 0;
my @first_pass;
while (my @svc = getservent) {
    push @first_pass, $svc[0];
    last if @first_pass == 10;
}
setservent 0;                       # cursor back to start
```

(continues on next page)

(continued from previous page)

```
my @second_pass = getservent;     # first entry again
endservent;
```

Batch-resolve with the file held open:

```
my @wanted = qw(http https ssh smtp);
setservent 1;
my %port = map { $_ => scalar getservbyname($_, "tcp") } @wanted;
endservent;
```

Edge cases

- **`STAYOPEN` is required at the Perl level.** `setservent` is parsed with list-operator precedence and reads one argument; a bare `setservent;` with no argument is a compile-time error. Pass `0` when you want the default "close after each lookup" behaviour.
- **Platform dependence.** The services database is whatever the system resolver uses - typically `/etc/services`, but NIS, LDAP, and other name-service backends can supply entries. `setservent` rewinds whichever source the C library selects; the Perl level has no say in it.
- **Non-Unix platforms.** On systems without `setservent(3)` the call croaks with `The setservent function is unimplemented`. See `perlport` for the availability matrix.
- **Threads.** Iterator state is per-process, not per-thread; two threads walking the services database concurrently will interleave and neither will see every record. See the shared note in `perlfunc` on `getpwnam`-family thread safety.
- **Pairing with `endservent`.** When `STAYOPEN` was true, call `endservent` to release the file descriptor; otherwise the handle leaks for the life of the process.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `getservent` - walk the services database one entry at a time; this is the iterator `setservent` rewinds
- `endservent` - close the services database file opened by `setservent 1`
- `getservbyname` - look up a single service by name; honours the open/closed state set here
- `getservbyport` - look up a single service by port number; same open/closed semantics
- `setprotoent` - same pattern for the protocols database when you are rewinding both together

- `Net::servent` - object-oriented wrapper over the `getserv*` family for when positional return lists get unwieldy

Sockets

setsockopt

Set a kernel-level option on an open socket.

`setsockopt` is the Perl wrapper around the POSIX `setsockopt(2)` system call. It writes the option named by `OPTNAME` at protocol `LEVEL` on the already-opened `SOCKET` filehandle, using the value packed into `OPTVAL`. It is the knob for everything the kernel lets you tune on a socket - address reuse, keepalives, timeouts, Nagle's algorithm, multicast membership, send and receive buffer sizes, `SO_LINGER` behaviour on `close`, and so on. Symbolic names for `LEVEL` and `OPTNAME` come from the `Socket` module.

Synopsis

```
setsockopt SOCKET, LEVEL, OPTNAME, OPTVAL
```

What you get back

A true value on success, `undef` on failure (with `$!` set to the underlying errno: `EBADF`, `ENOPROTOOPT`, `EINVAL`, `ENOTSOCK`, etc.). Always check the return value - a silently-ignored option means the program runs with defaults that differ from what the code reads:

```
setsockopt($sock, SOL_SOCKET, SO_REUSEADDR, pack("l", 1))
    or die "setsockopt SO_REUSEADDR: $!";
```

Global state it touches

None directly. `setsockopt` sets `$!` on failure; that is the only interpreter global involved. The effect is on the kernel-side state of the socket, not on any Perl-level variable.

OPTVAL - packed bytes or a bare integer

Most socket options take a C `int` flag. `setsockopt` accepts either form for convenience:

- A **packed byte string** whose layout matches the option's C type, typically built with `pack`: `pack("l", 1)` for a boolean, `pack("ll", $on_secs, $on_usecs)` for a `struct timeval`, `pack("ii", $on, $linger)` for `struct linger`.
- A **plain integer**, which is shorthand for `pack("i", OPTVAL)`. Handy for `SO_REUSEADDR`, `TCP_NODELAY`, and other boolean flags:

  ```
  setsockopt($sock, IPPROTO_TCP, TCP_NODELAY, 1);
  ```

Options whose value is a struct (`SO_LINGER`, `SO_RCVTIMEO`, `IP_ADD_MEMBERSHIP`) must be passed as an explicit `pack`ed string. A bare integer there produces `EINVAL` or, worse, silently writes wrong-sized memory that the kernel misinterprets.

Examples

Disable Nagle's algorithm on a TCP socket so small writes go out immediately - useful for latency-sensitive protocols (interactive shells, game traffic, RPC):

```
use Socket qw(IPPROTO_TCP TCP_NODELAY);

setsockopt($sock, IPPROTO_TCP, TCP_NODELAY, 1)
    or die "TCP_NODELAY: $!";
```

Allow a server to rebind a port that is still in `TIME_WAIT` from a previous process - the single most common `setsockopt` call in practice, and it must come **before** `bind`:

```
use Socket qw(SOL_SOCKET SO_REUSEADDR);

setsockopt($srv, SOL_SOCKET, SO_REUSEADDR, pack("l", 1))
    or die "SO_REUSEADDR: $!";
bind($srv, sockaddr_in(9000, INADDR_ANY))
    or die "bind: $!";
```

Set a receive timeout so a blocking `recv` or read from the socket returns with `EAGAIN` / `EWOULDBLOCK` after 5 seconds instead of hanging forever. The option value is a packed `struct timeval`:

```
use Socket qw(SOL_SOCKET SO_RCVTIMEO);

my $tv = pack("l!l!", 5, 0);    # 5 seconds, 0 microseconds
setsockopt($sock, SOL_SOCKET, SO_RCVTIMEO, $tv)
    or die "SO_RCVTIMEO: $!";
```

Enable TCP keepalive probes so half-open connections (peer crashed, cable yanked) are eventually noticed and the socket errors out rather than blocking indefinitely:

```
use Socket qw(SOL_SOCKET SO_KEEPALIVE);

setsockopt($sock, SOL_SOCKET, SO_KEEPALIVE, 1)
    or die "SO_KEEPALIVE: $!";
```

Make `close` block for up to 10 seconds while the kernel flushes unsent data, then discard and reset. `SO_LINGER` takes a two-integer struct: the on-flag and the timeout in seconds:

```
use Socket qw(SOL_SOCKET SO_LINGER);

setsockopt($sock, SOL_SOCKET, SO_LINGER, pack("ii", 1, 10))
    or die "SO_LINGER: $!";
```

Enlarge the kernel receive buffer for a bulk-transfer socket. The kernel typically doubles the value you request and caps it at `net.core.rmem_max`:

```
use Socket qw(SOL_SOCKET SO_RCVBUF);

setsockopt($sock, SOL_SOCKET, SO_RCVBUF, 1 << 20)   # 1 MiB
    or die "SO_RCVBUF: $!";
```

Edge cases

- **SOCKET must already be open.** `setsockopt` does not create the socket - call `socket` first. A closed or never-opened filehandle fails with `EBADF`.
- **Ordering matters.** `SO_REUSEADDR` must be set *before* `bind`. `SO_KEEPALIVE` can go before or after `connect`, but for server sockets set it on the accepted child filehandle returned by `accept`, not on the listening one - options do not always propagate.
- **Integer shorthand is `pack("i", N)`**, not `pack("l", N)`. On platforms where `int` and `long` differ in size, passing a plain integer and passing `pack("l", N)` write different byte counts. Both happen to work for the common `int`-sized boolean options on LP64 Linux, but the shorthand form is the portable default.
- **Struct-valued options need explicit `pack`.** `SO_LINGER`, `SO_RCVTIMEO`, `SO_SNDTIMEO`, `IP_ADD_MEMBERSHIP`, `IP_MREQ`, and similar take fixed-layout structs. Pass a packed string of the right length or the kernel rejects the call with `EINVAL`.
- **`LEVEL` picks the option namespace.** `SOL_SOCKET` for generic options, `IPPROTO_TCP` for TCP-specific (`TCP_NODELAY`, `TCP_KEEPIDLE`), `IPPROTO_IP` for IPv4 (`IP_TTL`, `IP_ADD_MEMBERSHIP`), `IPPROTO_IPV6` for IPv6. Mismatching `LEVEL` and `OPTNAME` gives `ENOPROTOOPT`.
- **Read-only or unsupported options** fail with `ENOPROTOOPT`. This also fires when the kernel simply does not know the option - common when copying code between operating systems.
- **Privileged options.** A few options (`SO_PRIORITY` above certain values, `IP_TRANSPARENT`, `SO_BINDTODEVICE`) require `CAP_NET_ADMIN` on Linux; an unprivileged call fails with `EPERM`.
- **Reading back is `getsockopt`, not `setsockopt`.** `setsockopt` has no query form; use `getsockopt` to confirm what the kernel actually stored (it may round, clamp, or double the value).

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `getsockopt` - the query counterpart; read back an option to see what the kernel actually applied
- `socket` - create the socket filehandle that `setsockopt` configures; must be called first
- `bind` - set `SO_REUSEADDR` with `setsockopt` before calling `bind` on a server socket
- `accept` - options set on the listener do not all propagate; re-apply per-connection options on the accepted handle
- `pack` - builds the `OPTVAL` byte string for struct-valued options like `SO_LINGER` and `SO_RCVTIMEO`
- `Socket` - source of `SOL_SOCKET`, `IPPROTO_TCP`, `SO_REUSEADDR`, `TCP_NODELAY`, and the rest of the symbolic constants

Arrays

shift

Remove and return the first element of an array.

`shift` mutates its argument: it removes the element at index `0`, shifts every remaining element down by one position, and returns the removed value. The array shrinks by one. If the array is empty, nothing is removed and `undef` is returned.

Synopsis

```
shift @arr
shift @$aref
shift                   # @_ inside a sub, @ARGV outside
```

What you get back

The scalar that used to be at index `0`, or `undef` if the array was empty. Because `undef` is also a legal array element, a bare `undef` return does not by itself mean the array was empty - use `scalar @arr` or a prior `defined` check when the distinction matters:

```
my @arr  = (undef, 'two');
my $item = shift @arr;      # undef - but @arr was not empty
```

Default array: @ARGV vs @_

Called without an argument, `shift` picks its array from the surrounding context:

- Inside a named or anonymous subroutine - **`@_`**. This is the idiomatic way to consume positional arguments one at a time.
- Outside any sub (including in the file scope of a script, and in `eval STRING`, `BEGIN`, `INIT`, `CHECK`, `UNITCHECK`, `END` blocks) - **`@ARGV`**. This is how top-of-script argument consumption is usually written.

The choice is made at compile time based on lexical position, not at runtime. A `shift` inside a sub always targets `@_` even if the sub is called from file scope.

Examples

Consume a single positional argument inside a method. This is the canonical first line of almost every Perl OO method:

```
sub greet {
    my $self = shift;
    my $name = shift;
    return "hello, $name, from $self";
}
```

The same method written with a list-unpack `@_` slice. Both forms are idiomatic; `shift` wins when you want to peel off the invocant before deciding what to do with the rest of `@_`:

```
sub greet {
    my ($self, $name) = @_;
    return "hello, $name, from $self";
}
```

Consume `@ARGV` at the top of a script, one argument at a time:

```
my $mode = shift // 'default';       # first CLI arg, or 'default'
my $path = shift // die "need a path\n";
# @ARGV now holds whatever is left
```

A FIFO queue: `push` on the tail, `shift` off the head:

```
my @queue;
push @queue, 'a', 'b', 'c';
while (defined(my $job = shift @queue)) {
    handle($job);
}
```

Dereference an array ref in place - `shift` works on `@$aref` exactly as it does on a named array:

```
my $aref = [10, 20, 30];
my $head = shift @$aref;             # 10, $aref now [20, 30]
```

Edge cases

- **Empty array**: returns `undef` and leaves the array untouched. `shift` on an empty array is not an error.
- **Complexity is O(n)**: every remaining element is moved down by one slot. For large arrays used as queues this adds up. When FIFO ordering is not required, `pop` is O(1); when it is, consider a deque-style data structure or a pair of stacks rather than `shift`-ing a million-element array in a loop.
- **`shift` with no argument is context-sensitive by lexical position**: inside a sub it targets `@_`, outside it targets `@ARGV`. Moving a line of code between those scopes silently changes which array it mutates.
- **Tied arrays**: `shift` calls the `SHIFT` method on the tied object if one is defined; otherwise it falls back to `FETCH` on element `0`, then `DELETE`, then a sequence of `FETCH`/`STORE` pairs to shift the remaining elements down, then `STORESIZE`. A custom `SHIFT` handler is almost always worth writing for performance.
- **Array references**: `shift @$aref` and `shift @{ $expr }` both work. `shift $aref` (without the `@` sigil) was an experimental feature added in Perl 5.14 and removed in 5.24 - it is a syntax error in the Perl version pperl targets.
- **`shift` is an lvalue-consumer, not an lvalue**: you cannot assign to the result of `shift` to modify the array. Use `splice` or direct index assignment for that.

- **Return value in list context**: `shift` always returns a single scalar. Assigning its result to a list - `my ($x) = shift @arr` - works only because the single scalar fills the first list slot; the remaining slots are `undef`. Use `splice` to remove multiple elements at once.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `pop` - remove and return the **last** element; O(1), the right choice when FIFO order is not needed
- `push` - append to the tail; pairs with `shift` to form a FIFO queue
- `unshift` - prepend to the head; the inverse of `shift`, and also O(n) for the same reason
- `splice` - remove, insert, or replace any contiguous run of elements; the general tool when `shift` / `pop` / `unshift` / `push` are not enough
- `@ARGV` - the default array `shift` consumes at file scope
- `@_` - the default array `shift` consumes inside a sub; holds the caller's argument list

SysV IPC

shmctl

Control or query a System V shared memory segment.

`shmctl` is the management primitive for shared memory segments created with `shmget`. It performs one of a small set of operations named by `CMD` - reading the segment's metadata, changing its permissions, or marking it for removal. `ID` is the segment identifier returned by `shmget`. `ARG` is a buffer scalar that either supplies an input `shmid_ds` structure (for `IPC_SET`) or receives the returned one (for `IPC_STAT`); for `IPC_RMID` it is ignored.

Synopsis

```
use IPC::SysV qw(IPC_STAT IPC_SET IPC_RMID);

shmctl $id, IPC_STAT, $buf  or die "shmctl STAT: $!";
shmctl $id, IPC_SET,  $buf  or die "shmctl SET: $!";
shmctl $id, IPC_RMID, 0     or die "shmctl RMID: $!";
```

What you get back

Return value follows the `ioctl` convention, not the usual `0`-on-success Unix pattern:

- `undef` on error, with `$!` set to the errno.

- The string `"0 but true"` when the underlying call returned zero. This compares numerically equal to `0` but is boolean-true, so `shmctl(...) or die` still works.
- Any other non-zero value otherwise.

The practical rule: test for falsehood with `shmctl(...) or die "...: $!"` and never compare the return value to literal `0` with `==`.

The `IPC_STAT` form writes a packed `shmid_ds` structure into `ARG`. Unpack it with `IPC::SysV`'s helper or with a manual `unpack` template; the structure layout is platform-dependent so relying on a fixed template is not portable.

Global state it touches

- `$!` is set on failure to the errno from the underlying `shmctl(2)` call. Typical values: `EINVAL` (bad `ID` or `CMD`), `EACCES` (insufficient permissions for the operation), `EPERM` (non-owner attempting `IPC_SET` or `IPC_RMID`), `EFAULT` (bad `ARG` buffer for `IPC_STAT`/`IPC_SET`).

Examples

Read a segment's metadata. `IPC::SysV` provides the `shmid_ds` unpacker; without it, the structure is an opaque byte string:

```
use IPC::SysV qw(IPC_STAT);

my $buf = "";
shmctl($id, IPC_STAT, $buf) or die "stat failed: $!";
# $buf now holds the packed shmid_ds structure
```

Change the permissions on an existing segment. The pattern is read-modify-write on the packed structure:

```
use IPC::SysV qw(IPC_STAT IPC_SET);

my $buf = "";
shmctl($id, IPC_STAT, $buf) or die $!;
# ... modify the mode bits inside $buf ...
shmctl($id, IPC_SET,  $buf) or die $!;
```

Mark a segment for removal. The segment is destroyed once the last process detaches from it; `IPC_RMID` only flags it:

```
use IPC::SysV qw(IPC_RMID);

shmctl($id, IPC_RMID, 0) or die "rmid failed: $!";
```

Guarding on the `"0 but true"` return. Because the zero case is a string that is boolean-true, the idiomatic check is the `or die` form, not a numeric comparison:

```
my $rv = shmctl($id, IPC_STAT, $buf);
defined $rv                          or die "error: $!";
# do NOT write: $rv == 0 and ...   # misses the "0 but true" case
```

Edge cases

- **Missing constants**: `IPC_STAT`, `IPC_SET`, `IPC_RMID` are not built-in. Without `use IPC::SysV` they are bareword function calls and the code either fails to compile under `use strict` or silently passes the wrong integer. Always `use IPC::SysV qw(...)`.
- **Owner-only operations**: `IPC_SET` and `IPC_RMID` require the effective uid to match the segment's creator or owner, or the process to have `CAP_SYS_ADMIN`. Otherwise `$!` becomes `EPERM`.
- **`IPC_RMID` semantics**: marking for removal does not detach the caller and does not invalidate existing attachments. The segment stays usable until every process has called `shmdt` on it; only then is it actually destroyed. A process that attaches after `IPC_RMID` gets `EIDRM` or `EINVAL` depending on timing.
- **Structure layout is not portable**: `shmid_ds` fields and offsets vary between kernels and libc versions. Do not hand-code `unpack` templates for scripts that need to run on multiple platforms; use `IPC::SysV::ShmId` or a similar wrapper.
- **Buffer sizing for `IPC_STAT`**: `ARG` must be a scalar of at least `sizeof(shmid_ds)` bytes on output. Perl grows it as needed; pre-sizing with `$buf = "\0" x 256` is common but not required.
- **Unconfigured kernels**: on Linux built without `CONFIG_SYSVIPC`, the call fails with `ENOSYS`. Container images that drop SysV IPC exhibit the same behaviour.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `shmget` - create or look up the segment whose `ID` `shmctl` operates on
- `shmread` - copy bytes out of an attached segment; uses the `ID` returned by `shmget`, not an `shmctl` result
- `shmwrite` - copy bytes into an attached segment
- `msgctl` - same control pattern for System V message queues; same `"0 but true"` return convention
- `semctl` - same control pattern for System V semaphore sets; richer `CMD` set but identical return-value rules
- `IPC::SysV` - provides the `IPC_STAT` / `IPC_SET` / `IPC_RMID` constants and the `shmid_ds` packers

SysV IPC

shmget

Create or look up a System V shared memory segment and return its identifier.

`shmget` is the allocation step for SysV shared memory. Given a numeric `KEY` (either `IPC_PRIVATE` for a fresh anonymous segment or a well-known number agreed with

other processes), a requested `SIZE` in bytes, and a bit-mask of `FLAGS` (creation flags OR'd with a 9-bit mode), it returns a kernel-assigned integer identifier. That identifier - the `shmid` - is what every subsequent call (`shmctl`, `shmread`, `shmwrite`) uses to refer to the segment. The segment persists in the kernel until it is explicitly removed with `shmctl($shmid, IPC_RMID, 0)` or the system reboots; closing the creating process does **not** release it.

Synopsis

```
shmget KEY, SIZE, FLAGS
```

What you get back

An integer shared-memory identifier (`shmid`) on success, suitable for passing to `shmctl`, `shmread`, and `shmwrite`. On failure returns a false value and sets `$!` to the underlying `errno`.

```
use IPC::SysV qw(IPC_PRIVATE IPC_CREAT S_IRUSR S_IWUSR);

my $shmid = shmget(IPC_PRIVATE, 4096, IPC_CREAT | S_IRUSR | S_IWUSR)
    or die "shmget: $!";
```

The returned identifier is an opaque small integer. Do not do arithmetic on it, do not compare it to `0` for truth - the identifier `0` is a legitimate value on some systems. Always check the return against the documented failure sentinel (see *Differences from upstream*).

Global state it touches

- `$!` - set on failure to the underlying `errno`: `EACCES` (permissions), `EEXIST` (segment for that key already exists and both `IPC_CREAT` and `IPC_EXCL` were given), `EINVAL` (`SIZE` below `SHMMIN` or above `SHMMAX`, or `SIZE` differs from the existing segment when looking one up), `ENFILE` / `ENOSPC` (kernel segment-table full), `ENOENT` (no segment for that key and `IPC_CREAT` was not given), `ENOMEM` (insufficient memory).

The function is otherwise pure with respect to Perl-visible state: no special variables beyond `$!` are touched, `$_` is not read.

Examples

Create a fresh private segment - the usual pattern when parent and children share memory via `fork`:

```
use IPC::SysV qw(IPC_PRIVATE IPC_CREAT S_IRUSR S_IWUSR);

my $size  = 4096;
my $shmid = shmget(IPC_PRIVATE, $size, IPC_CREAT | S_IRUSR | S_
 ↪IWUSR)
    or die "shmget: $!";
# children inherit $shmid across fork and attach via shmread/
 ↪shmwrite
```

Attach to a pre-existing segment by its well-known key. `SIZE` and mode bits are ignored in this form; pass `0` for both:

```
my $key   = 0xDEADBEEF;
my $shmid = shmget($key, 0, 0)
    or die "shmget: $!";
```

Create-or-open with `IPC_CREAT`. If the key already exists the existing `shmid` is returned; otherwise a new segment of `SIZE` bytes is created with the given mode:

```
use IPC::SysV qw(IPC_CREAT S_IRUSR S_IWUSR S_IRGRP);

my $shmid = shmget(0x1234, 8192, IPC_CREAT | S_IRUSR | S_IWUSR | S_
↪IRGRP)
    or die "shmget: $!";
```

Fail if a segment already exists, using `IPC_EXCL`. This is the safe pattern for a process that wants to be the unique creator:

```
use IPC::SysV qw(IPC_CREAT IPC_EXCL S_IRUSR S_IWUSR);
use Errno qw(EEXIST);

my $shmid = shmget(0x1234, 4096, IPC_CREAT | IPC_EXCL | S_IRUSR | S_
↪IWUSR);
if (!$shmid) {
    die "shmget: $!" unless $! == EEXIST;
    # another process won the race; fall back to plain lookup
    $shmid = shmget(0x1234, 0, 0) or die "shmget (lookup): $!";
}
```

Full create-use-destroy cycle with guaranteed cleanup via `END`:

```
use IPC::SysV qw(IPC_PRIVATE IPC_CREAT IPC_RMID S_IRUSR S_IWUSR);

my $shmid = shmget(IPC_PRIVATE, 1024, IPC_CREAT | S_IRUSR | S_IWUSR)
    or die "shmget: $!";

END {
    shmctl($shmid, IPC_RMID, 0) if defined $shmid;
}

shmwrite($shmid, "hello", 0, 5)  or die "shmwrite: $!";
shmread($shmid, my $buf, 0, 5)   or die "shmread: $!";
```

Edge cases

- **`use IPC::SysV` is effectively mandatory.** The flag constants (`IPC_PRIVATE`, `IPC_CREAT`, `IPC_EXCL`, `IPC_RMID`) and mode bits (`S_IRUSR`, `S_IWUSR`, `S_IRGRP`, ...) are not built-ins. Without the import you end up passing bare barewords that stringify to `0` and silently do the wrong thing.
- **A literal `0` key *is* `IPC_PRIVATE`.** `IPC_PRIVATE` is defined as `0`, so a bare `0` never

looks up the segment at key `0` - it creates a fresh anonymous one, whether or not you passed `IPC_CREAT`. If `FLAGS` then carries no permission bits the new segment is mode `000` and the next `shmread`/`shmwrite` fails with `EACCES`. Import the constant by name so the intent is visible.

- **`SIZE` is rounded up to a page boundary** by the kernel; the returned segment may be larger than requested. Ask the kernel for the effective size with `shmctl($shmid, IPC_STAT, $ds)` and unpack the `shmid_ds` structure if you need the real number.
- **Segment persists beyond process exit.** A crashed or killed creator leaves the segment in the kernel. Use `ipcs -m` to list orphaned segments and `ipcrm -m <shmid>` to remove them. Install an `END` block or a signal handler to call `shmctl IPC_RMID` on shutdown.
- **`IPC_RMID` is deferred, not immediate.** It marks the segment for destruction; actual removal happens once the attach count drops to zero. Safe to call right after `shmget` if you know no one else will attach.
- **Kernel limits.** `SHMMNI` caps the total number of segments system-wide (`/proc/sys/kernel/shmmni`). `SHMMAX` caps the size of one segment (`/proc/sys/kernel/shmmax`). On a shared host these are easy to hit; treat `ENOSPC` and `EINVAL`-from-large-size as configuration issues, not bugs.
- **SysV IPC is not available on every platform.** Upstream Perl built on a system without `HAS_SHM` dies with `"System V IPC is not implemented on this machine"`. pperl targets Linux only, where SysV IPC is always present, so the call is always dispatched.

Differences from upstream

- Returns `-1` rather than `undef` on failure. Both values are false under Perl's truth rules, so the idiomatic `shmget(...) or die "$!"` works unchanged, but code that distinguishes "not an identifier" via `defined $shmid` will read `-1` as defined. Prefer checking for truth, or compare against `-1` explicitly when you want to branch on the failure sentinel. The `-1` is the raw `shmget(2)` return value, which is the documented error sentinel for that syscall.

See also

- `shmctl` - control operations on the segment `shmget` returned; use `IPC_RMID` to delete it, `IPC_STAT` to inspect size and permissions
- `shmread` - read `SIZE` bytes from the segment into a Perl scalar
- `shmwrite` - write a Perl scalar into the segment at an offset
- `msgget` - the equivalent allocation step for SysV message queues when you want a queue, not a flat buffer
- `semget` - the equivalent for SysV semaphores when the goal is synchronisation rather than shared data
- `IPC::SysV` - source of the `IPC_*` and `S_*` flag constants `shmget` requires; no local reference page, see upstream POD

SysV IPC

shmread

Copy bytes out of a System V shared memory segment into a Perl scalar.

`shmread` attaches to the shared memory segment identified by `ID`, reads `SIZE` bytes starting at byte offset `POS`, writes them into `VAR`, and detaches. `VAR` is assigned to as a whole - whatever it held before is replaced by exactly `SIZE` bytes of data. The variable is marked **tainted** under `-T`, because its contents came from outside the program.

Synopsis

```
use IPC::SysV qw(IPC_PRIVATE S_IRUSR S_IWUSR);
my $id = shmget(IPC_PRIVATE, 4096, S_IRUSR | S_IWUSR)
    or die "shmget: $!";
shmread($id, my $buf, 0, 128)
    or die "shmread: $!";
```

What you get back

Truthy on success, false on failure with `$!` set. On success `VAR` has been overwritten with exactly `SIZE` bytes; no short reads - either the whole range is copied in or the call fails. On failure `VAR`'s contents are unspecified; don't rely on them.

Always check the return value. A bad `ID`, an out-of-range `POS`/`SIZE`, or a segment the process lacks permission to attach all surface here as a false return:

```
shmread($id, my $buf, $pos, $size)
    or die "shmread(id=$id, pos=$pos, size=$size): $!";
```

Global state it touches

- `$!` - set on failure (any of the underlying `shmat`/`memcpy`/`shmdt` errors surface here).
- Taint flag on `VAR` - set under `-T` regardless of how `VAR` was declared, because the bytes originate outside the program.

Examples

Read a fixed-size record out of a segment:

```
use IPC::SysV qw(IPC_PRIVATE S_IRUSR S_IWUSR);
my $id = shmget(IPC_PRIVATE, 1024, S_IRUSR | S_IWUSR) or die
 ↪"shmget: $!";
shmwrite($id, "hello, world", 0, 12)                  or die
 ↪"shmwrite: $!";
shmread($id, my $buf, 0, 12)                          or die
 ↪"shmread: $!";
print $buf, "\n";                                     # hello, world
```

Read at an offset. `POS` and `SIZE` are in bytes, not elements - the segment has no notion of records:

```
shmread($id, my $chunk, 256, 64)
    or die "shmread: $!";                              # bytes 256..
↪319
```

Unpack a binary record straight out of shared memory:

```
shmread($id, my $raw, 0, 16) or die "shmread: $!";
my ($magic, $len, $flags) = unpack "a4 N N", $raw;
```

Reading tainted data under `-T`. The assignment is guaranteed to produce a tainted scalar even if `VAR` already held an untainted value:

```
# perl -T
shmread($id, my $buf, 0, $n) or die "shmread: $!";
# $buf is tainted; untaint via a regex match before using
# it in a sensitive context.
```

Edge cases

- **`VAR` is assigned to, not appended to.** Any previous value is discarded. Use a fresh lexical if you need both old and new data.
- **`SIZE` is exact, not a maximum.** `shmread` does not do short reads; requesting more bytes than the segment holds from `POS` onward fails rather than returning what fits.
- **`POS` and `SIZE` must lie within the segment** declared at `shmget` time. `POS + SIZE` past the segment's end fails with `EINVAL`.
- **Zero `SIZE`** produces an empty string in `VAR` and succeeds, assuming `POS` itself is valid.
- **Permissions.** The calling process must have read permission on the segment, as dictated by the mode bits passed to `shmget` (and, where applicable, by the segment's owner and current uid/gid).
- **`VAR` need not be pre-sized.** Unlike `sysread`, you do not `substr`-pad or preallocate `VAR` - `shmread` assigns the result directly.
- **Taint.** Under `-T`, `VAR` is tainted on success regardless of what it held before. Launder through a capture-group match (`$data =~ /\A(.*)\z/s and $data = $1`) only once you've validated the bytes are safe for the sensitive operation you're about to perform.
- **Not every platform has SysV IPC.** On systems without `shmat(2)` the call fails at runtime (typically `ENOSYS`); portable code should guard with an `eval` or feature probe.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `shmwrite` - the inverse: copy a string into a segment at a given offset
- `shmget` - obtain or create the segment `ID` that `shmread` operates on
- `shmctl` - query segment metadata (`IPC_STAT` fills in an `shmid_ds` you can use to bound-check `POS + SIZE`) or remove the segment when done
- `sysread` - the file-descriptor analogue; same "bytes into a scalar" shape, different source
- `unpack` - turn the raw bytes returned in `VAR` into typed fields
- `IPC::SysV` - the module that supplies `IPC_PRIVATE`, `S_IRUSR`, and the other constants you'll pass to `shmget`

SysV IPC

shmwrite

Copy bytes into a System V shared memory segment.

`shmwrite` attaches to the shared memory segment identified by `ID`, copies up to `SIZE` bytes from `STRING` starting at byte offset `POS` inside the segment, and detaches. If `STRING` is longer than `SIZE`, only the first `SIZE` bytes are written; if it is shorter, the remainder of the `SIZE`-byte window is zero-filled with NUL bytes. The segment `ID` is the value returned by an earlier `shmget` call (or obtained from another process via some out-of-band channel).

Synopsis

```
shmwrite ID, STRING, POS, SIZE
```

What you get back

True on success, false on error, with `$!` set to the reason (typically `EINVAL`, `EACCES`, or `EIDRM`). Unlike most I/O built-ins the return value is a plain boolean - there is no byte-count variant, because the count is always exactly `SIZE` when the call succeeds.

```
shmwrite($id, $payload, 0, 4096)
    or die "shmwrite($id) failed: $!";
```

Global state it touches

- `$!` is set on failure to the underlying `shmat` / `memcpy` / `shmdt` errno.
- No other special variables are read or written. Unlike `shmread`, `shmwrite` does not taint anything - it is the *source* of bytes, not a sink.

Examples

Publish a fixed-size record at the start of a segment:

```
use IPC::SysV qw(IPC_PRIVATE S_IRUSR S_IWUSR);
my $id = shmget(IPC_PRIVATE, 4096, S_IRUSR | S_IWUSR)
    or die "shmget: $!";
shmwrite($id, "hello, world", 0, 4096)
    or die "shmwrite: $!";
```

A `STRING` shorter than `SIZE` is NUL-padded to fill the window - useful for clearing stale trailing bytes from a previous writer:

```
shmwrite($id, "short", 0, 64);   # 5 bytes of "short", then 59 NULs
```

A `STRING` longer than `SIZE` is truncated; the tail is silently discarded, so size your window to the largest record you ever intend to publish:

```
shmwrite($id, "x" x 10_000, 0, 128);   # only the first 128 x's land
```

Writing to a non-zero offset lets multiple fixed-size records share one segment. Pair it with a matching `shmread` on the reader side:

```
my $slot = 3;
my $len  = 64;
shmwrite($id, $record, $slot * $len, $len)
    or die "shmwrite slot $slot: $!";
```

Check the return value - a segment that has been marked for deletion with `IPC_RMID` continues to accept writes from processes already attached, but fails for new `shmwrite` calls with `EIDRM`:

```
unless (shmwrite($id, $msg, 0, $SIZE)) {
    warn "segment $id gone: $!";
    # reacquire via shmget() and retry
}
```

Edge cases

- **Kernel must be built with SysV IPC.** On systems without SysV shared memory `shmwrite` croaks with `shmwrite not implemented`. This is a compile-time property of the kernel / libc, not a runtime condition you can probe for.
- **`SIZE` of `0`** is accepted by the kernel and writes nothing; `shmwrite` returns true. Use it only when you genuinely mean "no-op", not as an error signal.
- **`POS + SIZE` must fit inside the segment.** Overrunning the segment boundary fails with `EINVAL` and leaves the segment untouched - the copy is not partial.
- **Binary payloads are fine.** `STRING` is treated as a byte string; embedded NULs are written literally. If the string carries the UTF-8 flag, the raw internal bytes are what hits the segment - use `pack` or `Encode::encode` first if you need a specific encoding on the wire.

- **No locking.** The kernel guarantees that each `shmwrite` call is atomic with respect to `shmat` / `shmdt`, but two concurrent writers can still interleave byte ranges within the segment. Coordinate with `semop` or a user-space lock if multiple processes write the same region.
- **Permissions** come from the segment's mode at `shmget` time. Attaching read-only and calling `shmwrite` fails with `EACCES`.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `shmread` - inverse operation; copy bytes *out* of a segment into a Perl scalar
- `shmget` - obtain the `ID` that `shmwrite` writes into
- `shmctl` - inspect segment attributes, change permissions, or mark a segment for deletion
- `semop` - the usual companion primitive for serialising access to a shared segment
- `pack` - build the fixed-layout byte string you hand to `shmwrite` when the segment holds structured records
- `IPC::SysV` - constants (`IPC_PRIVATE`, `S_IRUSR`, `IPC_RMID`) used to construct the arguments to `shmget` and `shmctl`

Sockets

shutdown

Shut down one direction of a socket connection, or both.

`shutdown` tells the kernel to stop a socket doing further reads, further writes, or both - without closing the file descriptor. Unlike `close`, the effect is immediate and propagates through every forked copy of the descriptor, not just this process's. `HOW` has the same meaning as the argument to the `shutdown(2)` syscall.

Synopsis

```
shutdown SOCKET, HOW
```

What you get back

1 on success. On failure, `undef` if `SOCKET` is not a valid filehandle, or `0` with `$!` set for any other error (the socket is not connected, already shut down in that direction, etc.).

Two distinct failure returns, so a three-way test is sometimes warranted:

```
my $r = shutdown($sock, 1);
if    (!defined $r) { die "bad filehandle" }
elsif (!$r)         { die "shutdown failed: $!" }
```

The HOW argument

`HOW` is a small integer matching the POSIX `SHUT_*` constants:

- `0` - stop reading. The peer can still send, but data will be discarded by the kernel and `read` / `sysread` on this end will return end-of-file.
- `1` - stop writing. Queued output is flushed, then the peer sees end-of-file on its read side. `print` / `syswrite` on this end will fail and raise `SIGPIPE`.
- `2` - stop both directions. Equivalent to calling `shutdown` with `0` and then `1`.

The symbolic names `SHUT_RD`, `SHUT_WR`, `SHUT_RDWR` live in `Socket` if you prefer readable code:

```
use Socket qw(SHUT_WR);
shutdown $sock, SHUT_WR;
```

shutdown vs close

`close` drops the file descriptor in the current process only. Other processes that inherited the descriptor through `fork` keep their copies open, and the peer does not see end-of-file until the last copy is closed.

`shutdown` acts on the underlying socket, not on the descriptor. After `shutdown($sock, 1)` every forked copy sees the write half closed and the peer reads end-of-file right away - even though the descriptor itself is still open and can still be read from.

That is the usual reason to reach for `shutdown`: a parent and child share a socket, the child has finished sending, and the parent must keep reading replies. `close` in the child would be invisible to the peer; `shutdown($sock, 1)` is not.

Half-close for request/response protocols

A client that sends a complete request and then needs the server to know there is no more input uses `shutdown` with `HOW = 1`:

```
use IO::Socket::INET;
my $sock = IO::Socket::INET->new(PeerAddr => "example.com:80")
    or die "connect: $!";

print $sock "GET / HTTP/1.0\r\nHost: example.com\r\n\r\n";
shutdown $sock, 1;          # signal end-of-request to the server

while (my $line = <$sock>) {
    print $line;
}
close $sock;
```

Without the `shutdown`, servers that read until end-of-input would block waiting for more request data.

Examples

Stop reading but keep writing - the peer may continue to send, we just will not look at it:

```
shutdown $sock, 0;
```

Stop writing, keep reading - typical half-close after finishing a request:

```
shutdown $sock, 1;
```

Full shutdown, then close the descriptor:

```
shutdown $sock, 2;
close $sock;
```

Using `Socket` constants for readability:

```
use Socket qw(SHUT_RD SHUT_WR SHUT_RDWR);
shutdown $sock, SHUT_WR;
```

Check the specific error when a shutdown fails on an already-closed direction:

```
unless (shutdown $sock, 1) {
    warn "shutdown write failed: $!";
}
```

Edge cases

- **Writing after `shutdown($sock, 1)`** fails and raises `SIGPIPE`. The default action terminates the process; install `$SIG{PIPE} = 'IGNORE'` (or a handler) to convert it into a normal write failure you can check.
- **Reading after `shutdown($sock, 0)`** returns end-of-file immediately, not an error. Any data already queued in the kernel's receive buffer is discarded.
- **Not a socket**: calling `shutdown` on a regular filehandle sets `$!` to `ENOTSOCK` and returns `0`.
- **Not connected**: `shutdown` on a socket that was never connected (a fresh `socket` result, or a listening socket) sets `$!` to `ENOTCONN` and returns `0`.
- **Double shutdown**: shutting down the same half twice typically returns `0` with `ENOTCONN`. Portable code ignores the second failure rather than treating it as fatal.
- **Closed filehandle**: returns `undef`; under `use warnings` a `shutdown()` on `closed socket` warning is emitted.
- **Descriptor is not closed**: `shutdown` disables the directions you asked for but leaves the file descriptor open. Call `close` (or let the handle go out of scope) when you are actually done with it.
- **Shared descriptors across `fork`**: this is the case `shutdown` is designed for. All processes holding the descriptor see the shutdown take effect immediately; they do not get separate views.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `close` - release the descriptor in this process; does not affect forked copies and does not signal end-of-file to the peer until the last copy closes
- `socket` - create the socket that `shutdown` operates on
- `connect` - establish the connection; `shutdown` on an unconnected socket fails with `ENOTCONN`
- `accept` - produces the connected socket on the server side that typically gets half-closed with `shutdown`
- `Socket` - exports `SHUT_RD`, `SHUT_WR`, `SHUT_RDWR` and the rest of the socket-level constants
- `$!` - system error string to check after a `0` return

Numeric functions

sin

Return the sine of a number given in radians.

`sin` is the standard trigonometric sine: it takes an angle measured in **radians** and returns the ratio of the opposite side to the hypotenuse of the corresponding right triangle. If `EXPR` is omitted, `sin` operates on `$_`. The result is always a floating-point number in the closed interval `[-1, 1]`.

Synopsis

```
sin EXPR
sin
```

What you get back

A double-precision float in `[-1, 1]`. Perl forwards the call to the platform's C `sin(3)` routine, so the precision and rounding behaviour match your libm. Special values follow IEEE 754: `sin(0)` is exactly `0`, `sin("inf")` and `sin("nan")` are both `NaN`.

Radians, not degrees

The single most common mistake with `sin` is feeding it degrees. A full turn is `2 * π` radians, not `360`. Convert once and keep the converted value:

```
use POSIX qw(acos);
my $pi = acos(-1);                  # π to full double precision

sub deg2rad { $_[0] * $pi / 180 }
```

(continues on next page)

(continued from previous page)

```
print sin( deg2rad(30) ), "\n";      # 0.5
print sin( 30 ), "\n";               # -0.988031624092862 - wrong if␣
↪you meant degrees
```

exports `deg2rad` and `rad2deg` if you would rather not write the conversion yourself.

Examples

`sin(π/2)` is mathematically exactly 1, but because π is not representable in binary floating-point the computed result falls a few ulps short:

```
use POSIX qw(acos);
my $pi = acos(-1);

printf "%.17f\n", sin($pi / 2);      # 1.00000000000000000
printf "%.17f\n", sin($pi);          # 0.00000000000000012 - not␣
↪exactly 0
```

Treat any "should be zero" trig result as approximately zero. Compare with a tolerance, not with ==.

Degree-to-radian conversion inline, without pulling in a module. `3.14159` is accurate to six digits - fine for plotting, not for numeric analysis:

```
for my $deg (0, 30, 45, 60, 90) {
    printf "sin(%2d°) = %+.4f\n",
           $deg, sin($deg * 3.14159 / 180);
}
# sin( 0°) = +0.0000
# sin(30°) = +0.5000
# sin(45°) = +0.7071
# sin(60°) = +0.8660
# sin(90°) = +1.0000
```

For work that needs full double precision, replace `3.14159` with `acos(-1)` or `atan2(1, 1) * 4`.

Numerical derivative of `sin` by finite differences. The symmetric two-point formula `(f(x+h) - f(x-h)) / (2h)` should recover `cos(x)`:

```
my $x = 1.0;
my $h = 1e-6;
my $deriv = (sin($x + $h) - sin($x - $h)) / (2 * $h);

printf "derivative:  %.10f\n", $deriv;   # 0.5403023059
printf "cos(%.1f):     %.10f\n", $x, cos($x);
```

Picking `$h` is a trade-off: too large and truncation error dominates, too small and subtraction cancellation destroys the result. `1e-6` is a reasonable default for double precision.

Walk the unit circle and print `(cos θ, sin θ)` at eight evenly spaced angles:

```
use POSIX qw(acos);
my $pi = acos(-1);

for my $k (0 .. 7) {
    my $theta = $k * $pi / 4;
    printf "%.3f  %+.3f  %+.3f\n",
           $theta, cos($theta), sin($theta);
}
```

Inverse sine (arcsine) via the identity from `perlfunc`:

```
sub asin { atan2( $_[0], sqrt(1 - $_[0] * $_[0]) ) }

print asin(0.5), "\n";              # 0.523598775598299 (= π/6)
```

Edge cases

- **Non-numeric arguments coerce to `0`.** `sin("hello")` returns `0` because the string is numified to `0` first. Under `use warnings` this produces an `Argument "hello" isn't numeric in sin` warning. Validate input with `looks_like_number` from `Scalar::Util` if the source is untrusted.
- **Very large arguments lose precision.** `sin` first reduces its argument modulo 2π, and for `|x|` on the order of `1e16` the nearest representable double differs from `x` by more than π. The result is still a number in `[-1, 1]`, but it is essentially random. Reduce the angle yourself before the call if you are accumulating a phase over many iterations.
- **`undef` coerces to `0`** and returns `0`, with the usual `uninitialized` warning under `use warnings`.
- **IEEE specials**: `sin("inf")`, `sin("-inf")`, and `sin("nan")` all return `NaN`. Compare with `$result != $result` (the only value not equal to itself) to detect `NaN`, or use `POSIX::isnan`.
- **Default argument.** `sin;` with no argument reads `$_`. Inside a `while (<>)` loop this operates on the current line, which is almost always a bug.
- **No complex-number support** in the built-in. For complex arguments use , which overloads `sin` via operator overloading:

  ```
  use Math::Complex;
  my $z = cplx(1, 2);
  print sin($z), "\n";              # 3.16577851321617+1.
  ↪9596010414216i
  ```

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `cos` - cosine of an angle in radians; the companion function most often used alongside `sin`
- `atan2` - two-argument arctangent, the usual building block for inverse trig (`asin`, `acos`) in pure Perl
- `sqrt` - square root; appears in the classical `asin` identity `asin(x) = atan2(x, sqrt(1 - x*x))`
- - `asin`, `acos`, `atan`, hyperbolic variants, `deg2rad`/`rad2deg`, and great-circle helpers
- - overloads `sin` (and the rest of the trig family) for complex arguments
- `POSIX` - exposes `asin` directly, plus `isnan`/`isinf` for detecting the non-finite results discussed above

Processes

sleep

Pause the process for a number of whole seconds.

`sleep` suspends execution of the current process for `EXPR` seconds, or - with no argument - until a signal wakes it. The argument is truncated to an integer; sub-second delays require a different tool (see below). The return value is the number of seconds actually slept, which can be less than requested if a signal interrupted the call.

Synopsis

```
sleep EXPR
sleep
```

What you get back

An integer: the number of seconds the process was actually suspended. On an uninterrupted call this equals the (truncated) argument. If a signal delivered during the sleep unblocks the call early, the return value is the shorter, real elapsed time. For bare `sleep`, the value is whatever the implementation slept before a signal arrived.

```
my $slept = sleep 10;
warn "woken early after $slept s" if $slept < 10;
```

Argument handling

- **Positive integer** - the normal case. Sleep that many seconds.
- **Non-integer** (e.g. `2.9`) - the fractional part is **discarded**. `sleep 2.9` sleeps two seconds, not nearly three. For fractional delays use `Time::HiRes::sleep`, `Time::HiRes::usleep`, or the four-argument form of `select` - see below.

- **Negative integer** - `sleep` does **not** sleep. It emits a warning, sets `$!` (errno), and returns zero.
- **Zero** - `sleep 0` is permitted, but still performs a call into the underlying platform sleep primitive with whatever side effects that entails. It is therefore **not** exactly identical to skipping the call. In particular, `sleep 0` is not the same as a bare `sleep` with no argument.
- **No argument** - `sleep` with no argument sleeps indefinitely, until a signal is delivered. This is distinct from `sleep 0`.

Interruption by signals

`sleep` returns early if the process receives a signal whose handler runs (or whose default action does not terminate the process). The canonical pattern for a bounded wait that can be cancelled from elsewhere is an `alarm` combined with a signal handler:

```
eval {
    local $SIG{ALRM} = sub { die "Alarm!\n" };
    sleep;
};
die $@ unless $@ eq "Alarm!\n";
```

Because `sleep` is itself commonly implemented on top of `alarm` on Unix systems, you generally **cannot** mix `alarm` and `sleep` calls in the same program without surprises. Pick one.

Sub-second delays

`sleep` is integer-only by design. For finer-grained waits use one of:

- `Time::HiRes::sleep($seconds)` - accepts a floating-point number of seconds. Since Perl 5.8 `Time::HiRes` ships with the core distribution.
- `Time::HiRes::usleep($microseconds)` - integer microseconds.
- The four-argument form of `select` with the first three arguments `undef`:

  ```
  select undef, undef, undef, 0.25;   # sleep 250 ms
  ```

- `syscall` into `setitimer(2)` if your platform supports it and you need tight control over timers.

See also `POSIX::pause`, which blocks the process until any signal arrives - effectively a bare `sleep` without an implementation-defined maximum.

Examples

Fixed delay:

```
print "working...\n";
sleep 3;
print "done\n";
```

Poll loop with a back-off:

```
until (ready()) {
    sleep 1;
}
```

Detecting an early wake:

```
my $want = 30;
my $got  = sleep $want;
if ($got < $want) {
    warn "interrupted after $got of $want seconds\n";
}
```

Wait for a signal without burning CPU - bare `sleep` suspends until `SIGALRM` (or any other caught signal) arrives:

```
$SIG{USR1} = sub { warn "got USR1\n" };
sleep;                                  # wakes on USR1 delivery
```

Cancelling a long wait with `alarm`:

```
eval {
    local $SIG{ALRM} = sub { die "timeout\n" };
    alarm 5;
    sleep 3600;                         # at most 5 seconds really
    alarm 0;
};
alarm 0;
die $@ if $@ && $@ ne "timeout\n";
```

Edge cases

- **Bare `sleep` does not truly sleep "forever"**. The implementation has a maximum - in practice, whatever the underlying C `sleep(3)` or equivalent accepts - but for all realistic purposes it blocks until a signal arrives. Treat it as an indefinite wait.
- **`sleep 0` ≠ no call**. The platform `sleep(0)` syscall still runs. On most systems this yields the CPU briefly; do not rely on the specific behaviour across platforms.
- **`sleep` versus wall-clock time**. On older systems `sleep` could return up to a full second *early* due to coarse second counting. Modern systems sleep the full amount, but may overshoot: the scheduler may not wake your process immediately on a busy system. `sleep N` guarantees *at least* `N` seconds of suspension on modern kernels, not *exactly* `N`.
- **Negative arguments are not clamped**. They are a hard no-op that sets `$!`. If you compute the argument, validate it first.
- **Mixing with `alarm` is fragile**. The alarm timer and the sleep timer may be the same resource. Establish one discipline (either `alarm` + bare `sleep`, or plain `sleep N` without alarms) and stick to it within a program.

- **Signals that merely set a flag still wake `sleep`.** Any signal whose delivery causes a handler to run returns control to Perl; `sleep` then returns early even if the handler itself does nothing visible.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `alarm` - schedule a `SIGALRM` after a given number of seconds; the classic way to bound a `sleep` or any other blocking call
- `select` - in its four-argument form with an undefined readfds/writefds/exceptfds triple, provides sub-second sleeps
- `wait`, `waitpid` - block on child processes rather than on time
- `Time::HiRes::sleep` - floating-point seconds
- `Time::HiRes::usleep` - integer microseconds
- `POSIX::pause` - block until any signal arrives

Sockets

socket

Create a socket filehandle.

`socket` opens a socket of the kind described by `DOMAIN`, `TYPE`, and `PROTOCOL` and attaches it to `SOCKET`, which becomes a usable Perl filehandle. It is the thin Perl veneer over the `socket(2)` system call - nothing more, nothing less. The new socket is unbound and unconnected; you still need `bind`, `connect`, or `listen`+`accept` to do anything with it. Pull the symbolic constants in with `use Socket;` before the call - writing the raw integers is portable only by accident.

Synopsis

```
use Socket;
socket SOCKET, DOMAIN, TYPE, PROTOCOL
```

What you get back

1 on success, `undef` on failure (with `$!` set to the `errno` from the underlying `socket(2)`). On success `SOCKET` is now a filehandle you can `bind`, `connect`, `send` / `recv` on, or read and write with the ordinary I/O operators once it reaches a connected state. The return value is worth checking on every call - a failure here means you have nothing useful to pass to the next socket call, which will then fail in a more confusing way.

```
socket my $sock, PF_INET, SOCK_STREAM, getprotobyname("tcp")
    or die "socket: $!";
```

SOCKET can be a bareword filehandle (SOCK), a glob (*SOCK), or an undefined scalar that will be auto-vivified into an anonymous filehandle reference. The auto-vivified form is the modern idiom and the only one that scopes cleanly with my.

Global state it touches

- $! - set to the errno from socket(2) on failure.
- $^F - the system-file-descriptor threshold. On systems with close-on-exec, the new descriptor gets FD_CLOEXEC set when its number exceeds $^F (default 2, so descriptors 0–2 stay inheritable and everything else is auto-closed across exec). Raise it before socket if you specifically want the new socket to survive an exec.

Examples

A client socket for a TCP connection. Create, connect, talk:

```
use Socket;
socket my $sock, PF_INET, SOCK_STREAM, getprotobyname("tcp")
    or die "socket: $!";
my $addr = pack_sockaddr_in(80, inet_aton("example.com"));
connect $sock, $addr or die "connect: $!";
syswrite $sock, "GET / HTTP/1.0\r\nHost: example.com\r\n\r\n";
```

A listening TCP server socket. Create, mark reusable, bind, listen:

```
use Socket;
socket my $srv, PF_INET, SOCK_STREAM, getprotobyname("tcp")
    or die "socket: $!";
setsockopt $srv, SOL_SOCKET, SO_REUSEADDR, pack("l", 1)
    or die "setsockopt: $!";
bind $srv, pack_sockaddr_in(8080, INADDR_ANY) or die "bind: $!";
listen $srv, SOMAXCONN                        or die "listen: $!";
```

A UDP socket - SOCK_DGRAM instead of SOCK_STREAM, and no connect is needed before send when you pass an explicit destination:

```
use Socket;
socket my $udp, PF_INET, SOCK_DGRAM, getprotobyname("udp")
    or die "socket: $!";
my $to = pack_sockaddr_in(53, inet_aton("1.1.1.1"));
send $udp, $query, 0, $to or die "send: $!";
```

A Unix-domain stream socket for local IPC. PF_UNIX with a filesystem path instead of an IP address:

```
use Socket;
socket my $sock, PF_UNIX, SOCK_STREAM, 0 or die "socket: $!";
connect $sock, pack_sockaddr_un("/run/app.sock") or die "connect: $!
 ↪";
```

Letting PROTOCOL default to 0. The kernel picks the default protocol for the domain/type pair - for PF_INET + SOCK_STREAM that is TCP, for PF_INET + SOCK_DGRAM that is UDP. The

`getprotobyname` form above is more explicit but the `0` form is equivalent and avoids one lookup:

```
use Socket;
socket my $sock, PF_INET, SOCK_STREAM, 0 or die "socket: $!";
```

Edge cases

- **`use Socket` is not optional.** Without it, `PF_INET`, `SOCK_STREAM`, and friends are bareword strings, not integer constants, and the call fails at the syscall boundary with `Invalid argument`. The import is what turns them into numbers.
- **`SOCKET` is clobbered on success.** If the slot holds an open filehandle, it is closed first. There is no warning - the previous socket just silently goes away. Use a fresh `my` variable for each call.
- **Failure leaves `SOCKET` unchanged.** On `undef` return the scalar is neither opened nor modified; a prior value (including an earlier, still-open socket) is still there.
- **Descriptor limits.** `socket` is the point where `EMFILE` (per-process) and `ENFILE` (system-wide) descriptor-table exhaustion shows up. Long-running servers should treat these as recoverable rather than fatal.
- **`EAFNOSUPPORT` / `EPROTONOSUPPORT` / `ESOCKTNOSUPPORT`.** The kernel rejected the domain/type/protocol triple. Common cause: asking for `PF_INET6` on a kernel built without IPv6, or an invalid protocol number passed through from a stale `getprotobyname` lookup.
- **Non-blocking mode is a separate step.** `socket` returns a blocking descriptor. Use `fcntl` with `F_SETFL | O_NONBLOCK` (or the `IO::Socket` equivalent) if you need non-blocking I/O.
- **Close-on-exec is automatic past `$^F`.** Perl sets `FD_CLOEXEC` on the new descriptor when its fd number is greater than `$^F`. A socket will therefore not survive an `exec` unless you lowered that threshold or clear the flag with `fcntl`.
- **Numeric domain/type values are not portable.** `2` happens to be `AF_INET` on Linux; it is something else elsewhere. Always go through the `Socket` module constants.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `bind` - attach a socket created by `socket` to a local address, required before `listen` or before sending from a known port
- `connect` - initiate an outgoing connection on a `SOCK_STREAM` socket, or set the default peer on a `SOCK_DGRAM` socket
- `listen` - mark a bound `SOCK_STREAM` socket as passive so `accept` can pull connections off it

- `accept` - return the next pending connection on a listening socket as a fresh filehandle
- `socketpair` - create a connected pair of sockets in one call; the usual choice when both ends live in the same process or parent/child
- `Socket` - the module that supplies `PF_INET`, `SOCK_STREAM`, `inet_aton`, `pack_sockaddr_in` and the rest of the constants and packers you pass to `socket`
- `$^F` - descriptor threshold that controls close-on-exec for the new socket

Sockets

socketpair

Create an unnamed, connected pair of sockets that talk to each other.

`socketpair` opens two fresh filehandles in `SOCKET1` and `SOCKET2` already wired together, so anything written to one comes out of the other. No addresses, no `bind`, no `listen`, no `accept`. The commonest use is talking to a child process created by `fork` - each side keeps one handle and `close`s the other.

Synopsis

```
use Socket;
socketpair(SOCKET1, SOCKET2, DOMAIN, TYPE, PROTOCOL) or die $!;
socketpair(my $s1, my $s2, AF_UNIX, SOCK_STREAM, PF_UNSPEC) or die
 ↪$!;
```

What you get back

True on success, false on failure (with `$!` set). On success both filehandle arguments are populated with fresh handles; on failure they are left untouched. Always check the return value - every domain/type/protocol combination is platform-dependent.

```
socketpair(my $a, my $b, AF_UNIX, SOCK_STREAM, PF_UNSPEC)
    or die "socketpair: $!";
```

The `DOMAIN`, `TYPE`, and `PROTOCOL` constants come from the `Socket` module. `AF_UNIX` with `SOCK_STREAM` and `PF_UNSPEC` is the portable default.

Global state it touches

- `$!` is set on failure to the system `errno`.
- `$^F` - the maximum system file descriptor below which close-on-exec is **not** set. On systems with `FD_CLOEXEC`, descriptors above `$^F` get close-on-exec automatically; below it, they survive `exec`. Default is 2 (stdin/stdout/stderr). Raise it before `socketpair` if you want the new handles to be inherited across `exec`.

Examples

Plain parent/child pipe using a stream socketpair. Each side closes the end it does not need, so EOF propagates correctly:

```
use Socket;

socketpair(my $child, my $parent, AF_UNIX, SOCK_STREAM, PF_UNSPEC)
    or die "socketpair: $!";

my $pid = fork // die "fork: $!";
if ($pid == 0) {
    close $child;
    print $parent "hello from child\n";
    close $parent;
    exit;
}
close $parent;
chomp(my $msg = <$child>);          # "hello from child"
waitpid $pid, 0;
```

Bidirectional stream: both processes read *and* write on the same handle. No `shutdown` is needed because neither side is closing a direction early:

```
use Socket;

socketpair(my $a, my $b, AF_UNIX, SOCK_STREAM, PF_UNSPEC) or die $!;
my $pid = fork // die $!;
if ($pid == 0) {
    close $a;
    while (my $line = <$b>) {
        chomp $line;
        print $b "echo: $line\n";
    }
    exit;
}
close $b;
print $a "ping\n";
chomp(my $reply = <$a>);            # "echo: ping"
close $a;
waitpid $pid, 0;
```

Emulating `pipe` with `socketpair`. `shutdown` turns a bidirectional socket into a one-way channel:

```
use Socket;
socketpair(my $rdr, my $wtr, AF_UNIX, SOCK_STREAM, PF_UNSPEC) or die
 ↪$!;
shutdown($rdr, 1);                  # reader cannot write
shutdown($wtr, 0);                  # writer cannot read
```

Datagram pair - messages preserve boundaries, so a single `recv` returns exactly one

send:

```
use Socket;
socketpair(my $a, my $b, AF_UNIX, SOCK_DGRAM, PF_UNSPEC) or die $!;
send($a, "one", 0)   or die $!;
send($a, "two", 0)   or die $!;
recv($b, my $m1, 64, 0);            # "one"
recv($b, my $m2, 64, 0);            # "two"
```

Edge cases

- **AF_UNIX is the only portable domain.** On most systems `socketpair` only works with `AF_UNIX`; other families return `EAFNOSUPPORT`. Use the constant, not a hard-coded 1.
- **Unbuffered output.** Stream socketpair handles buffer like any other filehandle. Set `$|` on the writer (after `select`ing it) or the reader will block waiting for output that is still in the parent's buffer. `autoflush` from `IO::Handle` is the usual form:

  ```
  $parent->autoflush(1);
  ```

- **Close both ends or EOF never arrives.** A stream socketpair delivers EOF on a reader only after *every* writable copy of the peer handle is closed. A child that forgets to `close` the parent's end will hang the parent on `<$handle>` forever.
- **Close-on-exec via `$^F`.** Filehandles whose numeric descriptor is greater than `$^F` get `FD_CLOEXEC` on platforms that support it, so they vanish across `exec`. Raise `$^F` before the call if the handle must survive an `exec`.
- **Unimplemented on the platform.** If the underlying `socketpair(2)` syscall does not exist, perl raises an exception rather than returning false. Trap with `eval` if you need to fall back.
- **Datagram vs stream.** `SOCK_STREAM` is byte-oriented (no message boundaries, use newline framing or length prefixes). `SOCK_DGRAM` preserves message boundaries but loses them silently if the receive buffer is smaller than the message.
- **PROTOCOL.** For `AF_UNIX` the protocol is ignored; `PF_UNSPEC` (or `0`) is conventional. Specifying a protocol that does not match the domain/type combination returns `EPROTONOSUPPORT`.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `socket` - create a single socket when only one end is yours (the other comes from the network via `accept` or `connect`)
- `pipe` - unidirectional byte channel; lighter weight than `socketpair` but cannot do bidirectional traffic on one call
- `send` / `recv` - message-oriented I/O, required for `SOCK_DGRAM` socketpairs

- `fork` - the typical partner: one `socketpair` before the fork gives parent and child a ready-made conduit
- `Socket` - source of `AF_UNIX`, `SOCK_STREAM`, `SOCK_DGRAM`, `PF_UNSPEC` and the other constants `socketpair` expects
- `$^F` - threshold above which new descriptors get close-on-exec; matters when the handles must survive `exec`

Lists

sort

Sort a list, returning the sorted list.

`sort` takes a list and returns a new list with the elements rearranged in order. With no comparator it uses standard string comparison. With a `BLOCK` or a `SUBNAME` you supply the comparison rule as a small function that is called for each pair of elements and must return a negative, zero, or positive value the way `cmp` and `<=>` do.

Sorting is stable - elements that compare equal keep their original relative order. The algorithm is mergesort.

Synopsis

```
sort LIST
sort BLOCK LIST
sort SUBNAME LIST
```

Three idiomatic forms:

```
my @out = sort @in;                          # default: string cmp, ascending
my @out = sort { $a <=> $b } @in;            # numeric, ascending
my @out = sort by_name @in;                  # named comparator sub
```

What you get back

A **new** sorted list. `sort` does **not** modify the input in place. To replace an array with its sorted version you assign back:

```
@a = sort @a;
```

In list context the return is the sorted list. In scalar context the return value is undefined - do not use `scalar sort`.

The returned list contains **aliases** into the original list, exactly like a `foreach` index variable. Modifying an element of the result inside a later `foreach`, `map`, or `grep` therefore modifies the original element too. This is almost never what you want; treat the result as read-only.

Global state: $a and $b

Inside the comparator BLOCK or SUBNAME, the two elements being compared are exposed as the package globals $a and $b. These are not parameters and they are not lexicals - they are real package variables that sort localises for you around the call.

- $a and $b live in **the package that called sort**. In main, that is $main::a and $main::b; in Foo, $Foo::a and $Foo::b.
- **Never declare my $a or my $b** anywhere the sort block can see. A lexical $a or $b shadows the package global; the block then compares two unrelated variables and the sort silently produces garbage. Under use warnings you get no warning for this.
- If you genuinely need $a / $b as lexicals elsewhere, refer to the sort variables by full name inside the block:

```
print sort { $::a cmp $::b }          qw(A C E G B D F H);
print sort { our $a cmp our $b }      qw(A C E G B D F H);
```

- A comparator defined with the prototype ($$) (or an equivalent signature attribute) receives the pair in @_ instead of $a and $b. This is slower than the $a/$b convention but decouples the sub from the caller's package.

Both elements are passed **by reference**; mutating $a or $b in the block mutates the original list. Don't.

Examples

Default string sort:

```
my @sorted = sort qw(banana apple cherry);
# ('apple', 'banana', 'cherry')
```

Numeric ascending - the default sort @nums would sort "10" before "2" because comparison is by string:

```
my @nums = sort { $a <=> $b } (10, 2, 33, 4);
# (2, 4, 10, 33)
```

Numeric descending - swap $a and $b:

```
my @nums = sort { $b <=> $a } (10, 2, 33, 4);
# (33, 10, 4, 2)
```

Multi-key sort with a tiebreaker. || runs the second comparator only when the first returns 0:

```
my @people = sort {
    $a->{last}  cmp $b->{last}
 || $a->{first} cmp $b->{first}
} @records;
```

Sort a hash's keys by the associated value:

```
my %age = (alice => 30, bob => 25, carol => 40);
my @by_age = sort { $age{$a} <=> $age{$b} } keys %age;
# ('bob', 'alice', 'carol')
```

Schwartzian Transform - when the sort key is expensive to compute, precompute it once per element, sort the `[key, value]` pairs, then strip the keys. Three stages, read bottom-up:

```
my @sorted =
    map  { $_->[1] }                      # 3. strip key
    sort { $a->[0] <=> $b->[0] }          # 2. sort by key
    map  { [ expensive_key($_), $_ ] }    # 1. attach key
    @input;
```

Named comparator sub. The sub reads `$a` and `$b` from the caller's package, so it must be in (or referenced from) that package:

```
sub by_name { $a->{name} cmp $b->{name} }

my @sorted = sort by_name @people;
```

Prototyped comparator - arguments arrive in `@_`, so it works from any package:

```
sub numeric ($$) { $_[0] <=> $_[1] }

my @sorted = sort numeric @nums;
```

Edge cases

- **`$a` and `$b` are package globals, not lexicals.** A stray `my $a` in scope silently breaks every sort in that scope. If you must, use `our $a` or `$::a` inside the block.
- **`<=>` is numeric, `cmp` is string.** Using the wrong one is the most common sort bug. `sort { $a cmp $b } (10, 2, 33)` yields `(10, 2, 33)` - lexicographic, `"10"` before `"2"`. Use `<=>` for numbers.
- **Empty list returns an empty list.** `sort ()` is `()`. No error.
- **Single element is returned unchanged.** No comparisons happen.
- **Comparator must be consistent.** If it sometimes says `$x < $y` and sometimes says the opposite for the same pair, the result is undefined. The comparator must return a total ordering: negative, zero, or positive, stably, for every call.
- **`NaN` poisons `<=>`.** `$a <=> $b` returns `undef` if either operand is `NaN`, which `sort` then treats as `0` - equal - giving an arbitrary placement. Filter first:

  ```
  my @clean = sort { $a <=> $b } grep { $_ == $_ } @input;
  ```

- **Stable sort.** Equal elements keep their input order. Rely on this for multi-pass sorts (sort by secondary key first, then by primary key). `use sort 'stable'` is the default and a no-op; `use sort '_mergesort'` / `'_qsort'` pragmas are historical and have no effect on current Perl.

- **`sort` is not in-place.** `sort @a` does not touch `@a`. Use `@a = sort @a` to replace it. Assigning to `sort`'s result list aliases back into the original, so `(sort @a)[0] = ...` modifies `@a`'s smallest element.
- **No loop control out of the block.** `last`, `next`, `redo`, `return`, and `goto LABEL` do not work inside a sort comparator - the block is not a loop body.
- **Parser trap: sorting a function's return value.** `sort` takes a `LIST`, so a following bareword can be mistaken for a `SUBNAME`. To sort the result of `find_records(@key)`:

```
my @out = sort { $a cmp $b } find_records @key;   # OK
my @out = sort +find_records(@key);               # OK
my @out = sort(find_records(@key));               # OK
```

  To use `find_records` as the comparator over `@key` instead:

```
my @out = sort find_records @key;                 # comparator
 ↪form
my @out = sort { find_records() } @key;           # block form
```

- **Locale-aware sorting.** Under `use locale` (without `':not_characters'`) the default string comparison follows the current collation locale instead of codepoint order.
- **XSUB comparators.** If the comparator is an XSUB, the pair is passed on the argument stack the way XS functions normally receive arguments; `$a` and `$b` are not set.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `reverse` - reverses a list; combine with `sort` for descending sorts when the comparator is hard to invert by swapping `$a` and `$b`
- `map` - the other half of the Schwartzian Transform; use to attach and strip sort keys around a `sort`
- `grep` - prefilter the list (e.g. remove `NaN` or `undef`) before handing it to `sort`
- `$a` - the first element in the comparison pair, aliased as a package global for the duration of the sort
- `$b` - the second element in the comparison pair, same lifetime and scoping rules as `$a`
- `<=>` - numeric three-way comparison; the usual operator inside a numeric sort block
- `cmp` - string three-way comparison; the usual operator inside a string sort block

Arrays

splice

Remove and/or replace a slice of an array in place, and return the removed elements.

`splice` is the general-purpose array surgeon. Every other array mutator - `push`, `pop`, `shift`, `unshift`, single-element assignment - can be written as a `splice` call. Pick the position with `OFFSET`, pick how many elements to cut with `LENGTH`, and optionally supply a `LIST` of replacements. The array grows or shrinks to fit, and you get back whatever was cut out.

Synopsis

```
splice @array;                                  # remove everything
splice @array, $offset;                         # remove from $offset to␣
↪the end
splice @array, $offset, $length;                # remove $length␣
↪elements
splice @array, $offset, $length, @list;         # remove and replace␣
↪with @list
```

What you get back

- **List context**: the list of elements that were removed, in their original order. Empty list if nothing was removed.
- **Scalar context**: the *last* element removed, or `undef` if nothing was removed. This is rarely what you want - reach for list context unless you specifically need only the tail.
- **Void context**: nothing; use this form when you only care about the in-place mutation.

Examples

Basic removal - cut two elements starting at index 1:

```
my @a = ('a', 'b', 'c', 'd', 'e');
my @gone = splice @a, 1, 2;
# @a    == ('a', 'd', 'e')
# @gone == ('b', 'c')
```

Insert without removing - `LENGTH` is `0`, so nothing comes out and the `LIST` is spliced in at `OFFSET`:

```
my @a = (1, 2, 5, 6);
splice @a, 2, 0, 3, 4;
# @a == (1, 2, 3, 4, 5, 6)
```

Replace a fragment - `LENGTH` and replacement size need not match. Here three elements leave, one arrives:

```
my @a = ('x', 'y', 'z', 'w', 'v');
my @gone = splice @a, 1, 3, 'Y';
# @a    == ('x', 'Y', 'v')
# @gone == ('y', 'z', 'w')
```

Remove the trailing tail - omit `LENGTH` entirely to cut from `OFFSET` to the end:

```
my @a = (1, 2, 3, 4, 5);
my @tail = splice @a, 3;
# @a    == (1, 2, 3)
# @tail == (4, 5)
```

Negative `OFFSET` counts from the end. Pop the last element:

```
my @a = ('a', 'b', 'c');
my $last = splice @a, -1;
# @a    == ('a', 'b')
# $last == 'c'
```

Classic idiom - consume an array `$n` elements at a time:

```
while (my @chunk = splice @queue, 0, $n) {
    process(@chunk);
}
```

All the common array primitives written as `splice` (assuming `$#a >= $i`):

```
push    @a, $x, $y;    #  splice @a, scalar(@a), 0, $x, $y
pop     @a;            #  splice @a, -1
shift   @a;            #  splice @a, 0, 1
unshift @a, $x, $y;    #  splice @a, 0, 0, $x, $y
$a[$i] = $y;           #  splice @a, $i, 1, $y
```

Edge cases

- **Negative `OFFSET`** counts from the end: `-1` is the last element, `-2` the one before, and so on. If the magnitude exceeds the array length, Perl issues a warning and clamps to `0`.
- **Negative `LENGTH`** leaves that many elements at the end of the array untouched. `splice @a, 2, -1` removes from index 2 up to but not including the final element.
- **`OFFSET` past the end**: with an explicit `LENGTH`, Perl issues a warning and splices at the end of the array (equivalent to `splice @a, scalar(@a), 0, LIST`). Without a `LENGTH`, nothing is removed.
- **`LENGTH` past the end**: silently clamped. `splice @a, 1, 999` simply removes everything from index 1 onward.
- **Both `OFFSET` and `LENGTH` omitted**: removes every element. The array is left empty.

- **Replacement list size differs from `LENGTH`**: the array grows or shrinks to match. There is no requirement that `scalar(@list)` equal `LENGTH` - that asymmetry is the whole point of `splice`.
- **Array reference**: dereference inline. `splice @$ref, 0, 1` and `splice @{ $obj->{items} }, -2` both work.
- **Splicing a slice is not supported**: `splice @a[1..3], 0, 1` is a syntax error. `splice` requires a named array or dereferenced array, not a list expression. To operate on a range, pass the bounds as `OFFSET` and `LENGTH` instead.
- **Void context is fine**. If the return value is not needed, call `splice` as a statement - no allocation of the removed-elements list happens in that case.
- **Scalar expressions as the first argument**: not allowed. An experimental feature permitted this from 5.14 through 5.22 and was removed in 5.24. The first argument must be an array (or dereferenced arrayref).

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `push` - append to the end; equivalent to `splice @a, scalar(@a), 0, LIST`
- `pop` - remove the last element; equivalent to `splice @a, -1`
- `shift` - remove the first element; equivalent to `splice @a, 0, 1`
- `unshift` - prepend; equivalent to `splice @a, 0, 0, LIST`
- `delete` - superficially similar but semantically different. `delete $a[$i]` leaves a hole (the slot becomes `undef`) and does **not** renumber later indices; the array's length is unchanged unless the deleted slot was the last one. `splice @a, $i, 1` physically removes the slot and shifts every later element down by one. Use `delete` for sparse arrays where index identity matters; use `splice` when you want the array to close up.
- `grep` - filter an array into a new one by predicate; reach for it instead of `splice` when you want "keep only the ones that match" rather than "cut out a contiguous range"

Regular expressions and pattern matching

split

Cut a string into a list of fields using a regex separator.

`split` scans `EXPR` for matches of `PATTERN` and returns the pieces between the matches as a list. The matched text itself is the separator and is **not** included in the result - unless the pattern contains capture groups, in which case each capture becomes an extra element in the output. In scalar context the list size is returned.

If `PATTERN` is omitted, `split` behaves like the `awk` default: runs of whitespace are the separator and any leading whitespace in `EXPR` is stripped. If `EXPR` is omitted, `$_` is split.

Synopsis

```
split /PATTERN/, EXPR, LIMIT
split /PATTERN/, EXPR
split /PATTERN/
split
```

What you get back

A list of substrings (fields) in list context; the count of fields in scalar context. The separator text is never part of a field.

```
my @fields = split /,/, "a,b,c";       # ("a", "b", "c")
my $n      = split /,/, "a,b,c";       # 3
```

Splitting an `EXPR` that is the empty string always yields zero fields, regardless of `LIMIT`:

```
my @x = split /,/, "", -1;             # ()
```

Prior to Perl 5.11, `split` in void or scalar context overwrote `@_`. Modern Perl does not; never rely on that old side effect.

The LIMIT argument

`LIMIT` controls how many fields are produced and, crucially, whether trailing empty fields are kept.

- **Positive `LIMIT`** - upper bound on the number of fields. `EXPR` is split at most `LIMIT - 1` times; the final field holds the rest of the string verbatim. `LIMIT = 1` means no splits at all - you get the whole string back as a single element.

  ```
  my @x = split /,/, "a,b,c", 1;       # ("a,b,c")
  my @x = split /,/, "a,b,c", 2;       # ("a", "b,c")
  my @x = split /,/, "a,b,c", 3;       # ("a", "b", "c")
  my @x = split /,/, "a,b,c", 4;       # ("a", "b", "c")
  ```

- **Negative `LIMIT`** - treated as arbitrarily large. As many fields as possible are produced, including **all** trailing empty fields.

  ```
  my @x = split /,/, "a,b,c,,,", -1;   # ("a", "b", "c", "", "", "
  ↪")
  ```

- **Omitted or zero `LIMIT`** - like negative, **except** trailing empty fields are stripped. Leading empty fields are always kept.

  ```
  my @x = split /,/, "a,b,c,,,";       # ("a", "b", "c")
  my @x = split /,/, ",,a,b";          # ("", "", "a", "b")
  ```

- **Implicit `LIMIT` in list assignment.** When assigning to a fixed list, Perl sets `LIMIT` to one more than the number of targets, so the trailing fields do not need to be scanned. The following gets `LIMIT = 3` automatically:

```
my ($login, $passwd) = split /:/;
```

In time-critical code, pass an explicit `LIMIT` rather than letting the whole string be scanned.

The awk-style whitespace case: `split " "`

A literal single space as the pattern - `split " "`, not `split / /` - triggers a special case: `PATTERN` is treated as `/\s+/` **and** any leading whitespace in `EXPR` is stripped before splitting. This matches classic `awk` behaviour.

```
my @x = split " ", "  Quick brown fox\n";
# ("Quick", "brown", "fox")

my @x = split " ", "RED\tGREEN\tBLUE";
# ("RED", "GREEN", "BLUE")
```

To split on a *single* literal space only, use the regex form `/ /` - it is not special-cased:

```
my @x = split / /, " abc";              # ("", "abc")
```

Since Perl 5.18, the trigger is any expression whose value is the single-character string `" "` (not just a literal), and since Perl 5.28 the rule works correctly under `use feature 'unicode_strings'`. Under Perl 5.39.9+ the `/x` default modifier does **not** affect `split STRING`, so `split " "` still means awk-emulation even inside `use re "/x"`. If you want to split on one literal space under `use re "/x"`, write `split /(?-x: )/` or `split /\x{20}/`.

If `PATTERN` is omitted entirely, it defaults to `" "`, so bare `split` is the awk-style form.

The empty-pattern case: splitting into characters

If `PATTERN` matches the empty string, the split happens at every match position - i.e. between characters.

```
my @x = split //, "abc";                # ("a", "b", "c")
```

As a `split`-specific rule, the bare match operator `//` is **not** the "repeat the last successful match" form here; it is the literal empty pattern. A zero-width match at the very start of `EXPR` never produces an empty leading field, which is why splitting a leading space is handled like this:

```
my @x = split //, " abc";               # (" ", "a", "b", "c")   - 4,␣
↪not 5
my @x = split //, " abc", -1;           # (" ", "a", "b", "c", "") -␣
↪trailing empty with -1
```

A positive-width match at the start *does* produce an empty leading field:

```
my @x = split / /, " abc";              # ("", "abc")
```

Capture groups become extra elements

If PATTERN contains capturing groups, each capture is inserted into the output list for every separator match, in the order the groups are declared. A group that does not participate in the match contributes undef. These extras do **not** count toward LIMIT.

```
my @x = split /-|,/    , "1-10,20", 3;
# ("1", "10", "20")

my @x = split /(-|,)/  , "1-10,20", 3;
# ("1", "-", "10", ",", "20")

my @x = split /-|(,)/  , "1-10,20", 3;
# ("1", undef, "10", ",", "20")

my @x = split /(-)|,/  , "1-10,20", 3;
# ("1", "-", "10", undef, "20")

my @x = split /(-)|(,)/, "1-10,20", 3;
# ("1", "-", undef, "10", undef, ",", "20")
```

Use this when you want the separators preserved alongside the fields - typical for tokenisers where the delimiters are themselves part of the output stream.

Global state it touches

- $_ - EXPR defaults to it when no second argument is given.
- @_ - modern Perl does **not** touch it. Only relevant if you support pre-5.11 perls.
- The regex engine's match variables ($1, $2, ..., $&, $', $`) are **not** set as a side effect of split, even when PATTERN captures - the captures go into the returned list.

Examples

Classic CSV-style line, no embedded commas:

```
my @cells = split /,/, "alice,bob,carol";    # ("alice", "bob",
↪"carol")
```

Parse a /etc/passwd line into known fields; implicit LIMIT stops at the first seven separators:

```
my ($user, $pw, $uid, $gid, $gecos, $home, $shell)
    = split /:/, $line;
```

Awk-style whitespace tokenising, leading whitespace discarded:

```
my @words = split " ", "   one\ttwo  three\n";
# ("one", "two", "three")
```

Characters of a string - the empty pattern:

```
my @chars = split //, "hello";                  # ("h", "e", "l", "l",
↪"o")
```

Keep every trailing empty field for a strict column-count reader:

```
my @cols = split /\t/, $line, -1;
```

Preserve separators by capturing - round-trip reconstructible:

```
my @parts = split /([,;])/, "a,b;c";            # ("a", ",", "b", ";",
↪"c")
my $same  = join "", @parts;                    # "a,b;c"
```

Default form reads `$_` and does awk-style whitespace splitting:

```
for (@lines) {
    my @f = split;                              # split " ", $_
    ...
}
```

Edge cases

- **Empty input** - `split /X/, ""` returns the empty list for every `LIMIT`, including `-1`.
- **`LIMIT = 1`** - no splitting occurs; the whole string is the one and only field. Useful when you want `split` semantics conditionally without losing the input.
- **`/^/` pattern** - treated as if `/^/m` so it matches at every line start, which is the only useful behaviour. Splits a string into lines keeping the newlines.
- **Zero-width match at position 0** - never produces an empty leading field (see `split //, " abc"` above).
- **Positive-width match at position 0** - produces an empty leading field, always.
- **Match at end of string** - produces an empty trailing field, which is then stripped unless `LIMIT` is non-zero.
- **Patterns with modifiers** - the usual `qr//` modifiers (`/i`, `/m`, `/s`, `/x`, `/u`, `/a`, `/l`, `/n`) apply. The pattern need not be a literal; any expression producing a regex or string works, and `qr//` objects are accepted.
- **Single-space string that is not a literal** - from Perl 5.18 on, any expression evaluating to `" "` triggers the awk-style special case, not just the literal `" "`.
- **`split " "` vs `split / /`** - the first is awk-style whitespace (`\s+` with leading-ws stripping), the second is a single literal space. The two are not interchangeable.
- **Unicode whitespace** - under `use feature 'unicode_strings'` (default since 5.28 for the awk-style case), `split " "` treats Unicode whitespace as a separator too. Outside that scope the behaviour is affected by the “Unicode Bug”.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `join` - inverse operation; stitches a list back into a string with a chosen glue
- `m` - the match operator; when you want to *find* something rather than cut the string around it
- `qr` - pre-compile a pattern once for repeated `split` calls in a hot loop
- `index` - locate a fixed substring without building the full list of fields; cheaper when you only need the first split
- `substr` - extract by byte/character offset when the field boundaries are positional, not delimited

SCALARs and strings

sprintf

Build a formatted string from a format template and a list of values.

`sprintf` takes a `FORMAT` string containing literal text interspersed with `%`-introduced conversion specifiers, consumes values from `LIST` to fill each specifier, and returns the resulting string. It is the same machinery that drives `printf`, minus the write to a file-handle. Perl implements its own formatter; it does not call the C library's `sprintf(3)` except for the final floating-point digit generation, so non-standard extensions that happen to exist in your libc are not available here.

Synopsis

```
my $s = sprintf $format, @args;
my $s = sprintf "%-10s %5d", $name, $count;
my $s = sprintf "%.3f", $x;
```

What you get back

A single scalar string. Never a list, regardless of context. If `FORMAT` contains no `%` specifiers, the return value is `FORMAT` unchanged and `LIST` is ignored. If `FORMAT` references more arguments than `LIST` supplies, the missing values are treated as `undef` (numeric zero or the empty string depending on the conversion), and a `Missing argument in sprintf` warning is issued under `use warnings`.

Passing an array as the first argument is almost always a mistake - the array is evaluated in scalar context, so Perl uses its element count as the format. Use `sprintf $format, @rest` with the format held separately, or `sprintf "@array"` for the array-as-string case.

Global state it touches

- `LC_NUMERIC` locale, when `use locale` is in effect and `POSIX::setlocale` has been called, determines the decimal-point character used by %e, %f, %g and their upper-case variants. Without `use locale` the decimal point is always `.`, even if the process locale says otherwise.
- No other special variables are consulted. `sprintf` does not read `$_`; an explicit list is required.

Conversion specifiers

The core set, in the order you will actually reach for:

Spec	Meaning
%s	String. Stringifies the argument.
%d, %i	Signed integer in decimal. Floats are truncated toward zero.
%u	Unsigned integer in decimal.
%f, %F	Fixed-point floating point. Default 6 digits after the decimal.
%e, %E	Scientific notation. Upper-case E selects the upper-case form.
%g, %G	%e or %f, whichever is shorter. Precision is *significant digits*, not digits after the decimal.
%x, %X	Unsigned integer in hexadecimal; %X uses upper-case A–F.
%o	Unsigned integer in octal.
%b, %B	Unsigned integer in binary.
%c	Single character whose codepoint is the given integer.
%%	A literal %. Consumes no argument.
%p	Perl-value address, in hexadecimal. Diagnostic use only.
%n	Stores the number of characters written so far into the next argument (which must be a writable scalar).
%a, %A	Hexadecimal floating point.

Upper-case size synonyms %D %U %O and %F exist for historical compatibility; prefer the lower-case forms.

Flags, width, precision

Between % and the conversion letter, in order:

1. **Argument index** - N$ picks the N-th argument (1-based): `sprintf '%2$s %1$s', 'world', 'hello'` yields `"hello world"`.
2. **Flags** - any combination of:
   - - left-justify within the field (default is right-justify).

- + prefix non-negative numbers with +.
- ` ` (a space) prefix non-negative numbers with a space. When both space and + are given, + wins.
- `0` pad numeric conversions with zeros instead of spaces. Ignored when - is also set or a precision is given for integers.
- # alternate form: `0` prefix for octal, `0x`/`0X` prefix for non-zero hex, `0b`/`0B` prefix for non-zero binary.

3. **Minimum width** - a decimal number, or `*` to take the width from the next argument, or `*N$` for an explicit argument. Negative width (via `*`) is equivalent to the - flag.
4. **Precision** - `.` followed by a decimal number, or `.*` (next argument), or `.*N$`. Meaning depends on the conversion:
   - `%f` / `%e`: digits after the decimal point (default 6).
   - %g: total significant digits.
   - %d / %x / %o / %b: minimum digits; the number is zero-padded to this width. The `0` flag is ignored when a precision is present for integers.
   - `%s`: maximum width; truncates.
5. **Size modifier** - hh, h, `j`, `l`, q, `L`, `ll`, `t`, `z`, `V`. Interprets the argument as a specific C type. Almost never useful from Perl - Perl has one integer type and one float type, and Perl-level values do not overflow into the sizes these modifiers select. `V` is a no-op meaning "Perl's default size".
6. **Conversion letter** - one of the table entries above.

The vector flag

Prefix the conversion letter with `v` (optionally `*v` or `*N$v` to supply a separator) and Perl treats the argument as a vector of codepoints, applying the conversion to each codepoint and joining the results with `.` (or the supplied separator):

```
sprintf "%vd", "AB\x{100}";             # "65.66.256"
sprintf "%vX", "\x01\x02\x1f";          # "1.2.1F"
sprintf "%*vX", ":", "\xde\xad\xbe\xef"; # "DE:AD:BE:EF"
```

Common uses: rendering `$^V`, formatting IPv4 / IPv6 byte sequences, or dumping bitstrings. The argument is always read as a string of codepoints, not as a list.

Examples

Pad and align strings in a fixed-width report column:

```
printf "%-10s %5d\n", "widgets", 42;   # "widgets        42\n"
printf "%-10s %5d\n", "sprockets", 7;  # "sprockets       7\n"
```

Zero-pad a fixed-width integer, e.g. for lexicographic sort keys or line numbers:

```
sprintf "%08d", 1234;                   # "00001234"
sprintf "%05d", -7;                     # "-0007"  (sign consumes a␣
↪slot)
```

Round to a fixed number of decimal places. `sprintf` does the rounding; `printf` is a common way to coerce a display value:

```
sprintf "%.3f", 3.14159265;             # "3.142"
sprintf "%.0f", 2.5;                    # "2" or "3" - banker's␣
↪rounding
                                        # is platform-dependent
```

Scientific notation, with and without a chosen precision:

```
sprintf "%e",    6.022e23;              # "6.022000e+23"
sprintf "%.2e",  6.022e23;              # "6.02e+23"
sprintf "%g",    0.0001234;             # "0.0001234"
sprintf "%g",    0.00001234;            # "1.234e-05"  (switches to
↪%e)
```

Hex, octal, and binary with the # alternate-form prefix:

```
sprintf "%#x",  255;                    # "0xff"
sprintf "%#o",  8;                      # "010"
sprintf "%#b",  10;                     # "0b1010"
sprintf "%08b", 10;                     # "00001010"   (no prefix,␣
↪width 8)
```

Character from a codepoint - `%c` is essentially `chr` routed through the format engine:

```
sprintf "%c", 65;                       # "A"
sprintf "%c", 0x2014;                   # "-"  (em dash)
```

Field width from the argument list - useful when the width is computed at runtime:

```
my $w = 12;
sprintf "%-*s|", "name", $w;            # "name        |"
```

Out-of-order arguments - useful when the same value appears multiple times or when localised formats reorder placeholders:

```
sprintf "%2\$s loves %1\$s", "Alice", "Bob";   # "Bob loves Alice"
```

Build an IPv4 address string:

```
sprintf "%vd", "\x7f\x00\x00\x01";      # "127.0.0.1"
```

Capture length via %n (the receiving argument must be a writable scalar):

```
my $written;
my $s = sprintf "hello%n, world", $written;
```

(continues on next page)

(continued from previous page)

```
# $s       is "hello, world"
# $written is 5
```

Edge cases

- **Integer truncation on %d with floats**: %d converts via integer coercion, which truncates toward zero. `sprintf "%d", 3.9` is `"3"`, not `"4"`. Use `%.0f` if you want banker's-rounded display, or `int` / `sprintf "%.0f"` explicitly first.
- **%s on an object**: stringification runs, which may invoke overloaded `""` or fall back to `"ClassName=HASH(0x...)"`. There is no way to suppress overloading inside `sprintf`; call `overload::StrVal($obj)` first if you need the raw form.
- **Missing arguments**: `sprintf "%s %s", "one"` returns `"one "` and warns `Missing argument in sprintf at ...` under `use warnings`. No exception is raised.
- **Too many arguments**: extras are ignored silently. No warning.
- **%s with precision truncates, %d with precision zero-pads**. `sprintf "%.3s", "abcdef"` is `"abc"`. `sprintf "%.3d", 7` is `"007"`.
- **%0 flag vs precision on integers**: precision wins. `sprintf "%05.3d", 7` is `" 007"`, not `"00007"` - the `0` flag is ignored once you give an explicit integer precision.
- **Negative width via ***: `sprintf "%*s", -6, "x"` is `"x "`. Negative precision via `.*` counts as no precision at all.
- **NaN and Inf**: `sprintf "%f", 9**9**9` yields `"Inf"`; `sprintf "%f", -sin(9**9**9)` yields `"NaN"`. Exact spelling (`"Inf"` vs `"inf"`, `"NaN"` vs `"nan"`) follows the Perl build's convention, not the `%f` / `%F` case distinction.
- **Locale decimal point**: with `use locale` and a de_DE locale, `sprintf "%.2f", 1.5` can produce `"1,50"`. Without `use locale` the result is always `"1.50"`. Code that writes numeric output for consumption by other programs should either avoid `use locale` in that scope or use `use locale ':not_characters'` and format numbers outside locale influence.
- **%p is diagnostic-only**: the hexadecimal pointer value changes from run to run and is meaningless outside the running process. Do not use it as an identity key - two references may share the same address after the first has been freed.
- **Array passed as FORMAT**: `sprintf @a` formats using the scalar count of `@a` as the format string. Almost always a bug. Always keep the format in its own scalar.
- **Unsupported size modifier**: on platforms lacking e.g. long doubles, the `L`/`q`/`ll` size on a float conversion silently falls back to the default size, and a `printf` warning is issued under `use warnings`. To promote this to an exception, use `use warnings FATAL => 'printf'`.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `printf` - same formatting engine, but writes to a filehandle instead of returning the string
- `print` - use with a pre-built `sprintf` result when you want to format once and write to several handles
- `chr` - single codepoint to single character; equivalent to `sprintf "%c", $n`
- `hex`, `oct` - inverse operations: parse a string in hex/octal/binary back into a number
- `pack` - build a binary string from a template; reach for it when the output is bytes, not a human-readable format
- `$_` - *not* consulted by `sprintf`; unlike many built-ins it always needs an explicit list

Numeric functions

sqrt

Return the non-negative square root of EXPR.

`sqrt` computes the **principal square root** - the unique non-negative real number whose square equals EXPR. For any `EXPR >= 0` the result is `>= 0`; `sqrt` never returns the negative root. If EXPR is omitted, `$_` is used.

The computation is performed in double-precision floating point (IEEE 754 `binary64`), so the result is a float even when the input is an integer and even when the mathematical answer is itself an integer.

Synopsis

```
sqrt EXPR
sqrt
sqrt(EXPR)
```

What you get back

A floating-point number: the non-negative square root of EXPR. For negative input the result is `NaN` (and a warning is emitted under `use warnings`); see *Edge cases*. For complex-valued roots load , which overloads `sqrt` so that `sqrt(-1)` returns `i`.

```
my $r = sqrt 2;                    # 1.4142135623731
my $r = sqrt;                      # sqrt of $_
```

The default argument, precedence, and parsing

Three parser details shape every `sqrt` call:

- **No argument uses `$_`.** `sqrt` with no argument and no parentheses reads the topic variable. Handy inside `map` and `grep` blocks that already bind `$_`.
- **Named-unary precedence.** `sqrt` is a named unary operator, so it binds tighter than most binary operators but looser than multiplication and exponentiation. `sqrt 2 + 3` means `sqrt(2) + 3`, not `sqrt(2 + 3)` - parenthesise when in doubt.
- **`sqrt(...)` is a function call.** As with every named unary, writing an opening paren immediately after the name turns the remainder into the argument list: `sqrt(4) * 2` is 4 (the sqrt of 4) times 2, i.e. 4.

```
my @roots = map { sqrt } 1, 4, 9;   # (1, 2, 3); $_ is the topic
my $x = sqrt 2 + 3;                 # 4.41421…  - sqrt(2) + 3
my $y = sqrt(2 + 3);                # 2.23606…  - sqrt(5)
```

Examples

Classic irrational root. The printed decimal is the closest `binary64` approximation, not the exact value:

```
printf "%.15f\n", sqrt(2);          # 1.414213562373095
```

Pythagorean distance between two points:

```
sub distance {
    my ($x1, $y1, $x2, $y2) = @_;
    return sqrt(($x2 - $x1) ** 2 + ($y2 - $y1) ** 2);
}

print distance(0, 0, 3, 4), "\n";   # 5
```

Zero is its own square root, and so is one:

```
print sqrt(0), "\n";                # 0
print sqrt(1), "\n";                # 1
```

Perfect squares round-trip exactly for small integers, because the mathematical result is representable in `binary64`:

```
print sqrt(16), "\n";               # 4
print sqrt(144), "\n";              # 12
print sqrt(1_000_000), "\n";        # 1000
```

Large perfect squares may **not** round-trip exactly - the result is the nearest representable float, which can be slightly off:

```
my $n = 9_007_199_254_740_996;      # 2**53 + 4
printf "%.3f\n", sqrt($n) ** 2 - $n; # non-zero in general
```

Compare to the exponentiation operator, which is the general way to take any real power:

```
print 2 ** 0.5, "\n";                # same as sqrt(2)
print 27 ** (1/3), "\n";             # cube root of 27 ≈ 3
```

Using to get a complex result from a negative input:

```
use Math::Complex;
my $z = sqrt(-1);                    # i
print $z, "\n";                      # "i"
print sqrt(-4), "\n";                # "2i"
```

Edge cases

- **Negative input without** : returns `NaN` and triggers an `Invalid operation` warning under `use warnings`. `NaN` compares unequal to itself, so never test with `==`; use `POSIX::isnan` or check `$x != $x`.

  ```
  my $r = sqrt(-1);                  # NaN
  print "bad\n" if $r != $r;         # NaN self-compare trick
  ```

- **Negative input with** : once is loaded, `sqrt` is overloaded for complex scalars and returns the principal complex root. Real inputs still go through the built-in.

- **Non-numeric input**: coerced to a number using the usual rules - leading whitespace and a numeric prefix are consumed, the rest is ignored. Under `use warnings` an `isn't numeric` warning fires when the string has no numeric prefix at all. `undef` coerces to `0`.

  ```
  print sqrt("9abc"), "\n";          # 3, with a warning under
   ↪warnings
  print sqrt(""), "\n";              # 0, with a warning
  print sqrt(undef), "\n";           # 0, with an uninitialized
   ↪warning
  ```

- **Floating-point imprecision for perfect squares**: for sufficiently large integers, the mathematically exact root may not be representable. `sqrt($n*$n)` is not guaranteed to equal `$n` for large `$n`. If you need an exact integer square root, do not use `sqrt` - use 's `bsqrt` or a dedicated isqrt implementation.

- **Special IEEE values**: `sqrt("Inf")` is `Inf`; `sqrt("NaN")` is `NaN`; `sqrt(-0.0)` is `-0.0` (negative zero, which compares equal to `0`). `sqrt("-Inf")` is `NaN`.

- **Integer overflow is not an issue**: because the result is always a float, `sqrt` never overflows for any finite numeric input.

- **List context is irrelevant**: `sqrt` always returns a single scalar. In list context that scalar is the only element of the returned list.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `abs` - absolute value; pair with `sqrt` when you want `sqrt(abs($x))` to sidestep negative input
- `**` - exponentiation operator; `$x ** 0.5` is `sqrt`, and `$x ** (1/$n)` generalises to other roots
- `hypot` - from `POSIX`, computes `sqrt($x*$x + $y*$y)` without intermediate overflow
- - overloads `sqrt` and friends so that `sqrt(-1)` returns `i` instead of `NaN`
- , - arbitrary-precision square roots via `bsqrt` when `binary64` is not enough

Numeric functions

srand

Seed the pseudo-random number generator.

`srand` sets the internal state of the PRNG that `rand` draws from. Call it with an explicit integer to get a reproducible stream, or call it with no argument to reseed from the best entropy source the platform offers. Since Perl 5.14 `srand` **returns the seed it used**, so you can log it and replay the same run later.

Synopsis

```
srand EXPR
srand
```

What you get back

The seed that was installed, as an integer. For `srand($n)` the return value is `$n` (after the usual integer truncation). For argument-less `srand()` the return value is the auto-chosen seed drawn from system entropy - capture it if you ever want to reproduce the run.

```
my $seed = srand();                  # auto-seed, remember what we␣
↪used
warn "seed for this run: $seed\n";
```

Before Perl 5.14 `srand` returned nothing useful. Code that relies on the seed return should declare the minimum version:

```
use v5.14;                           # so srand returns the seed
```

Global state

`srand` mutates a single process-wide PRNG. Every subsequent call to `rand` in the same process - from any package, any module, any thread-free context - reads from that one state. There is no per-package or per-scope RNG. Consequences:

- Calling `srand($k)` anywhere in the program affects *every* later `rand` call, including those inside modules you did not write.
- Two processes that call `srand($k)` with the same `$k` produce identical `rand` streams. This is the basis of reproducible tests and the reason you must reseed after `fork`.
- If `srand` is never called explicitly, it is called implicitly **without argument** the first time `rand` is invoked.

Examples

Reproducible test case with an explicit seed - same input, same sequence, every run:

```
srand(42);
my @sample = map { rand() } 1..5;   # identical on every invocation
```

Seed from `time()` for rare-collision randomness across separate runs started at different wall-clock seconds. This is the classical pattern; it is **not** cryptographic-grade:

```
srand(time() ^ $$);                 # time XOR pid, old-school␣
↪spread
```

Save-seed, restore-seed pattern for a test that picks a random subset of its cases but logs enough to replay any failure:

```
my $seed = srand();                 # auto-seed, capture it
eval { run_randomised_tests() };
if ($@) {
    warn "FAILED with seed=$seed - reproduce with: srand($seed)\n";
    die $@;
}
```

Reseed after `fork` so parent and child do not share a stream:

```
my $pid = fork();
die "fork: $!" unless defined $pid;
if ($pid == 0) {
    srand();                        # child gets fresh entropy
    exec_child_work();
}
```

Two independent streams from the same seed - useful when you want to checkpoint and resume, or diff two runs that diverged after a decision point:

```
srand(12345);
my @first  = map { rand() } 1..10;
```

(continues on next page)

(continued from previous page)

```
srand(12345);
my @second = map { rand() } 1..10;  # identical to @first
```

Edge cases

- **No-argument form auto-seeds with the best available entropy.** On Linux this draws from `/dev/urandom` (or `getrandom(2)`). The seed is wide enough that you should treat it as opaque, not as a small integer.
- **Do not call `srand()` without arguments more than once per process.** The PRNG's internal state already carries more entropy than any single seed can restore, so re-seeding *loses* randomness rather than adding it. The two legitimate reasons to call it repeatedly are (a) with an explicit seed for reproducibility, and (b) once in each child after `fork`.
- **Decimal arguments are silently truncated to integer.** `srand(42)` and `srand(42.9)` install the same seed. Always pass an integer so intent is obvious.
- **Same seed ⇒ identical `rand` stream.** That is the whole point for tests; it is the whole problem for security.
- **Not cryptographically secure.** `rand`/`srand` use a fast non-cryptographic PRNG. Do not use them to generate session tokens, nonces, keys, password salts, or anything an adversary must not be able to predict. Use `Crypt::URandom` or read from `/dev/urandom` directly.
- **`PERL_RAND_SEED` environment variable.** If set to a non-negative integer at process startup, argument-less `srand()` (including the implicit call triggered by the first `rand`) installs a deterministic seed derived from that value. The derivation is deliberately unspecified; only same-value-same-binary-same-code reproducibility is guaranteed. Intended for debugging and performance analysis. The variable is read **once** at startup; mutating `%ENV` later has no effect on the current process.
- **Threads** (`use threads`): each interpreter clone gets its own PRNG state. Seeding in one thread does not affect another.
- **Scope of the seed**: there is no way to `local`-ise the PRNG state. If a library you call in between does an `srand`, your stream is perturbed. Capture and restore manually if that matters.

Differences from upstream

Fully compatible with upstream Perl 5.44. `PERL_RAND_SEED` is honoured with the same "deliberately unspecified but reproducible" contract as perl5.

See also

- `rand` - the consumer of the state `srand` installs; draws the next value from the seeded stream
- `time` - seconds since the epoch; the classic quick-and-dirty seed source for non-security code (`srand(time() ^ $$)`)

- `Crypt::URandom` - cryptographically secure random bytes from the OS entropy pool; use this, not `rand`, for tokens and keys
- `Math::Random::MT` - Mersenne Twister PRNG with its own independent state, when you need a second stream that is not perturbed by library calls to `rand`
- `perlrun` - documents the `PERL_RAND_SEED` environment variable and other startup knobs

Filehandles, files, directories

stat

Get a file's status information.

`stat` asks the operating system for the metadata record of a file - size, ownership, permissions, timestamps, inode - and returns it as a 13-element list. It is the underlying primitive behind the `-X` file tests (`-e`, `-f`, `-r`, ...), behind modules like `File::stat`, and behind any script that needs to inspect a file without opening it. `stat` dereferences symlinks and reports on the target; use `lstat` to get the link itself.

Synopsis

```
stat FILEHANDLE
stat EXPR
stat DIRHANDLE
stat
stat _
```

What you get back

In list context, a 13-element list of numeric fields. In scalar context, a boolean: true on success, false on failure. Either way, on success the kernel's stat buffer is cached in the special filehandle `_`, which any subsequent `stat`, `lstat`, or `-X` test can re-use without a new syscall.

On failure, list context returns the empty list and `$!` is set to the `errno` value from the failed `stat(2)`.

The fields, in order:

Index	Field	Meaning
0	`dev`	device number of the filesystem holding the file
1	`ino`	inode number
2	`mode`	file mode - type bits **and** permission bits combined
3	`nlink`	number of hard links to the file
4	`uid`	numeric user ID of the owner
5	`gid`	numeric group ID of the owner
6	`rdev`	device identifier, for character/block special files
7	`size`	total size in bytes
8	`atime`	last access time, seconds since the epoch (1970-01-01 00:00 GMT)
9	`mtime`	last modify time, seconds since the epoch
10	`ctime`	inode change time, seconds since the epoch - **not** creation time
11	`blksize`	preferred I/O block size for reads/writes against this file
12	`blocks`	number of filesystem blocks actually allocated (often 512 B each)

Field 2 (`mode`) packs both the file type and the permission bits into one number. Mask with `07777` to see just the permissions; use the `S_IF*` constants and `S_IS*` helpers from `Fcntl` to test the type portably.

```
my $mode = (stat($filename))[2];
printf "permissions are %04o\n", $mode & 07777;
```

Not every filesystem populates every field. `atime`, `blksize`, and `blocks` may come back as `0` on filesystems that do not track them (for instance some FUSE mounts or network filesystems); `ctime` is not a portable "creation time" - it tracks the last change to the inode itself (permissions, ownership, link count), not when the file was first created.

Global state it touches

- `$_` - read when `stat` is called with no argument. `stat` with no EXPR stats the filename held in `$_`, **not** the cached _ handle.
- `$!` - set to the `errno` of the failed `stat(2)` syscall when the call fails.
- The **special _ filehandle** - a per-interpreter cache of the most recent successful `stat`, `lstat`, or `-X` result. Passing _ as the argument reuses that cached buffer and issues no syscall. Successful calls refresh it; failed calls leave the previous contents in place.

Examples

Basic list-context unpack:

```
my ($dev, $ino, $mode, $nlink, $uid, $gid, $rdev, $size,
    $atime, $mtime, $ctime, $blksize, $blocks) = stat($filename);
```

Pick out a single field with a list slice:

```
my $size  = (stat($filename))[7];
my $mtime = (stat($filename))[9];
```

Scalar context as an existence-plus-readable-metadata probe:

```
if (stat($path)) {
    my $size = (stat _)[7];          # reuse the cached buffer
    print "size = $size\n";
} else {
    warn "stat $path: $!";
}
```

Reusing the cache across an `-X` test and a `stat`:

```
if (-x $file && (($d) = stat(_)) && $d < 0) {
    print "$file is executable NFS file\n";
}
```

Named-accessor access through `File::stat`:

```
use File::stat;
my $sb = stat($filename);
printf "file %s, size %s, perm %04o, mtime %s\n",
       $filename, $sb->size, $sb->mode & 07777,
       scalar localtime $sb->mtime;
```

Decoding the mode with `Fcntl`:

```
use Fcntl ':mode';
my $mode = (stat($filename))[2];
printf "permissions %04o\n", S_IMODE($mode);
print "is a directory\n" if S_ISDIR($mode);
```

Edge cases

- **No argument**: `stat;` stats `$_`, **not** the cached _ handle. To re-read the cache you must write `stat(_)` with the underscore as an explicit bareword.
- **The _ bareword is special**: it names the stat cache, not a file. Quoting it (`stat("_")`) stats a file literally called _ in the current directory.
- **Symlinks are followed**: `stat` reports on the target of a symlink. For the link itself, use `lstat`. A broken symlink makes `stat` fail with `ENOENT`; `lstat` on the same path succeeds.
- **DIRHANDLE form**: passing a directory handle from `opendir` returns the stat of the open directory itself. Not every platform supports this; on those that do not, the call fails and `$!` is set.
- **Inode numbers may overflow integers**: on filesystems with 64-bit inode numbers beyond Perl's integer range, `ino` is returned as a decimal string to preserve the full value. Compare inode numbers with eq, not ==, to stay correct on those filesystems. eq works fine on numeric inodes too.
- **`ctime` is not creation time**: it is the inode-change time - updated by `chmod`, `chown`, `rename`, link-count changes, and the file's own writes. Linux exposes a true creation time (`btime`) via `statx(2)`, but that is not part of `stat`'s return list.

- **Zero fields on some filesystems**: `atime` (on `noatime` mounts), `blksize`, and `blocks` may be reported as `0`. Treat `0` as "unavailable", not as a meaningful value.
- **Scalar context does not populate _ on failure**: a failed `stat` leaves the previous cached buffer untouched. Always check the return value before a follow-up `stat(_)` or `-X _`.
- **Race windows**: the metadata returned is a snapshot. Anything that acts on a path based on `stat`'s result (permission checks, TOCTOU patterns) must be written to tolerate the file changing between the `stat` and the follow-up operation - or better, open the file first and `stat` the filehandle.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `lstat` - like `stat` but does **not** follow symlinks; reports on the link itself
- `-X` file-test operators (`-e`, `-f`, `-r`, `-w`, `-x`, ...) - built on top of the same stat buffer; share the _ cache with `stat` and `lstat`
- `chmod` - change the permission bits reported in field 2 (`mode`)
- `chown` - change the `uid` and `gid` reported in fields 4 and 5
- `File::stat` - named accessors (`$sb->size`, `$sb->mtime`, ...) over the same 13-field list; `use File::stat` overrides `stat` in the current scope
- `Fcntl` - `:mode` tag exports the `S_I*` constants and `S_IS*` helpers for decoding field 2

Scoping

state

Declare a lexically scoped variable whose value persists across calls to its enclosing subroutine.

A `state` variable has the same visibility rules as a `my` variable - it is visible only inside the block that declares it - but it is **initialised exactly once** per enclosing closure, not on every entry to the block. This is Perl's built-in way to write persistent private variables: counters, memoisation caches, one-time-computed lookup tables, anything you would otherwise hide inside a file-scoped lexical above a `sub`.

Synopsis

```
state $scalar
state $scalar = EXPR
state ($a, $b, undef, $c)
state @array
state %hash
```

(continues on next page)

(continued from previous page)

```
state TYPE $var
state $var : ATTRS
```

What you get back

`state` is a declaration, not a function. Like `my` it returns the variable(s) it declares, so it can be used in any expression position where the declared variable would appear:

```
(state $n //= 0)++;                # declare-and-use in one␣
↪expression
print ++(state $calls), "\n";      # count calls to this line
```

The initialiser on the right of = is evaluated **only the first time control flows through the declaration** in that particular closure instance. On every subsequent entry, the declaration re-exposes the already-initialised variable and the initialiser is skipped.

Persistence scope - what "once" means

"Once per enclosing closure" is the precise rule. A `state` variable lives as long as the CV (code value) that contains its declaration:

- In a **named sub**, the CV is created once at compile time. The `state` variable is initialised on the first call and retains its value across every later call for the life of the process.
- In an **anonymous sub** / **closure**, each `sub { ... }` evaluation produces a fresh CV with its own fresh `state` slot. Two closures made from the same source do **not** share state.
- In a **nested block** inside a sub, the `state` still belongs to the surrounding CV - it is not re-created on each entry to the inner block.

```
sub counter { state $n = 0; ++$n }
counter(); counter(); counter();    # returns 1, 2, 3

my $make = sub { sub { state $n = 0; ++$n } };
my $a = $make->();
my $b = $make->();
$a->(); $a->(); $b->();             # $a sees 1, 2; $b sees 1
```

Feature gate and availability

`state` has always been feature-gated. Two ways to enable it:

- `use feature 'state';` - turns on just this feature.
- `use v5.10;` (or any later version bundle, including `use v5.36` and `use v5.44`) - implicit via the version feature bundle. Every modern program already has `state` available through its `use vX.Y` line.

Without the feature in scope, `state` is parsed as an ordinary bareword. To reach the built-in unconditionally, write `CORE::state`.

```
use v5.36;                              # 'state' feature is on by␣
↪default
sub hits { state $n = 0; ++$n }
```

List declarations and initialisers

`state` supports the parenthesised list form for declaring several variables at once, with `undef` as a placeholder:

```
state ($x, $y, undef, $z);
```

What it does **not** yet support is **initialising a list** of `state` variables. Upstream explicitly notes this: initialisation of `state` variables in a parenthesised list is not currently possible, so the list form is useful only for declaration.

Aggregates - `state @array` and `state %hash` - have historically had the same restriction on initialisers (Perl emitted an experimental warning, and in several versions a syntax error, for `state @a = (1, 2, 3)`). Build the aggregate inside the body instead:

```
sub primes_below_100 {
    state @p;
    unless (@p) {
        @p = _sieve(100);               # fill on first call only
    }
    return @p;
}
```

The `unless (@p)` / `unless (%h)` idiom above is the portable replacement for an aggregate initialiser.

Declaration placement and same-statement semantics

Like `my`, `our`, and `local`, a `state` declaration can appear anywhere an expression is allowed (except inside string interpolation). The new binding takes effect for **subsequent statements**, not for other mentions of the same variable in the same statement:

```
package main;
use feature 'state';
our $x = 2;
foo($x, state $x = $x + 1, $x);     # foo() receives (2, 3, 2)
foo($x, $main::x);                  # foo() receives (3, 2)
```

The first `$x` and the trailing `$x` in the same call still refer to `our $x`; only the next statement sees the `state $x`.

Examples

A hit counter - the canonical one-liner:

```
sub hits { state $n = 0; ++$n }
hits(); hits(); hits();             # returns 1, 2, 3
```

Memoisation of an expensive pure function:

```
sub fib {
    my ($n) = @_;
    state %cache;
    return $cache{$n} //= $n < 2 ? $n : fib($n-1) + fib($n-2);
}
```

Lazy one-time setup of a read-only table:

```
sub months {
    state @m = qw(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec);
    return @m;
}
```

Here the initialiser is a simple list, which works because the RHS of `state @m = ...` is constant-folded to a list literal and current Perl permits this form; the more general case of a computed list still wants the `unless (@m) { ... }` pattern above for portability.

Per-closure state (fresh slot per `sub { ... }` evaluation):

```
sub make_counter {
    return sub { state $n = 0; ++$n };
}
my $a = make_counter();
my $b = make_counter();
$a->(); $a->();                         # 1, 2
$b->();                                 # 1 - independent of $a
```

Declare-and-use in a single expression:

```
while (my $line = <$fh>) {
    warn "first line was: $line" if (state $first //= $line) eq
 ↪$line;
}
```

Edge cases

- **Aggregate initialisers.** `state @a = (1, 2, 3)` and `state %h = (k => 1)` have a long history of being restricted or experimental. When in doubt, declare the aggregate and fill it under `unless (@a) { ... }` on the first call. Scalar initialisers (`state $x = EXPR`) have always been fine.
- **Same-statement reuse.** A fresh `state` declaration does **not** apply to other mentions of the same name within the same statement; those still resolve to whatever was in scope before the declaration. See the `foo($x, state $x = $x + 1, $x)` example above.
- **Shadowing warning.** Redeclaring the same name in the same scope - `state $n; state $n;` - shadows the first. Under `use warnings` this emits a warning in the `shadow` category. Almost always a bug.
- **Anonymous-sub multiplicity.** Every evaluation of `sub { ... }` produces a new CV with its own `state` slot. If you build many closures from the same source, each

has its own independent `state` - that is a feature, but it surprises people expecting file-scoped persistence.

- **Threads and forks.** `state` lives in the CV, which is cloned on `fork` (copy-on-write in the OS sense - each process continues with its own copy) and on `threads->create` (deep-cloned). Two processes or threads do not share a `state` counter.
- **`local` does not apply.** `state` variables are lexicals, not package variables, so `local` cannot save and restore them. Wrap the whole sub in its own scope or use a different mechanism if you need dynamic scoping.
- **Access via `CORE::state`.** When writing code that must parse even without the feature enabled, prefix the keyword: `CORE::state $n = 0;`.
- **Destruction.** `state` variables live as long as the CV. For a named sub in the main program that means "until global destruction." Do not put a file-handle or a large buffer in a `state` variable expecting it to be freed at sub exit - it will not be.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `my` - ordinary lexical; reinitialised on every block entry, the natural contrast to `state`
- `our` - lexical alias for a package variable; use when callers need to see or `local`-ise the value
- `local` - dynamic scoping for package variables; orthogonal to `state` and cannot be applied to it
- `sub` - `state` lives inside a CV; understanding how named vs anonymous subs differ explains per-closure state
- `use` - `use v5.10` and later enable the `state` feature as part of the version bundle
- `undef` - legal placeholder inside `state (...)` list declarations

Regular expressions and pattern matching

study

A no-op retained for source compatibility with older Perl code.

`study` accepts a scalar argument (or operates on `$_` when called with none) and does nothing with it. The call is valid, the argument is evaluated, and control returns immediately. No index is built, no regex engine state is altered, no later match runs any faster or slower because of it.

The name survives because decades of Perl code - books, FAQs, CPAN distributions, in-house scripts - sprinkle `study` in front of long strings about to be matched against many patterns. Those programs still parse and still run. They just no longer get the optimisation the keyword once promised.

Synopsis

```
study SCALAR
study
```

What you get back

Always 1. The return value is not documented as useful and no idiomatic code inspects it. Treat `study` as returning nothing worth capturing.

History

Through Perl 5.14 `study` built an inverted index of the bytes in `SCALAR`, then biased the regex engine to anchor each subsequent match on the rarest character in the pattern (rarity estimated from a static frequency table baked into the interpreter). The intended workload was "one long string, many `m//` against it": the index amortised across the matches.

The optimisation was removed in Perl 5.16. It interacted badly with Unicode, `/i` case folding, and the post-5.10 regex engine, and the maintenance cost had long outweighed the narrow workload it helped. The keyword was kept as a no-op so existing source kept compiling.

Examples

Legacy idiom - still parses, no longer has any effect:

```
study $big_text;
while ($big_text =~ /$pattern/g) {
    # ...
}
```

No argument - applied to `$_`, also a no-op:

```
for (@lines) {
    study;
    print if /error/;
}
```

Replacement guidance: if the original code relied on `study` for speed, the modern answer is to compile the pattern once with `qr` and reuse it, or to restructure so the repeated work is the pattern side, not the string side:

```
my $re = qr/$pattern/;
for my $line (@lines) {
    print $line if $line =~ $re;
}
```

Edge cases

- **No effect on regex performance.** Removing every `study` call from a program changes nothing observable, including timing beyond measurement noise.
- **Argument is still evaluated.** `study some_expensive_call()` runs `some_expensive_call()` for its side effects and return value, then discards the result. This is almost never what the author intended.
- **No warning on use.** `study` does not warn under `use warnings` and is not deprecated, so a linter is the only way to flag stale occurrences in a codebase.
- **Not a pragma, not a hint.** `study` is a regular built-in expression. It does not persist across statements and there is no "studied" state attached to the scalar.
- **Tainting and magic are untouched.** `study $tainted` does not launder taint, does not fire `FETCH` on a tied scalar more than a plain read would, and does not mark the scalar in any way.

Differences from upstream

Fully compatible with upstream Perl 5.44. `study` is a no-op in both.

See also

- `qr` - compile a pattern once and reuse the compiled form; this is the modern answer to "make many matches against the same text faster"
- `m` - the match operator whose performance `study` used to influence
- `index` - fixed-string search, faster than a regex when the pattern is a literal substring
- `pos` - where the last `/g` match left off, relevant to the "one long string, many matches" workload `study` once targeted

Control flow

sub

Declare or define a subroutine.

`sub` is the keyword that introduces a subroutine. With a `NAME` and a `BLOCK` it defines a named subroutine in the current package. Without a `BLOCK` it is a forward declaration. Without a `NAME` it is an anonymous sub expression that evaluates to a code reference - `sub { ... }` is the only form of `sub` that is actually an expression; the named forms are declarations and return nothing useful.

Synopsis

```
sub NAME BLOCK                  # named definition
sub NAME (PROTO) BLOCK          # with prototype
sub NAME : ATTRS BLOCK          # with attributes
sub NAME (PROTO) : ATTRS BLOCK  # prototype + attributes
```

(continues on next page)

(continued from previous page)

```
sub NAME;                                   # forward declaration
sub NAME (PROTO);                           # forward with prototype

my $ref = sub BLOCK;                        # anonymous, returns code ref
my $ref = sub (PROTO) : ATTRS BLOCK;        # anonymous with proto +␣
 ↪attrs

use feature 'signatures';
sub NAME ($x, $y = 0, @rest) BLOCK          # named with signature
my $ref = sub ($x, $y) BLOCK;               # anonymous with signature
```

What you get back

The **named** forms (`sub NAME BLOCK`, `sub NAME (PROTO) BLOCK`, etc.) are declarations. They install the subroutine into the current package's symbol table at compile time and contribute nothing to the surrounding expression. Writing `my $x = sub foo { 1 };` is a common mistake - that is two statements glued together, and `$x` is left undefined.

The **anonymous** form `sub BLOCK` is an expression. It evaluates - at the point the expression is reached at runtime, not at compile time - to a code reference closing over the lexical variables in scope. The same `sub { ... }` expression evaluated twice produces two distinct code references that share nothing:

```
my @refs = map { sub { $_ } } 1..3;     # three independent closures
```

Forward declarations (`sub NAME;` with no block) do nothing at runtime and return nothing. Their purpose is to announce the name to the parser so that later calls to it can parse without parentheses.

Global state it touches

- The current package (set by `package`) determines the fully-qualified name of a named sub. `sub foo { }` inside `package Acme;` installs `Acme::foo`, not `main::foo`.
- `@_` inside the body holds the argument list (aliased to the caller's scalars, except when a signature is in effect - see *Signatures* below). Assigning to `@_` as a whole breaks the aliasing for the rest of the call.
- `$_[N]` elements are aliases to the caller's scalars. Modifying `$_[0]` modifies the caller's variable; passing a literal and then modifying `$_[0]` is a fatal error.
- `wantarray` inside the body reflects the context the subroutine was called in.

Named definition

```
sub greet {
    my ($name) = @_;
    return "Hello, $name";
}

print greet("world"), "\n";             # Hello, world
```

The body is compiled at compile time. The name is visible from that point in the file onward; forward references inside the same package work because Perl defers method-style and bareword calls.

Redefining a named sub at runtime (`eval "sub foo { ... }"` or a second `sub foo { }` in the same package) replaces the previous definition and emits a `Subroutine foo redefined` warning under `use warnings`. Suppress it with `no warnings 'redefine'` for the scope of the redefinition.

Anonymous sub and closures

```
sub make_counter {
    my $n = 0;
    return sub { ++$n };
}

my $c = make_counter();
print $c->(), $c->(), $c->(), "\n";     # 123
```

The inner sub captures `$n` from its enclosing scope. Each call to `make_counter` creates a fresh `$n` and a fresh closure over it.

```
my @doublers = map { my $k = $_; sub { $k * 2 } } 1..3;
print $doublers[2]->(), "\n";           # 6
```

Note the intermediate `my $k = $_` - closures capture variables, not values, and `$_` is shared across the `map` iterations. Without the copy, every closure would see the last value of `$_`.

Signatures

With `use feature 'signatures'` (on by default under `use v5.36` and later), the parameter list appears between the name and the block, and arguments are bound to lexical variables automatically:

```
use feature 'signatures';

sub add ($x, $y) { $x + $y }

sub greet ($name, $greeting = "Hello") {
    return "$greeting, $name";
}
```

(continues on next page)

(continued from previous page)

```
sub log_all ($level, @msgs) {
    print "[$level] $_\n" for @msgs;
}
```

Signature forms:

- `$x` - required positional parameter.
- `$x = EXPR` - optional positional with default; `EXPR` is evaluated each call when the argument is missing.
- `$x = undef` - optional, defaults to `undef` (useful for documenting that a slot is nullable rather than omitted).
- `@rest` / `%rest` - slurpy, consumes all remaining arguments.
- `$` / `@` / `%` - unnamed placeholder, asserts arity without binding a variable.

Calling with the wrong arity croaks: `Too many arguments for subroutine` or `Too few arguments for subroutine`.

Under a signature, `@_` still exists but accessing it is a warning under `use warnings 'experimental::args_array_with_signatures'` in earlier Perls and discouraged in 5.44. Use the signature variables instead.

Prototypes

```
sub mypush (\@@) {
    my $aref = shift;
    push @$aref, @_;
}

my @a = (1, 2);
mypush @a, 3, 4;                    # @a is now (1, 2, 3, 4)
```

Prototypes are a **compile-time parser hint**, not a type check. They change how call sites are parsed: `\@` in the prototype above forces the first argument to be an array and passes a reference to it. See `prototype` to inspect a sub's prototype.

Prototypes do nothing when the sub is called via `&name(...)` or through a code reference. If you want a signature, use `use feature 'signatures'`; prototypes and signatures can coexist with the `:prototype(...)` attribute:

```
use feature 'signatures';
sub each_pair :prototype(\@) ($aref) { ... }
```

Attributes

Attributes appear after `:` and modify how the sub is compiled or registered:

```
sub critical :lvalue { $state }         # usable on the left of =
sub noop     :method { }                # marked as a method (for␣
↪some tools)
```

Common core attributes: `lvalue`, `method`, `prototype(...)`, `const`. Modules like `Attribute::Handlers` let a package define its own. Unknown attributes are a compile-time error.

Forward declaration

```
sub later;                              # announce the name
plan later(3);                          # now parses without parens
sub later { ... }                       # defined later in the file
```

A forward declaration is only useful to tell the parser that `later` is a subroutine so that `later ARG` parses as a call rather than an indirect-object construct or a comma expression. It is not required if you call with parentheses.

Examples

Named sub, explicit argument unpacking:

```
sub area {
    my ($w, $h) = @_;
    return $w * $h;
}
print area(3, 4), "\n";                 # 12
```

Anonymous sub passed to a higher-order function:

```
my @lengths = map { sub { length shift } }->(@_),
              ("alpha", "beta", "gamma");   # not quite - see below
```

More useful: passing a `sub` directly:

```
use List::Util qw(first);
my $first_even = first { $_ % 2 == 0 } 1, 3, 4, 7;   # 4
```

Implicit return - the value of the last expression is returned:

```
sub double { $_[0] * 2 }                # no return needed
print double(21), "\n";                 # 42
```

Context-sensitive return:

```
sub names {
    my @n = qw(alice bob carol);
    return wantarray ? @n : scalar @n;
}
```

(continues on next page)

(continued from previous page)

```
my @all = names();                          # ("alice","bob","carol")
my $n   = names();                          # 3
```

Modifying caller arguments via `@_` aliasing:

```
sub upcase_in {
    for (@_) { $_ = uc }
}
my $s = "hello";
upcase_in($s);
print $s, "\n";                             # HELLO
```

Recursion with `__SUB__` - a reference to the current sub without naming it:

```
use feature 'current_sub';
my $fact = sub {
    my $n = shift;
    $n < 2 ? 1 : $n * __SUB__->($n - 1);
};
print $fact->(5), "\n";                     # 120
```

Edge cases

- **A named sub definition is a statement, not an expression.** It contributes nothing to its enclosing expression. `my $x = sub foo {};` parses as two statements; `$x` ends up `undef`.
- **The `sub NAME BLOCK` forms are compile-time.** The block body is parsed and compiled when the enclosing file or `eval` is compiled. Side effects inside the block run only when the sub is *called*, not when it is defined. By contrast, `sub BLOCK` (anonymous) evaluates at runtime every time the expression is reached, producing a new closure each time.
- **Nested named subs close over nothing useful.** A `sub` defined inside another `sub` is still installed in the package at compile time and does **not** capture lexicals from the enclosing call. Each call to the outer sub sees the same inner sub, warped around the *first* call's lexicals. Use an anonymous sub for closures.
- **Trailing `return` is optional but explicit is clearer.** Without `return`, the value of the last *expression* is returned. If the last statement is a loop or a block, the return value is unspecified - do not rely on it.
- **Assigning to `@_` wholesale breaks aliasing.** `@_ = @_` or `@_ = (1, 2)` replaces the array and later `$_[0] = X` no longer reaches the caller.
- **Passing a literal then modifying `$_[0]` is fatal.** `upcase_in("x")` dies with `Modification of a read-only value attempted` because `$_[0]` is aliased to the literal.
- **Redefining a sub** emits `Subroutine X redefined` under `use warnings`. Wrapping a redefinition requires `no warnings 'redefine'` for the scope:

```
no warnings 'redefine';
*Foo::bar = sub { ... };
```

- **sub inside a BEGIN block** is defined at compile time of the surrounding unit, same as any sub declaration; the BEGIN wrapping does not change the timing of the declaration itself.
- **A prototype after a space gotcha.** `sub foo ($$)` with `use feature 'signatures'` is parsed as a signature, not a prototype. To attach a prototype under signatures, use `sub foo :prototype($$) ($x, $y) { ... }`.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `return` - explicit return from a sub; without it the last expression's value is returned
- `wantarray` - inspect the calling context (list / scalar / void) from inside the body
- `caller` - information about who called this sub and from where
- `prototype` - read back the prototype of a named or anonymous sub
- `__SUB__` - reference to the current sub without naming it; the tool for anonymous recursion
- `my` - the usual way to unpack `@_` into named lexicals when not using signatures
- `method` - class-syntax method declaration; `use feature 'class'` equivalent of `sub` for methods

SCALARs and strings

substr

Extract, replace, or alias a contiguous slice of a string.

`substr` picks out a run of characters from `EXPR` starting at `OFFSET` and spanning `LENGTH` characters, and returns that slice. The same call can also *modify* the original string in three distinct ways: as an lvalue on the left of `=`, through the four-argument `REPLACEMENT` form, or through an aliased lvalue stored in a variable that keeps tracking the slice across subsequent writes. Character zero is the first character. Negative `OFFSET` counts from the end; negative `LENGTH` stops that many characters short of the end. If `LENGTH` is omitted, the slice runs to the end of the string.

Synopsis

```
substr EXPR, OFFSET
substr EXPR, OFFSET, LENGTH
substr EXPR, OFFSET, LENGTH, REPLACEMENT
substr(EXPR, OFFSET, LENGTH) = NEWVALUE       # lvalue form
```

What you get back

In rvalue position, a new scalar containing the extracted slice. In lvalue position, a magic scalar that, when assigned to, splices the new value into `EXPR` at the recorded position - the returned scalar does **not** hold a copy of the original characters. The four-argument form assigns `REPLACEMENT` into `EXPR` in place and returns the substring that was there before, so the extract and the replace happen in one call.

If the requested span lies partially outside the string, only the in-bounds portion comes back. If it lies entirely outside the string, the rvalue forms return `undef` and warn; the lvalue forms raise an exception.

Global state it touches

None. `substr` is pure with respect to interpreter globals. It does not read or write `$_`, `$/`, `$\`, `$!`, or any other special variable. Warnings about out-of-range access are governed by the usual `use warnings` / `$^W` scope.

Positions and negative indices

`OFFSET` is measured from the start of the string when non-negative, and from the end when negative: `-1` is the position of the last character, `-2` the second-to-last, and so on.

`LENGTH`, when non-negative, is the number of characters to include. When negative, it is a right-anchor: the slice stops that many characters short of the end of the string. Omitting `LENGTH` is the same as "to the end."

```
my $s = "The black cat climbed the green tree";
my $color  = substr $s, 4, 5;       # "black"
my $middle = substr $s, 4, -11;     # "black cat climbed the"
my $end    = substr $s, 14;         # "climbed the green tree"
my $tail   = substr $s, -4;         # "tree"
my $z      = substr $s, -4, 2;      # "tr"
```

Using substr as an lvalue

If `EXPR` is itself an lvalue, `substr(EXPR, OFFSET, LENGTH)` can appear on the left-hand side of an assignment. The right-hand value replaces the slice. The replacement does not have to match `LENGTH`: a shorter replacement shortens the string, a longer one extends it.

```
my $name = 'fred';
substr($name, 0, 2) = 'SH';         # $name is now "SHed"
substr($name, 1, 2) = 'ayfiel';     # $name is now "Shayfield"
```

Assigning past the current end of the string is allowed, up to one position past the end (append-at-end). Any further and the assignment raises an exception.

```
my $name = 'fred';
substr($name, 4) = 'dy';            # $name is now "freddy"
substr($name, 7) = 'gap';           # exception
```

To keep the string length fixed, pad or truncate the replacement explicitly - for example with `sprintf "%-*s", $len, $val`.

The four-argument replacement form

`substr EXPR, OFFSET, LENGTH, REPLACEMENT` replaces the slice in place and returns the substring that *was* there. This is usually preferable to the lvalue form: the operation is a single expression, and the old value is available as the return value.

```
my $s = "The black cat climbed the green tree";
my $z = substr $s, 14, 7, "jumped from";    # $z is "climbed"
# $s is now "The black cat jumped from the green tree"
```

This mirrors the extract-and-replace idiom of `splice` on arrays.

The aliased-lvalue trick

The scalar returned by the three-argument form is **not** a copy of the characters - it is a magic bullet that remembers which region of the original string it refers to. Stashing that lvalue in a variable (via `for`, or `\` taken through an lvalue accessor) lets repeated writes keep splicing into the same slot of the original string, even as that string changes size:

```
my $x = '1234';
for (substr($x, 1, 2)) {
    $_ = 'a';    print $x, "\n";    # 1a4
    $_ = 'xyz';  print $x, "\n";    # 1xyz4
    $x = '56789';
    $_ = 'pq';   print $x, "\n";    # 5pq9
}
```

With a negative `OFFSET`, the alias tracks the end of the string:

```
my $x = '1234';
for (substr($x, -3, 2)) {
    $_ = 'a';    print $x, "\n";    # 1a4
    $x = 'abcdefg';
    print $_, "\n";                 # f
}
```

This is useful when repeatedly rewriting the same slot of a buffer (header fields in a fixed-layout record, for instance) - but it is also a classic source of surprises. Prefer the four-argument form unless you specifically want the aliasing.

Examples

Extract a fixed field from a record:

```
my $line = "2026-04-22  richard.jelinek\@example.com";
my $date = substr $line, 0, 10;          # "2026-04-22"
```

Strip a leading prefix of known length:

```
my $msg = "ERROR: disk full";
my $rest = substr $msg, length("ERROR: ");   # "disk full"
```

Overwrite a fixed field in place with the four-argument form, and keep the old value:

```
my $record = "name:       John      age: 42";
my $old    = substr $record, 12, 10, sprintf("%-10s", "Jane");
# $record is now "name:       Jane      age: 42"
# $old is "John      "
```

Walk a string in windows of three characters without copying the whole string each time:

```
my $s = "ABCDEFGH";
for (my $i = 0; $i + 3 <= length $s; $i++) {
    print substr($s, $i, 3), "\n";  # ABC BCD CDE DEF ...
}
```

Rewrite the last byte of a buffer repeatedly via an aliased lvalue:

```
my $buf = "packet: 0";
my $tail = \substr($buf, -1, 1);
$$tail = '1';  # $buf is now "packet: 1"
$$tail = '2';  # $buf is now "packet: 2"
```

Edge cases

- **Out-of-range rvalue, partially in bounds**: returns just the in-bounds portion, no warning.

  ```
  my $s = "abcd";
  substr($s, 2, 100);                  # "cd"
  ```

- **Out-of-range rvalue, entirely past the end**: returns `undef` and warns under `use warnings`. The empty-tail case (exactly at the end) returns `""` without a warning:

  ```
  my $name = 'fred';
  my $null = substr $name, 4, 2;       # "" (no warning)
  my $oops = substr $name, 7;          # undef, with warning
  ```

- **Out-of-range lvalue**: raises an exception. `substr` refuses to silently extend the string by a gap of unspecified contents.
- **Negative `OFFSET` larger than the string**: counts as "before the start", and is clamped to position `0` in current Perls; under `use warnings` this issues a "substr outside of string" warning in some versions. Prefer to clamp explicitly.
- **Negative `LENGTH` that exceeds the remaining tail**: yields an empty string, because the right anchor falls at or before the left anchor.

  ```
  my $s = "abcde";
  substr($s, 1, -10);                  # ""
  ```

- **Unicode**: `substr` operates in characters, not bytes. On a string with the UTF-8 flag on, positions count codepoints; on a byte string, positions count bytes. Mixing the two by treating a decoded string as a byte buffer produces wrong offsets - decode first, then index.
- **Tied or magical scalars**: lvalue `substr` honours `STORE`-time magic on the target, so tied variables see the mutated whole string, not the slice.
- **Aliased lvalue kept beyond its target's lifetime**: if the original variable goes out of scope while the alias is still alive, later writes through the alias act on a freed string; keep the alias lifetime inside the target's.
- **Pre-5.10 corner**: repeated writes through a single lvalue had unspecified behaviour in older perls, as did negative offsets before 5.16. Modern Perl (and pperl) follow the semantics documented on this page.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `index` - find the position of a substring, to feed into `substr` as `OFFSET`
- `rindex` - same, searching from the right
- `length` - the upper bound on any meaningful `OFFSET` or `LENGTH` you pass to `substr`
- `splice` - the array-level analogue of the four-argument replacement form
- `pack` / `unpack` - for fixed-layout records where many fields are read or rewritten in one shot, prefer these over a chain of `substr` calls
- `tr///` - bulk character-level substitution that doesn't need positional indices

Filehandles, files, directories

symlink

Create a symbolic link at `NEWFILE` that points at the string `OLDFILE`.

`symlink` is the thin wrapper around the POSIX `symlink(2)` system call. It does not copy anything, does not read `OLDFILE`, and does not require `OLDFILE` to exist. It writes a new directory entry at `NEWFILE` whose payload is the literal bytes of `OLDFILE`, and the kernel later resolves that payload whenever something opens, stats, or traverses the link.

Synopsis

```
symlink OLDFILE, NEWFILE
```

What you get back

1 on success, `0` on failure (with `$!` set). On systems that do not support symbolic links at all, `symlink` raises an exception rather than returning `0`. To probe for support at runtime without dying, wrap a harmless call in `eval`:

```
my $symlink_exists = eval { symlink("", ""); 1 };
```

The `eval` pattern is the idiomatic capability check - it runs a no-op `symlink` whose only job is to fail cheaply on platforms where the call exists, and to throw on platforms where it does not.

How symlinks differ from hard links

A symlink and a hard link (`link`) look similar from the shell but behave very differently:

- `link` creates a second directory entry pointing at the same inode. Both names are equal; deleting either one leaves the file alive under the other.
- `symlink` creates a tiny file of type `lnk` whose contents are a *pathname*. The target is resolved by name at every lookup. The two names are **not** equal - the symlink has its own inode, its own permissions bits (mostly ignored), and its own timestamps.

Practical consequences:

- `OLDFILE` is stored **verbatim**. It is not canonicalised, not resolved against the current working directory, and not checked for existence. A symlink can be created pointing at a path that does not exist yet (a *dangling* symlink) and will start working the moment the target appears.
- Because `OLDFILE` is just a string, it can be relative. Relative symlink targets are resolved at lookup time relative to the directory containing `NEWFILE`, **not** the process's current working directory. This is the single biggest source of surprise.
- Symlinks can cross filesystem boundaries. Hard links cannot.
- Symlinks can point at directories. Hard links to directories are forbidden on every mainstream Unix.
- `unlink` on a symlink removes the **link**, not the target. To remove the target, unlink the target's path directly.

Examples

Create a symlink to an existing file:

```
symlink "/etc/hostname", "hostname.lnk"
    or die "symlink failed: $!";
```

Dangling symlink - the target does not exist yet, and that is fine:

```
symlink "/will/appear/later", "pending.lnk"
    or die "symlink failed: $!";
# opening pending.lnk now fails with ENOENT;
# it will start working once /will/appear/later exists.
```

Relative target resolved against the link's directory, not the cwd:

```
chdir "/var/log" or die $!;
symlink "syslog", "/tmp/current.lnk"
    or die "symlink failed: $!";
# readlink("/tmp/current.lnk") eq "syslog"
# opening /tmp/current.lnk resolves syslog in /tmp, not /var/log.
```

Symlink to a directory - this works, unlike `link`:

```
symlink "/var/log", "logs"
    or die "symlink failed: $!";
opendir my $dh, "logs" or die $!;   # traverses the symlink
```

Inspect and remove a symlink without touching its target:

```
my $target = readlink "hostname.lnk";   # the stored OLDFILE string
unlink "hostname.lnk" or die $!;        # removes the link only
# /etc/hostname is untouched.
```

Atomic-swap pattern - replace a symlink pointing at the "current" release directory by writing a new symlink and renaming over it:

```
symlink "release-2", "current.new"  or die $!;
rename  "current.new", "current"    or die $!;
# rename over an existing symlink is atomic on POSIX filesystems.
```

Edge cases

- **`OLDFILE` is stored verbatim.** No normalisation, no resolution, no existence check. `symlink "", "foo"` succeeds on most systems and creates an empty-target symlink that every lookup rejects with `ENOENT`. This is why the capability-probe idiom uses `("", "")` - a cheap call that always fails on supporting platforms.
- **Relative `OLDFILE`** is resolved relative to the directory of `NEWFILE`, not the current working directory. If you build `NEWFILE` from a relative path, think carefully about what directory the kernel will see it in.
- **`NEWFILE` already exists** → failure with `$!` set to `EEXIST`. There is no "replace" flag; to overwrite, write to a fresh name and `rename` over the old one (atomic).
- **Permissions on the symlink itself** are almost always ignored. The kernel enforces the permissions of the *target*. On most filesystems the symlink is reported as `lrwxrwxrwx` regardless of what `chmod` is called on it.
- **`stat` follows symlinks, `lstat` does not.** To examine the link itself - its type, its link count, the length of its target string - use `lstat`. To examine what the link points at, use `stat`.
- **`unlink` removes the link, not the target.** There is no way to delete the target through the symlink except by resolving it first.
- **Platforms without symlink support** raise an exception on any call, even one that would otherwise fail. Use the `eval` probe shown above rather than assuming a return value.

- **Filesystems without symlink support** (some FAT variants, certain network mounts) fail with `EPERM` or `EOPNOTSUPP` even on a kernel that implements `symlink(2)`. Check `$!` on failure rather than assuming the call was rejected for the obvious reason.
- **Symlink loops** are not detected at creation time. `symlink "a", "b"` followed by `symlink "b", "a"` succeeds; the loop only surfaces when something tries to resolve either name, which fails with `ELOOP` after the kernel hits its chase limit (typically 40).

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `link` - create a hard link (same inode, same filesystem, file only); the strict counterpart to `symlink`
- `readlink` - read the `OLDFILE` string stored in a symlink without resolving it
- `lstat` - stat the symlink itself rather than its target; the only way to see that a path is a symlink from Perl
- `stat` - stat what the symlink points at, following the chain of links
- `unlink` - remove a symlink (the link, not the target); also the way to remove a dangling symlink
- `$!` - `errno` set on failure; inspect it to distinguish `EEXIST`, `EPERM`, `EOPNOTSUPP`, `ENOENT` on the parent directory

I/O · Fixed-length data

syscall

Invoke a raw system call by its kernel number, passing the remaining arguments as `int` or as a pointer to a string buffer.

`syscall` is the escape hatch for operating-system calls the rest of the language does not wrap. The first argument is the syscall number - on Linux, the value of `SYS_<name>` from `syscall.ph`. Each remaining argument is passed either as an integer (if Perl already considers it numeric) or as a pointer to the scalar's string buffer. It returns whatever the kernel returns. On failure it returns `-1` and sets `$!`.

Synopsis

```
syscall NUMBER, LIST
```

What you get back

Whatever the kernel returned. Success values are syscall-specific - often a byte count, a file descriptor, or `0`. Failure is `-1` with `$!` set to the errno value.

Some syscalls (`lseek`, `read` on a pipe at EOF with errors, `ptrace`) can legitimately return `-1` on success. To tell them apart from failures, clear `$!` before the call:

```
$! = 0;
my $r = syscall(SYS_lseek(), fileno($fh), 0, 1);
die "lseek: $!" if $r == -1 && $! != 0;
```

Global state it touches

- `$!` - set when the kernel returns `-1`. Not touched on success.

Argument passing rules

Each argument in `LIST` is inspected at call time:

- **Numeric** argument (a literal number, or a scalar that has been used in a numeric context) - passed as a C `int`.
- **Non-numeric** argument - the scalar's string buffer is passed by pointer. You are responsible for pre-extending the buffer to hold any data the kernel might write. A too-short buffer is a memory corruption bug, not a Perl error.
- **Read-only string** (a literal, a `__DATA__` string, anything Perl refuses to write through) - rejected, because `syscall` has to assume any string pointer might be written through.
- **Integer stored as a string** (e.g. a value read from a file that was never used arithmetically) - may be passed as a pointer rather than an int. Force numeric interpretation by adding `0`:

  ```
  syscall(SYS_write(), 0 + $fd, $buf, length $buf);
  ```

Up to 14 arguments are supported after the syscall number.

Examples

Write a buffer to `STDOUT` using the raw `write(2)` syscall:

```
require 'syscall.ph';
my $s = "hi there\n";
syscall(SYS_write(), fileno(STDOUT), $s, length $s);
```

Pre-extend a buffer before a read:

```
require 'syscall.ph';
my $buf = "\0" x 4096;                # 4 KiB writable buffer
my $n = syscall(SYS_read(), fileno($fh), $buf, 4096);
die "read: $!" if $n == -1;
substr($buf, $n) = '';                # trim to actual bytes read
```

Call `gettid(2)` to get the kernel thread id (no libc wrapper on older glibc):

```
require 'syscall.ph';
my $tid = syscall(SYS_gettid());
```

Distinguish a legitimate `-1` return from a real failure:

```
$! = 0;
my $off = syscall(SYS_lseek(), fileno($fh), 0, 1);   # SEEK_CUR
if ($off == -1 && $! != 0) {
    die "lseek failed: $!";
}
```

Force a stringified integer to be passed as `int`:

```
my $fd = read_config("fd");           # returns "3"
syscall(SYS_close(), 0 + $fd);        # without "0 +" it'd pass a␣
↪pointer
```

Edge cases

- **`syscall.ph` missing**: `require 'syscall.ph'` fails unless `h2ph` has been run against the system headers. Without it the `SYS_*` constants are undefined and calls become hand-maintained magic numbers.
- **Read-only argument**: passing a string literal or other read-only scalar croaks with `Modification of a read-only value`. Copy into a lexical first: `my $s = "literal"; syscall(..., $s, ...)`.
- **Buffer too short**: the kernel will happily write past the end of a scalar's buffer if you lied about the length. This corrupts the Perl heap. Always pre-extend with `$buf = "\0" x $n` or `vec($buf, $n-1, 8) = 0` before the call, and never pass `length $buf` larger than the actual capacity.
- **Argument count ceiling**: 14 arguments after `NUMBER`. Syscalls that take more (rare) are not reachable via `syscall`.
- **`SYS_pipe()` on Linux**: the raw syscall returns the read-end fd but gives you no handle on the write-end fd. Use `pipe` instead, which exposes both.
- **64-bit values on 32-bit platforms**: some syscalls take 64-bit arguments (offsets, sizes) that do not fit in a Perl `int` on 32-bit builds. There is no portable way to pass them through `syscall` - use a dedicated wrapper (`Fcntl::lseek`, `sysseek`, etc.) or a CPAN binding.
- **Unimplemented**: on platforms without a `syscall(2)` entry point (or when built without support), `syscall` raises `The syscall function is unimplemented`.
- **Signal interruption**: a syscall interrupted by a signal returns `-1` with `$!` set to `EINTR`. `syscall` does not restart automatically - retry in user code if needed.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `syswrite` - buffered-bypass write through a Perl filehandle; almost always what you want instead of `syscall(SYS_write(), ...)`
- `sysread` - buffered-bypass read; pairs with `syswrite` and handles buffer management for you
- `sysopen` - `open(2)` with the full flag vocabulary, without needing the raw syscall
- `pipe` - the right way to create a pipe; avoids the `SYS_pipe()` write-end problem
- `pack` - build the binary structs (`struct timeval`, `struct stat`, etc.) that syscalls expect in pointer arguments
- `unpack` - decode the binary structs a syscall writes into your buffer

Filehandles, files, directories

sysopen

Open a file the low-level way, passing an integer `MODE` bitmask straight through to the underlying `open(2)` system call.

`sysopen` is the escape hatch for when `open`'s string-mode DWIM is in your way - when you need `O_EXCL` for an atomic create-if-not-exist, `O_NOFOLLOW` to refuse symlinks, `O_NONBLOCK` on a FIFO, or any other flag that has no corresponding `"<"` / `">"` / `">>"` string. The flag constants live in `Fcntl` and are combined with `|`.

Synopsis

```
sysopen FILEHANDLE, FILENAME, MODE
sysopen FILEHANDLE, FILENAME, MODE, PERMS
```

What you get back

1 on success, `undef` on failure (with `$!` set to the `errno` from the failed syscall). Unlike `open`, there is no magic - `FILENAME` is taken literally, no leading `>`, `<`, `|`, or `-` is interpreted, no layered mode string is parsed.

If `FILEHANDLE` is an undefined scalar, it is autovivified into a fresh handle:

```
sysopen my $fh, $path, O_RDONLY
    or die "sysopen $path: $!";
```

MODE - the Fcntl bitmask

MODE is an integer built by OR-ing constants from Fcntl. The access mode is mandatory and mutually exclusive; the modifier flags are optional and combined freely:

Flag	Meaning
O_RDONLY	Open for reading only.
O_WRONLY	Open for writing only.
O_RDWR	Open for reading and writing.
O_CREAT	Create the file if it does not exist.
O_EXCL	With O_CREAT: fail if the file already exists.
O_APPEND	Every write seeks to end-of-file first (atomic append).
O_TRUNC	Truncate the file to zero length on open.
O_NONBLOCK	Non-blocking mode for the resulting descriptor.
O_NOFOLLOW	Fail with ELOOP if the final path component is a symlink.

Import them by name or with a tag:

```
use Fcntl qw(O_RDWR O_CREAT O_EXCL O_APPEND);
# or
use Fcntl ':DEFAULT';
```

The legacy values 0, 1, 2 for read-only, write-only, read-write also work on every system Perl supports, but name the constants - the resulting code survives cross-platform review and reads without a glossary.

PERMS - the create-time permission bits

PERMS is an octal mode (like chmod takes) applied to the inode **only when O_CREAT actually creates the file**. If PERMS is omitted, Perl passes 0666. The process umask is applied by the kernel on top, so a default umask of 022 yields 0644 on disk.

```
sysopen my $fh, $path, O_WRONLY | O_CREAT | O_EXCL, 0600
    or die "create $path: $!";
```

Do not write 0644 as the PERMS argument without a reason. A hard-coded 0644 strips group-write even from users who deliberately set a permissive umask; 0666 plus the user's umask is the portable default and is what sysopen uses when you omit the argument entirely.

Global state it touches

- $! - set on failure to the errno returned by open(2).
- ${^OPEN} - the open pragma's layer configuration. sysopen applies the **same** default PerlIO stack open would apply for a layer-less call. Use binmode immediately after a successful sysopen when you need to override or strip layers (typical for binary files, sockets-as-files, or O_NONBLOCK descriptors).
- Process umask - masks PERMS when O_CREAT creates the file.

Examples

Atomic create-if-not-exist - the canonical reason to reach for `sysopen`:

```
use Fcntl qw(O_WRONLY O_CREAT O_EXCL);
sysopen my $fh, "lock.pid", O_WRONLY | O_CREAT | O_EXCL, 0644
    or die "another instance is already running: $!";
print $fh "$$\n";
close $fh;
```

Open-or-create for read/write, preserving existing contents:

```
use Fcntl qw(O_RDWR O_CREAT);
sysopen my $fh, $path, O_RDWR | O_CREAT, 0666
    or die "sysopen $path: $!";
```

Append-only log with the append guarantee - every write lands at EOF regardless of interleaving with other writers:

```
use Fcntl qw(O_WRONLY O_CREAT O_APPEND);
sysopen my $log, "/var/log/app.log", O_WRONLY | O_CREAT | O_APPEND,␣
↪0644
    or die "open log: $!";
print $log "startup\n";
```

Open a FIFO without blocking on the reader-side handshake:

```
use Fcntl qw(O_RDONLY O_NONBLOCK);
sysopen my $fifo, "/tmp/q", O_RDONLY | O_NONBLOCK
    or die "sysopen fifo: $!";
```

Refuse to follow a symlink at the leaf - useful in directories writable by untrusted users:

```
use Fcntl qw(O_RDONLY O_NOFOLLOW);
sysopen my $fh, "$dir/config", O_RDONLY | O_NOFOLLOW
    or die "sysopen config: $!";
```

Force byte-level I/O after opening - strip any `:utf8` or `:crlf` layer the default PerlIO stack would otherwise install:

```
sysopen my $fh, $path, O_RDONLY or die $!;
binmode $fh;
```

Edge cases

- **`O_EXCL` without `O_CREAT` is a no-op.** The exclusivity check only triggers when `O_CREAT` asks the kernel to create the file. Always write them together: `O_CREAT | O_EXCL`.
- **`O_EXCL` is not a lock.** It prevents a successful open on an already-existing file, nothing more. Once the file exists, every subsequent `sysopen` with `O_CREAT | O_EXCL` fails. It does not serialize access to the contents - use `flock` for that.

- **O_EXCL on network filesystems.** NFS implementations vary; older NFS versions silently lose the exclusive semantics. Do not rely on O_EXCL across NFSv2.
- **O_CREAT | O_EXCL and symlinks.** With both flags set, the kernel refuses to open a pre-existing symlink at the final path component. It does **not** protect intermediate path components; use O_NOFOLLOW or resolve the directory with opendir plus openat semantics if you need that.
- **O_TRUNC with O_RDONLY.** Behavior is undefined by POSIX. Do not combine them.
- **Omitted PERMS.** The default is 0666, masked by umask. It is **not** 0644. Omit PERMS unless you actively want to override the user's umask.
- **FILEHANDLE as expression.** If FILEHANDLE is a bareword, it names a package global. If it is an expression evaluating to a glob, globref, or IO::Handle, that handle is opened. If it is an undefined scalar, a fresh handle is autovivified into it - the lexical form used throughout these examples.
- **No magic on FILENAME.** A FILENAME of "-" opens a file literally named -, not STDIN. A leading > opens a file literally named >filename, not a write to filename. This is the entire point of sysopen versus open.
- **Default PerlIO layers still apply.** sysopen is "low-level" relative to the mode string, not relative to PerlIO. The resulting handle has the same layer stack as a layer-less open; call binmode to change it.
- **Closed or invalid FILEHANDLE expression.** Returns undef with $! set (typically EBADF for a bad existing descriptor; ENOENT, EACCES, EEXIST, ELOOP for the path-level errors).

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- open - higher-level opener with string-mode parsing, pipes, - as STDIN / STDOUT, and layered modes; use it unless you specifically need a flag sysopen exposes
- fcntl - change flags on an already-open descriptor (e.g. set O_NONBLOCK after the fact); shares the Fcntl constant vocabulary
- sysread - unbuffered read; works on any handle, not just one obtained from sysopen, despite the name
- syswrite - unbuffered write, same note
- Fcntl - the module that exports O_* constants and the :DEFAULT / :flock / :mode tags
- binmode - set or strip PerlIO layers on the handle after opening
- umask - the process mask that subtracts bits from PERMS when O_CREAT creates the file

I/O · Fixed-length data

sysread

Read raw bytes from a filehandle by calling the underlying `read(2)` system call.

`sysread` tries to read `LENGTH` bytes from `FILEHANDLE` and store them in `SCALAR`. It bypasses the PerlIO buffering stack, which is the whole point - reads go straight to the kernel and return whatever a single `read(2)` returns, which may be fewer bytes than requested. Use it for sockets, pipes, devices, and any other situation where you want control over individual system calls rather than line- or record-oriented input.

Synopsis

```
sysread FILEHANDLE, SCALAR, LENGTH
sysread FILEHANDLE, SCALAR, LENGTH, OFFSET
```

What you get back

- The number of bytes actually read, which may be **less than** `LENGTH` on sockets, pipes, ttys, and during signal interruption.
- `0` at end of file.
- `undef` on error, with `$!` set to the `errno` from the failed `read(2)`.

`SCALAR` is grown or shrunk so that the last byte read becomes the last byte of the scalar. After a short read, the scalar holds only the bytes that were actually received - nothing is zero-padded to `LENGTH`.

```
my $n = sysread $sock, my $buf, 4096;
defined $n           or die "read failed: $!";
$n == 0              and return;          # peer closed
process_chunk($buf);                      # length($buf) == $n
```

Global state it touches

- `$!` - set when `sysread` returns `undef`. Interrupted reads show up as `EINTR`; would-block reads on non-blocking handles show up as `EAGAIN` / `EWOULDBLOCK`.
- `SCALAR` itself - always assigned to, even on failure (it is truncated to `OFFSET` bytes when `OFFSET` is given and the read fails or returns zero).
- The filehandle's byte position advances by the number of bytes returned; this is the same kernel-level position manipulated by `sysseek`.

`sysread` does not consult or touch `$/`, `$\`, `$,`, or `$_`. It is a byte-level primitive; record semantics do not apply.

OFFSET - where the bytes land inside SCALAR

Without `OFFSET`, `SCALAR` is replaced by the bytes read.

With `OFFSET`, the bytes are written **into** `SCALAR` starting at that position:

- `OFFSET >= 0` - start writing at byte `OFFSET`. If `OFFSET` is greater than the current length of `SCALAR`, the string is first padded with `"\0"` bytes up to `OFFSET`, then the new data is appended.
- `OFFSET < 0` - counted backwards from the end of `SCALAR`. `-1` means "overwrite the last byte", `-10` means "overwrite the last 10 bytes".

This is how you grow a buffer incrementally without repeatedly concatenating:

```
my $buf = "";
while ((my $n = sysread $fh, $buf, 8192, length $buf) > 0) {
    # each call appends at the current end of $buf
}
defined $n or die "read failed: $!";
```

Examples

Read up to 64 bytes from a file. `$buf` ends up exactly `$n` bytes long:

```
open my $fh, "<", "input.bin" or die $!;
my $n = sysread $fh, my $buf, 64;
defined $n or die "sysread: $!";
```

Read up to 32 bytes and place them starting at position 512 in `$buf`, padding with `"\0"` if `$buf` was shorter:

```
my $buf = "header";
my $n = sysread $fh, $buf, 32, 512;   # $buf is now 512 + $n bytes
```

Drain a socket until the peer closes. Checking for `0` - not `eof` - is the only correct end-of-stream test for `sysread`:

```
while (1) {
    my $n = sysread $sock, my $chunk, 4096;
    defined $n or die "read: $!";
    last if $n == 0;                  # orderly shutdown from peer
    handle($chunk);
}
```

Restart a read interrupted by a signal. On a blocking handle, `EINTR` is the one failure you typically retry rather than propagate:

```
use Errno qw(EINTR);
my $n;
RETRY: {
    $n = sysread $fh, my $buf, $want;
    redo RETRY if !defined $n && $! == EINTR;
}
defined $n or die "read: $!";
```

Fill a fixed-size record, tolerating short reads. A single `sysread` can return fewer bytes than asked for even on a regular file near EOF, and routinely does on sockets and pipes:

```
sub read_exact {
    my ($fh, $want) = @_;
    my $buf = "";
    while (length($buf) < $want) {
        my $n = sysread $fh, $buf, $want - length($buf), length
↪$buf;
        defined $n        or die "read: $!";
        $n == 0 and die "short read: got ", length $buf, " of $want
↪";
    }
    return $buf;
}
```

Edge cases

- **Never mix with buffered I/O on the same handle.** `readline`, `read`, `<$fh>`, `getc`, `eof`, `seek`, and `tell` all go through PerlIO buffers; `sysread` goes around them. Interleaving them on the same handle leaves data stranded in the buffer that `sysread` will never see, or skips over data the buffered reader already consumed. Use one style per handle.
- **`:utf8` is forbidden.** A handle with a `:utf8` layer (including the implicit layer added by `:encoding(...)`) makes `sysread` throw an exception. Strip the layer with `binmode` - `binmode $fh` with no second argument restores raw bytes - before calling `sysread`.
- **`:crlf` / `:perlio` still buffer.** Even without `:utf8`, the default `:perlio` stack buffers reads and translates line endings under `:crlf`. `sysread` ignores those layers, so bytes the buffered reader already pulled into the PerlIO buffer are silently lost to `sysread`. Open handles destined for `sysread` with `sysopen` or call `binmode $fh` to disable `:crlf`.
- **Short reads are normal, not errors.** A return of `$n` with `0 < $n < LENGTH` is success. Only `undef` indicates failure; `0` indicates EOF. Loop until you have what you need.
- **No `syseof`.** There is no separate end-of-file test for `sysread`; `eof` looks at the PerlIO buffer and is meaningless here. The return value of `0` is the end-of-file signal.
- **`OFFSET` beyond current length.** `sysread $fh, $buf, 10, 1_000_000` pads `$buf` with a million `"\0"` bytes before writing. This is occasionally useful for preallocating, but usually a bug.
- **Negative `OFFSET`** counts from the current end of `SCALAR`, not from `LENGTH`. `sysread $fh, $buf, 4, -4` overwrites the last four bytes of whatever `$buf` currently holds.
- **Non-blocking handles.** On a handle with `O_NONBLOCK` set, a read with nothing available returns `undef` with `$!` equal to `EAGAIN` or `EWOULDBLOCK`. Treat that as "try again later", not as an error.
- **Signals.** A blocking `sysread` interrupted by a signal returns `undef` with `$!` equal to `EINTR`. Retry the call.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `syswrite` - the write-side counterpart; same buffering rules, same short-I/O semantics
- `sysopen` - opens a handle with raw `open(2)` flags, the natural partner for `sysread` on files
- `sysseek` - byte-accurate seek that composes correctly with `sysread`; do not mix `seek` with `sysread`
- `read` - the buffered counterpart; use it for line- and record-oriented input, not for sockets or partial reads
- `binmode` - strip `:utf8` or `:crlf` from a handle before using `sysread`
- `$!` - the `errno` value after a failed `sysread`, including `EINTR`, `EAGAIN`, and `EWOULDBLOCK`

I/O · Fixed-length data

sysseek

Reposition a filehandle at the system level, bypassing PerlIO buffering.

`sysseek` moves `FILEHANDLE`'s kernel file-descriptor offset to a new byte position by calling the underlying `lseek(2)` directly. It is the companion of `sysread` and `syswrite`: the three form the unbuffered I/O family, which talks to the kernel without going through Perl's `:perlio` or stdio-like buffer stack. Use `sysseek` on handles you access with the sys-family; use `seek` on handles you access with `read`, `readline`, or `print`.

Synopsis

```
sysseek FILEHANDLE, POSITION, WHENCE
sysseek $fh, 0, 0              # rewind to start
sysseek $fh, 0, 1              # report current position
sysseek $fh, -1024, 2          # 1024 bytes before EOF
```

What you get back

The new absolute byte offset on success, or `undef` on failure (with `$!` set). A position of zero is returned as the dualvar string `"0 but true"` so that the value prints as `0` in numeric context yet remains truthy in boolean context - a check like `if (sysseek ...)` is safe at offset zero.

```
defined(sysseek $fh, $offset, 0)
    or die "sysseek to $offset failed: $!";
```

Because the return value carries the new offset, `sysseek` doubles as a "where am I" primitive - the reason the idiomatic `systell` is one line:

```
use Fcntl 'SEEK_CUR';
sub systell { sysseek($_[0], 0, SEEK_CUR) }
```

WHENCE values

`WHENCE` is an integer with three meaningful values. Prefer the symbolic constants from `Fcntl` - they also make the call portable to platforms that do not use `0` / `1` / `2`:

- `0` / `SEEK_SET` - `POSITION` is measured from the **start of the file**. `POSITION` must be non-negative.
- `1` / `SEEK_CUR` - `POSITION` is added to the **current position**. Negative values move backward, positive values forward. `sysseek $fh, 0, 1` returns the current offset without moving the pointer.
- `2` / `SEEK_END` - `POSITION` is added to the **end-of-file** offset. `POSITION` is typically zero or negative.

```
use Fcntl qw(SEEK_SET SEEK_CUR SEEK_END);

sysseek $fh, 0,   SEEK_SET;    # rewind
sysseek $fh, 0,   SEEK_END;    # move to EOF
sysseek $fh, -$n, SEEK_CUR;    # $n bytes back from here
```

Bytes, not characters

`sysseek` operates on **byte offsets**, always. Even when the handle has a character-oriented layer such as `:encoding(UTF-8)`, the offset it accepts and returns is a raw byte count. This matches the kernel view but is irrelevant in practice: `sysseek` is meant for handles that do not use PerlIO layers, and mixing it with a decoding layer is one of the confusion traps listed below.

If the data is character-oriented, use `seek` and `tell` on the buffered side instead of `sysseek`.

Do not mix with buffered I/O

`sysseek` talks directly to the file descriptor. `read`, `readline`, `print`, `write`, `seek`, `tell`, and `eof` talk to the PerlIO buffer sitting in front of the descriptor. The buffer's position and the kernel's position are independent - a `sysseek` moves the kernel pointer but does not flush or invalidate the buffer, so the next buffered read returns stale bytes from before the `sysseek`, and the next buffered write lands wherever the buffer thought it was.

The rule is simple: pick one family per handle.

- Handle used with `sysread` / `syswrite` → reposition with `sysseek`.
- Handle used with `read` / `print` / `readline` → reposition with `seek`.

Global state it touches

- `$!` - set to the system error message on failure.
- Unlike `seek`, `sysseek` does **not** flush or discard the PerlIO buffer and does **not** clear the EOF flag on the handle. That asymmetry is the source of the "mixing" hazard above and the reason the two families must not share a handle.

Examples

Random-access read of a fixed-length record. The file holds 128-byte records; jump straight to record 42:

```
use Fcntl 'SEEK_SET';
sysopen my $fh, $path, O_RDONLY or die $!;
sysseek $fh, 42 * 128, SEEK_SET or die "sysseek: $!";
sysread $fh, my $rec, 128       or die "sysread: $!";
```

Report the current offset without moving the pointer - the `systell` idiom:

```
use Fcntl 'SEEK_CUR';
my $pos = sysseek $fh, 0, SEEK_CUR;
defined $pos or die "sysseek: $!";
print "at byte $pos\n";
```

Append to a file held open for read-write, then rewind to replay from the start:

```
use Fcntl qw(SEEK_SET SEEK_END);
sysseek $fh, 0, SEEK_END or die $!;
syswrite $fh, $record     or die $!;
sysseek $fh, 0, SEEK_SET or die $!;
```

Zero is still true - the dualvar lets you write the terse form without accidentally treating a successful rewind as a failure:

```
if (my $pos = sysseek $fh, 0, 0) {
    # entered even when $pos stringifies to "0"
    printf "rewound; pos=%d\n", $pos;
}
else {
    die "sysseek failed: $!";
}
```

Seek past end-of-file to create a sparse hole, then write the tail marker:

```
use Fcntl 'SEEK_SET';
sysseek $fh, 1_000_000, SEEK_SET or die $!;
syswrite $fh, "END"               or die $!;
```

Edge cases

- **Unseekable handles**: pipes, sockets, TTYs, and most special devices fail with `$!` set to `ESPIPE` (`Illegal seek`). The return value is `undef`; always check with `defined`.
- **Do not mix with buffered I/O on the same handle**. `sysseek` bypasses the PerlIO buffer; `read`, `readline`, `print`, `write`, `seek`, `tell`, and `eof` use it. Interleaving the two families produces silently wrong results. Pick one family per handle and stay with it.
- **Zero stringifies as `"0 but true"`**. Boolean tests work at offset zero - `if (sysseek ...)` stays truthy - but string comparisons against the literal `"0"` do not:

```
my $pos = sysseek $fh, 0, 0;
$pos == 0     or die;           # true
$pos eq "0"   and die "nope";   # false - the string is "0 but
 ↪true"
```

  Use `defined` for success/failure detection and numeric comparison for the offset value.
- **Encoding layers**: `sysseek` on a handle with an `:encoding(...)` layer is almost always a mistake - the byte offset can land mid codepoint, and the sys-family is meant for raw bytes anyway. If the data needs decoding, use `seek` with the buffered family.
- **Negative offsets with `SEEK_SET`**: `POSITION` must be non-negative when `WHENCE` is `0`. A negative value fails with `EINVAL`.
- **Very large files**: `sysseek` returns the kernel's full 64-bit offset; positions beyond 2 GiB round-trip correctly through subsequent `sysseek` calls on all supported 64-bit builds.
- **Closed filehandle**: returns `undef` and sets `$!`; under `use warnings` a `sysseek() on closed filehandle` warning is emitted.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `seek` - buffered counterpart; use it on handles accessed via `read`, `readline`, or `print`
- `sysread` - the unbuffered read paired with `sysseek`; both skip PerlIO and speak directly to the file descriptor
- `syswrite` - the unbuffered write; the third member of the family that belongs on the same handle as `sysseek`
- `sysopen` - opens a handle at the descriptor level, the usual way to get a handle you will later drive with the sys-family

- `tell` - buffered-side current-offset query; pair it with `seek`, not with `sysseek`
- `Fcntl` - source of the `SEEK_SET`, `SEEK_CUR`, `SEEK_END` constants

Processes

system

Run a separate program and wait for it to finish.

`system` does exactly the same thing as `exec`, except that a `fork` is done first and the parent process waits for the child to exit. The return value is the wait status of the child, not its output - if you want the output of a command, use backticks or `qx//` instead.

Synopsis

```
system LIST
system PROGRAM LIST
```

The indirect `PROGRAM LIST` form lies to the program about its own name: `PROGRAM` is the file actually executed, while the first element of `LIST` becomes `argv[0]` inside the child. Same trick as `exec`.

What you get back

The return value is the raw status word from the underlying `wait` call - the same value that lands in `$?` after the call. To recover the program's actual exit code, shift right by eight:

```
system("make", "install");
my $exit = $? >> 8;
```

`-1` means the program could not be started at all (or the `wait` itself failed); inspect `$!` for the reason. A return of `0` means the child exited successfully with status `0`.

Shell-vs-direct execution

Argument processing depends on the shape of `LIST`, and mirrors `exec` exactly:

- **More than one argument** (or an array with more than one element): the first element is the program, the rest are its arguments. The shell is **not** invoked; `execvp` runs the program directly. This is the safe form.
- **Exactly one scalar argument**: Perl scans it for shell metacharacters. If any are present, the entire string is passed to `/bin/sh -c` for parsing. If none are present, Perl splits on whitespace and calls `execvp` directly - saving a shell.

Prefer the list form whenever the arguments come from untrusted input or contain characters the shell would interpret:

```
system("ls", "-l", $dir);              # safe, no shell
system("ls -l $dir");                  # shell parses $dir -␣
↪hazardous
```

Signals vs. exit

The status word combines three pieces of information: a signal number (if the child died on a signal), a core-dump flag, and the exit value. Decode it explicitly when you need to tell them apart:

```
system(@cmd);
if ($? == -1) {
    print "failed to execute: $!\n";
}
elsif ($? & 127) {
    printf "child died with signal %d, %s coredump\n",
        ($? & 127), ($? & 128) ? 'with' : 'without';
}
else {
    printf "child exited with value %d\n", $? >> 8;
}
```

For a structured decode, inspect `${^CHILD_ERROR_NATIVE}` - the full native status word - with the `WIFEXITED`, `WEXITSTATUS`, `WIFSIGNALED`, `WTERMSIG` macros from `POSIX`:

```
use POSIX qw(WIFEXITED WEXITSTATUS WIFSIGNALED WTERMSIG);
system(@cmd);
my $n = ${^CHILD_ERROR_NATIVE};
if (WIFEXITED($n))   { my $code = WEXITSTATUS($n); ... }
if (WIFSIGNALED($n)) { my $sig  = WTERMSIG($n);    ... }
```

`SIGINT` and `SIGQUIT` are ignored in the parent while the child runs. If you want your own program to die when the user hits Ctrl-C during `system`, check the return status yourself and re-raise:

```
system(@cmd);
kill 'INT', $$ if ($? & 127) == 2;    # re-raise SIGINT on self
```

Stdio flushing

Before the fork, Perl attempts to flush every filehandle opened for output. This prevents buffered lines from being duplicated by both parent and child. The flush is best-effort; on platforms where it is unreliable, set `$|` (`$AUTOFLUSH`) or call the `autoflush` method from `IO::Handle` on handles you care about.

The die idiom

The canonical check - terse, and it embeds the status word in the message so the caller can still decode it:

```
my @args = ("command", "arg1", "arg2");
system(@args) == 0
    or die "system @args failed: $?";
```

For a module that should throw on any failed `system` (and many other bits of Perl), use the `autodie` pragma.

Examples

Run a command and ignore the result:

```
system("clear");
```

Run a command, die on failure, and keep the status intact for downstream inspection:

```
system("tar", "xf", $archive) == 0
    or die "untar $archive failed: $?";
```

Detect whether a program is available by running it with `--version` and discarding its output:

```
my $ok = system("which $prog >/dev/null 2>&1") == 0;
```

Run under a specific shell explicitly, bypassing Perl's metacharacter-detection heuristic:

```
system("/bin/bash", "-c", $pipeline);
```

Lie about `argv[0]` using the indirect form - the kernel executes `/usr/bin/vim`, but the program sees its name as `editor`:

```
system { "/usr/bin/vim" } "editor", $file;
```

Edge cases

- **Return of -1**: the child was never created, or `wait` itself failed. Distinct from "child exited with status 255". Check `$!` for the reason - typically `ENOENT` (program not found) or `EACCES` (not executable).
- **Exit value 255 from a shell**: when the shell form is used and `/bin/sh` cannot find the program, the shell itself exits with 127. Programs that return 255 from Perl typically come through `die` in another Perl program.
- **`SIGCHLD` handler interaction**: `system` internally forks and waits. A `$SIG{CHLD}` handler can observe or race with this wait; see `perlipc` for the reaping discipline.
- **Shell quirks**: when the shell form is triggered, return codes and error reporting are whatever `/bin/sh` says they are - not always what the underlying program returned. Use the list form to avoid this.
- **Not for capturing output**: `system` writes the child's output to the inherited `STDOUT`/`STDERR`. Use `qx//` or open a pipe with `open` if you need the output as a string.
- **Windows**: only the `system PROGRAM LIST` syntax reliably avoids the shell. pperl is Linux-only, so this does not apply here; noted for portability.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `exec` - replace the current process instead of forking and waiting; same argument-processing rules as `system`
- `fork` - create a child process without running a new program; `system` is `fork` + `exec` + `waitpid`
- `waitpid` - reap a specific child, used when managing multiple children directly
- `qx` - run a command and capture its output as a string; what you want when `system`'s stdio targeting is not enough
- `$?` - child error status word set by `system`, backticks, pipe `close`, and `wait`
- `$!` - errno after a failed fork or exec (return value `-1`)
- `${^CHILD_ERROR_NATIVE}` - full native wait status for decoding with `POSIX W*` macros

I/O · Fixed-length data

syswrite

Write bytes to a filehandle with the raw `write(2)` system call, bypassing Perl's buffered I/O.

`syswrite` hands `LENGTH` bytes from `SCALAR` directly to the kernel via `write(2)`, skipping the `PerlIO` buffering that `print`, `printf`, and `write` go through. The kernel may accept fewer bytes than requested - `syswrite` returns whatever it actually wrote, and well-written code always checks. If `LENGTH` is omitted, the whole `SCALAR` is written. An optional `OFFSET` starts the write at a position other than the beginning of the string.

Synopsis

```
syswrite FILEHANDLE, SCALAR, LENGTH, OFFSET
syswrite FILEHANDLE, SCALAR, LENGTH
syswrite FILEHANDLE, SCALAR
```

What you get back

The number of bytes actually written, or `undef` on error (with `$!` set). **The returned count can be less than `LENGTH`.** This is normal on pipes, sockets, and non-blocking handles; it is not an error. Code that needs every byte delivered must loop:

```
my $off = 0;
my $len = length $buf;
while ($off < $len) {
    my $n = syswrite $fh, $buf, $len - $off, $off;
    defined $n or die "write failed: $!";
    $off += $n;
}
```

A return of `0` is legal and means "no bytes written this call, try again" - not end-of-stream. `undef` is the only error signal.

Global state it touches

- `$!` - set on error (any return of `undef`).
- No interaction with `$,`, `$\`, or `$|`. `syswrite` does not format, does not append record separators, and does not honour the autoflush flag because there is no buffer to flush - bytes go straight to the kernel.
- No interaction with the selected filehandle from `select`. `FILEHANDLE` is mandatory; there is no default.

Examples

Write a string verbatim to `STDOUT`, no separators added:

```
syswrite STDOUT, "hello\n";        # 6 bytes written
```

Write a fixed-size record and check for a short write:

```
my $rec = pack "A8 N", $name, $id;   # 12 bytes
my $n = syswrite $fh, $rec;
defined $n or die "syswrite: $!";
$n == length $rec or die "short write: $n of ${\length $rec}";
```

Write from the middle of a buffer with `OFFSET`. Useful inside a drain loop where `$off` tracks how much has been accepted so far:

```
my $buf = "ABCDEFGHIJ";
syswrite $fh, $buf, 4, 3;          # writes "DEFG"
```

Negative `OFFSET` counts from the end:

```
my $buf = "ABCDEFGHIJ";
syswrite $fh, $buf, 3, -4;         # writes "GHI"
```

Write to a non-blocking socket and handle `EAGAIN`:

```
use Errno qw(EAGAIN EWOULDBLOCK);

my $n = syswrite $sock, $buf;
if (!defined $n) {
    if ($! == EAGAIN || $! == EWOULDBLOCK) {
        # kernel buffer full - try again after select()
    } else {
        die "syswrite: $!";
    }
}
```

Edge cases

- **Do not mix with buffered I/O on the same handle.** `syswrite` bypasses `PerlIO`, but `print`, `printf`, `write`, `seek`, `tell`, and `eof` all go through the `:perlio` and `:crlf` buffering layers. Interleaving them on one handle corrupts the output order. Pick one mode per handle. If you must mix, use `read` / `sysread` symmetry rules and be deliberate.
- **Short writes are normal.** Pipes, sockets, terminals, and non-blocking handles routinely accept fewer bytes than offered. The drain-loop idiom above is the correct pattern for any handle where short writes matter.
- **Return value `0`** means the kernel accepted zero bytes this call. It is not end-of-stream and not an error; retry.
- **Length larger than available data**: if `LENGTH` exceeds `length(SCALAR) - OFFSET`, only the available bytes are written and that count is returned.
- **Empty SCALAR**: if `SCALAR` has length zero, `OFFSET` must be `0`. Any other value raises an exception.
- **`:utf8` and `:encoding(...)` layers raise an exception.** `syswrite` operates on bytes, not characters, and refuses any handle carrying a UTF-8 encoding layer. `:encoding(...)` implicitly adds `:utf8`, so it is equally forbidden. Use `binmode` to remove the layer, or write through `print` instead.
- **Characters above U+00FF** on a byte-mode handle also raise an exception - there is no way to serialise a codepoint greater than `0xFF` as a single byte. Encode the string explicitly before calling `syswrite`:

```
use Encode qw(encode);
syswrite $fh, encode("UTF-8", $text);
```

- **Broken pipe / closed socket** raises `SIGPIPE`. The default action terminates the process. Install `$SIG{PIPE} = 'IGNORE'` (or a handler) to turn it into a normal `syswrite` error with `$!` set to `EPIPE`.
- **Closed filehandle** returns `undef` with `$!` set to `"Bad file descriptor"`; under `use warnings` a `syswrite() on closed filehandle` warning is emitted.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `sysread` - the read counterpart; same buffer-bypassing semantics and same short-read caveat
- `sysopen` - open a handle at the `open(2)` level so `syswrite` can talk to it without any `PerlIO` translation layers in the way
- `sysseek` - reposition the handle using `lseek(2)`, paired with `syswrite` for random-access I/O on raw handles

- `print` - the buffered counterpart you almost always want instead; honours `$,`, `$\`, and `$|`
- `binmode` - remove a `:utf8` or `:encoding(...)` layer from a handle so `syswrite` can be used on it
- `$!` - inspect after a `undef` return to distinguish `EAGAIN` from a real failure

I/O

tell

Return the current byte position of a filehandle.

`tell` reports where the next read or write on `FILEHANDLE` will happen, measured in bytes from the start of the file. Pair it with `seek` to save a position now and return to it later. If `FILEHANDLE` is omitted, `tell` reports the position of the handle last read from - including the implicit handle inside `while (<FH>)` loops.

Synopsis

```
tell FILEHANDLE
tell
```

What you get back

A non-negative integer byte offset on success, `-1` on error. The value is suitable as the `POSITION` argument to `seek` with `WHENCE` set to `0` (absolute seek). On pipes, FIFOs, sockets, and sometimes on the standard streams, the underlying OS has no seekable position and `tell` returns `-1`; `$!` is not necessarily set in that case, so treat `-1` as the authoritative failure signal.

```
my $pos = tell $fh;
die "not a seekable handle" if $pos < 0;
# ... read or write ...
seek $fh, $pos, 0;                  # back to where we were
```

Global state it touches

- **Last-read filehandle**: the no-argument form `tell` uses the handle most recently read from in the current scope. Every `readline`, `<FH>`, or `<>` updates this. Mixing `tell` with no argument and multiple active handles is a source of surprising results - name the handle explicitly when in doubt.
- `$!` is not a reliable error channel here. Check for `-1` directly rather than testing `$!` after the call.

Bytes, not characters

`tell` always returns a **byte** offset, even when the handle has a character-oriented PerlIO layer such as `:encoding(UTF-8)` or `:utf8`. This is deliberate: translating a character count into a byte offset in a variable-width encoding would require rereading the file from the beginning. The same rule applies to `seek` and `sysseek` - they all speak bytes.

Consequence: on a UTF-8 text file you cannot in general use `tell` to count characters. Use it only to mark a position you intend to feed back into `seek`.

Examples

Remember a position, read ahead, return:

```
open my $fh, "<", "log.txt" or die $!;
my $mark = tell $fh;                # 0
my $first = <$fh>;                  # read one line
seek $fh, $mark, 0;                 # rewind to the mark
my $again = <$fh>;                  # same line again
```

Record the end-of-header offset so later code can rescan the body without reparsing the header:

```
while (my $line = <$fh>) {
    last if $line =~ /^\r?\n\z/;    # blank line ends headers
}
my $body_start = tell $fh;
# ... process body ...
seek $fh, $body_start, 0;           # second pass over the body
```

The no-argument form uses the handle last read from. Inside a `while (<>)` loop that handle is the implicit `ARGV`:

```
while (<>) {
    print "at ", tell, ": $_";      # offset within the current file
}
```

`tell` on a pipe returns `-1` because pipes are not seekable:

```
open my $ph, "-|", "ls" or die $!;
my $pos = tell $ph;                 # -1
```

Filehandles inside complex expressions need the same block form as with `print` - `tell $handles[0]` is a syntax error. Use:

```
my @handles = (\*STDIN, $fh);
my $pos = tell { $handles[1] };
```

Edge cases

- **Closed filehandle**: returns `-1`. Under `use warnings` a `tell()` on closed `filehandle` warning is emitted.
- **No-argument form, nothing read yet**: returns `-1`. The "last-read filehandle" is only meaningful after an actual read.
- **After `sysread` / `syswrite` / `sysseek`**: do not use `tell` on a handle you have manipulated with the `sys*` family. Those functions bypass PerlIO buffering; `tell` reports the buffered position, which will be wrong by the buffer contents. Use `sysseek` with a whence of 1 and an offset of `0` as the `sys*`-safe position query.
- **Standard streams**: `tell STDIN`, `tell STDOUT`, `tell STDERR` may return `-1` or a meaningful offset depending on whether the stream is backed by a regular file (shell redirection) or a terminal, pipe, or socket.
- **No `systell`**: there is no separate function for buffered-bypassing position. The idiom is:

```
my $pos = sysseek $fh, 0, 1;
```

- **Binary-mode vs text-mode on non-Linux platforms**: pperl targets Linux only, so `tell` offsets always match raw file bytes. CRLF translation layers that complicate `tell` on Windows-family perls do not apply here.
- **Append mode (`>>`)**: after `open my $fh, ">>", ...`, the initial `tell $fh` may report `0` until the first write, because some platforms position the handle at end-of-file only at each write. Don't rely on the initial position; write once, then `tell`.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `seek` - jump to an absolute byte offset previously returned by `tell`
- `sysseek` - position query and jump for handles used with `sysread` / `syswrite`, bypassing PerlIO buffering
- `telldir` - the directory-handle analogue; pairs with `seekdir`
- `eof` - test for end-of-file rather than absolute position
- `open` - produce the filehandle in the first place; mode and PerlIO layers determine whether the handle is seekable

I/O

telldir

Return the current read position of a directory handle as an opaque token.

`telldir` reports where the next `readdir` on `DIRHANDLE` will pick up. The returned value is a bookmark, not a byte offset: hand it back to `seekdir` to resume iteration at the

same entry, or discard it. The number has no arithmetic meaning - it is not an index into the directory, not a count of entries read, and not comparable across directory handles. Treat it the way you would treat the return value of `tell` on a file handle: opaque to you, meaningful only to the matching `seekdir`.

Synopsis

```
telldir DIRHANDLE
```

What you get back

An integer on success, `undef` on failure (for example, if `DIRHANDLE` is not an open directory handle). The integer is only guaranteed to round-trip through `seekdir` on the **same** open directory handle; closing and reopening the directory invalidates every previously returned token.

```
opendir my $dh, $path or die "opendir $path: $!";
my $pos = telldir $dh;
# ... readdir a few entries ...
seekdir $dh, $pos;           # back to where $pos was taken
```

Global state it touches

Sets `$!` on failure.

Examples

Bookmark a position, then return to it:

```
opendir my $dh, "." or die "opendir: $!";
readdir $dh;                 # consume "."
my $mark = telldir $dh;
my @rest = readdir $dh;      # drain the rest
seekdir $dh, $mark;
my @again = readdir $dh;     # same entries as @rest
closedir $dh;
```

Record a token before each entry so you can rewind one step:

```
opendir my $dh, $path or die "opendir $path: $!";
my $prev;
while (defined(my $entry = readdir $dh)) {
    last if $entry eq "STOP";
    $prev = telldir $dh;     # position *after* the current entry
}
seekdir $dh, $prev if defined $prev;
```

Rewinding to the start is better done with `rewinddir` than by remembering the token from before the first `readdir`:

```
opendir my $dh, $path or die "opendir $path: $!";
my @first_pass  = readdir $dh;
rewinddir $dh;                # cleaner than seekdir $dh, $start_
 ↪token
my @second_pass = readdir $dh;
```

Check for failure explicitly when the handle might be closed:

```
my $pos = telldir $dh;
defined $pos or die "telldir failed: $!";
```

Edge cases

- **Token is opaque.** Do not compare, add, subtract, or persist the value. It is valid only for `seekdir` on the same open directory handle within the same process.
- **Not portable across handles.** Two `opendir` calls on the same directory produce independent handles; a token from one is meaningless to the other.
- **Not stable across `closedir` / `opendir`.** Reopening the directory starts a fresh iteration; old tokens no longer refer to anything.
- **Directory compaction.** If the underlying filesystem or another process modifies the directory (adds, removes, or renames entries) between `telldir` and the matching `seekdir`, the resumed position may skip entries, repeat entries, or land on a different entry than the one originally bookmarked. This is a property of the C library `telldir(3)` / `seekdir(3)` pair and is not something Perl papers over. Snapshot the directory with `readdir` into a list if you need stability.
- **Closed or never-opened handle.** Returns `undef` and sets `$!`.
- **Bareword vs. scalar handle.** `telldir DH` and `telldir $dh` both work. The bareword form is a package global; the scalar form holds a reference to an anonymous directory handle created by `opendir my $dh, ...`.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `seekdir` - the only consumer of a `telldir` token; restores `DIRHANDLE` to the bookmarked position
- `readdir` - the iterator whose position `telldir` reports; advances the cursor that `telldir` observes
- `rewinddir` - cheaper than `seekdir` to a start-of-stream token when you just want to iterate the directory again
- `opendir` - creates the `DIRHANDLE` that `telldir` queries; every `telldir` token is tied to one open call
- `closedir` - invalidates every `telldir` token for the handle

- `tell` - the file-handle analogue; same opaque-token contract, different underlying system call

Classes and OO

tie

Bind a variable to a class so that every access to the variable dispatches through that class's methods.

`tie` enchants `VARIABLE` so that subsequent reads, writes, and other operations on it are forwarded to `CLASSNAME`'s tie-interface methods (`FETCH`, `STORE`, `FIRSTKEY`, `PRINT`, ...). `LIST` is passed to the class's constructor (`TIESCALAR`, `TIEARRAY`, `TIEHASH`, or `TIEHANDLE`, depending on the sigil of `VARIABLE`). The constructor returns the underlying tied object, and that object - **not** `VARIABLE` - is the return value of `tie`. Keep it if you want to call extra methods on the implementation directly.

The method contract each class must satisfy is documented at length in `perltie`; this page documents `tie` the built-in.

Synopsis

```
tie my $scalar, 'ScalarClass', @args;
tie my @array,  'ArrayClass',  @args;
tie my %hash,   'HashClass',   @args;
tie *FH,        'HandleClass', @args;

my $obj = tie my %h, 'HashClass', @args;   # keep the tied object
```

What you get back

The object returned by the class's `TIE*` constructor - an ordinary blessed reference, same as any other object. Call methods on it directly when the class exposes operations that don't map onto plain variable access:

```
my $db = tie my %cache, 'My::Cache', path => '/var/cache/app.db';
$db->flush;                 # method on the tied object
$cache{key} = 'value';      # goes through STORE
```

`tie` does not return the variable itself. Code that writes `my $x = tie ...` and then treats `$x` as the scalar is a common beginner mistake; the scalar is `$x` only if you also tied `$x`, and the assignment target is the tied object, not the variable.

To recover the tied object from a variable later, use `tied`. To dissolve the binding, use `untie`.

Which constructor is called

The sigil of `VARIABLE` picks the constructor. `LIST` is forwarded verbatim:

VARIABLE form	Constructor called	Interface reference
`$scalar`	`CLASSNAME->TIESCALAR(LIST)`	`Tie::Scalar`
`@array`	`CLASSNAME->TIEARRAY(LIST)`	`Tie::Array`
`%hash`	`CLASSNAME->TIEHASH(LIST)`	`Tie::Hash`
`*FH` (glob)	`CLASSNAME->TIEHANDLE(LIST)`	`Tie::Handle`

The constructor's arguments are conventionally those you'd pass to something like `dbm_open(3)` - a filename, a mode, a permission mask - but anything the class expects is fine.

Methods each class must provide

`tie` itself only invokes `TIE*`. Every other operation on the variable dispatches to a method named after that operation. Not all methods are mandatory - missing methods make the corresponding operations fail with a runtime error. The full menus are:

Tied scalar (`TIESCALAR`):

```
TIESCALAR classname, LIST
FETCH     this
STORE     this, value
DESTROY   this
UNTIE     this
```

Tied array (`TIEARRAY`):

```
TIEARRAY  classname, LIST
FETCH     this, index
STORE     this, index, value
FETCHSIZE this
STORESIZE this, count
CLEAR     this
PUSH      this, LIST
POP       this
SHIFT     this
UNSHIFT   this, LIST
SPLICE    this, offset, length, LIST
EXTEND    this, count
DELETE    this, index
EXISTS    this, index
DESTROY   this
UNTIE     this
```

Tied hash (`TIEHASH`):

```
TIEHASH   classname, LIST
FETCH     this, key
STORE     this, key, value
DELETE    this, key
CLEAR     this
```

(continues on next page)

(continued from previous page)

```
EXISTS    this, key
FIRSTKEY  this
NEXTKEY   this, lastkey
SCALAR    this
DESTROY   this
UNTIE     this
```

Tied filehandle (`TIEHANDLE`):

```
TIEHANDLE this, LIST
READ      this, scalar, length, offset
READLINE  this
GETC      this
WRITE     this, scalar, length, offset
PRINT     this, LIST
PRINTF    this, format, LIST
BINMODE   this
EOF       this
FILENO    this
SEEK      this, position, whence
TELL      this
OPEN      this, mode, LIST
CLOSE     this
DESTROY   this
UNTIE     this
```

See `perltie` for the full specification (method signatures, when each is called, defaults inherited from the `Tie::*` base classes).

Examples

Bind a hash to an on-disk DBM file and iterate with `each`. `keys` and `values` would materialise the whole set at once - for large DBM files that's expensive:

```
use NDBM_File;
tie(my %HIST, 'NDBM_File', '/usr/lib/news/history', 1, 0);
while (my ($key, $val) = each %HIST) {
    print $key, ' = ', unpack('L', $val), "\n";
}
```

Keep the constructor's return value so you can talk to the tied object directly:

```
my $impl = tie my %cache, 'My::Cache', ttl => 60;
$cache{session} = $data;    # STORE
$impl->expire_now;           # direct method call
```

Tie a scalar to a read-only class that logs every access:

```
package Tie::Logged {
    sub TIESCALAR { my ($class, $v) = @_; bless \$v, $class }
```

(continues on next page)

(continued from previous page)

```
    sub FETCH     { my $self = shift; warn "read\n";  $$self }
    sub STORE     { my ($s, $v) = @_;  warn "write\n"; $$s = $v }
}

tie my $x, 'Tie::Logged', 42;
print $x;           # warns "read",  prints 42
$x = 99;            # warns "write"
```

Tie a filehandle to a class that captures output into a string:

```
package Tie::StringHandle {
    sub TIEHANDLE { my $c = shift; bless { buf => '' }, $c }
    sub PRINT     { my $s = shift; $s->{buf} .= join '', @_; 1 }
    sub buf       { $_[0]->{buf} }
}

tie *OUT, 'Tie::StringHandle';
print OUT "hello\n";
print OUT "world\n";
my $obj = tied *OUT;
print $obj->buf;    # "hello\nworld\n"
```

Edge cases

- **Return value is the object, not the variable.** `my $r = tie my %h, ...` binds `$r` to the tied object. Assigning to `$r` assigns to a scalar; it does not write through to %h.
- **Module must be loaded first.** Unlike `dbmopen`, `tie` does **not** `use` or `require` `CLASSNAME` for you. Load it explicitly:

  ```
  use My::TiedClass;
  tie my %h, 'My::TiedClass';
  ```

  Forgetting this is the typical cause of `Can't locate object method "TIEHASH" via package ...`.
- **Missing methods are fatal at the point of use**, not at `tie` time. A tied hash whose class defines `FETCH` but not `STORE` ties successfully, then raises `Can't locate object method "STORE" ...` on the first assignment.
- **Sigil mismatch is a `tie`-time error.** `tie` picks the constructor from the variable's sigil, so a class that only defines `TIESCALAR` cannot be `tie`-d to a hash - the lookup of `TIEHASH` fails at the `tie` call itself.
- **`tie`ing a tied variable** implicitly `untie`s the previous binding first, triggering `UNTIE` and eventually `DESTROY` on the old object (once the last reference to it drops).
- **`local` on a tied variable** saves and restores the *binding*, not just the value. At scope exit the variable is re-tied to its original object.
- **Holding an extra reference to the tied object** keeps it alive past `untie`. Under `use warnings` this produces a `untie attempted while N inner references`

`still exist` warning. Drop the reference (e.g. `undef $obj`) before calling `untie` if you want `DESTROY` to fire promptly.

- **`tie` does not stash the object inside the variable in any user-visible way** - use `tied` to retrieve it; there's no dereference syntax that exposes it.
- **Performance.** Every operation on a tied variable dispatches through a method call. In tight loops this is visibly slower than a plain variable; for bulk work, read into a plain array or hash first, operate, then write back.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `tied` - recover the object a variable is currently tied to, or `undef` if it isn't tied
- `untie` - dissolve the binding and let the tied object be destroyed
- `bless` - the low-level primitive `TIE*` constructors use to turn a reference into an object
- `dbmopen` - older shortcut for the specific case of tying a hash to a DBM file; loads the DBM class for you, which `tie` does not
- `ref` - inspect the class of the object returned by `tie`

Classes and OO

tied

Return the object backing a tied variable.

`tied` hands you the same reference that `tie` originally returned when `VARIABLE` was bound to a class. Through that reference you can call methods directly on the tie implementation - bypassing the usual `FETCH` / `STORE` dispatch - which is how tie classes expose extra operations that have no natural scalar / array / hash syntax. If `VARIABLE` is not tied, `tied` returns `undef`.

Synopsis

```
tied VARIABLE
```

What you get back

A reference to the underlying tie object when `VARIABLE` is tied, and `undef` when it is not. The reference is the exact value originally returned by the matching `tie` call - same class, same blessed identity - so `ref(tied $v)` names the tie class and method calls on it dispatch normally:

```
my $obj = tied %hash;
$obj->flush if defined $obj;
```

Use `defined` (or a boolean test, when the tie class never blesses into a false-overloading package) to distinguish a tied variable from an untied one before calling methods on the result.

The "tie handle" trick

`tied` is the only supported way to talk to the tie implementation directly. Any method on the tie class that is not one of the standard hooks (`FETCH`, `STORE`, `EXISTS`, `DELETE`, `CLEAR`, `FIRSTKEY`, `NEXTKEY`, and their array / scalar / handle equivalents) is reachable only through the object `tied` returns:

```
tie my %cache, 'My::LRU', size => 1024;
tied(%cache)->stats;           # class-specific method, no hash␣
↪syntax
tied(%cache)->evict_older_than(3600);
```

This is also the pattern for introspection - asking a tied variable "who are you tied to?" without assuming knowledge of the binding site:

```
my $class = ref tied @array;   # e.g. "Tie::File"
```

Examples

Basic test for tiedness:

```
tie my %h, 'My::Tied::Hash';
print defined(tied %h) ? "tied\n" : "not tied\n";   # tied
```

Calling a class-specific method through the tie object:

```
tie my @log, 'Tie::File', 'app.log' or die $!;
tied(@log)->flock(2);          # LOCK_EX; Tie::File API
# ... mutate @log ...
tied(@log)->flock(8);          # LOCK_UN
```

A plain scalar, array, hash, or filehandle that was never tied:

```
my $x;
print defined(tied $x) ? "yes" : "no", "\n";        # no
```

`tied` does **not** untie - the variable stays tied after the call, and successive calls return the same reference:

```
tie my $t, 'My::Counter';
my $a = tied $t;
my $b = tied $t;
print $a == $b ? "same\n" : "different\n";          # same
```

Round-tripping through `ref` to learn the tie class:

```
tie my %db, 'DB_File', 'data.db', O_RDWR | O_CREAT, 0644;
print ref tied %db, "\n";      # DB_File
```

Edge cases

- **Not tied**: returns `undef`. No warning, no exception. This is the idiomatic way to check whether a variable is tied at all - there is no separate `is_tied` built-in.
- **Filehandles**: `tied *FH` returns the object for a tied filehandle (bound with `tie *FH, ...`). A glob that was never tied yields `undef` just like any other untied variable.
- **`tied` does not untie**. It is a pure accessor. To release the binding, call `untie`. Holding the reference returned by `tied` past the point where the variable is untied keeps the tie object alive, which is deliberate - that is how `untie` warnings about "untie attempted while N inner references still exist" are triggered.
- **Tied element of an aggregate**: when an individual array or hash element is itself tied (separate from a tie on the container), `tied $array[0]` or `tied $hash{key}` returns the per-element object, not the container's. A container tie and an element tie are independent bindings.
- **Copied values are not tied**. `my $copy = $tied_scalar;` stringifies through `FETCH`; `$copy` is a plain scalar and `tied $copy` returns `undef`. Tie is a property of the variable, not of the value.
- **References to tied variables preserve the tie.** `\%hash` where `%hash` is tied still refers to the tied container, so `tied %{ $ref }` returns the same tie object.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `tie` - bind a variable to a class; establishes the tie relationship that `tied` queries
- `untie` - release the tie; after calling it, `tied` on the same variable returns `undef`
- `ref` - get the class name of the object returned by `tied` without a separate method call
- `bless` - `tied` returns a blessed reference when the tie class blessed the object it returned from its `TIE*` constructor
- `defined` - the standard test for "is this variable tied?" is `defined tied $var`

Time

time

Return the current wall-clock time as an integer number of seconds since the system epoch.

`time` is the starting point for every other date and time operation in Perl. It reads the system clock and returns a single integer - the number of non-leap seconds since the epoch, which on every supported platform is `00:00:00 UTC, 1 January 1970` (Unix time). The result is suitable for feeding directly to `gmtime` or `localtime` to break it down into a calendar date, or for subtracting two readings to measure elapsed time.

Synopsis

```
time
```

What you get back

A plain integer: seconds since `1970-01-01 00:00:00 UTC`, truncated (not rounded) toward zero. No sub-second precision - two calls within the same second return the same value. For millisecond or microsecond granularity, use `Time::HiRes::time`.

`time` takes no arguments; the bare word is the entire call. Writing `time()` is also fine and sometimes clearer next to other function calls.

Examples

Raw reading of the clock:

```
my $now = time;
print "$now\n";                     # e.g. 1745424000
```

Human-readable local date and time - `time` pairs with `localtime` in scalar context for a formatted string:

```
print scalar localtime, "\n";       # Wed Apr 23 12:00:00 2026
```

The same reading broken into fields for calendar arithmetic:

```
my ($sec, $min, $hour, $mday, $mon, $year) = localtime time;
$year += 1900;
$mon  += 1;
printf "%04d-%02d-%02d %02d:%02d:%02d\n",
       $year, $mon, $mday, $hour, $min, $sec;
```

Elapsed-time measurement around a block of work - the idiomatic whole-second stopwatch:

```
my $t0 = time;
do_work();
my $elapsed = time - $t0;
print "took ${elapsed}s\n";
```

UTC rather than local time - feed the same integer to `gmtime`:

```
my @utc = gmtime time;              # same fields, UTC-based
print scalar gmtime, "\n";          # Wed Apr 23 10:00:00 2026
```

Edge cases

- **No sub-second resolution.** Back-to-back calls in the same second return the same integer, so `time - $t0` for fast work prints `0`. Use `Time::HiRes::time` for a floating-point reading with microsecond precision:

```
use Time::HiRes qw(time);
my $t0 = time;                          # now a float
do_work();
printf "%.6f s\n", time - $t0;
```

- **Leap seconds.** The returned count excludes leap seconds; it is POSIX "seconds since the epoch" as reported by the kernel. Code that subtracts two `time` readings measures wall-clock elapsed seconds, not SI seconds, across a leap-second boundary.
- **Integer width.** The value is a signed integer wide enough for current and far-future timestamps on all supported platforms (64-bit). The 2038 problem does not apply here.
- **Timezone.** `time` itself is timezone-agnostic - the integer names a single instant in UTC. Timezones only enter the picture when you convert via `localtime`; `gmtime` stays in UTC.
- **Monotonicity.** `time` follows the system clock. If the clock is stepped backward (NTP adjustment, manual change), a later call may return a smaller value than an earlier one. For elapsed measurements that must not go backward, use a monotonic clock via `Time::HiRes::clock_gettime(CLOCK_MONOTONIC)`.
- **In arithmetic.** `time + 3600` is one hour from now; `time - 86400` is 24 hours ago. Feed the result to `localtime` or `gmtime` to render it.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `gmtime` - break a `time` value into UTC calendar fields (or a formatted string in scalar context)
- `localtime` - same, but in the local timezone; the most common partner for `time`
- `sleep` - pause for a whole number of seconds; pairs naturally with `time` for coarse scheduling
- `alarm` - schedule a `SIGALRM` a given number of seconds in the future
- `Time::HiRes::time` - floating-point seconds with microsecond resolution when whole-second granularity is not enough

Processes · Time

times

Report CPU time consumed by this process and its terminated children.

`times` asks the kernel how many seconds of user-mode and kernel-mode CPU have been charged to the current process, and - separately - to children that have already been reaped. The return is four floating-point seconds: user, system, child-user, child-system.

It answers "how much CPU have I burned?", not "how much wall-clock time has passed?" - for that, see `time`.

Synopsis

```
my ($user, $system, $cuser, $csystem) = times;
my $user_only                         = times;    # scalar context
```

What you get back

In list context, four values:

- `$user` - CPU time spent in user-mode code of this process.
- `$system` - CPU time spent in the kernel on behalf of this process.
- `$cuser` - user-mode CPU time of **terminated** child processes.
- `$csystem` - kernel-mode CPU time of **terminated** child processes.

All four are seconds as double-precision floats; resolution is microsecond on Linux (sourced from `getrusage`).

In scalar context, `times` returns just `$user`.

`times` takes no arguments and cannot fail in a way visible to the caller.

Children's times - the "terminated" rule

`$cuser` and `$csystem` aggregate CPU only from children that have been **reaped** - i.e. a `wait` or `waitpid` has successfully collected their exit status, or they were autoreaped because `$SIG{CHLD}` was set to `'IGNORE'`. A child that is still running, or has exited but not yet been reaped (a zombie), contributes nothing.

This matters when measuring a pipeline that spawns workers:

```
my $pid = fork;
if ($pid == 0) { do_work(); exit }
# (busy child runs here)
my ($u0, $s0, $cu0, $cs0) = times;      # $cu0, $cs0 still 0
waitpid($pid, 0);
my ($u1, $s1, $cu1, $cs1) = times;      # now $cu1, $cs1 reflect␣
 ↪child
```

Until the `waitpid` returns, the child's CPU does not appear in the parent's `times`.

Examples

Measure CPU cost of a block of work, excluding wall-clock sleeps or I/O waits:

```
my ($u0, $s0) = times;
compute_heavy();
my ($u1, $s1) = times;
```

(continues on next page)

(continued from previous page)

```
printf "compute used %.3f s user + %.3f s system\n",
       $u1 - $u0, $s1 - $s0;
```

Compare CPU time to wall-clock time - a low ratio means the process was mostly waiting rather than computing:

```
use Time::HiRes qw(time);
my $t0 = time;
my ($u0, $s0) = times;
do_mixed_io_and_compute();
my $wall = time - $t0;
my ($u1, $s1) = times;
printf "cpu/wall = %.2f\n", ($u1 - $u0 + $s1 - $s0) / $wall;
```

Attribute a parallel workload to its children. Reap every child before reading `times`:

```
my @pids = map { my $p = fork; $p == 0 ? (worker($_), exit) : $p }
 ↪1..4;
waitpid($_, 0) for @pids;
my (undef, undef, $cu, $cs) = times;
printf "workers used %.2f s user + %.2f s system\n", $cu, $cs;
```

Scalar context when only the user time matters - e.g. a quick self-profiling print:

```
printf STDERR "[%.2f] reached checkpoint\n", scalar times;
```

Edge cases

- **No arguments, no prototype to worry about.** `times` is a zero-argument built-in; `times()` and `times` parse identically. It is **not** the transliteration operator - that is `tr///` (alias `y///`), which is a separate token.
- **Scalar vs list context is load-bearing.** Writing `my $t = times;` binds `$user` to `$t`, discarding the other three values silently. Use `my ($u,$s,$cu,$cs) = times;` when you want them all.
- **Unreaped children don't count.** See the "terminated" rule above. Reading `times` before `wait` gives `$cuser == $csystem == 0` even when the child is visibly burning CPU in `top`.
- **Threads do count.** On Linux, kernel-threaded work in the current process shows up in `$user` / `$system` just like single-threaded work. Threads in a separate process don't, until that process exits and is reaped.
- **Clock is monotonic-within-process, not wall-clock.** `times` never goes backwards across a single process's lifetime, but successive `times` calls are not directly comparable between parent and child - each `getrusage` accounting is local.
- **Resolution.** Linux reports microsecond-resolution ticks via `getrusage`, so short measurements (sub-millisecond) are noisy but not zero. For high-precision wall-clock timing, use `Time::HiRes::time`, not `times`.

- **Overflow.** Values are floats; they can in principle lose sub-microsecond precision after very long runs (years of CPU), but this is not a concern for normal programs.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `time` - wall-clock seconds since the epoch; use it when you want elapsed real time, not CPU time
- `Time::HiRes::time` - high-resolution wall-clock time, the usual partner for CPU accounting
- `POSIX::times` - POSIX wrapper returning an additional real-time "clock ticks since boot" value; use it when you need the elapsed field the built-in omits
- `wait` - reap a child so its CPU starts counting in `$cuser` / `$csystem`
- `waitpid` - reap a specific child for the same reason, without blocking on unrelated ones
- `fork` - the operation whose cost shows up in the child columns once the child is reaped

SCALARs and strings

tr///

Character-by-character substitution. `tr` scans a string and replaces every occurrence of a character from `SEARCHLIST` with the positionally corresponding character from `REPLACEMENTLIST`, returning the count of characters it touched.

It is not a regex. `SEARCHLIST` is a set of literal characters (with range shorthand like `a-z`), not a pattern. No metacharacters, no `\d`, no variable interpolation. The translation table is built once, at compile time.

Synopsis

```
tr/SEARCHLIST/REPLACEMENTLIST/cdsr
y/SEARCHLIST/REPLACEMENTLIST/cdsr
$str =~ tr/abc/xyz/;
$count = ($str =~ tr/aeiou//);
```

`y///` is a sed-compatibility synonym for `tr///` - identical behaviour, identical modifiers. See `y///`.

What you get back

Without the `/r` modifier, `tr` mutates the target string in place and returns the **number of characters matched** - or, under `/d`, matched-or-deleted; under `/s`, matched-before-squashing. The return value is an integer and is commonly used to *count* characters without changing anything (see the `tr/*/*/` idiom below).

With `/r`, `tr` leaves the original string alone, builds a transliterated copy, and returns **that copy**. The returned string is always a plain string - even if the target was a tied variable or a blessed object.

Target without `=~`: `tr` operates on `$_`. Target with `=~`: the left-hand side must be an lvalue (scalar, array element, hash element) unless `/r` is used, in which case any expression works.

Modifiers

The four trailing flags change what `tr` does with each character it sees. They can be combined:

- **`c`** - complement `SEARCHLIST`. The set of characters acted on becomes *every character NOT in* `SEARCHLIST`. Under `/c`, `SEARCHLIST` is sorted by codepoint after complementing, and any `REPLACEMENTLIST` is applied to that sorted set - so the order of characters you wrote in `SEARCHLIST` no longer matters.
- **`d`** - delete. Characters matched by `SEARCHLIST` that have no position in `REPLACEMENTLIST` are removed from the target entirely rather than replaced with the final replacement character.
- **`s`** - squash. A run of characters that all translate to the same output character collapses to a single instance of that output.
- **`r`** - return modified copy. Do not touch the target. Return the transliterated string instead of a count.

Replacement list length rules

The relationship between `SEARCHLIST` and `REPLACEMENTLIST` changes with the modifiers:

- If `REPLACEMENTLIST` is **shorter** than `SEARCHLIST` and `/d` is **not** in effect, the final replacement character is *replicated* to fill the gap. `tr/abcd/AB/` is the same as `tr/abcd/ABBB/`.
- If `REPLACEMENTLIST` is shorter than `SEARCHLIST` **and** `/d` is in effect, surplus `SEARCHLIST` characters are deleted. `tr/abcd/AB/d` does `tr/ab/AB/` plus `s/[cd]//g`.
- If `REPLACEMENTLIST` is **empty** and `/d` is not in effect, `REPLACEMENTLIST` becomes a copy of `SEARCHLIST`. This is the count-only form: `tr/abc//` counts a, b, c without changing anything.
- If `REPLACEMENTLIST` is empty **and** `/d` is in effect, every character in `SEARCHLIST` is deleted. `tr/abc//d` removes all a, b, c from the target.

Character ranges

A hyphen between two characters specifies an inclusive codepoint range: `tr/A-J/0-9/` is `tr/ABCDEFGHIJ/0123456789/`. A hyphen at the start, at the end, or preceded by a backslash is taken literally - `tr/-ab-/xyz/` and `tr/\-abc/wxyz/` treat the hyphen as the literal character `-`.

Only use ASCII-alphabet (same case) or ASCII-digit ranges. `a-z`, `A-Z`, `0-9`, and clean subsets like `h-k` or `B-E` are portable and mean exactly what they read as. Mixed-case ranges, `\x`-ranges, or ranges straddling the alphabet/digit boundary look obvious but produce different results on non-ASCII platforms - spell them out.

Portable Unicode ranges use `\N{U+...}` endpoints:

```
$str =~ tr/\N{U+20}-\N{U+7E}//d;    # delete all ASCII printables
```

Delimiters

The three delimiters around `SEARCHLIST` and `REPLACEMENTLIST` can be any matching pair of printable characters, not just `/`. Bracketing pairs (`{}`, `()`, `[]`, `<>`) require two independent pairs, one for each list:

```
tr(aeiouy)(yuoiea);
tr[+\-*/]"ABCD";
```

When the delimiter is a single quote (`tr'...'...'`), the lists are treated almost literally - only `\\` pairs are collapsed. Hyphens inside single-quoted `tr` are **not** range specifiers; they are literal hyphens.

Global state it touches

- `$_` - default target when neither `=~` nor `!~` is used. `tr///` alone reads and (unless `/r` is used) mutates `$_`.
- No other special variables. `tr` does not set `$&`, `$1`, etc. - those belong to regex matching, and `tr` is not regex.

Examples

ASCII lowercase to uppercase:

```
my $name = "Perl";
$name =~ tr/a-z/A-Z/;          # $name is now "PERL"
```

Count without changing - the `tr/X/X/` idiom. The replacement is the same character, so the string is unchanged; the return value is the count:

```
my $stars = "**hello**";
my $cnt   = ($stars =~ tr/*/*/);    # $cnt == 4
```

Empty `REPLACEMENTLIST` without `/d` counts, too:

```
my $digits = ($text =~ tr/0-9//);   # count ASCII digits in $text
```

Delete characters with `/d`:

```
my $s = "(555) 123-4567";
$s =~ tr/0-9//cd;               # keep only digits: "5551234567"
```

Squash runs with `/s`:

```
my $line = "hello    world";
$line =~ tr/ //s;               # "hello world"
```

Non-destructive `/r` - build a copy, leave the original:

```
my $host  = "example.com";
my $HOST  = $host =~ tr/a-z/A-Z/r;    # $host unchanged, $HOST␣
↪uppercased
```

Chain with other `/r` operators:

```
my $tag = $host =~ tr/a-z/A-Z/r
               =~ s/\./_/gr;     # "EXAMPLE_COM"
```

`map` with `/r` - turn a list of strings into a transformed list:

```
my @upper = map tr/a-z/A-Z/r, @names;
```

Whole-word compacting - collapse non-alphabetic runs to a single space:

```
$sentence =~ tr/a-zA-Z/ /cs;    # "foo, bar; BAZ!" -> "foo bar BAZ "
```

Strip the high bit off every byte:

```
$bytes =~ tr[\200-\377][\000-\177];
```

Edge cases

- **Not a regex.** `tr/\d/X/` matches the literal characters `\` and `d`, not "any digit". Character classes, anchors, quantifiers, and backreferences are all silently literal. If you want regex, use `s///`.
- **No variable interpolation.** `$var` and `@var` inside the lists are literal `$` / `@` plus the following name. The table is compiled once, when the program is parsed. To build the lists dynamically, wrap in `eval`:

  ```
  eval "tr/$from/$to/";
  die $@ if $@;
  ```

- **Repeated characters in `SEARCHLIST`** - only the first mapping counts. `tr/AAA/XYZ/` translates every `A` to `X`; the `Y` and `Z` positions are unreachable.
- **No rescan.** Transliterated output is not re-fed into the table, even if the replacement character appears in `SEARCHLIST` too. `tr/ox/xo/` turns `"oxxo"` into `"xoox"`, not some fixed point.
- **`!~` with `tr`** - returns `0` if any character was changed, `1` otherwise. Useful only for quick "nothing happened" checks:

  ```
  $foo !~ tr/A/a/     # true iff no 'A' was in $foo
  ```

- **Lvalue requirement.** Without `/r`, the `=~` target must be assignable. `"constant" =~ tr/a/b/` is a compile-time error.

- **`/r` always returns a plain string.** If the target is a tied scalar or a blessed string-overloaded object, `/r` strips the magic and gives you a fresh plain string.
- **`tr` is not the shell `tr(1)`.** Similar syntax, overlapping semantics, but case-folding beyond ASCII belongs to `lc` / `uc` / `lcfirst` / `ucfirst` or to `s///` with `\U` / `\L` - not to `tr`.
- **`/c` with a multi-character `REPLACEMENTLIST`** is non-portable across character sets. Under `/c` the search set is sorted by codepoint, and that sort order differs between ASCII and EBCDIC. Stick to single-character `REPLACEMENTLIST` when combining with `/c`, or use `/c` only with `/d` (where the replacement doesn't matter).

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `y///` - sed-compatibility synonym for `tr///`; identical behaviour, pick whichever reads better in context
- `s///` - full regex substitution; reach for it when you need patterns, captures, or variable interpolation in the search side
- `lc` - Unicode-aware lowercase; use instead of `tr/A-Z/a-z/` when the input may contain non-ASCII letters
- `uc` - Unicode-aware uppercase; complement to `lc`
- `tr` *in* `perlop` - the upstream reference for the quote-like operator form, delimiter rules, and range portability
- `$_` - default target when `tr` is used without `=~` or `!~`

I/O

truncate

Shorten (or extend) a file to an exact byte length.

`truncate` adjusts the size of the file opened on `FILEHANDLE`, or the file named by `EXPR`, so it is exactly `LENGTH` bytes long. Bytes past `LENGTH` are discarded. The file's current read/write position is **not** changed - a handle sitting past the new end of file stays there, which is almost never what you want, so plan a `seek` before the next write.

Synopsis

```
truncate FILEHANDLE, LENGTH
truncate EXPR, LENGTH
```

What you get back

1 on success, `undef` on failure with `$!` set to the underlying `errno`. A missing platform implementation raises an exception rather than returning `undef`; that is an install-time property, not something to branch on at runtime.

```
truncate $fh, 0
    or die "truncate failed: $!";
```

Filehandle vs. filename form

Two forms, chosen by what you already have in hand:

- **`truncate FILEHANDLE, LENGTH`** - the handle must be open for writing (or read/write). Use this when the caller already owns an open write handle; it avoids a fresh `open`/`close` round-trip and works even after the file has been unlinked.
- **`truncate EXPR, LENGTH`** - `EXPR` is stringified as a pathname. The file is opened, truncated, and closed internally. Use this when you only have a path and do not need the handle afterwards.

Both forms require write permission on the target. On the filehandle form, that is a property of the open mode; on the pathname form, of the filesystem permissions.

LENGTH semantics

- `LENGTH` is in **bytes**, not characters. Encoding layers on the filehandle are irrelevant - the length argument goes straight to the OS.
- `LENGTH` less than the current file size discards the tail.
- `LENGTH` equal to the current size is a no-op that still succeeds.
- `LENGTH` greater than the current size extends the file with a hole of zero bytes on filesystems that support sparse files, or with literal zero bytes otherwise. Upstream `perlfunc` calls this case undefined; in practice it behaves like POSIX `ftruncate(2)` on Linux, which is what pperl targets.

Global state it touches

- `$!` is set on failure to the underlying `errno`.

Examples

Empty a logfile without closing the handle the rest of the program is still writing to:

```
open my $log, "+<", "app.log" or die $!;
truncate $log, 0 or die "truncate: $!";
seek $log, 0, 0;                # rewind; see "Position is not reset
 ↪"
```

Truncate a file by name, no handle involved:

```
truncate "scratch.dat", 0
    or die "truncate scratch.dat: $!";
```

Rotate a pre-allocated buffer file down to the used portion:

```
my $used = tell $fh;            # bytes actually written
truncate $fh, $used
    or die "truncate: $!";
```

Extend a file to a fixed size (sparse hole on Linux):

```
open my $fh, ">", "image.raw" or die $!;
truncate $fh, 1024 * 1024 * 1024   # 1 GiB sparse file
    or die "truncate: $!";
```

Edge cases

- **Position is not reset.** After `truncate $fh, 0`, the handle still remembers whatever offset it had. A following `print $fh ...` without an intervening `seek` may write past the new end of file, leaving a zero-filled hole.
- **Read-only handle**: truncating a handle opened with `<` fails with `EBADF` / `EINVAL` depending on the platform. Open with `+<` (read/write) or `>>` plus an appropriate mode if you need to truncate through the same handle you read from.
- **Negative `LENGTH`**: the kernel rejects it and `truncate` returns `undef` with `$!` set to `EINVAL`.
- **Directory as `EXPR`**: fails with `EISDIR`. `truncate` is strictly a file operation.
- **File unlinked but still open**: the filehandle form works fine - the inode is still there as long as any handle holds it open.
- **`LENGTH` larger than current size**: on Linux this extends with a sparse hole; reading the hole yields `"\0"` bytes. Upstream `perlfunc` flags this as undefined, so portable code should not rely on it.
- **Platform without `ftruncate`/`truncate`**: raises an exception (`The truncate function is unimplemented`). Not a concern on pperl's supported platform (Linux), but matters for portable code.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `open` - produce a writable filehandle suitable for the filehandle form of `truncate`
- `seek` - reposition the handle after truncating, before writing again
- `tell` - current offset, useful as the `LENGTH` argument when trimming to "what I have written so far"

- `sysopen` - lower-level open when you need explicit flags (`O_TRUNC` truncates at open time; `truncate` is the after-the-fact equivalent)
- `unlink` - reach for this instead when you want the file gone, not merely emptied
- `$!` - `errno` on failure

Control flow

try

Run a block and divert any exception it throws into a `catch` block, with an optional `finally` block that always runs on the way out.

`try`/`catch` is Perl's structured exception-handling syntax. The `try` block executes; if any statement in it throws (via `die`, an uncaught exception from a callee, or a runtime error), control transfers to the `catch` block with the exception value bound to the parenthesised lexical. If nothing throws, the `catch` block is skipped. Unlike `eval`, a `return` inside `try` returns from the enclosing sub, not from the block - `try` is control flow, not a call frame.

The syntax must be enabled with a feature guard:

```
use feature 'try';
```

Synopsis

```
try BLOCK catch (VAR) BLOCK
try BLOCK catch (VAR) BLOCK finally BLOCK
```

What you get back

`try` is a statement, not an expression, but it yields the last evaluated value of whichever branch ran - the `try` block on success, or the `catch` block on failure. Used as the final statement of a sub or a `do` block, that value becomes the result:

```
my $value = do {
    try {
        fetch_thing(@args);
    }
    catch ($e) {
        warn "fetch failed: $e";
        $DEFAULT;
    }
};
```

Do **not** write `return` inside a `try` that is being used this way - `return` exits the enclosing sub, bypassing the assignment.

A `finally` block's final expression is discarded; it cannot contribute to the value.

Global state it touches

- `$@` is **not** the mechanism here. `try`/`catch` binds the exception to the lexical named in `catch (VAR)`, not to `$@`. Code inside `catch` sees whatever `$@` happened to hold before the `try`; rely on the lexical.
- `$_`, `$!`, and other dynamic globals are untouched by the construct itself - only by whatever code runs inside the blocks.

The `catch` variable

`catch` must be followed immediately by a parenthesised variable declaration. `my` is implicit - write `catch ($e)`, not `catch (my $e)`. The variable is lexical to the `catch` block and holds the thrown value exactly as `die` delivered it: a string, a blessed reference, or any scalar:

```
try {
    risky();
}
catch ($e) {
    if (ref $e && $e->isa('My::Exception')) {
        $e->rethrow if $e->is_fatal;
        warn $e->message;
    }
    else {
        warn "plain die: $e";
    }
}
```

There is no multi-`catch` dispatch on exception class - write the classification yourself inside the single `catch` block.

`finally`

An optional third block runs on every exit path from the `try`/`catch` construct: normal fall-through, exception caught by `catch`, exception rethrown from `catch`, or a non-local transfer (`return`, `last`, `next`, `redo`, `goto`) out of either block.

```
try {
    open_resource();
    use_resource();
}
catch ($e) {
    warn "failed: $e";
}
finally {
    close_resource();
}
```

`finally` is the right place for release code that must run no matter how the protected section exits - file handles, locks, restoring global state.

A `finally` block has restrictions the other two blocks do not:

- No `return`, no `goto`, no `last` / `next` / `redo`. Attempting any of these is a compile-time or runtime error.
- The final expression value is ignored - `finally` cannot contribute to the construct's yielded value.

`finally` remains **experimental** in Perl 5.44 and emits a warning in the `experimental::try` category when used. `try` and `catch` themselves were de-experimentalised in 5.40 and no longer warn.

Contrast with `eval { ... } / $@`

The older pattern is still supported, but `try`/`catch` fixes three long-standing pitfalls:

Pitfall	`eval` / `$@`	`try` / `catch`
`$@` clobbering between `eval` exit and the `if ($@)` check	real hazard, needs `local $@` dance	exception bound to a fresh lexical
`return` inside the protected block	returns from the `eval`, not the sub	returns from the enclosing sub
Distinguishing "no error" from "error was false"	`$@` can be falsey-but-set	`catch` only runs on a real throw

Use `try` for new code. Reach for `eval EXPR` only when you actually need string-`eval`'s runtime compilation.

Examples

Trap an exception and log it:

```
use feature 'try';

try {
    my $x = call_a_function();
    $x < 100 or die "too big: $x";
    send_output($x);
}
catch ($e) {
    warn "unable to output a value: $e";
}
print "finished\n";
```

Use `try` as an expression inside `do` to pick a fallback:

```
use feature 'try';

my $config = do {
    try { load_config($path) }
    catch ($e) { warn "using defaults: $e"; default_config() }
};
```

Release a resource unconditionally with `finally`:

```
use feature 'try';
no warnings 'experimental::try';   # finally is still experimental

my $lock = acquire_lock();
try {
    do_critical_work();
}
catch ($e) {
    warn "critical work failed: $e";
}
finally {
    release_lock($lock);
}
```

Classify exceptions inside a single `catch` block:

```
use feature 'try';

try {
    fetch_remote();
}
catch ($e) {
    if (ref $e && $e->isa('Net::Timeout')) {
        retry_later();
    }
    elsif (ref $e && $e->isa('Net::Auth')) {
        die $e;                     # rethrow auth failures
    }
    else {
        warn "unexpected: $e";
    }
}
```

`return` inside `try` returns from the enclosing sub - not just the block:

```
use feature 'try';

sub first_valid {
    for my $path (@_) {
        try {
            return load($path);     # exits first_valid on success
        }
        catch ($e) {
            warn "skip $path: $e";
        }
    }
    return;                         # all paths failed
}
```

Edge cases

- **Feature guard required.** Without `use feature 'try'` (or a `use v5.34` or later bundle that enables it), `try` is a plain identifier and the construct is a syntax error.
- **`catch` is mandatory.** `try { ... }` on its own is not legal - every `try` needs a `catch`. Use `defer` or `finally` if you want cleanup without exception handling, paired with a trivial `catch ($e) { die $e }` rethrow if you also want to let the exception propagate.
- **Rethrowing.** Inside `catch`, `die $e` rethrows. There is no implicit rethrow if you simply fall off the end of `catch`; doing so **swallows** the exception. Decide deliberately.
- **`$@` is not cleared** by entering `try` and is not set by a caught exception. Do not read `$@` inside `catch` expecting the thrown value - read the lexical.
- **`caller()` does not see `try`.** Like `while` or `foreach`, `try` is not a call frame. `caller` skips it, which is correct: since `return` propagates through `try`, `try` has no independent return semantics worth exposing to introspection.
- **Loop controls pass through.** `last`, `next`, and `redo` inside `try` or `catch` act on the enclosing loop, not on the `try` construct. A `finally` block will still run before the transfer completes.
- **`finally` restrictions are enforced.** `return`, `goto`, and loop controls inside `finally` are errors. If you need conditional cleanup that may abort, do it in `catch` or outside the construct.
- **No try-expression sugar beyond last-statement yield.** `try` is not a general expression; it can only produce a value when it is the final statement of a block that yields (`do`, a sub). `my $x = try { ... } catch ($e) { ... };` is a syntax error.

Differences from upstream

Fully compatible with upstream Perl 5.44.

- `try` and `catch` are non-experimental (as of 5.40) and emit no warnings.
- `finally` is still experimental in 5.44 and emits `experimental::try` warnings when used; silence with `no warnings 'experimental::try'`.

See also

- `catch` - the mandatory second block; receives the exception in a parenthesised lexical
- `finally` - optional third block for unconditional cleanup; still experimental in 5.44
- `die` - the throw side of the construct; any scalar (string, blessed ref, object) can be the thrown value
- `eval` - the older block-eval / `$@` pattern, still needed for string `eval`; prefer `try` for pure exception handling
- `defer` - unconditional cleanup without a surrounding `try`/`catch`; the semantic basis on which `finally` is defined

- `return` - inside `try` returns from the enclosing sub, not from the block; inside `finally` it is an error

SCALARs and strings

uc

Return an uppercased copy of a string.

`uc` walks `EXPR` character by character and returns a new string in which every cased character has been replaced by its uppercase counterpart. Characters that have no uppercase mapping - digits, punctuation, already-uppercase letters, symbols - are passed through unchanged. If `EXPR` is omitted, `uc` operates on `$_`. The input is never modified; `uc` always returns a fresh string.

`uc` is the internal function implementing the `\U` escape in double-quoted strings, so `"\Ufoo\E"` and `uc("foo")` produce the same result.

Synopsis

```
uc EXPR
uc
```

What you get back

A new scalar string containing the uppercased version of `EXPR`. The length in characters is unchanged for all cased letters in the Basic Multilingual Plane; a few characters fold to a longer form under Unicode rules (see *Edge cases*), so the byte length and character length of the result can both grow.

```
my $s = uc("Perl is GREAT");   # "PERL IS GREAT"
```

Global state it touches

- `$_` - used as the implicit argument when `EXPR` is omitted.
- `LC_CTYPE` locale - consulted when `use locale` is active. `uc` otherwise ignores the environment.
- `use bytes` / `use feature 'unicode_strings'` / `use locale` - the lexical pragmas in scope at the **call site** select which of the casing rule sets below is applied.

Which rules apply

`uc` picks one of three rule sets based on the pragmas in effect and the internal representation of the string. The logic matches `lc` exactly; the mapping table is just the reverse.

- **`use bytes` in effect** - ASCII rules. Only `a-z` change, to `A-Z` respectively. Every byte outside `0x61..0x7A` is passed through unchanged, regardless of what it would mean as Latin-1 or UTF-8. Use this only when you have deliberately opted out of Unicode handling.

- **`use locale` for `LC_CTYPE` in effect** - the current locale governs code points below 256; Unicode rules govern the rest (the latter only reachable if the string has the UTF-8 flag set). From v5.20 onward, a UTF-8 locale gives full Unicode rules for the whole string. Under non-UTF-8 locales, case changes that cross the 255/256 boundary are not well-defined and, since v5.22, trigger a locale warning. See `perllocale`.
- **String has the UTF-8 flag set** - Unicode rules.
- **`use feature 'unicode_strings'` or `use locale ':not_characters'` in effect** - Unicode rules, even for byte strings.
- **Otherwise** - ASCII rules. Anything outside `a-z` is passed through unchanged. This is the historic default for strings that have neither the UTF-8 flag nor a lexical pragma opting into Unicode.

The practical takeaway: if you want predictable Unicode uppercasing of arbitrary input, put `use feature 'unicode_strings';` (or a modern `use v5.12;` or higher) at the top of the file and stop worrying about which representation the string happens to have.

Examples

Basic ASCII:

```
my $s = uc("hello");            # "HELLO"
```

Omitted argument operates on `$_`:

```
for ("alpha", "beta") {
    print uc, "\n";             # "ALPHA\n", "BETA\n"
}
```

`\U` in a double-quoted string is the same operation:

```
my $s = "Perl is \Ugreat\E";    # "Perl is GREAT"
```

Unicode uppercasing under `unicode_strings`:

```
use feature 'unicode_strings';
my $s = uc("straße");           # "STRASSE"
```

The German sharp s has no single-character uppercase in Unicode; it maps to the two-character sequence `SS`. The returned string is therefore one character longer than the input.

Greek lowercase letters uppercase into capitals, with unchanged characters passed through:

```
use feature 'unicode_strings';
my $s = uc("Καλημέρα, world!"); # "ΚΑΛΗΜΈΡΑ, WORLD!"
```

Byte mode confines the operation to ASCII, which is occasionally what you want for protocol tokens:

```
use bytes;
my $s = uc("héllo");           # "HéLLO" - only the ASCII letters␣
↪change
```

Edge cases

- **undef**: `uc(undef)` returns `""` and triggers an `uninitialized` warning under `use warnings`. Guard inputs that may be undefined if that warning matters.
- **Empty string**: `uc("")` returns `""`. No warning.
- **One-to-many mappings**: A small number of characters expand to multiple characters on uppercase - the best-known is U+00DF (ß) → SS, and the Greek U+0390 / U+03B0 expansions. The result string is longer than the input in those cases. Do not assume `length(uc($s)) == length($s)`.
- **Titlecase vs uppercase**: `uc` does **not** titlecase the first letter. Titlecase is a distinct Unicode category; `ucfirst` applies it to the first character and leaves the rest alone.
- **Non-cased characters**: Digits, punctuation, whitespace, symbols, and CJK ideographs are returned unchanged. `uc("42!")` is `"42!"`.
- **In-place update is not a thing**: `uc` returns a new value; it never modifies its argument. To uppercase in place, assign back: `$s = uc $s`.
- **Context**: `uc` is always scalar. Calling it in list context still produces a single string.
- **Byte-string / Unicode-flag surprises**: A string built from a `read` without a `:utf8` layer has no UTF-8 flag, so `uc` without `use feature 'unicode_strings'` will apply **ASCII rules** even if the bytes are valid UTF-8. Decode first, or enable the pragma, to get the casing the bytes look like they should get.
- **`use locale` and non-UTF-8 locales (v5.22+)**: Case changes that would cross the 255/256 boundary emit a `Can't do uc("…") on non-UTF-8 locale` warning and return the input character unchanged.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `lc` - the inverse operation; lowercases every cased character using the same rule selection
- `ucfirst` - uppercase (titlecase) only the first character of the string; for a Unicode-aware `ucfirst . lc $rest` pattern when titlecasing a whole word
- `lcfirst` - lowercase only the first character
- `fc` - Unicode casefolding for case-insensitive comparison; use this, not `uc` or `lc`, when comparing strings for equivalence
- `sprintf` - the `%s` conversion does not casefold; combine with `uc` when a format needs an uppercased field

- `tr///` - the `tr/a-z/A-Z/` idiom uppercases only ASCII and is faster than `uc` for guaranteed-ASCII input

SCALARs and strings

ucfirst

Return a copy of a string with its first character titlecased.

`ucfirst` takes a string, converts its **first character** to the Unicode **titlecase** form, and returns the result. Every other character is left untouched. The original is never modified. If the argument is omitted, `ucfirst` operates on `$_`. Titlecase usually coincides with uppercase - `"a"` becomes `"A"` - but differs for a small number of digraphs in scripts that distinguish the two (see *Edge cases*). The same pragma, locale, and UTF-8-flag rules that govern `lc` and `uc` govern `ucfirst`.

Synopsis

```
ucfirst EXPR
ucfirst
```

What you get back

A new string of the **same length in characters** as the input - only the first character is altered. The return value is always a fresh scalar; modifying it does not affect the argument, and modifying the argument afterward does not affect the returned string. The byte length may change by a few bytes if the first character's titlecase form has a different UTF-8 encoding from its original form.

```
my $str = ucfirst("hello world!");   # "Hello world!"
```

Global state it touches

- Reads `$_` when called without an argument.
- Observes `use locale` for `LC_CTYPE` - with locale in effect the current locale's uppercasing table applies to a first character whose code point is below 256.
- Observes `use bytes` - under `use bytes` only a leading `a-z` changes, to `A-Z`.
- Observes `use feature 'unicode_strings'` - forces Unicode rules regardless of the UTF-8 flag on the input.

Which casing rules apply

`ucfirst` picks one of five rulesets for the **first character**, in priority order. The first one whose condition holds wins. Subsequent characters are always passed through unchanged, regardless of which ruleset applied to the first.

1. **`use bytes` in effect** - ASCII rules. A leading `a-z` maps to `A-Z`; every other leading byte is left alone, including bytes in the 128-255 range that would otherwise be Latin-1 letters.

2. **`use locale for LC_CTYPE` in effect** - the current locale's tables apply to a first character below code point 256; a first character at 256 or above (only reachable when the string already carries the UTF-8 flag) uses Unicode rules. From Perl 5.20 onward, a UTF-8 locale uses full Unicode rules throughout.
3. **The argument has the UTF-8 flag set** - Unicode titlecase rules apply to the first character.
4. **`use feature 'unicode_strings'` or `use locale ':not_characters'` in effect** - Unicode titlecase rules apply to the first character, regardless of the UTF-8 flag.
5. **Otherwise** - ASCII rules. A leading character outside a-z is returned unchanged, including Latin-1 lowercase letters like à-þ.

The upshot: if you want predictable Unicode behaviour on every string regardless of how it was constructed, enable `use feature 'unicode_strings'` (or `use v5.12` and above, which turns it on for you).

Titlecase vs uppercase

For the overwhelming majority of characters, titlecase and uppercase are the same code point - `ucfirst("abc")` and `uc(substr("abc", 0, 1)) . substr("abc", 1)` produce identical results. They differ only for a handful of Unicode characters whose uppercase form is a sequence of two capital letters but whose titlecase form is a single capital-followed-by-lowercase letter. The standard example is the Latin digraph dz (U+01F3):

- uppercase is DZ (U+01F1) - `"DZ"` rendered as one capital glyph.
- titlecase is Dz (U+01F2) - `"Dz"`, capital D followed by small z.

`ucfirst` returns titlecase, which is the correct form when the character stands at the start of a capitalised word. `uc` on the same first character would return the two-capital form, which reads as if the whole word were in caps.

Examples

Basic ASCII first-character uppercasing:

```
my $s = ucfirst("hello, world!");     # "Hello, world!"
```

Default to `$_`:

```
for ("foo", "bar", "baz") {
    print ucfirst, "\n";              # Foo / Bar / Baz
}
```

Unicode characters only titlecase when the rules allow it:

```
use feature 'unicode_strings';
my $s = ucfirst("äpfel");             # "Äpfel"
```

Without `unicode_strings` and without the UTF-8 flag on the input, a non-ASCII first character passes through unchanged:

```
my $bytes = "\xE4pfel";                 # ä as a single Latin-1 byte
my $cap   = ucfirst $bytes;             # unchanged: "\xE4pfel"
```

`ucfirst` is what backs the `\u` escape inside double-quoted strings:

```
my $str = "\uperl\E is great";          # "Perl is great"
```

Capitalise every word - combine with `split`, `map`, and `join`:

```
my $title = join " ", map { ucfirst lc $_ } split / /, "HELLO world
 ↪";
# "Hello World"
```

Titlecase a digraph that distinguishes titlecase from uppercase:

```
use feature 'unicode_strings';
my $t = ucfirst("\x{01F3}xyz");          # "\x{01F2}xyz" - Dz, not DZ
```

Edge cases

- **`undef` argument** raises an `uninitialized` warning under `use warnings` and returns the empty string.
- **Empty string** returns the empty string - there is no first character to change.
- **Single-character string** behaves exactly like `uc` would for that one character under the same ruleset, except in the digraph cases described above where titlecase and uppercase differ.
- **Numbers are stringified first**: `ucfirst(42)` returns `"42"` - the leading `"4"` is not a cased character, so nothing changes.
- **Leading whitespace or punctuation** is **not** skipped. `ucfirst(" hello")` returns `" hello"` - the first character is the space, which has no titlecase form. If you need to capitalise the first *letter*, strip leading non-letters first or use a regex (`s/\b(\w)/\u$1/`).
- **`LATIN SMALL LETTER SHARP S` (U+00DF, ß)** titlecases to `"Ss"` under full Unicode rules - a length change that contradicts the usual "same length in characters" guarantee. Under `use locale` on a non-UTF-8 locale, Perl leaves the character unchanged rather than guessing, and from Perl 5.22 onward this raises a locale warning.
- **Locale and the UTF-8 flag interact**: under `use locale` on a non-UTF-8 locale, a first character below 256 follows the locale while a first character 256 or above follows Unicode. In practice `ucfirst` only ever touches one character, so the interaction is subtler than for `lc` / `uc`, but it is still observable in strings that begin with supplementary-plane characters.
- **Tied variables** have their `FETCH` called once. `ucfirst` does not modify the tied value; it returns a plain (non-tied) scalar.
- **`ucfirst` is a unary named operator**, not a list operator. `ucfirst $a, $b` parses as `(ucfirst $a), $b` - only `$a` is transformed, and the comma operator discards the result. Use parentheses or `map` when you mean to titlecase multiple strings:

```
my @capped = map { ucfirst } @words;
```

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `lcfirst` - the lowercasing counterpart; turns the first character into its lowercase form
- `uc` - uppercase every character in the string; use when you want the whole string capitalised, not just the first letter
- `fc` - Unicode casefold, the correct choice for case-insensitive comparison when input may contain non-ASCII
- `$_` - the default subject `ucfirst` reads when called with no argument
- `use locale` - controls whether locale tables or ASCII/Unicode rules apply to the first character when it is below code point 256
- `use feature 'unicode_strings'` - forces full Unicode titlecasing regardless of the UTF-8 flag on the input

Filehandles, files, directories

umask

Set or read the process file-creation mode mask.

`umask` controls which permission bits are **cleared** from the `MODE` argument of every file- or directory-creation call made by this process from now on. It is a property of the running process, not of any particular filehandle, and it is inherited by child processes. Call `umask EXPR` to install a new mask and receive the previous one; call bare `umask` to read the current mask without changing it.

Synopsis

```
umask EXPR
umask
```

What you get back

The **previous** umask as an integer, so you can save it and restore it later:

```
my $old = umask 0077;
# ... create private files ...
umask $old;
```

Bare `umask` returns the current mask without changing it. If the system has no `umask(2)` call and the caller is trying to restrict access for itself (`(EXPR & 0700) > 0`), an exception is raised; if the call is not implemented and the caller is not restricting its own access, `undef` is returned. On Linux this case does not arise.

How the mask is applied

The mask is **subtractive**. Every bit set in the umask is cleared from the `MODE` passed to `mkdir`, `sysopen`, `open` in creation modes, and the underlying `creat(2)` / `mkdir(2)` system calls. The effective permissions of the new file are `MODE & ~umask`.

A Unix permission like `rwxr-x---` is three sets of three bits, or three octal digits - `0750`. The leading `0` marks the literal as octal and is not itself a permission digit. A umask is written the same way and names the bits you want **disabled** for newly created files.

Worked example. With `umask 0022`, asking `sysopen` for `0666` actually creates a file with mode `0644`:

```
  0666  &~  0022   =   0644
  rw-rw-rw-    ----w--w-      rw-r--r--
```

With `umask 0027` (group may not write, others may not read, write, or execute), the same `0666` request yields `0640` (`0666 & ~0027 == 0640`).

Global state it touches

The mask itself is process-global kernel state set via `umask(2)`. `umask` does not read or write any Perl special variable directly, but every creation call in the process observes it, and `$!` is set by the underlying system calls that *consume* the mask (`mkdir`, `sysopen`, `open`), not by `umask` itself. Child processes spawned via `fork`, `exec`, or `system` inherit the current value.

Examples

Install a mask that hides the file from everyone but the owner, keeping the previous value for restoration:

```
my $old = umask 0077;
open my $fh, ">", "secret.txt" or die "open: $!";
# secret.txt is created mode 0666 & ~0077 == 0600
close $fh;
umask $old;
```

Read the current mask without changing it - useful for logging or diagnostics:

```
printf "current umask: %04o\n", umask;      # e.g. "current umask:␣
↪0022"
```

Typical values to pass - these are the three masks actually used in practice:

```
umask 0022;   # group + other: read/execute, no write   (default on␣
↪most systems)
```

(continues on next page)

(continued from previous page)

```
umask 0027;   # group: read/execute; other: nothing
umask 0077;   # owner-only; nothing for group or other
```

Scoped change around a block of file-creating code. `umask` has no `local` magic of its own - save and restore explicitly:

```
sub write_private_file {
    my ($path, $data) = @_;
    my $old = umask 0077;
    open my $fh, ">", $path or do {
        umask $old;
        die "open $path: $!";
    };
    print $fh $data;
    close $fh;
    umask $old;
}
```

Octal literal vs. string - a common trap. A umask is a **number**, not a string of octal digits. Reading one from configuration requires `oct`:

```
my $from_config = "0027";                    # string
umask oct $from_config;                      # correct: 0027 as a␣
↪number
umask $from_config;                          # WRONG: decimal 27 ==␣
↪033 octal
```

Edge cases

- **Bare `umask` in list context** still returns a single integer. The function is scalar-valued.
- **Negative or out-of-range values** are masked to the nine low-order permission bits by the kernel; only bits `0777` survive. Passing `0777` disables all permissions on every newly created file, which is almost never what you want.
- **`umask` does not affect existing files.** It only filters the `MODE` argument of *creation* calls. To change an existing file's permissions use `chmod`.
- **`umask` does not affect `open` for an existing file** - opening a file that already exists does not re-apply the mask. It applies only when a file is being created (`O_CREAT` in `sysopen`, or `>`/`>>`/`+>` modes creating a new pathname in `open`).
- **Inheritance across `fork`/`exec`.** Children start with the parent's current umask. Setting the mask before `system` or `exec` changes the environment the child inherits; restore it afterwards if the parent still cares.
- **Threads.** The umask is a per-process attribute in the kernel, so on systems where Perl threads share a process (the usual case) every thread sees the same mask. Changing it in one thread changes it for all.
- **`umask(2)` not implemented.** The POD documents a fallback path for systems without the syscall: raise on attempts to restrict the owner's own access, otherwise re-

turn `undef`. Every platform pperl targets has `umask(2)`, so this path is effectively unreachable here.

- **Octal presentation**. Print the mask with `printf "%04o"`; the default `%d` prints decimal, which is unreadable as permission bits.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `chmod` - change permissions on an **existing** file; the counterpart to the creation-time filtering `umask` performs
- `mkdir` - directory creation call whose `MODE` argument is filtered through the current umask
- `sysopen` - low-level file creation whose `MODE` argument is filtered through the current umask; the canonical place where the mask is felt
- `open` - in creation modes (`>`, `>>`, `+>`) the underlying `creat(2)` applies the umask to the default `0666`
- `oct` - convert a string like `"0027"` to the integer value `umask` expects
- `$!` - set by the creation calls that consume the mask, not by `umask` itself; check it after the `open` / `sysopen` / `mkdir` that fails

Misc

undef

The undefined value, and the operator that produces it.

`undef` wears two hats. Bare `undef` is a **value** - the canonical undefined scalar, the thing `defined` returns false for. `undef EXPR` is a **unary operator** that undefines its argument in place: the variable, array, hash, subroutine, or typeglob named by `EXPR` is emptied out and, where possible, its storage is released.

Both forms evaluate to the undefined value.

Synopsis

```
undef                  # the undefined value
undef EXPR             # undefine the lvalue EXPR, return undef
```

`EXPR` must be an lvalue: a scalar, an array (with `@`), a hash (with `%`), a subroutine (with `&`), or a typeglob (with `*`).

What you get back

Always the undefined value. The operator form returns `undef` *after* wiping its argument; the value form just hands you `undef`.

```
my $x = undef;                          # $x is now undef
return undef if $oops;                  # explicit "I mean no value"
my $was = undef $counter;               # $counter wiped, $was is undef
```

The value vs. the operator

Bare `undef` is the undefined scalar. Assigning it to a variable leaves the variable's storage in place and sets its value to undef:

```
my @big = (1) x 1_000_000;
$big[0] = undef;                        # slot 0 holds undef; @big still␣
↪huge
@big    = ();                           # empties the array, keeps the␣
↪AV
@big    = undef;                        # assigns the ONE-element list␣
↪(undef)
```

Note the last line: `@big = undef` is a list assignment of a one-element list whose only element is undef, so `@big` ends up with length 1. It is almost never what you want. Use `@big = ()` to empty, or `undef @big` to empty *and* release the backing storage.

The operator form does more than assign undef:

```
undef $scalar;          # scalar: sets value to undef, frees any PV␣
↪buffer
undef @array;           # array: frees all elements AND the AV's␣
↪storage
undef %hash;            # hash:  frees all entries AND the HV's␣
↪buckets
undef &sub;             # sub:   frees the CV body; the name still␣
↪exists
undef *glob;            # glob:  destroys $glob, @glob, %glob, &
↪glob, etc.
```

The distinction matters for memory. `@array = ()` leaves the allocated capacity intact so the next push is cheap; `undef @array` releases it. For a large, one-shot array that you want to truly let go of, use `undef`. For a working array you will refill, use `= ()`.

Examples

Returning "no value" from a subroutine. In scalar context this is a clean way to signal failure; in list context `undef` alone produces a one-element list containing undef, which is truthy:

```
sub find_user {
    my ($id) = @_;
```

(continues on next page)

(continued from previous page)

```
    return undef unless $id;        # scalar-context caller: false
    ...
}
```

For subroutines meant to signal failure in both contexts, return an empty list instead - `return` with no argument does the right thing:

```
sub find_user {
    my ($id) = @_;
    return unless $id;              # () in list ctx, undef in␣
↪scalar ctx
    ...
}
```

Skipping positional values in a list assignment. `undef` on the **left-hand side** of a list assignment is a placeholder that throws away the matching value:

```
my ($x, undef, $z) = foo();         # discard the middle value
my (undef, @rest)  = @_;            # drop the first arg, keep the␣
↪rest
```

Classic `select` idiom for a sub-second sleep - three ignored filehandle sets plus a timeout:

```
select undef, undef, undef, 0.25;   # sleep 250 ms
```

Dual-context return:

```
return wantarray ? (undef, $errmsg) : undef if $they_blew_it;
```

Releasing a large working set:

```
my %cache = build_huge_cache();
use_cache(\%cache);
undef %cache;                       # free buckets and entries now
```

Undefining a subroutine. The name in the symbol table stays, but calling it will complain about an undefined subroutine:

```
sub greet { print "hi\n" }
undef &greet;
greet();                            # Undefined subroutine &
↪main::greet
```

Undefining a typeglob nukes *every* slot at once:

```
our $xyz = 1;
our @xyz = (1, 2);
our %xyz = (a => 1);
sub xyz { 42 }
```

(continues on next page)

(continued from previous page)

```
undef *xyz;                             # $xyz, @xyz, %xyz, &xyz all␣
↪gone
```

Edge cases

- **`undef $hash{$key}` is not `delete`.** It sets the value at that key to undef; the key itself still exists and `exists` still returns true. Use `delete $hash{$key}` to remove the key. The same applies to array elements: `undef $arr[3]` leaves index 3 in place with value undef, it does not shorten the array.
- **On tied variables and DBM values**, `undef $tied{$key}` invokes the tie's `STORE`, not its `DELETE`. The perldoc warning ("probably not do what you expect on most predefined variables or DBM list values") is specifically about this.
- **`undef` is a unary operator, not a list operator.** It takes exactly one argument, so `undef $a, $b` parses as `(undef $a), $b` - only `$a` is undefined. To undefine several variables, call it several times, or use a loop:

  ```
  undef $_ for $a, $b, $c;
  ```

- **`undef EXPR` requires an lvalue.** `undef 42` or `undef func()` is a compile-time error (*Can't modify ... in undef operator*).
- **Left-hand `undef` in list assignment is syntax, not a value.** `(undef, $x) = @pair` is parsed specially; you cannot write `my $slot = undef; ($slot, $x) = @pair` and expect the first value to be discarded - that assigns into `$slot`.
- **Constant `undef` in boolean context is false**, in numeric context is `0`, in string context is `""`. Under `use warnings` each of those conversions emits a `Use of uninitialized value` warning at the point of use, not at the point the variable became undef.
- **`undef &sub` does not remove the subroutine name** from its package stash - only the code. To remove the name as well, undefine the whole glob slot: `delete $Package::{sub_name}` or `undef *Package::sub_name` if you also want the sibling slots gone.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `defined` - the predicate that asks "is this value not undef?"; the natural counterpart to the `undef` value
- `delete` - remove a hash key or array element entirely; use this, not `undef`, when you want `exists` to become false
- `exists` - check whether a hash key or array index is present, regardless of whether its value is defined

- `local` - temporarily replace a package variable's value (often with `undef`) for the duration of a scope
- `wantarray` - pairs with `undef` in the idiomatic `wantarray ? (undef, $err) : undef` dual-context failure return

Filehandles, files, directories

unlink

Remove one or more directory entries that name files.

`unlink` asks the kernel to drop the link between each name in `LIST` and the file it refers to. If that link was the last one and no process still holds the file open, the file's storage is reclaimed; otherwise the file remains on disk, nameless but accessible through any open descriptor, until the last handle is closed. `unlink` operates on **directory entries**, not on file contents.

Synopsis

```
unlink LIST
unlink
```

What you get back

The number of files `unlink` actually removed, as an integer. On total success that equals `scalar @LIST`; on partial or total failure it is the count of names that were removed before and after the ones that failed. Zero is a valid "false" return when nothing could be removed. `$!` is set to the error from the **last** failing call - it does **not** tell you which name failed.

```
my $removed = unlink 'a', 'b', 'c';
unlink @goners;
unlink glob "*.bak";
```

If `LIST` is omitted, `unlink` operates on the single filename in `$_`.

Global state it touches

- `$_` - used as the operand when `unlink` is called with no arguments.
- `$!` - set to `errno` from the last `unlink(2)` that failed. Because it reflects only the final failure in the list, it is not a reliable way to diagnose which name could not be removed.

Finding out which file failed

`unlink` returns a count, not a per-file status, and overwrites `$!` on every iteration. To attribute errors to individual names, call `unlink` once per file:

```
foreach my $file (@goners) {
    unlink $file
        or warn "could not unlink $file: $!";
}
```

The per-file form also lets you keep going after a failure, which the list form does implicitly but silently.

Directories

unlink does **not** remove directories. On Linux it returns `0` and sets `$!` to `EISDIR` when asked to. The perl binary recognises a `-U` command-line switch that, combined with running as superuser, lifts the check - but even then unlinking a directory bypasses the filesystem's directory-integrity guarantees and can corrupt the parent directory's link count. Use `rmdir` instead; it calls `rmdir(2)`, which is the correct syscall for removing an empty directory entry.

```
rmdir $dir  or die "rmdir $dir: $!";    # correct
unlink $dir;                            # fails with EISDIR
```

POSIX deletion semantics

unlink removes the **name**, not the file. As long as any process holds the file open, the file's data and inode remain valid and that process can keep reading and writing through its descriptor. Storage is reclaimed only when the last name is gone **and** the last descriptor is closed. This is the idiomatic pattern for creating a temporary file that cleans itself up even on a crash:

```
open my $fh, "+>", $path or die "open $path: $!";
unlink $path or die "unlink $path: $!";
# $fh is still usable; $path is gone from the directory.
# When $fh closes or the process exits, the storage is freed.
```

It is also why a long-running program can keep writing to a log file whose name has been unlinked by an operator - the writes succeed, but nothing on disk is visible under the old name. `df` will report the space as used until the handle closes.

Symbolic links

If a name in `LIST` is a symbolic link, unlink removes the **link itself**, not the file it points at. The target is left untouched even if the link is dangling or points to a directory. This matches the `unlink(2)` syscall and is almost always what you want; use `readlink` first if you specifically need to act on the target.

Examples

Remove a single file, check the result:

```
unlink $path
    or die "cannot remove $path: $!";
```

Remove several and report the tally:

```
my @files  = glob "*.tmp";
my $gone   = unlink @files;
warn "removed $gone of ", scalar @files, " files\n"
    if $gone != @files;
```

Per-file diagnostics (the only reliable way to know which name failed):

```
my @failed;
for my $f (@goners) {
    unlink $f or push @failed, "$f: $!";
}
die "unlink failures:\n", join("\n", @failed), "\n" if @failed;
```

Rely on `$_` in a chain of file tests:

```
for (glob "*.bak") {
    next unless -f && -M _ > 30;    # older than 30 days
    unlink or warn "skip $_: $!";
}
```

Self-cleaning scratch file (POSIX unlink-while-open idiom):

```
open my $scratch, "+>", "/tmp/work.$$" or die $!;
unlink "/tmp/work.$$"                  or die $!;
# write through $scratch, no cleanup needed on exit
```

Edge cases

- **Empty LIST**: `unlink ()` returns `0` and does not touch `$!`. Passing an empty array behaves the same way.
- **Non-existent name**: returns `0` for that entry and sets `$!` to `ENOENT`. The rest of the list is still processed.
- **Permission denied**: `unlink(2)` checks write permission on the **containing directory**, not on the file itself. Removing a file you cannot write is allowed as long as you can write its parent directory; removing a file you own in a directory you cannot write is not. `$!` will be `EACCES` or `EPERM`.
- **Sticky directories** (`/tmp`, mode 1777): non-root users may only unlink names they own. Failures surface as `EPERM`.
- **Return value in boolean context**: `unlink @files or die` fires only when **every** name failed (count is `0`). To die on **any** failure, compare against `scalar @files` or use the per-file loop.
- **Busy file on Linux**: Linux does not prevent `unlink` of a file that is mmap'ed, executing, or open - the name is removed and the POSIX semantics above apply. Other operating systems may return `ETXTBSY` or `EBUSY`; this page covers Linux.
- **Taint mode**: names coming from tainted data are allowed as operands to `unlink`; see `perlsec` for the usual caveats about accepting untrusted paths.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `rmdir` - remove an **empty** directory; the correct tool when the name refers to a directory entry of type `DIR`
- `rename` - move or re-link a name without dropping it, often combined with `unlink` when replacing a file atomically
- `link` - create an additional hard link to a file; `unlink` is its inverse for one name at a time
- `symlink` - create a symbolic link; note that `unlink` removes the symlink itself, not its target
- `truncate` - shrink a file in place without unlinking it when you want to reuse the name

Lists · Fixed-length data

unpack

Extract typed values from a binary or fixed-width string according to a template.

`unpack` is the inverse of `pack`. It walks `EXPR` left to right, consuming the bytes described by each directive in `TEMPLATE` and turning them into Perl values. The result is a list - one value per directive, or per repetition when a directive has a count. If `EXPR` is omitted, `unpack` reads from `$_`.

This page is a **directive reference**. For a narrative introduction with worked protocol and file-format examples, start at the *pack/unpack tutorial*.

Synopsis

```
unpack TEMPLATE, EXPR
unpack TEMPLATE                                  # reads from $_
my @fields = unpack "A10 A10 A*", $line;
my ($ver, $len, $payload) = unpack "n N A*", $msg;
```

What you get back

A list of values, one per directive (or per repetition when a directive has a count). In **scalar context**, only the **first** value produced is returned:

```
my @all   = unpack "A4 A4 A*", $rec;    # three values
my $first = unpack "A4 A4 A*", $rec;    # just the first - not a
↪count
```

Each directive determines the Perl type of its result - see the “Result type” column in the directive table below.

Global state it touches

`unpack TEMPLATE` with no `EXPR` reads `$_`. No other interpreter globals are consulted. The `C0` / `U0` directives switch template byte-vs-character interpretation locally within the template but do not mutate any outside state.

Template syntax

Identical to `pack`: a sequence of directive letters, optionally followed by repeat counts `N` / `*` / `[…]` and modifiers `!` / `<` / `>`. Parentheses form groups; `#` introduces a template comment. See `pack` for the full syntax rules - this page covers only the unpack-specific semantics.

Directive table

Every unpack directive in one place. **W** is the width in bytes consumed from the input per scalar produced.

Directive	W	Signed	Endian	Result type	Modifiers	Notes
a	count	-	-	string	-	Returns bytes untouched
A	count	-	-	string	-	Strips trailing whitespace and NUL
Z	count	-	-	string	-	Returns everything up to the first NUL
b	count/8	-	-	bit string	-	“0”/”1” characters, LSB of each byte first
B	count/8	-	-	bit string	-	“0”/”1” characters, MSB of each byte first
h	count/2	-	-	hex string	-	`[0-9a-f]` characters, low nybble first

continues on next page

Table 2 – continued from previous page

Directive	W	Signed	Endian	Result type	Modifiers	Notes
H	count/2	-	-	hex string	-	[0-9a-f] characters, high nybble first
c	1	yes	-	integer	-	Signed char
C	1	no	-	integer	-	Unsigned char
W	1	no	-	integer	-	Unsigned char; yields codepoint in U0 mode
s	2	yes	native	integer	! < >	s! = native short
S	2	no	native	integer	! < >	S! = native unsigned short
l	4	yes	native	integer	! < >	l! = native long
L	4	no	native	integer	! < >	L! = native unsigned long
i	native	yes	native	integer	! < >	sizeof(int)
I	native	no	native	integer	! < >	sizeof(unsigned int)
q	8	yes	native	integer	< >	Requires 64-bit-integer Perl
Q	8	no	native	integer	< >	Requires 64-bit-integer Perl
n	2	no	big	integer	!	Network order; n! = signed
N	4	no	big	integer	!	Network order; N! = signed
v	2	no	little	integer	!	"VAX" order; v! = signed

continues on next page

Table 2 – continued from previous page

Directive	W	Signed	Endian	Result type	Modifiers	Notes
V	4	no	little	integer	!	"VAX" order; V! = signed
j	IV size	yes	native	integer	< >	Perl-internal IV
J	UV size	no	native	integer	< >	Perl-internal UV
f	4	-	native	number	< >	Single-precision IEEE 754
d	8	-	native	number	< >	Double-precision IEEE 754
F	NV size	-	native	number	< >	Perl-internal float
D	varies	-	native	number	< >	Long double
p	ptr size	-	native	string	< >	Dereferences pointer to NUL-terminated string
P	ptr size	-	native	string	< >	Dereferences pointer; count = bytes to read
u	varies	-	-	string	-	Uudecoded bytes
U	varies	-	-	integer	-	Unicode codepoint number
w	varies	no	-	integer	-	BER-compressed integer
x	1	-	-	(none)	!	Skip one byte forward; x!N aligns forward
X	−1	-	-	(none)	!	Back up one byte; X!N aligns backward

continues on next page

Table 2 – continued from previous page

Directive	W	Signed	Endian	Result type	Modifiers	Notes
`@`	absolute	-	-	(none)	!	Jump to position N within the innermost group
`.`	-	-	-	integer	!	Return current position (relative to start of group / string)
`( … )`	-	-	-	-	`< > !`	Group: repeat count and endianness propagate inside
`/`	-	-	-	-	-	See “Length-prefixed payloads” below
`%N`	varies	-	-	integer	-	**Prefix.** N-bit checksum of the following directive’s values

The modifiers have the same meaning as in `pack`: `!` selects native sizes, alignment semantics, signedness for `n` / `N` / `v` / `V`, or byte offsets for `@` / `.`; `<` / `>` force endianness.

Differences from pack

- **`a` strips nothing** (returns bytes raw); **`A` strips trailing whitespace and NUL**; **`Z` stops at the first NUL.** With `pack` all three pad; with `unpack` all three un-pad differently.
- **`x` skips forward by W, `X` backs up by W.** `X` before the start of the string is a fatal error.
- **`/` reads a count from the data, then applies it to the next directive**:

```
unpack("W/a", "\004Gurusamy")          # ("Guru")
unpack("a3/A A*", "007 Bond  J ")      # (" Bond", "J")
unpack("n/a*", "\x00\x0chello, world") # ("hello, world")
```

With `pack` the form is *length-item*`/`*item* and the length is computed from the value's length. With `unpack` the form is `/`*item* (no length-item before) - the preceding integer directive supplies the count.

- **`%N` is unpack-only.** It replaces the following directive's normal output with the `N`-bit sum of the values that directive would otherwise have produced. `%32W*` is the System V `sum` checksum. `%32b*` counts set bits.
- **`.` returns the current byte position** rather than zero-filling to one. Useful after `x` / `X` navigation to learn where you are.
- **`p` and `P` dereference pointers read from the input** - almost always unsafe unless you know the input was produced by `pack "p"` / `pack "P"` in the same process.
- **Under-run / over-run behaviour:** if the template asks for more data than `EXPR` contains, the result is not well-defined (may yield empty strings, zeros, a decreased repeat count, or an exception). If `EXPR` is longer than the template consumes, trailing bytes are silently ignored.

Examples

Parse a minimal binary protocol header - a 16-bit version, a 32-bit length, then a variable-length payload, all network byte order:

```
my ($ver, $len, $payload) = unpack "n N A*", $msg;
```

Split a fixed-width text record - faster and clearer than chained `substr` calls:

```
my ($date, $desc, $amount) = unpack "A10 A27 A*", $line;
# "2026-04-22 coffee at the station           3.50"
# $date = "2026-04-22", $desc = "coffee at the station",
# $amount = "3.50"
```

Extract a System V-style checksum with the `%` prefix - the 32-bit sum of every byte, masked to 16 bits:

```
my $checksum = do {
    local $/;                              # slurp
    unpack "%32W*", readline $fh;
} % 65535;
```

Count set bits in a bit vector - `%32b*` reads the whole mask as bits and returns their sum:

```
my $setbits = unpack "%32b*", $selectmask;
```

Read a sequence of little-endian 32-bit integers off a raw buffer:

```
my @ints = unpack "V*", $buf;
```

Use `x` to skip filler bytes, `X` to back up after peeking:

```
# 2-byte tag, skip 2 reserved bytes, then 4 32-bit big-endian values
my ($tag, @vals) = unpack "n x2 N4", $frame;
```

Edge cases

- **Scalar context returns only the first value.** Assigning `unpack` to a scalar is almost never what you want. To count fields, assign to a list first, or use `() = unpack ...` in list context:

  ```
  my $n = () = unpack "A10 A10 A*", $line;   # 3
  ```

- **`*` is greedy, not a placeholder.** `A*` consumes all remaining bytes as one string; there is no backtracking. A `*` directive must be the last one in its sequence, or nothing after it runs.
- **Templates are not regular expressions.** Whitespace is ignored, but there is no alternation, no `|`, no lookahead. If data shape depends on earlier fields, unpack in stages: decode a header, then call `unpack` again on the payload using a template derived from what you just read.
- **Byte vs. character semantics.** `unpack` runs in byte mode by default on a byte string and in character mode on a string with the UTF-8 flag set. `C0` forces byte mode from that point; `U0` forces UTF-8 mode. Mixing a UTF-8-flagged string with numeric directives like `N` reads codepoints, not bytes - usually not what you want. Call `utf8::encode` first, or operate on byte-level data.
- **`X` past the start of the string is fatal.**
- **`p` / `P` on an arbitrary input is undefined behaviour.** Never use them on data you did not pack yourself in the same process.
- **No EXPR argument**: `unpack TEMPLATE` reads `$_`. Useful inside `while (<$fh>) { ... }` loops over fixed-width records.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `pack` - the inverse operation; same template language
- `substr` - extract a single fixed slice; simpler for one field, slower for many
- `sprintf` - round-trip between binary and human-readable
- `read` - read raw bytes off a filehandle into a buffer for `unpack`
- `vec` - indexed access to a bit vector without a template
- *pack/unpack tutorial* - task-oriented walkthrough with worked protocol and file-format examples

Arrays

unshift

Prepend one or more values to the start of an array and return the array's new length.

`unshift` is the mirror of `push`: where `push` appends at the tail, `unshift` inserts at the head. The values in `LIST` are prepended as a group, in the order given, so the first argument of `LIST` becomes the new element 0, the second becomes element 1, and so on, with the original contents sliding to higher indices.

Synopsis

```
unshift @arr, $val
unshift @arr, @more
unshift @arr, $first, $second, $third
```

What you get back

The new number of elements in the array, as an integer. This is the same value you would get from `scalar @arr` after the call.

```
my @colors = ("red");
my $n = unshift @colors, "blue", "green";   # $n == 3
```

Most calls discard the return value; it is occasionally useful when you want to prepend and branch on the resulting size in one step.

Examples

Prepend a single element:

```
my @animals = ("cat");
unshift @animals, "mouse";
# @animals is now ("mouse", "cat")
```

Prepend several elements - **the order is preserved**, not reversed. This is the single most common `unshift` confusion:

```
my @a = (10, 20, 30);
unshift @a, 1, 2, 3;
# @a is now (1, 2, 3, 10, 20, 30)
# NOT (3, 2, 1, 10, 20, 30)
```

If you genuinely want the prepended values reversed, reverse them explicitly:

```
my @a = (10, 20, 30);
unshift @a, reverse(1, 2, 3);
# @a is now (3, 2, 1, 10, 20, 30)
```

Prepend a whole array - the elements are spliced in, not the array as a single value:

```
my @head = ("a", "b");
my @tail = ("c", "d");
unshift @tail, @head;
# @tail is now ("a", "b", "c", "d")
```

Insert at the front vs. insert at the rear. These two calls produce mirror-image results on the same starting array:

```
my @x = (2, 3);
push    @x, 4;      # @x is (2, 3, 4)   - 4 goes to the rear
unshift @x, 1;      # @x is (1, 2, 3, 4) - 1 goes to the front
```

Work on an array through a reference. `unshift` accepts a dereferenced array on the left:

```
my $ref = [10, 20, 30];
unshift @$ref, 0;
# $$ref[0] is now 0, @$ref is (0, 10, 20, 30)
```

The arrow-deref form works the same way; write the deref as its own expression rather than stretching a method chain across the `unshift` argument:

```
my @list = @$ref;
unshift @list, -1;
```

Edge cases

- **Cost is O(n).** Every existing element is shifted up by `scalar(LIST)` slots, so prepending to a large array is proportionally expensive. If you are building a list by repeated prepending, build it with `push` and `reverse` at the end - or build it backwards with indices - instead of `unshift`ing in a loop.
- **List is prepended whole, not reversed.** `unshift @a, 1, 2, 3` places 1 at index 0, 2 at index 1, 3 at index 2. Readers coming from languages where "push to front" naturally reverses (because each element becomes the new head) often expect the opposite; Perl does not.
- **Empty `LIST` is a no-op.** `unshift @arr` with nothing after it leaves the array unchanged and returns the current element count.
- **Tied arrays**: `unshift` on a tied array calls the `UNSHIFT` handler if the package provides one; otherwise it falls back to a sequence of `SPLICE` or `STORE` / `STORESIZE` calls. A tie class that only implements `FETCH` and `STORE` can still be `unshift`ed, but every call rewrites every index.
- **Dereferenced array works**: `unshift @$ref, $x` modifies the array referenced by `$ref` in place. `unshift @{ $h{key} }, $x` autovivifies `$h{key}` to an empty array reference if it did not previously exist.
- **Scalar expression on the left is a syntax error.** A Perl 5.14 experiment briefly accepted `unshift $scalar, ...` to mean "treat `$scalar` as an array reference and prepend"; it was removed in Perl 5.24 and stays removed. Write `unshift @$scalar, ...`.

- **Return value in scalar vs. list context** is the same integer. Unlike many builtins, `unshift` does not behave differently in list context - it always returns the new length.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `push` - the other end: append to the tail of the array and return the new length
- `shift` - the inverse operation: remove and return the first element
- `pop` - symmetric partner to `shift`, removes from the tail
- `splice` - general-purpose insert/remove at an arbitrary position; `unshift @a, LIST` is equivalent to `splice(@a, 0, 0, LIST)`
- `reverse` - pair with `unshift` when you do want the prepended values to end up in reverse order
- `List::Util` - no dedicated `prepend`; for head-insert semantics `unshift` is the idiomatic tool, and `List::Util::pairs` / `List::Util::zip` compose with it when you are restructuring list shapes rather than just adding to one

Classes and OO

untie

Break the binding between a variable and its tied class.

`untie` reverses a prior `tie`: after the call, `VARIABLE` goes back to being a plain scalar, array, hash, or filehandle, and subsequent reads and writes touch the raw storage rather than being routed through the tied package's `FETCH`, `STORE`, and friends. If the variable is not tied, `untie` is a no-op.

Synopsis

```
untie VARIABLE
```

What you get back

Returns a true value if the variable was tied and the binding was removed, false otherwise. The return value is rarely checked - call sites know whether they are unwinding a tie they set up themselves.

As a side effect, if the tied package defines an `UNTIE` method, it is invoked on the tied object before the binding is dropped. `UNTIE` receives the object and a count of any outstanding inner references held to it (see *Edge cases* below).

What it does

1. Locate the magic on `VARIABLE` that a prior `tie` installed.
2. If the tied package provides `UNTIE`, call it on the object.
3. Remove the magic so the variable reverts to ordinary storage.
4. Release the interpreter's reference to the tied object. The object is destroyed (triggering `DESTROY` if defined) once no other references to it remain.

After `untie`, any value `tied` previously returned for this variable no longer corresponds to a live binding.

Examples

Tie a hash, use it, then release it:

```
use Tie::File;
tie my @lines, 'Tie::File', 'log.txt' or die $!;
push @lines, "new entry";
untie @lines;                       # flush and release
```

A round-trip through `tied` - grab the underlying object so you can call a non-tied method on it, then drop the binding:

```
my $obj = tied %cache;
$obj->flush;
untie %cache;
```

`untie` on a variable that was never tied is harmless:

```
my %h;
untie %h;                           # no-op, no warning
```

An `UNTIE` hook that runs before the binding goes away:

```
package My::Tied;
sub TIEHASH { bless {}, shift }
sub UNTIE   { my ($self, $count) = @_;
              warn "untie with $count outstanding refs" if $count }
# ...
tie   my %h, 'My::Tied';
untie %h;                           # calls My::Tied::UNTIE($obj, 0)
```

Edge cases

- **Outstanding inner references**: if code stashed the object returned by `tie` or `tied` somewhere that still holds it at the moment `untie` runs, Perl issues

  ```
  untie attempted while N inner references still exist
  ```

under `use warnings`. The binding is removed anyway, but those stashed references continue to point at the now-detached object. Drop those references first, or accept that they will now see the un-tied object in isolation.

- **Calling `untie` on a non-tied variable** is a no-op. No warning, no error, no method call.
- **Filehandle tie**: `untie *FH` removes the tie installed by `tie *FH, $class, ...`. The underlying glob and any real file descriptor behind it are unaffected - `untie` does not `close`.
- **`UNTIE` is optional**: if the tied package does not define one, the binding is simply removed. Do not rely on `UNTIE` running; many tied classes omit it.
- **Destruction order**: `untie` does not itself call `DESTROY`. The object is destroyed when its refcount hits zero, which may be immediately after `untie` (if nothing else holds it) or later (if the caller kept a copy from `tied`).
- **Localised tied variables**: a tie installed inside a scope on a `local`ised variable is undone automatically when the scope exits; you do not need an explicit `untie`, and calling one is harmless.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `tie` - install the binding that `untie` removes; shows the full set of tie-hook method names
- `tied` - fetch the object currently bound to a tied variable, typically just before calling `untie` on it
- `bless` - `tie` blesses the object it is given; `untie` does not unbless, it only detaches the variable from it
- `ref` - inspect what kind of reference `tied` returned before deciding to `untie`
- `local` - scope-bounded alternative when you want the tie to go away on block exit without an explicit `untie`

Scoping · Modules · Classes and OO

use

Load a module at compile time and import its symbols into the current package.

`use` is the everyday mechanism for pulling in library code. It runs at **compile time**, not at run time: the named module is located on `@INC`, loaded (if not already in `%INC`), and given a chance to alter the caller's namespace - typically by aliasing subroutine or variable names into it. The same syntax also drives **pragmas** (compiler directives like `strict` or `warnings`); the only difference is convention - pragmas are lowercase and usually have lexical, not package-wide, effects.

A `use` statement is defined to be exactly equivalent to:

```
BEGIN { require Module; Module->import(LIST); }
```

The `BEGIN` forces both the load and the import to happen while the surrounding file is still being parsed, so the imported names are visible to the compiler for the rest of the file.

Synopsis

```
use Module VERSION LIST
use Module VERSION
use Module LIST
use Module
use Module ()
use VERSION
```

What you get back

`use` is a declaration, not an expression - it has no useful return value and cannot appear on the right-hand side of anything. Its effect is a **side effect on the current compilation unit**: package `%INC` gets an entry, the importing package gets whatever symbols `import` aliased in, and - for `use VERSION` - the current lexical scope gets a feature bundle turned on.

If the module cannot be loaded, or if `import` dies, the `BEGIN` block propagates the exception and compilation aborts.

The four module forms

use Module

Loads the module and calls its `import` method with **no arguments**. The module decides what "no arguments" means; for `Exporter`-based modules it typically means "export your `@EXPORT` defaults":

```
use File::Spec;                 # calls File::Spec->import()
```

use Module LIST

Loads the module and calls `import` with `LIST`:

```
use List::Util qw(first sum max);
```

This is the common "import exactly these names" form. `LIST` is passed verbatim to `import`; its meaning is the module's business, not the language's.

use Module ()

Empty parentheses disable the import call entirely. This is distinct from omitting the list - no `import` method is invoked at all:

```
use POSIX ();                  # load, but do not import anything
POSIX::strftime(...);          # call by full name instead
```

Equivalent to:

```
BEGIN { require POSIX }
```

Use this when you want the module loaded but do not want your namespace altered - typically because you will call everything by fully qualified name, or because the module's default exports would clobber something you care about.

use Module VERSION / use Module VERSION LIST

When a VERSION appears immediately after `Module` (no comma), `use` calls the module's `VERSION` method with that version as an argument **before** calling `import`:

```
use List::Util 1.45 qw(uniq);
```

is equivalent to:

```
BEGIN {
    require List::Util;
    List::Util->VERSION(1.45);
    List::Util->import(qw(uniq));
}
```

The default `VERSION` method, inherited from `UNIVERSAL`, croaks if the requested version is greater than `$Module::VERSION`.

VERSION must look like a version literal - a digit, or `v` followed by a digit. Arbitrary expressions in that position are parsed as the start of `LIST`, not as a version. Note there is **no comma after VERSION**.

use VERSION - request a Perl language level

`use VERSION` (with no module name) declares the minimum Perl version the file requires, and **lexically enables the matching feature bundle**:

```
use v5.36;                     # v-string form - preferred
use 5.036;                     # numeric form - equivalent
use 5.036_000;                 # older numeric form
```

An exception is raised at compile time if the running Perl is older than VERSION; the rest of the file is not even parsed.

Beyond the version check, `use VERSION` turns on the features bundled with that release:

- **`use v5.12`** or higher - implicit `use strict`.

- **`use v5.35`** or higher - implicit `use warnings`.
- **`use v5.36`** - the common modern baseline: adds `say`, `signatures`, `isa`, and the earlier bundles' features.
- **`use v5.39`** or higher - lexically imports `builtin` functions for the matching bundle.
- **`use v5.41`** or higher - enables `source::encoding 'ascii'`.

`use VERSION` does **not** load `feature.pm`, `strict.pm`, `warnings.pm`, or `builtin.pm` - it implements the equivalent behaviour directly. Place it at the top of the file, before anything that depends on those features being active.

From Perl 5.39 onward, issuing a second `use VERSION` while one is already in effect is a fatal error. In older bundles it was merely deprecated.

Modules vs pragmas

Syntactically, pragmas and modules are identical - both are just packages whose `import` method you invoke via `use`. The conventional differences:

- **Naming.** Pragmas are lowercase (`strict`, `warnings`, `feature`, `integer`); modules are `Studly::Caps`.
- **Scope of effect.** Pragmas usually alter the **current lexical scope** (a block, not a package) and unwind at the closing brace. Modules usually alter the **current package** and persist until end-of-file.
- **`no`.** The counterpart `no` statement calls `unimport` instead of `import`. Pragmas typically implement `unimport` to undo their effect; most ordinary modules do not.

```
use strict;                     # lexical - affects this block /␣
↪file
use warnings;                   # lexical
{
    no warnings 'uninitialized';
    # warnings off only in this block
}

use List::Util qw(sum);         # package-scoped - affects main::
```

Conditional loading

Because `use` runs at compile time, it **ignores ordinary runtime control flow**. A `use` inside the false branch of an `if` is still executed:

```
if (0) {
    use Heavy::Module;          # loaded anyway - compile time!
}
```

For genuinely conditional loading, either use `require` at run time, or delegate to the `if` pragma:

```
use if $] >= 5.036, 'experimental', 'builtin';
use if $^O eq 'MSWin32', 'Win32::API';
```

Examples

Load a module with its default exports:

```
use Carp;                           # croak, carp, confess visible here
```

Import a specific subset:

```
use List::Util qw(first min max sum);
```

Load without importing - call by full name:

```
use Scalar::Util ();
my $ok = Scalar::Util::looks_like_number($x);
```

Require a minimum module version:

```
use Moo 2.000000;
```

Declare the Perl language level and pick up its feature bundle:

```
use v5.36;                          # strict, warnings, say, signatures.
 ↪..
sub greet ($name) {
    say "hello, $name";
}
```

Turn a pragma on for a file, then off for one block:

```
use warnings;

sub legacy_parser {
    no warnings 'uninitialized';
    return join ",", @_;            # undefs render as "" without␣
 ↪warnings
}
```

Conditional pragma via the `if` helper:

```
use if $] < 5.010, 'MRO::Compat';
```

Edge cases

- **`Module` must be a bareword.** `use $name;` is a syntax error - `use` does not accept a runtime expression for the module name. Use `require` for that.
- **No comma between `Module` and `VERSION`.** `use List::Util 1.45` is a version check; `use List::Util, 1.45` is a syntax error.

- **The version literal rule is strict.** Anything that does not start with a digit or with v followed by a digit is parsed as the beginning of LIST, not as a version. `use Module "1.45"` passes the string `"1.45"` to `import`; it does **not** check a version.
- **Empty LIST vs omitted LIST.** `use Module;` calls `Module->import()`. `use Module ();` does not call `import` at all. The distinction matters for modules whose `import` has side effects even with no arguments.
- **Missing `import` method.** If the module does not define (and does not inherit) an `import` method, the call is silently skipped - no error, even if AUTOLOAD is defined.
- **`use` inside `eval`.** BEGIN runs at compile time of the `eval`'d code, so `eval "use Some::Module"` delays loading until the `eval` runs. This is the common way to probe for an optional module at run time:

```
eval { require Some::Module; Some::Module->import };
my $have_it = !$@;
```

- **Ordering of `use VERSION` and other pragmas.** Place `use VERSION` first, so the feature bundle is in effect before later pragmas (which may depend on it) run. A later `no strict 'refs'` still overrides the strictures implied by `use v5.12+`, but relying on that is discouraged.
- **Pragmas that look like modules.** Some pragmas ship with a capitalised name (UNIVERSAL, Internals); some modules ship lowercase (if, lib, ok). The naming convention is a hint, not a rule - what matters is what the module's `import` does.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `require` - the run-time half of `use`; loads a module without invoking `import`, and accepts a runtime expression for the module name
- `no` - the counterpart declaration; calls `unimport` instead of `import` to undo a pragma's effect
- `import` - not a built-in; an ordinary class method invoked by `use`, usually provided via inheritance from `Exporter`
- `package` - declares the package whose namespace `use` will alter
- `@INC` - the list of directories searched for the module file
- `%INC` - records every file loaded by `use` or `require`, keyed by relative path

Filehandles, files, directories

utime

Set access and modification times on a list of files.

`utime` applies the numeric access time ATIME and modification time MTIME - both Unix epoch seconds - to every file named in LIST. It is the Perl-level interface to the `utime(2)`

and `utimes(2)` system calls, and the tool behind the Unix `touch(1)` command. The inode change time (`ctime`) of each file is set to the current time as a side effect - this is a kernel invariant, not a Perl choice, and cannot be suppressed.

Synopsis

```
utime ATIME, MTIME, LIST
utime undef, undef, LIST      # set both to "now" (since 5.8.0)
```

What you get back

The number of files whose timestamps were successfully updated. Any file that could not be touched is silently skipped in the count; `$!` reflects the error from the *last* failing file only. Check the return value against `scalar @files` when you need to detect partial failure:

```
my @files = ("a", "b", "c");
my $n = utime $atime, $mtime, @files;
warn "only $n of ", scalar @files, " files updated: $!"
    if $n != @files;
```

Arguments

- `ATIME` - new access time, in seconds since the Unix epoch. Must be numeric; non-numeric scalars are coerced the usual way and a string like `"now"` becomes `0` (midnight 1970-01-01).
- `MTIME` - new modification time, same format.
- `LIST` - one or more filenames. On systems that support `futimes(2)` (Linux does), filehandles may appear in `LIST` as well. Filehandles must be passed as globs (`*FH`) or glob references (`\*FH`); a bareword in that position is parsed as a filename, not a handle.

The `undef, undef` form

Since Perl 5.8.0, passing `undef` for both `ATIME` and `MTIME` calls `utime(2)` with a null second argument. The kernel then sets both times to the current wall-clock time, and - crucially - this form succeeds on files the caller does not own, provided the caller has **write permission** on the file. The owner-or-root requirement only applies to the explicit-timestamp form.

```
for my $file (@ARGV) {
    utime undef, undef, $file
        or warn "Couldn't touch $file: $!";
}
```

This is the form the real `touch(1)` command uses internally, and is the right choice for a generic “touch” script.

Permissions

Two distinct rules, depending on which form you use:

Form	Who may succeed
`utime $atime, $mtime, ...`	File owner, or root
`utime undef, undef, ...`	Anyone with write permission on the file

A non-owner attempting the explicit-timestamp form gets `EPERM` in `$!` and the file is left untouched (it still counts as a failure, not a silent no-op).

Examples

Plain Unix-style `touch` on a list of files you own:

```
my $now = time;
utime $now, $now, @ARGV;
```

Generic `touch` that also works on files you merely have write access to:

```
for my $file (@ARGV) {
    utime undef, undef, $file
        or warn "Couldn't touch $file: $!";
}
```

Copy timestamps from one file to another using `stat`:

```
my @st = stat $src or die "stat $src: $!";
utime $st[8], $st[9], $dst
    or die "utime $dst: $!";
```

Touch an already-open filehandle on Linux (needs `futimes(2)`):

```
open my $fh, ">>", "log" or die $!;
utime undef, undef, *$fh;       # glob deref, not bareword
```

Set a file one hour into the past, useful for build-system tests:

```
my $t = time - 3600;
utime $t, $t, "stale.tmp";
```

Edge cases

- **Mixed `undef` and number**: passing `undef` for only one of the first two arguments does **not** trigger the "both-undef" path. The `undef` is coerced to `0` (epoch midnight 1970-01-01), the other value is used verbatim, and an `uninitialized` warning fires under `use warnings`. This is almost never what you want.

  ```
  use warnings;
  utime undef, time, $file;       # atime = 1970-01-01, warning
  ```

- **Non-numeric timestamps**: `ATIME` and `MTIME` must be numeric. String values are coerced - `"1700000000"` works; `"yesterday"` silently becomes `0`.
- **`ctime` always moves**: there is no way to preserve a file's inode change time across a `utime` call. If you need that, you need filesystem-level tools or a snapshot, not Perl.
- **Filehandle in `LIST`**: must be a glob (`*FH`) or glob reference (`\*FH`). A bareword like `FH` in that position is parsed as the filename `"FH"`, and a string scalar holding a handle name is likewise treated as a filename.
- **Non-existent file**: counts as a failure and does not increment the return value; `$!` is set to `ENOENT` (`No such file or directory`).
- **Partial failure**: the count reflects successes only. To know *which* file failed, loop over the list one at a time.
- **NFS clock skew**: the server's clock is authoritative, not the client's. A noticeable difference between the two shows up as timestamps that do not match what `time` on the client returned.
- **Sub-second precision**: Perl's `utime` operates on whole seconds. Filesystems and kernels that support nanosecond timestamps will see the fractional component zeroed. Reach for `Time::HiRes::utime` (XS) if you need better resolution.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `stat` - read the timestamps `utime` writes; elements 8 (`atime`), 9 (`mtime`), and 10 (`ctime`) of the returned list
- `time` - current epoch seconds, the usual source for both `ATIME` and `MTIME`
- `open` and `close` - filehandles that `utime` can touch on systems with `futimes(2)`
- `$!` - system error from the last failing file when the return count is less than `scalar @files`
- `utime(2)`, `utimes(2)`, `futimes(2)` - the underlying system calls

Arrays · Hashes

values

List every value of a hash (or every element of an array).

`values` is the value-side companion to `keys`. Given a hash, it returns all the values; given an array, it returns all the elements. The elements come back as **aliases** into the underlying container, so assigning to them from a loop modifies the container in place. As a side effect, calling `values` resets the iterator that `each`, `keys`, and `values` share for that hash or array.

Synopsis

```
my @v = values %hash;
my @v = values @array;
my $n = scalar values %hash;    # number of values
for (values %hash) { $_ *= 2 }  # mutate in place via aliasing
```

What you get back

In list context: the full list of values from a hash, or the full list of elements from an array. Array entries come back in index order (lowest index first). Hash entries come back in an implementation-defined, per-hash random order - see *Edge cases*.

In scalar context: the number of entries, identical to `scalar keys %h` for hashes and `scalar @a` for arrays. `values` in void context does nothing observable **except** reset the iterator - that is the idiomatic way to force-reset an iterator mid-traversal:

```
values %h;                          # void context: iterator reset only
```

The returned list is not a copy. Each element is an alias to the live value in the container. Aliasing is documented behaviour and is the basis for the in-place mutation idiom under *Examples*.

Global state it touches

`values` resets the per-container iterator state that `each` reads and advances. Any `while (my ($k, $v) = each %h) { ... }` loop in progress on the same hash will restart from the beginning the next time `each` is called. `keys` has the same effect. The iterator is a property of the container, not of any particular call site - so a caller in one sub can disturb an `each` loop running in another sub if they operate on the same hash.

Examples

Iterate a hash in key-sorted order, which `values` alone does not give you:

```
my %score = (alice => 90, bob => 75, carol => 82);
for my $k (sort keys %score) {
    print "$k: $score{$k}\n";
}
```

Sum every value of a hash:

```
use List::Util qw(sum0);
my %count = (apples => 3, pears => 5, plums => 2);
my $total = sum0 values %count;     # 10
```

Mutate every value in place through the alias:

```
my %prices = (bread => 3.00, milk => 2.50, eggs => 4.20);
$_ *= 1.10 for values %prices;      # apply 10% price rise
# %prices is now (bread => 3.30, milk => 2.75, eggs => 4.62)
```

The same idiom works on an array, though for arrays a plain `@array` loop is more conventional:

```
my @nums = (1, 2, 3, 4);
$_ **= 2 for values @nums;          # @nums is now (1, 4, 9, 16)
```

Force-reset an in-progress `each` iterator without consuming its output:

```
while (my ($k, $v) = each %h) {
    last if $v > $threshold;
}
values %h;                          # void context: drop iterator
 ↪state
# next each %h starts from the beginning
```

Count entries without building the list - scalar context never materialises the values:

```
if (scalar values %cache > 10_000) {
    prune(\%cache);
}
```

Edge cases

- **Aliasing is the feature and the footgun.** `for (values %h)` gives you aliases, not copies. A harmless-looking `s///` inside such a loop rewrites the hash. If you want copies, assign first: `my @v = values %h; for (@v) { ... }`.
- **Hash order is randomised.** Perl 5.18+ randomises the traversal order per-hash. Two hashes built from the same keys can yield values in different orders. Never rely on the order for anything that leaves the process - serialise via `sort keys` if you need stability.
- **Order is stable only while the hash is unmodified.** Insertions and deletions may permute the order. The one documented exception is deleting the key most recently returned by `each` or `keys`, which leaves the remaining order intact.
- **`keys`, `values`, `each` agree.** For a hash that has not been modified between calls, `keys %h` and `values %h` produce corresponding lists in the same order - `(keys %h)[$i]` and `(values %h)[$i]` name the same entry.
- **Array form behaves like `@array` plus a reset.** `values @a` in list context is equivalent to `@a`, with the side effect of resetting the array's iterator. For the iterator-reset alone, `keys @a` in void context is the conventional choice.
- **Empty hash or array** returns the empty list in list context and `0` in scalar context. No warning, no error.
- **Tied hashes.** Behaviour is delegated to the tie class's `FIRSTKEY` / `NEXTKEY` / `FETCH` methods. Order, aliasing, and iterator-reset semantics follow whatever the tie implementation provides - the pure-hash guarantees above do not automatically carry over.
- **Removed experiment: scalar argument.** Perl 5.14–5.22 experimentally accepted `values $ref`; that experiment was removed in 5.24. Dereference explicitly: `values %$ref`, `values @$ref`.

- **Array support requires 5.12+.** Running `values @a` on an older Perl raises a syntax error. Use `use v5.12;` at the top of a file that relies on array form to make the requirement explicit.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `keys` - the key-side companion; same ordering rules, same iterator, scalar context returns the count
- `each` - single-step iteration returning `($key, $value)` pairs; shares the iterator that `values` resets
- `exists` - test whether a key is present without triggering autovivification
- `delete` - remove a key/value pair; deleting the most-recently-returned key of `each` or `keys` leaves the remaining traversal order intact
- `map` - transform values into a derived list without mutating the hash (the non-aliasing alternative to `for (values %h)`)
- `sort` - pair with `keys` when you need a deterministic traversal order

Fixed-length data

vec

Read or write a fixed-width slot inside a string treated as a packed bit vector.

`vec` views `EXPR` as an array of unsigned integers, each `BITS` wide, packed back-to-back from the start of the string. `OFFSET` indexes into that array - not into bytes, not into bits, but into `BITS`-sized elements. Read form returns the integer at that slot; the lvalue form writes one. Widths are 1, 2, 4, 8, 16, 32, and on 64-bit builds 64.

Synopsis

```
my $n = vec($buf, $offset, $bits);
vec($buf, $offset, $bits) = $n;
```

What you get back

In rvalue context, an unsigned integer - the contents of the selected slot, zero-extended to a Perl number. In lvalue context, an assignable slot; assigning truncates the right-hand value to `BITS` and writes it into the string, extending the string with zero bytes if `OFFSET` lies past the current end.

The parentheses around `vec(...)` in the lvalue form are required - without them, `vec $buf, $o, $b = 3` parses the = as part of the argument list.

How the bits are laid out

The layout depends on `BITS`, and is chosen so code is portable across big- and little-endian machines:

- **`BITS == 8`**: each slot is one byte of the string. `vec($s, $i, 8)` is the unsigned value of `substr($s, $i, 1)`.
- **`BITS == 16, 32, 64`**: bytes of the string are grouped into chunks of `BITS/8` and interpreted in **big-endian** order - equivalent to `unpack` with `n`, `N`, or (on 64-bit builds) `Q>` / the moral equivalent. `vec($s, 0, 32)` reads the first four bytes as a big-endian `uint32`.
- **`BITS == 4, 2, 1`**: the string is broken into bytes, and each byte is split into `8/BITS` groups, numbered **little-endian-ish** within the byte. The bit values from low to high are `0x01`, `0x02`, `0x04`, `0x08`, `0x10`, `0x20`, `0x40`, `0x80`. So for `chr(0x36)` (`0b00110110`):
  - `BITS == 4` gives the two nibbles `(0x6, 0x3)`.
  - `BITS == 2` gives the four 2-bit groups `(0x2, 0x1, 0x3, 0x0)`.
  - `BITS == 1` gives the eight bits `(0, 1, 1, 0, 1, 1, 0, 0)`.

A slot entirely off the end of the string reads as `0`. Writing past the end grows the string with zero bytes to reach the slot. A negative `OFFSET` is a fatal error.

Global state it touches

None. `vec` operates purely on its arguments.

Examples

Read a byte at a given index:

```
my $s = "Perl";
print vec($s, 0, 8);            # 80   (== ord 'P')
print vec($s, 3, 8);            # 108  (== ord 'l')
```

Build a string by writing 32-bit big-endian words:

```
my $buf = '';
vec($buf, 0, 32) = 0x5065726C;  # "Perl"
vec($buf, 1, 32) = 0x50657270;  # "PerlPerp"
print $buf;                     # PerlPerp
```

Use `vec` as a compact boolean array - one bit per flag:

```
my $flags = '';
vec($flags, 17, 1) = 1;
vec($flags, 42, 1) = 1;
print vec($flags, 17, 1);       # 1
print vec($flags, 18, 1);       # 0  (slot never set, still zero)
print length $flags;            # 6  (string auto-extended to fit␣
 ↪bit 42)
```

Count the set bits in a bit vector without looping bit by bit - the idiomatic pattern uses `unpack`:

```
my $ones = unpack("%32b*", $flags);   # population count
```

Convert a bit vector into a string of `0`s and 1s for display:

```
my $bits = unpack("b*", $flags);      # "00...010...010..."
```

Combine two bit vectors with the bitwise string operators - those treat string operands as bit vectors of the same shape `vec` reads and writes:

```
my $union        = $flags_a | $flags_b;
my $intersection = $flags_a & $flags_b;
my $diff         = $flags_a ^ $flags_b;
```

Edge cases

- **Lvalue precedence**: `vec EXPR, O, B = N` is a syntax error. Always write `vec(EXPR, O, B) = N`.
- **Off-the-end read**: `vec($short, 1_000_000, 8)` returns `0`, never dies, never warns.
- **Off-the-end write**: the string is zero-padded up to the slot. For `BITS == 1`, writing bit 94 grows the string to 12 bytes.
- **Negative `OFFSET`**: fatal - `"Negative offset to vec in lvalue context"` or the rvalue equivalent.
- **`BITS` not a supported power of two**: fatal with `"Illegal number of bits in vec"`. Valid widths are 1, 2, 4, 8, 16, 32, and 64 on 64-bit builds.
- **UTF-8 encoded strings**: `vec` wants a byte string. If the scalar is flagged UTF-8, Perl first tries to downgrade it to a one-byte-per-character representation. If any character has a codepoint of 256 or higher, that fails fatally with `"Wide character in vec"`. Call `utf8::downgrade` deliberately, or pack the data with `pack "C*"` first, before reaching for `vec`.
- **Read on undef**: under `use warnings`, triggers an `uninitialized` warning on the string argument; returns `0`.
- **Assignment value wider than `BITS`**: the value is masked to the low `BITS` bits. `vec($s, 0, 4) = 0x1F` stores `0xF`.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `pack` - build multi-field binary structures; `vec` is the random-access counterpart when every field has the same width

- `unpack` - pull fields out of a binary string; use `unpack("b*", $v)` or `unpack("%32b*", $v)` to render or popcount a `vec` bit vector
- `substr` - byte-level random access when `BITS` would be `8` and you also want the lvalue to grow or shrink the string
- `sprintf` - format the integer a `vec` read returns, e.g. `sprintf "%08b", vec($s, $i, 8)`
- `ord` - one-shot equivalent of `vec($s, $i, 8)` when you only need the byte value and never assign back

Processes

wait

Block until any child process exits and reap it.

`wait` is the simplest front end to the underlying `wait(2)` system call. It suspends the current process until one of its children terminates, collects that child's exit status into `$?` and `${^CHILD_ERROR_NATIVE}`, and returns the reaped child's PID. If there are no children to wait on, it returns `-1` immediately. It is exactly equivalent to `waitpid(-1, 0)`.

Synopsis

```
my $pid = wait;
```

What you get back

The PID of the child that was reaped (a positive integer), or `-1` if there are no unreaped children left to wait on. The child's status goes into `$?`; the raw, OS-specific status word goes into `${^CHILD_ERROR_NATIVE}`.

To decode `$?` after a successful `wait`:

```
my $exit   = $? >> 8;       # normal exit code (0..255)
my $signal = $? & 0x7f;     # termination signal, if any
my $core   = $? & 0x80;     # core-dump flag
```

A `-1` return does **not** always mean "never had any children" - it can also mean "something already reaped them for me" (see below).

Global state it touches

- `$?` - set to the reaped child's 16-bit wait status (high byte: exit code, low byte: termination signal and core flag). Also readable as `$CHILD_ERROR` under `use English`.
- `${^CHILD_ERROR_NATIVE}` - the platform's raw status word, for callers that want to use POSIX macros (`WIFEXITED`, `WEXITSTATUS`, `WIFSIGNALED`, `WTERMSIG`) from `POSIX`.
- `%SIG` - if `$SIG{CHLD}` is set to `'IGNORE'`, or a handler in it calls `wait` itself, this call's result changes. See *Edge cases*.

Examples

Fork a worker, do something, then reap it:

```
my $pid = fork // die "fork: $!";
if ($pid == 0) {
    # child
    exec "/usr/bin/true" or die "exec: $!";
}
my $reaped = wait;              # blocks until the child exits
die "unexpected pid" if $reaped != $pid;
printf "child %d exited %d\n", $reaped, $? >> 8;
```

Reap every outstanding child in a loop. `wait` returns `-1` once the process has none left:

```
while ((my $kid = wait) > 0) {
    warn "reaped $kid, status " . ($? >> 8) . "\n";
}
```

Distinguish normal exit from signal termination:

```
my $pid = wait;
if ($pid > 0) {
    if    ($? == 0)       { say "clean exit" }
    elsif ($? & 0x7f)     { say "killed by signal " . ($? & 0x7f) }
    else                  { say "exit " . ($? >> 8) }
}
```

Decode the native status word with `POSIX` macros, which is more portable than manual bit-shifting:

```
use POSIX ":sys_wait_h";
my $pid = wait;
if ($pid > 0 && WIFEXITED(${^CHILD_ERROR_NATIVE})) {
    say "exit code: ", WEXITSTATUS(${^CHILD_ERROR_NATIVE});
}
```

Non-blocking reap of every pending zombie - use `waitpid` with `WNOHANG`, not `wait`:

```
use POSIX ":sys_wait_h";
1 while waitpid(-1, WNOHANG) > 0;
```

Edge cases

- **No children to wait on**: returns `-1` immediately. `wait` does not block in this case - it only blocks when at least one child is still alive and unreaped.
- **`$SIG{CHLD} = 'IGNORE'`**: on POSIX systems this asks the kernel to auto-reap children. Under this setting, `wait` almost always returns `-1` because the children are gone before user code can call it. `$?` is not set. If you want both fire-and-forget children **and** status reporting, use a real handler, not `'IGNORE'`.

- **Custom `$SIG{CHLD}` handler**: if the handler calls `wait` (or `waitpid`) itself, a subsequent `wait` in the main flow will find nothing left and return `-1`. Decide up-front which code path does the reaping.
- **Interaction with `qx//` and `system`**: both of these internally fork, exec, and wait on their own child. A `$SIG{CHLD}` handler that blindly calls `wait` can accidentally reap that internal child out from under `qx//` or `system`, leaving them unable to collect the status and typically returning `-1` with `$!` set to `ECHILD`. Handlers should loop with `waitpid(-1, WNOHANG)` and only act on PIDs they recognise.
- **`$?` is only written on a successful reap**. After a `-1` return, `$?` retains whatever it had before the call. Don't inspect it unless `wait` returned a positive PID.
- **Interrupted by a signal**: if a signal is delivered while `wait` is blocked and the handler returns normally, Perl restarts the wait - you do not see `EINTR` at the Perl level. If the handler `die`s or `exit`s, control leaves `wait` the usual way for that mechanism.
- **Threaded builds**: `wait` waits for children of the current process, not children of a specific thread. In a multi-threaded program the reaping thread is whichever one called `wait` first.
- **Return value is a PID, not a boolean**: `if (wait) { ... }` is true for both successful reaps and `-1`. Always compare to `0` or check `> 0` explicitly, as in the loop form above.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `waitpid` - wait for a specific child, or poll non-blockingly with `WNOHANG`; `wait` is the `waitpid(-1, 0)` form
- `fork` - create the children that `wait` reaps
- `exec` - what a forked child typically calls to become a different program
- `system` - fork + exec + wait in one call; uses its own internal `wait` and sets `$?` for you
- `kill` - send a signal to a child you no longer want to wait for
- `$?` - where the reaped child's status lands, encoded as exit-code high byte plus signal/core low byte

Processes

waitpid

Wait for a specific child process to terminate and reap it.

`waitpid` is the targeted counterpart of `wait`. Where `wait` reaps *any* child and blocks until one exits, `waitpid` lets you name which child (or which group of children) to wait for, and lets you poll instead of block by passing `WNOHANG` in `FLAGS`. The exit status of the reaped child lands in `$?`, identically to `wait`.

Synopsis

```
use POSIX ":sys_wait_h";

waitpid $pid, 0;                    # block until $pid exits
waitpid $pid, WNOHANG;              # poll: is $pid done yet?
waitpid -1,  WNOHANG;               # reap any ready zombie, don't␣
↪block
```

What you get back

- The PID of the reaped child on success.
- `0` when `WNOHANG` is set and at least one matching child exists but none have terminated yet - i.e. "still running, try again later".
- `-1` when no child process matches `PID` (never existed, already reaped, or - on systems that auto-reap - was harvested by the kernel without Perl's help).

The child's exit status is written to `$?` and the raw platform status to `${^CHILD_ERROR_NATIVE}` on every successful reap. Decode `$?` the same way you would after `wait` or a backtick call:

```
my $reaped = waitpid $pid, 0;
if ($reaped > 0) {
    my $exit   = $? >> 8;
    my $signal = $? & 127;
    my $core   = $? & 128;
}
```

What PID means

`PID` selects which children `waitpid` is willing to reap:

- **Positive PID** - wait for that specific child. `-1` on return means no such child (you already reaped it, or it was never yours).
- **`0`** - wait for any child in the *current* process group. Useful when your program has joined a process group with other cooperating processes and you only want to harvest your own.
- **`-1`** - wait for any child, same scope as `wait`.
- **Less than `-1`** - wait for any child whose process group ID equals the absolute value of `PID`. `waitpid -$pgid, 0` waits for any member of process group `$pgid`.

`FLAGS` is a bitmask. The portable flag is `WNOHANG`, which makes the call non-blocking: if no matching child has exited, `waitpid` returns `0` immediately instead of sleeping. `FLAGS` of `0` means "block until something is reapable", and is implemented on every platform Perl runs on - even those without a native `waitpid(2)` syscall, where Perl emulates it by remembering exit statuses of children it has already seen.

Global state it touches

- `$?` - set to the exit status of the reaped child (shifted: high byte = `exit` code, low 7 bits = signal, bit `0x80` = core-dumped flag).
- `${^CHILD_ERROR_NATIVE}` - set to the raw platform status word, before Perl normalises it into `$?`.

Neither variable is touched when `waitpid` returns `0` (`WNOHANG` and nothing ready) or `-1` (no matching child).

Examples

Block until a known child finishes, then inspect its exit code:

```
my $pid = fork // die "fork: $!";
if ($pid == 0) {
    exec "/usr/bin/gzip", "big.log" or die "exec: $!";
}
waitpid $pid, 0;
die "gzip failed with status ", $? >> 8 if $?;
```

Poll without blocking - useful inside an event loop or a `SIGCHLD` handler:

```
use POSIX ":sys_wait_h";

sub harvest_children {
    while ((my $kid = waitpid -1, WNOHANG) > 0) {
        warn "child $kid exited with ", $? >> 8, "\n";
    }
}
```

The classic “reap every pending zombie, don’t block” idiom:

```
use POSIX ":sys_wait_h";
1 while waitpid(-1, WNOHANG) > 0;
```

Wait only for children in a specific process group. `setpgid` first, then target the group by negating its ID:

```
my $leader = fork // die "fork: $!";
if ($leader == 0) {
    setpgid 0, 0;        # become process-group leader
    exec @worker_cmd or die "exec: $!";
}
setpgid $leader, $leader;  # parent matches, races with child

# ... later, reap any member of that group:
while ((my $kid = waitpid -$leader, WNOHANG) > 0) {
    log_exit($kid, $?);
}
```

Distinguish the three return values in one place:

```
my $r = waitpid $pid, WNOHANG;
if    ($r == $pid) { finalize_child($pid, $?) }
elsif ($r == 0)    { still_running($pid) }
elsif ($r == -1)   { warn "no such child $pid (already reaped?)" }
```

Edge cases

- **`WNOHANG` requires `use POSIX`.** The constant is not a Perl built-in; it comes from the `POSIX` module, typically via `use POSIX ":sys_wait_h";`. Writing a bare integer (`waitpid $pid, 1`) works on Linux but is not portable and hides intent.
- **`$SIG{CHLD} = 'IGNORE'` races with `waitpid`.** When you tell the kernel to auto-reap children, there is nothing left for `waitpid` to find - it returns `-1` with `$!` set to `ECHILD`. Choose one reaping strategy per program, not both.
- **Already-reaped PIDs** return `-1`. A single child's PID can only be reaped once; after that, it is forgotten. If you hand the same PID to `waitpid` twice, the second call returns `-1`.
- **Negative `PID` is a process group, not "any but this one".** `waitpid -1234, 0` waits for members of process group 1234, not "every child except PID 1234". Readers unfamiliar with the POSIX convention often misread this.
- **`WNOHANG` with `PID = -1`** is the standard "drain the zombie queue" form. Combined with a `while` loop it reaps every child that has already exited and stops as soon as the kernel has nothing more to hand back.
- **Return value of `-1` can also mean auto-reap.** On systems configured to reap children automatically (e.g. a `$SIG{CHLD} = 'IGNORE'` handler installed at startup), `waitpid` sees no child and returns `-1` even for a PID you just forked. See `perlipc` for the platform notes.
- **Blocking wait is always available.** Even on platforms without a native `waitpid(2)` or `wait4(2)` syscall, `waitpid PID, 0` works - Perl emulates it by retaining the statuses of children it has already collected.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `wait` - block for *any* child; equivalent to `waitpid -1, 0` on systems that support it
- `fork` - create the children you will later reap with `waitpid`
- `kill` - send a signal to the same PIDs and process groups `waitpid` accepts
- `$?` - exit status of the reaped child; decode as `$? >> 8` for the exit code, `$? & 127` for the signal
- `${^CHILD_ERROR_NATIVE}` - raw platform status word for the same child
- `POSIX` - source of `WNOHANG` and the rest of `:sys_wait_h`

Control flow

wantarray

Report the calling context of the currently executing subroutine.

`wantarray` lets a subroutine ask the runtime *how* its caller wrote the call: was the result assigned to a list, to a scalar, or thrown away? The answer is one of three distinct values, and a sub can use it to return different shapes of data, or to skip expensive work altogether when the caller asked for nothing.

Synopsis

```
wantarray
```

Takes no arguments and has no parentheses form - it is a keyword, not a function call.

What you get back

`wantarray` returns one of exactly three values, corresponding to the three contexts Perl distinguishes:

Return value	Context of the caller	Typical caller syntax
true (1)	list	`my @x = f(); (f())[0]`
false ("")	scalar	`my $x = f(); if (f()) {…}`
`undef`	void	`f();` as a statement

The three-way distinction is why the function "should have been named `wantlist()`": the boolean-looking return is actually tri-state, and `defined wantarray` is the right test for "did the caller want anything at all".

The canonical idiom

The pattern `wantarray` exists for is a sub that adapts its return shape to the caller:

```
sub records {
    my @rows = heavy_query();
    return wantarray ? @rows : "@rows";
}

my @r = records();          # gets the list
my $s = records();          # gets a space-joined string
records();                  # still runs heavy_query - see below
```

Pair that with an early-out when the caller asked for nothing:

```
sub records {
    return unless defined wantarray;    # caller wrote `records();`
    my @rows = heavy_query();           # skipped in void context
```

(continues on next page)

(continued from previous page)

```
    return wantarray ? @rows : "@rows";
}
```

`defined wantarray` is the void-context guard. `wantarray` alone is not - it is false in both scalar and void, and an early `return` on a false `wantarray` would break every scalar-context caller.

Context is propagated, not re-interpreted

`wantarray` reports the context the caller *imposed* on the sub, not what the sub does internally:

```
sub ctx { wantarray ? "list" : defined wantarray ? "scalar" : "void"
 ↪}

my @a = ctx();      # "list"
my $s = ctx();      # "scalar"
ctx();              # "void"
print ctx(), "\n";  # "list"   - print's LIST is list context
scalar ctx();       # "scalar" - scalar() forces scalar context
```

Note the last two: `print` imposes list context on its arguments, and `scalar` explicitly forces scalar context. Neither is about what `ctx` *does* with the result.

Calls from void context

A sub called as a bare statement runs in void context, and `wantarray` returns `undef` *inside that sub*. This does not cascade: a sub called from within a void-context sub sees whatever context *its own* caller imposed.

```
sub inner { defined wantarray ? "wanted" : "void" }
sub outer { my $x = inner(); return $x }

outer();            # outer is void; inner is scalar → "wanted"
```

Each call site establishes its own context independently. `wantarray` always reports the nearest enclosing sub's context, not the dynamic chain's.

Works inside `eval` too

`wantarray` also reports the context of an `eval` block or `eval EXPR`, not just named subroutines:

```
my @x = eval { wantarray ? (1, 2, 3) : "scalar" };  # (1,2,3)
my $x = eval { wantarray ? (1, 2, 3) : "scalar" };  # "scalar"
eval { defined wantarray ? 1 : 0 };                 # void → 0
```

This is the same mechanism - `eval` is a call frame in the same sense a sub is.

Where `wantarray` has no useful answer

`wantarray`'s result is **unspecified** at these places, and code should not rely on any particular value:

- The top level of a file (outside any sub).
- Inside a `BEGIN`, `UNITCHECK`, `CHECK`, `INIT`, or `END` block.
- Inside a `DESTROY` method.

Treat `wantarray` as meaningful only within an ordinary sub or `eval`. Anywhere else, design your code so the answer does not matter.

Examples

A getter that returns the full list to list callers, the count to scalar callers, and does nothing in void context:

```
sub warnings {
    return unless defined wantarray;
    my @w = collect_warnings();
    return wantarray ? @w : scalar @w;
}

my @all   = warnings();     # every warning
my $count = warnings();     # just the number
warnings();                 # collect_warnings() is not called
```

A sub that refuses to be called in void context because the return value is the whole point:

```
sub must_use {
    defined wantarray
        or croak "must_use: return value must be used";
    return compute();
}
```

A sub that emits one element per call in scalar context and the whole batch in list context - a shape some iterator-style APIs expose:

```
sub next_batch {
    state @queue;
    @queue = refill() unless @queue;
    return wantarray ? splice(@queue) : shift @queue;
}
```

Edge cases

- **No parentheses, no arguments.** `wantarray()` parses but the empty parens are noise; the keyword takes nothing. `wantarray $x` is a syntax error, not a call with an argument.

- **Not the caller's context for operators**. `wantarray` reports the call-frame context only. An expression like `$sub->() + 1` puts the call in scalar context; `wantarray` inside `$sub` sees scalar, as expected. There is no way to ask "was I called inside an arithmetic expression" - only list / scalar / void.
- **`return` honours the same context**. `return @list` in scalar context yields the last element, not the count - because `return` itself sees the caller's context. Use `return wantarray ? @list : scalar @list` when you want the count for scalar callers.
- **Boolean context is scalar context**. `if (f())` calls `f` in scalar context; `wantarray` returns false there, not `undef`.
- **List assignment to an empty list is still list context**. `() = f();` is list context even though the result is discarded. `wantarray` returns true. This is useful for forcing list context on a sub whose side effects depend on it.
- **`wantarray` is not available in XS callers the same way**. An XSUB inspects context through `GIMME_V`; a Perl sub called *from* an XSUB sees the context the XSUB established for the call.
- **Prototypes do not change `wantarray`**. A `($)` prototype forces an argument's context, not the sub's own call context.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `return` - honours the same list/scalar/void context `wantarray` reports; the two are designed to be used together
- `caller` - inspect the call stack itself (package, file, line, and, in its three-arg form, more context details including whether `wantarray` would return true there)
- `scalar` - force scalar context on an expression, which is what makes `wantarray` return false in the callee
- `eval` - establishes a call frame whose context `wantarray` also reports

I/O

warn

Emit a warning to `STDERR`.

`warn` stringifies `LIST` and prints the result to `STDERR` - the same formatting machinery as `die`, minus the unwind. Control returns to the caller; nothing is thrown, nothing is caught. Reach for it when the program should keep running but the user, the operator, or the log deserves to hear about something unexpected.

Synopsis

```
warn LIST
warn $message
warn                            # re-surface current $@ as a warning
```

What you get back

`warn` returns 1 under normal circumstances. The return value is almost never useful - the point of the call is the side effect on `STDERR` (or on whatever `$SIG{__WARN__}` does with the message).

Unlike `die`, `warn` does **not** unwind. Execution continues with the statement after the call.

Global state it touches

- `$@` - read by the no-argument form. If `$@` holds a non-empty value, `warn` re-surfaces it (see *Propagation* below). `warn` does **not** write `$@`.
- `$,`, `$\` - **not** consulted. `warn` is not `print`; the output separators and record separator play no role. The message is written as a single string.
- `$.` - the current input line number, appended to messages that lack a trailing newline.
- `$SIG{__WARN__}` - if set, the handler runs in place of writing to `STDERR`. See *The* `$SIG{__WARN__}` *hook* below.
- `STDERR` - the default destination. Any redirection of `STDERR` (shell-level `2>`, `open STDERR, ...`, a PerlIO layer) affects where `warn` output lands.

The trailing-newline rule

The same rule as `die` applies, and for the same reason:

- **Message ends in `"\n"`** - used verbatim. Nothing is appended. Use this for warnings that are complete user-facing diagnostics.
- **Message does not end in `"\n"`** - Perl appends `" at FILE line N"`, and if a file is being read, `", <HANDLE> line M"`, then a final `"."` and newline. Use this when pinpointing the call site is more useful than a tidy message.

```
warn "cache is stale\n";            # "cache is stale\n"
warn "cache is stale";              # "cache is stale at main.pl␣
 ↪line 42.\n"
```

Reference-valued warnings never get location information appended. Stringifying an object to add file and line would defeat the point of passing one.

Propagation: `warn` with no argument (or empty string)

When `LIST` is empty or stringifies to `""`, `warn` does not emit a fresh message - it re-surfaces whatever is in `$@`:

- If `$@` is **non-empty**, `warn` appends `"\t...caught"` to it and emits the result. This is the canonical way to notice that an exception was caught without letting it die silently:

```
eval { risky() };
warn if $@;                        # "<whatever died>\n\t...caught␣
 ↪at ..."
```

- If `$@` is also empty, the string `"Warning: Something's wrong"` is used.

This is deliberately asymmetric with `die`: `die` with no argument **re-throws** (triggering the propagation machinery and the `PROPAGATE` hook for objects); `warn` with no argument **reports** what is already in `$@`.

The `$SIG{__WARN__}` hook

Setting `$SIG{__WARN__}` installs a handler that runs **instead of** the default write to `STDERR`. The handler receives the exception derived from `LIST` as its single argument and is responsible for disposing of it - logging it, rewriting it, escalating it to `die`, or dropping it on the floor:

```
local $SIG{__WARN__} = sub {
    my ($msg) = @_;
    return if $msg =~ /^Use of uninitialized/;  # silently drop
    log_warning($msg);
};
```

Three asymmetries with `$SIG{__DIE__}` are worth internalising:

- **The default message is suppressed.** `$SIG{__DIE__}` runs *alongside* the exception machinery; `$SIG{__WARN__}` runs *instead of* the default `STDERR` write. If the handler does nothing, the warning is silently dropped.
- **To pass the warning through, call `warn` again.** The handler is not re-entered for its own `warn` call, so there is no infinite loop:

```
local $SIG{__WARN__} = sub {
    log_warning($_[0]);
    warn $_[0];                    # re-emit to STDERR as well
};
```

- **A `__WARN__` handler can silence *mandatory* warnings** - warnings that `no warnings` cannot turn off. This makes `$SIG{__WARN__}` a sledgehammer; prefer `no warnings '<category>'` for targeted suppression and reserve the handler for routing or escalation.

```
# wipe out *all* compile-time warnings, then switch on at runtime
BEGIN { $SIG{'__WARN__'} = sub { warn $_[0] if $DOWARN } }
```

(continues on next page)

(continued from previous page)

```
my $foo = 10;
my $foo = 20;                       # duplicate-my silenced
$DOWARN = 1;
warn "\$foo is alive and $foo";     # now shows up
```

`warn` the built-in vs. `use warnings`

These are different things. Do not confuse them:

- **`warn`** is a built-in function that emits a single message right now. You call it.
- **`use warnings`** is a pragma that enables categories of compile-time and run-time diagnostics (uninitialized values, deprecated features, unopened filehandles, and so on). The compiler and interpreter emit those diagnostics on your behalf, using the same machinery `warn` uses.

The pragma's diagnostics flow through `$SIG{__WARN__}` exactly like explicit `warn` calls. `no warnings '<category>'` suppresses the pragma's diagnostics in its lexical scope but has no effect on an explicit `warn` you wrote yourself - that call always fires.

Examples

Report a recoverable problem and keep going:

```
open my $fh, "<", $path
    or warn "skipping $path: $!\n" and next;
```

Development-mode diagnostic with location appended:

```
warn "unexpected record shape";
# unexpected record shape at parser.pl line 87.
```

Noticing a caught exception without suppressing it:

```
eval { load_config() };
warn if $@;                         # surfaces "...caught at ..."
```

Route every warning through a logger, still printing to `STDERR`:

```
local $SIG{__WARN__} = sub {
    my ($msg) = @_;
    $logger->warn($msg);
    print STDERR $msg;              # preserve default visibility
};
```

Escalate a specific warning to a fatal error:

```
local $SIG{__WARN__} = sub {
    die @_ if $_[0] =~ /corrupt database/;
    warn @_;                        # everything else: default path
};
```

Pass a blessed object for structured warnings (the receiver of a `$SIG{__WARN__}` hook gets the reference, not a stringified form):

```
local $SIG{__WARN__} = sub {
    my ($w) = @_;
    if (ref($w) && $w->isa('MyApp::Warning')) {
        $logger->record($w);
    }
    else {
        print STDERR $w;
    }
};
warn MyApp::Warning->new(code => 'SLOW_IO', detail => $path);
```

Edge cases

- **List with two or more items** is stringified and concatenated: `warn "bad record ", $n, ": ", $raw`. The newline rule applies to the concatenated result.
- **Empty list** (`warn;` or `warn "";`) triggers the propagation path, not a fresh message.
- **Reference-valued warning inside a `$SIG{__WARN__}` handler**: the handler receives the reference unchanged. If no handler is installed, the reference is stringified to `STDERR` via its `""` overload (or Perl's default `MyClass=HASH(0x...)`).
- **`warn` inside `$SIG{__WARN__}`**: does **not** re-enter the handler. The call writes to `STDERR` as if no handler were installed. This is what makes the "pass through" idiom safe.
- **`warn` during global destruction**: still works, but `STDERR` may already be closed in pathological cases. Warnings from `DESTROY` handlers running at interpreter exit can vanish.
- **`STDERR` closed or redirected**: `warn` writes to whatever `STDERR` currently points at. If `STDERR` is closed, the write fails silently; no error is raised, because `warn` itself does not check.
- **No relation to `$\` / `$,`**: adding `local $\ = "\n"` does not make `warn "x"` print a newline. The trailing-newline rule is the only newline-adding mechanism for `warn`.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `die` - same formatting rules (trailing-newline suppresses location, references pass through unchanged), but raises an exception instead of reporting and continuing
- `eval` - pairs with the no-argument form of `warn` for the "catch, inspect, re-surface" idiom
- `$@` - the source the no-argument `warn` reports from

- `$SIG{__WARN__}` - the hook that replaces the default STDERR write; use it for routing, filtering, or escalation
- `Carp` - `carp`, `cluck`, `croak`, `confess`: warnings and exceptions that report from the caller's perspective rather than the call site, which is almost always what a library wants

I/O

write

Render one record to a filehandle through its associated `format`.

`write` is the executor half of Perl's report-generation mechanism: it takes the picture declared by a `format`, plugs the values from the matching argument lines into the fields, and emits the resulting lines to `FILEHANDLE`. It also tracks the page: when not enough lines remain, it writes a form feed, runs the top-of-page format, and then writes the record. `write` is the **only** way a declared format produces output; a format is inert until `write` references it.

Do not confuse this with `syswrite` or the inverse of `read`: `write` is strictly the picture-record executor. It has nothing to do with raw byte output.

Synopsis

```
write FILEHANDLE
write EXPR
write
```

What you get back

1 on success, `undef` on failure (with `$!` set). A failure here is typically either a closed handle or an I/O error on the underlying write - not "format not found," which croaks rather than returning `undef`.

How `write` picks the format

Every filehandle has **two** format slots: the record format and the top-of-page format. By default:

- The **record format** is the format whose name matches the filehandle (format `STDOUT` for `STDOUT`, format `REPORT` for a handle opened as `REPORT`).
- The **top-of-page format** is the filehandle name with `_TOP` appended (`STDOUT_TOP`, `REPORT_TOP`), falling back to `top` in the current package.

Both are overridable per-handle through `$~` (current format name) and `$^` (current top-of-page format name). These variables act on the **currently selected** handle, so changing them for anything other than `STDOUT` requires a `select` dance:

```
my $ofh = select REPORT;
$~ = "My_Format";
```

(continues on next page)

(continued from previous page)

```
$^ = "My_Top";
select $ofh;
```

If `FILEHANDLE` is omitted, `write` targets the currently selected handle (default `STDOUT`). If the argument is an `EXPR` rather than a bareword, the expression is evaluated and its string value is looked up as the filehandle name at runtime.

Global state it touches

`write` reads and updates a cluster of per-handle format variables. All are per-filehandle, reached via `select`:

- `$~` - name of the record format for the selected handle. Read at every `write` call.
- `$^` - name of the top-of-page format. Consulted when a new page begins.
- `$=` - page length in lines. Default `60`.
- `$-` - lines remaining on the current page. Decremented by every `write`. Set to `0` to force a page break on the next call.
- `$%` - current page number. Incremented on each top-of-page.
- `$^L` - string emitted before every top-of-page except the first. Default `"\f"` (form feed).
- `$:` - characters on which fill-mode (^) fields may break.
- `$^A` - the format accumulator, the buffer `formline` writes into and `write` flushes. Rarely touched directly unless you are bypassing `write` with `formline`.

`write` also writes through the PerlIO stack of the target handle, so `$|` (autoflush on the selected handle) still applies.

Top-of-form processing

Before emitting the record, `write` checks whether the record fits in the lines remaining on the page (`$-`). If it does not:

1. Emit `$^L` (form feed by default) - skipped on the very first page.
2. Run the top-of-page format named by `$^` to produce the page header.
3. Reset `$-` to `$=` and increment `$%`.
4. Emit the record.

On the very first `write` to a handle, step 1 is skipped so the output does not begin with a form feed.

Examples

Basic report with a header:

```
format STDOUT_TOP =
                        Passwd File
 Name                Login    Office   Uid   Gid Home
 ------------------------------------------------------------------
.
format STDOUT =
 @<<<<<<<<<<<<<<<<<< @|||||| @<<<<<<@>>>> @>>>> @<<<<<<<<<<<<<<<<<
 $name,              $login,  $office,$uid,$gid, $home
.

while (my @row = getpwent) {
    ($name, $login, $uid, $gid, $office, $home) = @row[0,0,2,3,5,7];
    write;
}
```

Write to a named filehandle. The format name defaults to the handle name; no `select` is needed if the default is acceptable:

```
open my $fh, ">", "report.txt" or die $!;
format REPORT =
 @<<<<<<<<<<<<<<   @>>>>>>>
 $item,            $count
.
# rebind the glob so REPORT refers to $fh
*REPORT = *$fh;
for my $r (@records) {
    ($item, $count) = @$r;
    write REPORT;
}
close $fh;
```

Switch format at runtime by setting `$~` on the selected handle. `local` confines the change to the current scope:

```
sub print_summary {
    local $~ = "SUMMARY_FORMAT";
    write;
}
```

Force a page break by zeroing `$-` before the next call:

```
$- = 0;                 # next write starts a new page
write;
```

`write EXPR` picks the handle at runtime from a string - useful when the target is data-driven:

```
my $handle_name = $verbose ? "LOG" : "STDOUT";
write $handle_name;
```

Interleaving `print` with `write`: legal, but `$-` only tracks `write` output. If a `print` burst consumes page rows you care about, decrement `$-` by hand:

```
print "extra note: $msg\n";
$-  -= 1;                  # keep the page counter honest
write;
```

Edge cases

- **Format not found**: if the format named by `$~` does not exist, `write` croaks with `Undefined format "NAME" called`. This is a fatal error, not a return-`undef`-with-`$!` failure.
- **Closed filehandle**: returns `undef` and sets `$!`. Under `use warnings` a `write()` on `closed filehandle` warning is emitted.
- **Record taller than page**: if the record itself has more lines than `$=`, `write` still emits all of its lines on one page - it does not split a single record across a page boundary. The oversize is silently tolerated.
- **`_TOP` fallback**: if no `<HANDLE>_TOP` format exists, `write` falls back to the format named `top` in the current package. This is rarely what you want for autovivified handles; set `$^` explicitly when it matters.
- **Lexical variables in formats**: `my` variables declared outside a format are **not visible** inside it unless the format is declared within the lexical scope of those variables. This bites anyone who wraps `write` inside a subroutine and wonders why the report is blank - move the `format` declaration inside the sub, or use package globals for the argument-line expressions.
- **Top-of-page triggers mid-record, not mid-page**: `write` checks page room once per record. A record that starts near the bottom of the page will either fit in full or trigger top-of-page before being written; it never straddles.
- **`LC_NUMERIC` and numeric fields**: if `use locale` is in effect when the format is declared, the decimal point in numeric fields follows the locale at declaration time - not at `write` time. Switching locales after the format is compiled does not change the separator.
- **Control characters in text fields** are replaced with spaces so they cannot scramble column alignment. The sole exception is `\r` in fill-mode fields, where it forces a line break.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `format` - declare the picture that `write` renders; the two are meaningless without each other
- `formline` - run one picture line against an argument list into `$^A` without any page tracking; the low-level primitive behind `write`
- `select` - change which handle `write` targets when called without an argument, and the prerequisite for setting any of the `$~`, `$^`, `$=`, `$-` per-handle

- `print` - free-form output when columnar reporting is overkill; may be interleaved with `write` on the same handle
- `$~` - override the record format name for the selected handle
- `$-` - lines left on the current page; set to `0` to force a new page before the next `write`
- `perlform.pod` - full picture-line grammar, fill-mode rules, and worked report examples

SCALARs and strings

y///

Transliterate characters in a string. `y///` is a synonym for `tr///` - the two are identical in every respect.

The `y` spelling is provided for **sed** devotees, whose muscle memory produces `y/abc/ABC/` rather than `tr/abc/ABC/`. Perl's parser accepts either; the resulting op is the same. Use whichever spelling reads better in the surrounding code - if the file already quotes a sed script, `y///` often fits; in fresh Perl, `tr///` is more common.

Synopsis

```
y/SEARCHLIST/REPLACEMENTLIST/cdsr
$var =~ y/SEARCHLIST/REPLACEMENTLIST/cdsr
y/SEARCHLIST/REPLACEMENTLIST/      # transliterates $_
```

What you get back

Same as `tr///`:

- Without `/r`: the number of characters matched (and replaced or deleted), after modifying the target in place.
- With `/r`: a new string with the transliteration applied; the original is untouched.

Examples

Classic sed-style uppercase transliteration on `$_`:

```
while (<>) {
    y/a-z/A-Z/;
    print;
}
```

Counting vowels without modifying the string, using `/r` to discard the transliterated copy:

```
my $vowels = ($text =~ y/aeiouAEIOU//) ;   # count in place is fine␣
↪too
my $count  = () = $text =~ /[aeiouAEIOU]/g;
```

Strip everything except digits from a string (non-destructive copy via `/r`, complement + delete):

```
my $digits = $phone =~ y/0-9//cdr;          # "(415) 555-1212" →
↪"4155551212"
```

Rot13 - the textbook `y///` one-liner:

```
(my $coded = $plain) =~ y/A-Za-z/N-ZA-Mn-za-m/;
```

Edge cases

See `tr///` for the full list. Two worth repeating here:

- **No variable interpolation.** `y/$x/$y/` transliterates the literal characters `$`, `x` against `$`, `y` - not the contents of those variables. This matches `tr///` and is the single most common sed-to-Perl translation bug.
- **`y'...'...'` disables ranges.** With single-quote delimiters, `-` is literal: `y'a-z'A-Z'` transliterates only `a`, `-`, `z` against `A`, `-`, `Z`.

Differences from upstream

Fully compatible with upstream Perl 5.44.

See also

- `tr///` - the canonical spelling; identical semantics, same flags (`/c`, `/d`, `/s`, `/r`), same edge cases
- `s///` - pattern-based substitution when character-by-character mapping is not enough
- `lc`, `uc` - locale- and Unicode-aware case change; prefer these over `y/a-z/A-Z/` for anything beyond ASCII
- `$_` - the default target when no `=~` binding is given

5.1.3 Operators

Perl's operator inventory is large but the structure is regular. Operators group into a handful of families - arithmetic, comparison, logical, bitwise, regex binding, assignment - and a few one-off mechanisms (range, ternary, comma, arrow, subscripts). This reference is split into one page per family.

PetaPerl implements the full Perl 5 operator set with identical semantics: same precedence, same associativity, same context rules. Differences are localised to the *PetaPerl-specific notes* at the bottom of each page where they apply.

Choose a family

Arithmetic operators

The numeric four-function-calculator set, plus modulus and exponentiation, plus the unary sign operators.

Operator	Operation	Arity	Associativity
`**`	exponentiation	binary	right
`*`	multiplication	binary	left
`/`	division	binary	left
`%`	modulus	binary	left
`+`	addition	binary	left
`-`	subtraction	binary	left
`+`	unary plus	unary	right
`-`	unary negation	unary	right

All arithmetic operators evaluate their operands in numeric context - strings are coerced via the rule "leading numeric prefix wins, the rest is ignored (with an `Argument "..." isn't numeric` warning under `use warnings`)".

```
my $x   = 5 + 3;          # 8
my $y   = 10 / 3;         # 3.3333333333333…
my $z   = 2 ** 10;        # 1024
my $rem = 17 % 5;         # 2
my $neg = -$x;            # -8
```

Precedence and associativity

`**` binds tightest and is *right*-associative. `*`, `/`, `%` form one group; `+`, `-` form another. Unary `+` and `-` bind between `**` and `*` (high precedence).

```
2 ** 3 ** 2     # 2 ** 9       = 512   (right-assoc: top-down)
-2 ** 2         # -(2 ** 2)    = -4    (** binds tighter than unary
↪-)
(-2) ** 2       # 4                    (parens override)
6 / 2 / 3       # (6 / 2) / 3  = 1     (left-assoc on /)
```

The `-2 ** 2` case bites everyone exactly once. Parenthesise the sign when the base might be negative.

Division

`/` always returns a floating-point result, even for integer operands that divide exactly:

```
10 / 5           # 2     -- numerically equal, stored as int or
↪float
                 #          depending on the runtime; treat as
↪numeric
```

(continues on next page)

(continued from previous page)

```
10 / 4          # 2.5
10 / 3          # 3.3333333333333335
```

For integer division, use `int`:

```
int(10 / 4)     # 2
int(-7 / 2)     # -3   (truncation toward zero, not floor)
```

Division by zero is a fatal error (not a warning, not an infinity-producing operation):

```
my $bad = 1 / 0;   # dies: Illegal division by zero
```

Modulus

% returns the integer remainder. Both operands are converted to integers first; the sign of the result follows the right operand (Perl-flavoured "sign of divisor", same as Python, unlike C).

```
 17 %  5     #  2
-17 %  5     #  3      -- result follows sign of $b ( 5)
 17 % -5     # -3      -- result follows sign of $b (-5)
-17 % -5     # -2
```

For floating-point remainder, use `POSIX::fmod` from *POSIX*.

Exponentiation

** accepts any pair of numbers including non-integer exponents:

```
2 ** 10         # 1024
2 ** 0.5        # 1.4142135623730951    -- square root
$x ** -1        # 1/$x                   -- reciprocal
```

For `sqrt`, the dedicated *sqrt* function is clearer (and named-unary, so it parses with one argument greedily - see *precedence*).

Unary + and unary -

Unary - negates. Unary + is a no-op on numbers; its only practical use is to *force* numeric context, or to disambiguate a syntactic edge case where Perl would otherwise treat the next token as a function name:

```
sub foo { 42 }
print foo (1+2)*3;       # prints 42*3 ... wait, no:
                         # parses as: print( foo(1+2) ) * 3
                         # because foo(...) sticks to the parens.
print foo +(1+2)*3;      # parses as: print( foo + ((1+2)*3) )
                         # the unary + signals "not a function call"
```

You will rarely need this. It is documented because it appears in older code.

Compound assignment

Every binary arithmetic operator has a compound assignment form (`+=`, `-=`, `*=`, `/=`, `%=`, `**=`). See *assignment*.

See also

- *String operators* - `.` and `x`, the string equivalents of `+` and `*`.
- *Numeric comparison* - `< <= == > >= != <=>`.
- *Precedence* - full table.
- `abs`, `sqrt`, `int` - perlfunc forms of related operations.

String operators

Two operators in this family - concatenation and repetition. They play the same role for strings that `+` and `*` play for numbers.

Operator	Operation	Arity	Associativity	Context
.	concatenation	binary	left	string
x	repetition	binary	left	mixed

Both coerce their operands to string context (with the usual numeric-to-string conversion: `42` becomes `"42"`, `1.5` becomes `"1.5"`, etc.).

`.` - concatenation

Joins the two operands. The result is the concatenation of the left and right strings:

```
"Hello" . " " . "World"         # "Hello World"
"x = " . 42                     # "x = 42"            (numeric␣
↪coercion)
"err: " . $!                    # "err: <strerror>"
```

Compound `.=` appends in place:

```
my $s = "a";
$s .= "b";       # "ab"
$s .= "c";       # "abc"
```

`.=` is faster than `$s = $s . "..."` for repeated appending - not by an asymptotic margin in modern Perl, but enough to matter in tight loops on long strings. `pperl`'s runtime mutates the internal buffer in place when the string's reference count is 1 (see *PetaPerl-specific notes* below).

For large numbers of pieces, `join` is generally clearer than a chain of `.`:

```
join(", ", @items)
"hello, world, !"               # hand-built; readable for small␣
```

(continues on next page)

(continued from previous page)

```
↪arity

join(", ", $a, $b, $c, $d)      # cleaner than $a.", ".$b.", ".$c.",
↪".$d
```

x - repetition

Repeats its left operand a number of times given by the right operand. Behaviour depends on the *context* of the left operand:

- **Scalar context** on the left → string repetition.
- **List context** on the left (parens around the left operand) → list repetition.

```
"ab" x 3                # "ababab"          (string)
"-" x 40                # 40-character ruler

(0) x 5                 # (0, 0, 0, 0, 0)   (list)
(1, 2) x 3              # (1, 2, 1, 2, 1, 2)
```

The parens-on-the-left distinction trips beginners. Without parens, the left side is in scalar context regardless of how it looks:

```
1, 2 x 3                # (1, "222")        -- Whoops.
                        # x binds tighter than , so this is
                        # (1, (2 x 3)) - and 2 in scalar context
                        # stringifies, so 2 x 3 is "222".

(1, 2) x 3              # (1, 2, 1, 2, 1, 2)   -- list repetition
```

Compound x= replicates in place:

```
my $line = "=";
$line x= 40;            # "========================================"
```

A non-positive count returns the empty string (or empty list for list context), without warning:

```
"abc" x  0              # ""
"abc" x -3              # ""
("a", "b") x 0          # ()
```

A non-integer count is truncated toward zero:

```
"x" x 3.7               # "xxx"      (3.7 truncates to 3)
"x" x -0.5              # ""
```

Numeric vs string operators

Mixing numeric and string operators on the same operand produces predictable but sometimes surprising results, because each operator imposes its own context:

```
"5" + "3"              # 8        (+ wants numeric → both coerce to␣
↪numbers)
 5  .  3               # "53"     (. wants string  → both coerce to␣
↪strings)
"5" * "3"              # 15
 5  x  3               # "555"    (x wants string left, number right)
```

There is no operator that "guesses" - the operator picks the context, full stop.

See also

- *Arithmetic operators* - `+` and `*` are the numeric analogues of `.` and `x`.
- *String comparison* - comparing strings rather than building them.
- `join`, `sprintf`, `substr` - perlfunc forms of common string-building operations.

Numeric comparison operators

Seven operators that compare two numbers and return either a boolean (the first six) or a three-way result (`<=>`).

Operator	Question	Returns
`<`	less than	true / false
`<=`	less or equal	true / false
`==`	equal	true / false
`>=`	greater or equal	true / false
`>`	greater than	true / false
`!=`	not equal	true / false
`<=>`	three-way (sort)	`-1`, `0`, or `+1`

All seven coerce both operands to numbers before comparing.

```
$age   >= 18                  # true if 18 or older
$count == 0                   # true if numerically zero
$x     != $y                  # true if they differ numerically
$a    <=>  $b                 # -1 if $a < $b, 0 if equal, +1 if $a␣
↪> $b
```

== is for numbers, eq is for strings

This is the single most common bug new Perl programmers ship. `==` compares **numerically** - both operands are run through numeric coercion, and any non-numeric string becomes `0`:

```
"foo" == "bar"        # TRUE  -- both coerce to 0
"42x" == "42y"        # TRUE  -- both coerce to 42
"1e2" == 100          # TRUE  -- "1e2" is 100 in scientific notation
```

For string equality, use eq:

```
"foo" eq "bar"        # FALSE  -- the actual question
"42x" eq "42y"        # FALSE
```

If `use warnings` is in scope, the `Argument "..." isn't numeric` warning fires every time == (or any other numeric operator) sees a non-numeric string. That warning is your friend.

<=> - the spaceship operator

<=> returns -1 if the left is smaller, 0 if equal, +1 if greater. It is *the* sort comparator:

```
my @sorted = sort { $a <=> $b } @numbers;       # ascending
my @rev    = sort { $b <=> $a } @numbers;       # descending

# multi-key: by length, ties broken by content
my @lines = sort { length($a) <=> length($b) || $a cmp $b } @raw;
```

The pattern `cmp1 || cmp2 || cmp3` works because <=> returns 0 on ties and || falls through on 0. See *logical*.

<=> and cmp are the only two operators on this family that are *not* boolean - they are integer-valued and exist for sorting, not for `if`-conditions.

Floating-point equality

== on floats is exact bit-for-bit equality. For values produced by floating-point arithmetic this is almost never what you want:

```
0.1 + 0.2 == 0.3      # FALSE  -- 0.30000000000000004
```

Compare with a tolerance:

```
abs($a - $b) < 1e-9                # absolute tolerance
abs($a - $b) < 1e-9 * abs($a)      # relative tolerance, when $a ≠ 0
```

abs is a *named-unary* operator.

Chaining is not magic

Perl does **not** support mathematical chaining of comparisons. `$a < $b < $c` does not test "$b is between $a and $c" - it parses as `($a < $b) < $c`, where the first comparison yields a boolean ("" or 1) which is then numerically compared with `$c`:

```
1 < 5 < 3         # parses as (1 < 5) < 3
                  #         = 1 < 3
```

(continues on next page)

(continued from previous page)

```
                #          = TRUE   -- which is *not* what you␣
↪meant.
```

Write the conjunction explicitly:

```
$a < $b && $b < $c      # the actual "between" test
```

Precedence

Comparison operators sit at row 11 of the *precedence* table - looser than the arithmetic operators (so `$a + $b == $c + $d` works), tighter than the logical operators (so `$a < $b && $c > $d` works). They are **non-associative**: `$a < $b < $c` is not invalid, but the non-associativity warning explained above applies.

See also

- *String comparison* - `eq ne lt le gt ge cmp`, the lexical-order family.
- *Logical operators* - combining comparison results.
- *<=> in sort* - the canonical use case.
- *Truthiness* - the "two flavours of comparison" overview that motivates the numeric/string split.
- *Numbers and strings* - when a scalar gets coerced to a number, and the precision/Inf/NaN story behind ==.

String comparison operators

The string-flavoured counterparts of the numeric comparison family. Same shape, same precedence row, but operating lexicographically (Unicode code-point order by default) instead of numerically.

Operator	Question	Returns
`lt`	less than	true / false
`le`	less or equal	true / false
`eq`	equal	true / false
`ge`	greater or equal	true / false
`gt`	greater than	true / false
`ne`	not equal	true / false
`cmp`	three-way (sort)	`-1`, `0`, or `+1`

Both operands are coerced to strings before comparing.

```
$name eq "John"              # exact string match
$kind ne "guest"             # negated match
$word lt "m"                 # alphabetically before "m"
$a    cmp  $b                # sort comparator
```

Lexicographic order

Strings compare character by character, code-point by code-point. The first differing character decides; if one string is a prefix of the other, the shorter wins.

```
"abc" lt "abd"        # TRUE  -- 'c' < 'd' at position 2
"abc" lt "abcd"       # TRUE  -- prefix loses
"ABC" lt "abc"        # TRUE  -- ASCII: uppercase < lowercase
"10"  lt "9"          # TRUE  -- '1' < '9' at position 0 (lex, not␣
↪numeric!)
```

The last example is the canonical reason `eq`/`lt`/`gt` exist: when you want digit-strings sorted *numerically*, you must use `<=>` or pre-coerce.

Unicode

Code-point order is **not** the same as locale-aware "alphabetical" order:

- `"ä" gt "z"` is TRUE under code-point order because U+00E4 is beyond U+007A.
- Under German DIN 5007-1 ("dictionary") order, `"ä"` should sort with `"a"` - long before `"z"`.

For locale-correct collation, use `Unicode::Collate` from *perlfunc* or `use locale` with a suitable locale set. The bare `lt`/`gt`/`cmp` give you ordered, stable, language-independent comparison - which is exactly the right thing for `sort` keys, hash bucketing, deterministic test output, and so on. It is the *wrong* thing for human-facing alphabetical listings in any non-English language.

`cmp` for sorting

`cmp` is the string-comparison spaceship. It returns `-1`, `0`, or `+1` and chains the same way `<=>` does:

```
my @sorted = sort { $a cmp $b } @names;        # ascending lex order
my @cased  = sort {
    lc($a) cmp lc($b) || $a cmp $b             # case-insensitive,
                                               # ties broken by case
} @names;
```

Mixing flavours: a worked bug

The compound-key sort idiom from *numeric comparison* showed `||`-chaining of `<=>` and `cmp`. The bug to avoid is using the *wrong* operator for the *type* of the key:

```
# version strings like "1.10", "1.2", "1.20", ...
sort @versions                             # ASCII order:  "1.10",
↪"1.2","1.20"
sort { $a <=> $b } @versions               # numeric mash: works␣
↪only by accident
                                           # (everything past first␣
↪dot ignored)
```

(continues on next page)

(continued from previous page)

```
sort { sortkey($a) <=> sortkey($b) } @versions   # parse first, then␣
↪compare
```

The right answer for version strings is a parser like `Sort::Versions` or hand-rolled `(\d+)`-tokenisation; neither `cmp` nor `<=>` does it correctly on its own.

Precedence

String comparison shares row 11 of the *precedence* table with numeric comparison. They are non-associative - the chaining caveat from *numeric comparison* applies here too:

```
"a" lt "b" lt "c"     # parses as ("a" lt "b") lt "c"
                      #         = 1            lt "c"
                      #         = TRUE  (because "1" < "c"␣
↪lexically)
                      # - accidentally right for the wrong reason.
```

Write the conjunction explicitly with `&&`.

See also

- *Numeric comparison* - the parallel family.
- `sort`, `reverse`, `lc`, `uc`, `fc` - perlfunc tools that pair with string comparison.
- *Unicode in Perl* - the locale / code-point-order distinction in depth.

Logical operators

The boolean family - short-circuit AND, OR, defined-or; their low-precedence word-form siblings; the negations; and `xor`.

Operator	Reads as	Short-circuit	Returns	Precedence
`!`	not	n/a	normalised 1 or `""`	high
`&&`	and	yes	left if false, else right (the operand)	high
`||`	or	yes	left if true, else right (the operand)	high
`//`	defined-or	yes	left if defined, else right (operand)	high
`^^`	exclusive or	no	normalised 1 or `""`	high
`xor`	exclusive or	no	normalised 1 or `""`	very low
`not`	not	n/a	normalised 1 or `""`	very low
`and`	and	yes	same as `&&`	very low
`or`	or	yes	same as `||`	very low

The full conceptual treatment of these operators - what truthiness means, why && returns one of its operands, how implication and XOR fit in, the De Morgan transformations that let you simplify tangled conditions - lives in the *Boolean Logic for Perl Programmers* tutorial. This page is the operator-reference companion.

Short-circuit and operand return

&& and || evaluate left-to-right and stop as soon as the result is determined.

- `A && B` - if A is false, return A without evaluating B. Otherwise return B.
- `A || B` - if A is true, return A without evaluating B. Otherwise return B.

The return value is the **operand itself**, not a normalised true/false. This is what makes Perl's defaulting idioms work:

```
my $port = $config{port} || 8080;            # operand of ||, not␣
↪boolean
my $name = $user_input  || "anonymous";
my $val  = $cache{$key} ||= compute($key);   # ||= sets if currently␣
↪false
```

For comparison, `!`, `not`, and `xor` *do* normalise: they always return either 1 or the empty string `""`.

`//` - defined-or

`//` short-circuits on **definedness** instead of truthiness:

- `A // B` - if A is defined, return A. Otherwise return B.

The difference matters when `0`, `""`, or `"0"` is a legitimate value rather than a stand-in for "missing":

```
my $port    = $config{port} || 8080;    # 0 means "use default" -␣
↪wrong
my $port    = $config{port} // 8080;    # 0 means "0", undef means
↪"default"
my $verbose = $opt{verbose} // 0;       # 0 is a real choice
```

`//=` is the compound form:

```
$config{port} //= 8080;                  # set only if currently␣
↪undef
```

`xor` and `^^`

The exclusive-or operator, true if exactly one side is true. ^^ binds like || (row 16 of the precedence table), `xor` is its very-low-precedence word form. Unlike and/or, neither short-circuits (it cannot - both sides have to be evaluated to determine the result):

```
if ($admin xor $guest) { ... }          # exactly one of the two
my $one = $admin ^^ $guest;             # same test, as an␣
↪expression
```

The result is normalised - 1 or "", not an operand. For bitwise XOR on integers, see *bitwise*.

Symbolic vs word forms - precedence, not style

&&, ||, ! bind tightly (rows 15–16). and, or, not bind very loosely (rows 22–24). This matters at the boundary with assignment:

```
my $fh = open $h, '<', $path || die "no $path: $!";
#                        ^^^^^^^^^^^^^^^^^^^^^^^^^^
# parses as: open $h, '<', ($path || die "no $path: $!")
# - open of "either $path or the die-message". Catastrophe.

my $fh = open $h, '<', $path or die "no $path: $!";
# parses as: ($fh = open $h, '<', $path) or die "no $path: $!"
# - what you meant.
```

Rule of thumb:

- Inside an expression, where you want the value to flow into a variable or another operator: use &&, ||, !, //.
- After a statement, where you want a side-effect (die, warn, return) to fire conditionally: use or, and, not.

! and not

Both negate. Both return the canonical 1 or "":

```
my $missing = ! $config{port};          # 1 if port is unset/0/""/"0
↪"
return if not @items;                   # cleaner reading than `if !
↪@items`
```

! binds high (row 5); not binds very low (row 22).

Compound forms

Three of the logical operators have compound assignment forms:

Compound	Means	Use
&&=	$x = $x && $y	“narrow if currently true”
||=	$x = $x || $y	“default if currently false”
//=	$x = $x // $y	“default if currently undef”

```
$cache{$k} ||= compute($k);              # lazy populate
$cfg{port} //= 8080;                     # default that respects 0
$check     &&= validate($check);         # only if currently truthy
```

Tutorial cross-reference

The conceptual underpinnings of this page are covered in the boolean-logic tutorial:

- *Truthiness* - what Perl considers false.
- *Operators* - the operand-return rule, the precedence reasoning, and the ternary `?:` (which lives in *its own page* on this reference).
- *Truth tables* - all sixteen binary functions and how to spell each in Perl.
- *De Morgan* - reducing convoluted `unless` to clean `if`.
- *Functional completeness* - NAND/NOR.

See also

- *Bitwise* - the same boolean operators applied to integer bits in parallel.
- *Ternary* - `?:`, logical selection as an expression.
- *Numeric comparison* and *string comparison* - produce the boolean inputs that the operators on this page combine.
- *Error variables* - `$!` and `$@` are the values you typically put on the right-hand side of the `or die` idiom that this page's word-form operators were made for.

Bitwise operators

The boolean operators applied to each pair of bits in parallel, plus the two shifts. A 32- or 64-bit integer is, from these operators' point of view, a vector of independent one-bit values.

Operator	Operation	Per-bit rule
&	bitwise AND	each output bit = a ∧ b
|	bitwise OR	each output bit = a ∨ b
^	bitwise XOR	each output bit = a ⊕ b
~	bitwise NOT	each output bit = ¬a (unary)
<<	left shift	shift bits left, zero-fill from right
>>	right shift	shift bits right

Operands are coerced to integers (the floor toward zero, with warnings if a non-integer string is supplied under `use warnings`).

```
0xFF & 0x0F            # 0x0F     -- mask to low nibble
0x10 | 0x01            # 0x11     -- combine flag bits
0xFF ^ 0xAA            # 0x55     -- toggle alternating bits
~0                     # -1 (or all-ones, depending on integer width)
```

(continues on next page)

(continued from previous page)

```
1  << 4               # 16
32 >> 2               # 8
```

Setting, clearing, toggling, testing a flag

The four basic flag operations on a single bit:

```
use constant FLAG_VERBOSE  => 0x01;
use constant FLAG_DRY_RUN  => 0x02;
use constant FLAG_RECURSE  => 0x04;
use constant FLAG_FORCE    => 0x08;

my $flags = 0;

$flags |=  FLAG_VERBOSE;       # SET    -- OR with the bit
$flags |=  FLAG_DRY_RUN;       # SET another
$flags &= ~FLAG_DRY_RUN;       # CLEAR  -- AND with the inverted bit
$flags ^=  FLAG_VERBOSE;       # TOGGLE -- XOR with the bit
my $on = $flags & FLAG_RECURSE;# TEST   -- AND, then test truthiness
```

Each is a one-bit application of a boolean operator: set is `bit ∨ flag`, clear is `bit ∧ ¬flag`, toggle is `bit ⊕ flag`, test is `bit ∧ flag`.

Compound assignment

All six binary bitwise operators have compound forms:

```
$bits  &= $mask;       # AND-assign
$flags |= 0x01;        # OR-assign  (set bit)
$x     ^= $y;          # XOR-assign (toggle bits where $y is 1)
$x   <<= 1;            # multiply-by-2 (with overflow caveats)
$x   >>= 1;            # divide-by-2 (toward -∞ for signed values)
```

Shifts

<< shifts left; >> shifts right. The number of bit positions to shift is the right operand. Shifting by a negative amount, or by an amount equal to or greater than the integer's bit width, is implementation-defined - don't rely on it.

For unsigned arithmetic, left-shift is multiplication by `2 ** $n` (modulo overflow):

```
$x << 1               # like $x * 2
$x << 8               # like $x * 256
```

For unsigned arithmetic, right-shift is integer division by `2 ** $n`:

```
$x >> 1               # like int($x / 2)   (for non-negative $x)
$x >> 8               # like int($x / 256)
```

For *signed* values, >> is implementation-defined - Perl does not guarantee arithmetic vs logical shift. Mask explicitly when sign matters:

```
($x >> 8) & 0xFF      # extract a byte at offset 8, regardless of␣
↪sign
```

Numeric vs bytewise mode

Perl's bitwise operators act on **integers** by default. With `use feature 'bitwise'` (or `use v5.22+`), the suffixed forms `&.`, `|.`, `^.`, `~.` act on **strings** byte-by-byte:

```
use feature 'bitwise';
"\x80" |. "\x01"      # "\x81"  -- single-byte string OR
"abc"  ^. "abc"       # "\0\0\0" -- bytewise XOR
```

The undecorated &, |, ^, ~ operators always do integer math, and string operands are coerced to numbers first. The suffixed `.`-forms are the way to ask for byte-string operations explicitly.

XOR-swap

XOR has two properties - $a \oplus a = 0$ and $a \oplus b \oplus b = a$ - that combine into the famous swap-without-temporary trick:

```
my ($a, $b) = (0xFEED, 0xBEEF);

$a ^= $b;             # a := a ⊕ b
$b ^= $a;             # b := b ⊕ (a ⊕ b)  = a
$a ^= $b;             # a := (a ⊕ b) ⊕ a  = b

# $a == 0xBEEF, $b == 0xFEED
```

Not faster than `($a, $b) = ($b, $a)` in Perl. Worth knowing because it appears in interview folklore and embedded code, and because it is a clean illustration of the XOR identity.

Common bit tricks

Patterns you will see in performance-sensitive code:

```
$x &  ($x - 1)            # $x with its lowest set bit cleared
($x & ($x - 1)) == 0      # true when $x is a power of two (and non-
↪zero)
1 << $n                   # the integer with only bit $n set
$x & (1 << $n)            # is bit $n set in $x?
$x | (1 << $n)            # set bit $n
$x & ~(1 << $n)           # clear bit $n
$x ^ (1 << $n)            # toggle bit $n
($x >> $n) & 0xFF         # extract a byte at offset $n
```

Each is an exercise in tracking what each bit does - pure boolean reasoning applied 32 (or 64) times in parallel.

Tutorial cross-reference

The boolean-logic tutorial has a longer treatment of the bitwise-as-parallel-boolean model and the application patterns above:

- *Boolean Logic - Applications* - bitwise operators as parallel boolean, flag manipulation, XOR-swap, common bit tricks.
- *Boolean Logic - Operators* - the boolean foundations the bitwise operators inherit.

See also

- *Logical* - the same boolean operators on whole values.
- *Precedence* - &, |, ^ sit between comparison and &&/||. Almost always parenthesise around them.
- `pack`, `unpack`, `vec` - perlfunc tools for bit-level work beyond what raw operators offer.

Range and flip-flop operators

Two operators with two completely different jobs depending on context: in **list context** they generate ranges, and in **scalar context** they are stateful "flip-flop" boolean expressions for matching line ranges in stream-processing.

Operator	List-context job	Scalar-context job
`..`	inclusive range	flip-flop (false-tested entry)
`...`	inclusive range	flip-flop (true-tested entry)

`..` in list context - ranges

In list context, `..` produces the inclusive sequence between its endpoints:

```
0..9                    # (0,1,2,3,4,5,6,7,8,9)
'a'..'z'                # ('a','b',...,'y','z')      26 elements
'aa'..'zz'              # ('aa','ab',...,'zy','zz')  676 elements
'A'..'F'                # ('A','B','C','D','E','F')

for my $i (1..100) { ... }                      # numeric for-loop
for my $name ('alpha'..'omega') { ... }         # alphabetic,␣
 ↪magic-incremented
```

For numeric endpoints the result is the integer sequence (with strict ordering - `5..1` is the empty list, never reverse). For string endpoints the magic-increment from `++` is used, which gives you the natural alphabetic spread (and on `'aa'` through `'zz'` the expected 676 two-letter combinations).

A common Perl idiom:

```
my @hex = ('0'..'9', 'a'..'f');           # 16 hex digits as strings
my @rev = reverse 1..10;                   # (10,9,8,...,1)
@arr[ 5..9 ]                               # array slice, indexes 5.
 ↪.9
@arr[ -3..-1 ]                             # last three elements
```

The all-integer range can be very large - `1..1_000_000_000` is a billion-element list. Perl 5 lazily generates loop ranges in `for`, so `for my $i (1..1_000_000_000)` is safe, but `my @x = 1..1_000_000_000` allocates the whole array. PetaPerl follows the same rule.

Reversed endpoints

`5..1` is empty. There is no "reverse range" form; if you want descending, generate ascending and `reverse`:

```
1..5                    # (1,2,3,4,5)
5..1                    # ()          - empty, no warning
reverse 1..5            # (5,4,3,2,1)
```

Mixed-type endpoints

If either endpoint *looks* numeric, `..` does numeric ranging:

```
'1'..'5'                # (1,2,3,4,5)             - numeric
'1'..'10'               # (1,2,3,4,5,6,7,8,9,10) - numeric

'a'..'5'                # ('a')                   - magic-increment␣
 ↪from "a",
                        #                          stops because it␣
 ↪can't reach "5"
```

Make the type explicit to avoid confusion:

```
0+'1' .. 0+'5'          # numeric, no doubt
sprintf("%d", 1) .. sprintf("%d", 5)
```

In practice, write integer ranges with bare integer literals and character ranges with bare quoted-string letters; the magic-vs-arithmetic distinction follows automatically.

`..` and `...` in scalar context - flip-flops

Inside a boolean context (an `if`, a `while`, a `?:`, the right of `&&`/`||`/`//`), `..` becomes a *stateful* expression that remembers whether it is currently between a left-side match and a right-side match.

```
while (<$fh>) {
    if (/^BEGIN/ .. /^END/) {     # true from BEGIN line through END␣
 ↪line
        print;                     # print everything in between,␣
```

(continues on next page)

(continued from previous page)

```
 ↪inclusive
    }
}
```

The semantics:

1. The flip-flop is initially **false**.
2. When the **left** condition (`/^BEGIN/`) becomes true, the flip-flop transitions to **true**, and the right side is evaluated immediately on the same iteration.
3. The flip-flop stays true on subsequent iterations until the **right** condition (`/^END/`) becomes true. On that iteration it returns true *and then* transitions back to false.
4. So both the BEGIN line and the END line are included.

When the flip-flop is true, it returns a *sequence number* (1 on the entering iteration, 2 on the next, ...). On the closing iteration it returns the count followed by `E0` - so the value is true (non-empty string, not the false `"0"`) but stringifies as the number `0` for cosmetic clarity. You can usually treat it as plain boolean.

`...` differs from `..` in one detail:

- `..` (two dots) - when the left side becomes true, the right side is checked **on the same iteration**. So the same line can open and close the range.
- `...` (three dots) - when the left side becomes true, the right side is **not** checked until the next iteration. Always a multi-line range.

```
while (<$fh>) {
    print if /<head>/ .. /<\/head>/;     # one-line <head>...</head>␣
 ↪works
    print if /<head>/ ... /<\/head>/;    # forces multi-line; same-
 ↪line ignored
}
```

When you'd use a flip-flop

The flip-flop shines in awk-style line-range filtering - pulling sections out of a log file, isolating a fenced code block from markdown, processing a configuration block delimited by keywords. It is famously *un*-loved by code reviewers because its statefulness is implicit.

Two practical alternatives that some prefer:

```
# Explicit state variable - equivalent semantics, easier to grep␣
 ↪for:
my $in_block = 0;
while (<$fh>) {
    $in_block = 1 if /^BEGIN/;
    print if $in_block;
    $in_block = 0 if /^END/;
}
```

(continues on next page)

(continued from previous page)

```
# A simple two-step iterator:
while (<$fh>) {
    if (/^BEGIN/ .. /^END/) { print }
}
```

The flip-flop wins on conciseness; the explicit form wins on clarity at the cost of three more lines. Pick by audience.

See also

- `reverse`, `grep`, `map` - list operators commonly composed with ranges.
- *Subscript* - array slices `@arr[1..3]` rely on range-in-list-context.
- *Precedence* - `..` and `...` sit at row 17, between `||` `//` and `?:`. Almost always parenthesise when combining with logic.

Regex binding operators

Two operators that connect a string with a regular expression operation. They do not perform the match themselves; they merely say "this expression's input is that string".

Operator	Reads as	Use
`=~`	matches	bind a regex op (`m//`, `s///`, `tr///`)
`!~`	doesn't match	same, with the boolean result negated

```
$str =~ /pattern/                    # match: TRUE if $str contains␣
↪a match
$str =~ s/foo/bar/                   # substitution: returns count␣
↪of changes
$str =~ tr/a-z/A-Z/                  # transliteration: returns␣
↪count
$str !~ /pattern/                    # match negated: TRUE if NO␣
↪match
```

Without an explicit binding, regex ops act on `$_`:

```
$_ = "hello";
print "match\n" if /h/;              # implicit $_ binding
```

`=~` and `!~` are how you redirect that input to a different variable.

What they actually return

`=~` returns whatever the regex op on the right would return:

- `m//` - boolean (true on match, false on no match) in scalar context; the captured groups in list context.

- `s///` - the number of substitutions performed (which is boolean-true when ⩾ 1).
- `tr///` - the number of characters processed.

`!~` returns the boolean negation, regardless of the underlying op. It is mostly used with `m//`:

```
print "no digit"   if $str !~ /\d/;
print "$n changes" if $str =~ s/foo/bar/g;
my @hits           = $str =~ /(\w+)/g;       # list context:␣
↪captures
```

Three op partners

`=~` accepts three regex operations on its right:

- `m//` - match. The `m` is optional when the delimiters are slashes: `$s =~ /pattern/` and `$s =~ m{pattern}` both work.
- `s///` - substitution. Three pieces: pattern, replacement, flags. Returns the count of replacements.
- `tr///` (also spellable `y///`) - transliteration. Replaces characters one-for-one between two character sets. Returns the count of characters processed.

```
$line =~ /(\d+)/             # extract first run of digits
$line =~ s/^\s+//             # strip leading whitespace
$line =~ s/\s+/ /g            # collapse all whitespace to single␣
↪spaces
$line =~ tr/A-Z/a-z/          # ASCII lowercase
```

`!~` only with `m//`

`!~` is meaningful only with the *match* operation, since substitution and transliteration return *counts* and the "no-changes-made" case is a meaningful zero, not a "didn't match" boolean. Perl will let you write `$s !~ s/.../.../`, but you almost never want it - the result `!s///` is "true if zero substitutions" which reads strangely. Stick to `!~ /.../`.

Lvalue vs rvalue

`=~` itself does not assign. It only routes the regex op on its right at the string on its left. The op *itself* may then mutate that string - `s///` and `tr///` do, `m//` does not - but the mutation comes from `=~`:

```
my $s = "hello";
$s =~ s/l/L/g;           # mutates $s - now "heLLo"
my $n = $s =~ /(\w+)/;  # does NOT mutate $s; $n is the boolean␣
↪result
```

The string on the left of `=~` must be modifiable for `s///` and `tr///`. A literal string or `$1` (a regex capture variable) will fail with "Modification of a read-only value attempted":

```
"hello" =~ s/l/L/g;     # FATAL - literal is read-only
$1 =~ s/x/y/;           # FATAL - capture variable is read-only
```

Copy first if you need to modify a read-only source:

```
(my $copy = $1) =~ s/x/y/;     # idiom for "modify a copy of $1"
```

Precedence

=~ and !~ sit at row 6 of the *precedence* table - quite tight, between unary and the multiplicative operators. This is why you can write `$s =~ /foo/ && $t =~ /bar/` without parens around either match.

Tutorial cross-reference

The boolean-logic tutorial covers regex-as-set-algebra in its applications chapter - alternation as union, lookarounds as intersection and complement:

- *Boolean Logic - Applications* (the *Regular expressions: logic on sets of strings* section)

The full regex language reference lives in the regex guide:

- *Regular expressions guide*

See also

- `m//`, `s///`, `tr///`, `qr//`, `split` - the regex-consuming perlfunc forms.
- `$_` - the default input variable when no binding is given.
- *Regex match variables* - `$&`, `$1`..`$N`, `%+`, `@+`, `@-`: what every successful match populates.

Assignment operators

Plain assignment plus the compound forms - every binary operator that produces a value also has a compound assignment form that applies the operation in place.

Plain =

```
my $x = 10;                          # scalar assignment
my @a = (1, 2, 3);                   # list assignment
my ($x, $y, $z) = (1, 2, 3);         # destructuring
my ($first, @rest) = @items;         # one + remainder
my %h = (a => 1, b => 2);            # hash from key-value list
```

Assignment is **right-associative** - chains are evaluated right-to-left:

```
$a = $b = $c = 0;                    # parses as $a = ($b = ($c =
↪0))
                                     # all three become 0
```

The whole assignment expression has a value: in scalar context it is the value assigned; in list context it is the **list of elements assigned to**, in left-to-right order:

```
my $count = (my @items = qw(a b c)); # $count = 3
                                     # - number of elements assigned
                                     # to @items, in scalar context

while ((my $line = <$fh>) ne '') { ... }   # uses the assignment as
                                            # both side-effect and␣
↪value
```

List assignment in scalar context

Probably the most-used Perl idiom that nobody can quote the rule for:

```
my $n = () = $string =~ /\d/g;       # count digits in $string
```

Decoded:

1. `$string =~ /\d/g` in list context yields all digit captures.
2. `() = ...` is a list assignment to the empty list (assigns nothing, but evaluates the right-hand side in list context).
3. `$n = (...)` puts the *list assignment* in scalar context, which evaluates to the **number of elements** on the right.

Result: `$n` is the count of matches. There is no `scalar(grep ...)` or `count` built-in for this; the empty-list trick is the canonical idiom.

Compound assignment

Every binary operator that fits the pattern `OP=` exists. The mechanics are identical to `$x = $x OP $y`, *except* that `$x` is evaluated only once on the left - which matters when the lvalue has a side-effect (e.g. an autoviv, a function call):

```
$h->{counter}++;                     # autoviv $h->{counter},␣
↪increment
$h->{counter} += 1;                  # same effect, but spell out␣
↪the OP
```

Operator	Equivalent	Use
`+=`	`$x = $x + $y`	accumulate
`-=`	`$x = $x - $y`	subtract in place
`*=`	`$x = $x * $y`	scale
`/=`	`$x = $x / $y`	divide in place
`%=`	`$x = $x % $y`	modulo
`**=`	`$x = $x ** $y`	exponent in place
`.=`	`$x = $x . $y`	append (string)
`x=`	`$x = $x x $y`	repeat (string), or `(@) x= n`
`&=`	`$x = $x & $y`	bitmask AND-equals
`|=`	`$x = $x | $y`	bitmask OR-equals (set bits)
`^=`	`$x = $x ^ $y`	bitmask XOR-equals (toggle)
`<<=`	`$x = $x << $y`	shift-left in place
`>>=`	`$x = $x >> $y`	shift-right in place
`&&=`	`$x = $x && $y`	"narrow if currently true"
`||=`	`$x = $x || $y`	default if currently false
`//=`	`$x = $x // $y`	default if currently undef
`^^=`	`$x = $x ^^ $y`	logical XOR-equals

```
$count     += 5;                          # arithmetic
$path      .= "/user";                    # string append
$flags     |= FLAG_VERBOSE;               # set a flag bit
$flags     &= ~FLAG_VERBOSE;              # clear a flag bit
$cache     ||= load_data();               # lazy populate
$cfg{key} //= $default;                   # default that respects 0 and "
 ↪"
$str        x= 3;                         # repeat in place
```

Lvalues - what can appear on the left

The left operand of = (or any compound form) must be an **lvalue** - a location that can be written to. Perl's lvalues include:

- A simple scalar variable: `$x`, `$obj->{field}`.
- An array or hash variable: `@arr`, `%h`.
- An array element or slice: `$arr[3]`, `@arr[1..3]`.
- A hash element or slice: `$h{key}`, `@h{qw(a b c)}`.
- A dereferenced container: `$$ref`, `@$aref`, `%{$h}`.
- The `local`-/`my`-declared form of any of the above.
- A *substring*: `substr($s, $offset, $len) = "..."`.
- A *ternary* whose branches are lvalues (see *ternary*).
- The pseudo-operators `pos`, `vec`, `keys` on tied hashes, certain `@_` slots, and a handful of specialised forms.

Most of the time you write an lvalue without thinking. The cases where you stop and check are exotic - `substr` as lvalue, a ternary as lvalue, an array slice as the target of a list assignment.

Multiple-target list assignment

The right-hand side is evaluated *first* in list context, then distributed across the lvalues on the left. This makes swap easy:

```
($a, $b) = ($b, $a);                    # swap
($x, $y, $z) = ($y, $z, $x);            # rotate left
```

If the lists differ in length:

- More values on the right than slots on the left → extras silently discarded.
- More slots on the left than values on the right → unfilled slots become `undef`.

```
my ($a, $b) = (1, 2, 3, 4);             # $a=1, $b=2; 3 and 4 dropped
my ($a, $b, $c) = (1, 2);               # $a=1, $b=2, $c=undef
```

A trailing array or hash on the left slurps the remainder:

```
my ($head, @tail) = @items;             # $head = first element
                                        # @tail = the rest
my ($head, %tail) = @items;             # if @items has key/value pairs
                                        # after the first
```

Only **one** array or hash on the left side; otherwise it would not be unambiguous where one ends and the next begins.

Precedence

Assignment operators sit at row 19 - quite low, looser than comparison and `?:`, tighter only than the comma operators and the word-form logicals (`and`, `or`, `not`, `xor`). This is why `my $fh = open ... or die` works (`or` is even looser than `=`).

See also

- *Logical* - `||=`, `//=`, `&&=` are the workhorses of "default if missing" idioms.
- *Subscript* - most lvalue forms involve a subscript.
- *Comma* - `=` parses out from the surrounding `,`-list via precedence; understanding row 19 vs row 20 is what tells you when `(a, b)=(c, d)` distributes vs being a single list.

Ternary operator

One operator, three operands. Logical selection as an *expression* - the result has a value, can be assigned, returned, passed as an argument.

Form

```
COND ? THEN : ELSE
```

If `COND` is true, the value is `THEN`; otherwise it is `ELSE`. Only the chosen branch is evaluated:

```
my $label = $count == 1 ? "1 item" : "$count items";
return $ok ? \%result : undef;
my $abs   = $x < 0     ? -$x       : $x;
```

It is an expression, not a statement

The most common reason to reach for `?:` instead of `if`/`else` is that you want a *value* - to assign, to return, to put inside another expression:

```
# verbose
my $kind;
if ($n == 1) { $kind = 'singular' }
else         { $kind = 'plural'   }

# idiomatic
my $kind = $n == 1 ? 'singular' : 'plural';
```

Chaining (right-associative)

`A ? B : C ? D : E` parses as `A ? B : (C ? D : E)` - the chain descends through the false branch:

```
my $sign = $x < 0  ? '-'
         : $x > 0  ? '+'
         :           '0';
```

The columnar layout makes the chain readable. Past two or three levels, switch to `if`/`elsif`/`else` - the chain stops being clearer than the alternative.

Ternary as lvalue

Both branches must be lvalues, and the chosen branch is the lvalue you assign to:

```
my ($pos, $neg) = (0, 0);
($x > 0 ? $pos : $neg) = $x;              # assigns to $pos or $neg

# Common case: pick a buffer to append to
($verbose ? $log : $debug) .= "$msg\n";
```

This pattern is rare in modern code - an explicit `if` with two straightforward assignments reads better. Worth knowing for when you encounter it.

What you don't write

`?:` is *not* a statement separator. These are wrong:

```
$x ? do_a() : do_b();                          # OK: do_a/do_b have side-
 ↪effects,
                                               # but the value is␣
 ↪discarded
$x > 0 ? print "yes\n" : print "no\n";         # WORKS, but reads␣
 ↪strangely
```

When the goal is a side-effect (rather than a value), use `if`/`else` or the postfix `if`/`unless`:

```
$x > 0 ? do_yes() : do_no();                   # value-return phrasing
do_yes() if $x > 0;                             # side-effect phrasing
do_no()  unless $x > 0;
```

With short-circuit logical operators

`?:` interacts with `&&`, `||`, `//` in ways that can confuse the eye. Almost always parenthesise:

```
my $x = $a || $b ? $c : $d;                    # parses as ($a || $b) ? $c␣
 ↪: $d
                                               # because || binds tighter␣
 ↪than ?:
my $x = $a || ($b ? $c : $d);                  # if you meant the other␣
 ↪thing,
                                               # parenthesise
```

Precedence

`?:` sits at row 18 - looser than the boolean operators (`&&`/`||`/`//`), tighter than assignment. This is why `my $x = COND ? A : B` works without parens around the right-hand side.

See also

- *Logical* - short-circuit `&&` and `||`, the alternative-without-`if` family.
- *Boolean Logic - Operators* - `?:` in the context of all the boolean operators.
- *Assignment* - `?:` as lvalue.

Subscript and slice operators

The bracket-and-brace family that picks elements out of arrays and hashes - and the deref-brace family that does the same through a reference.

Form	Selects	Container	Result type
`$arr[i]`	one element	array	scalar
`$hash{key}`	one element	hash	scalar
`@arr[i, j, k]`	many elements (slice)	array	list
`@hash{k1, k2}`	many elements (slice)	hash	list
`%arr[i, j, k]`	key/value slice	array	list (k,v...)
`%hash{k1, k2}`	key/value slice	hash	list (k,v...)

A few rules cover all twelve combinations.

The sigil signals the *result*, not the container

`@arr` is the array; `$arr[3]` is one of its elements. The leading sigil tells you what *kind of value* you are accessing - scalar (`$`), list of values (`@`), key/value list (`%`):

```
my @arr = (10, 20, 30, 40, 50);

$arr[2]           # 30                - scalar (single element)
@arr[1, 3]        # (20, 40)          - list of values at those␣
 ↪indices
%arr[1, 3]        # (1 => 20, 3 => 40) - list of (index, value)␣
 ↪pairs
```

```
my %h = (a => 1, b => 2, c => 3);

$h{a}             # 1                 - scalar
@h{'a', 'c'}      # (1, 3)            - list of values
%h{'a', 'c'}      # (a => 1, c => 3)  - list of (key, value) pairs
```

The "%-slice" forms (`%arr[...]`, `%hash{...}`) are a Perl 5.20 addition that returns key/value pairs as a list. They are the right tool when you want a copy of *part of* a hash:

```
my %selected = %h{ qw(a c) };           # %selected = (a => 1, c =>␣
 ↪3)
```

Negative indices on arrays

Negative array indices count from the end:

```
$arr[-1]          # last element
$arr[-2]          # second-from-last
@arr[-3..-1]      # last three elements as a slice
```

Out-of-range positive indices read as `undef`. Out-of-range negative indices (further back than the start) raise a fatal runtime error:

```
my @a = (1, 2, 3);
$a[10]            # undef (with no warning)
```

(continues on next page)

(continued from previous page)

```
$a[-10]           # FATAL: Modification of non-creatable array value
                  # (the error wording reflects how the lookup is␣
 ↪implemented)
```

Slice as lvalue

Both array and hash slices are valid assignment targets:

```
@arr[0, 2, 4] = ('A', 'C', 'E');          # set positions 0, 2, 4
@h{qw(a b c)} = (1, 2, 3);                # set three hash keys

(@arr[0, 1], @arr[1, 0]) = ...             # be careful: aliasing!
```

You can swap elements via slice assignment without a temporary:

```
@arr[$i, $j] = @arr[$j, $i];              # swap two array elements
@h{$key1, $key2} = @h{$key2, $key1};      # swap two hash values
```

Hash key autoquoting

Inside {} for hash subscripts, **bareword identifiers are auto-quoted**:

```
$h{name}          # same as $h{'name'}
$h{first_name}    # same as $h{'first_name'}
$h{1_2}           # FATAL - not a valid bareword identifier
$h{"1_2"}         # quote it explicitly
```

This is one of the most-loved (and most-bug-causing) Perl conveniences. The bareword-or-not decision happens at parse time based on the literal token; constants and computed values do not benefit:

```
use constant KEY => 'name';
$h{KEY}           # WRONG - this is the bareword "KEY", not the␣
 ↪constant!
$h{KEY()}         # use empty parens to force function-call␣
 ↪evaluation
$h{+KEY}          # or unary +, the canonical workaround
```

The +KEY form is the most compact way to say "this is *not* a bareword, evaluate it as an expression".

Array indices: integer-coerced, no autoquoting

[] indices are evaluated as expressions and coerced to integers (truncating toward zero):

```
$arr[1.7]         # $arr[1]    - 1.7 truncates to 1
$arr["3"]         # $arr[3]    - string coerces to integer
$arr[$#arr]       # last element  ($#arr is last valid index)
```

The `$#arr` syntax inside subscripts is idiomatic for "last":

```
$arr[$#arr]       # last
@arr[0..$#arr]    # all of @arr (verbose; just write @arr)
@arr[1..$#arr]    # all except first
```

Mixing - multidimensional access

Perl 5 does not have native multi-dimensional arrays; nested data is built from references:

```
my @grid = ([1,2,3], [4,5,6], [7,8,9]);

$grid[1][2]       # 6                  - chained subscripts on AoA
$grid[1]->[2]     # same - the arrow is optional between subscripts
$grid[$y][$x]
```

The arrow-between-subscripts is implied by Perl's grammar: when two subscripts immediately follow each other on a deref chain, the `->` can be omitted. See *arrow*.

See also

- *Arrow* - `->[]` and `->{}` for going through a reference; the arrow-elision rule.
- *Assignment* - slice assignment; multi-target destructuring.
- `exists`, `delete` - perlfunc forms that take a subscript expression.
- `keys`, `values` - whole-container forms; the slice operators are the partial forms.
- *Arrays*, *Hashes* - the data-types pages explain the sigil-vs-result rule that governs which subscript form to use.

Arrow operator

The single most heavily used operator in working Perl, after =. It does three things at three syntactic positions, and they all start by *dereferencing* something on the left.

Form	Means	Example
`$ref->[i]`	element of an arrayref	`$aref->[0]`
`$ref->{k}`	element of a hashref	`$href->{name}`
`$ref->(...)`	call a coderef	`$cref->(@args)`
`$obj->meth`	method call on an object	`$obj->print`
`$obj->meth(@a)`	method call with arguments	`$obj->set_x(42)`
`$class->meth`	class method call	`Foo::Bar->new`

The first three forms are *dereferencing*. The last three are *method dispatch*. Both are spelled with `->`.

Dereferencing

A reference is a scalar that points at a container or a coderef. You cannot index into a reference directly with `[]` or `{}`; you must dereference first, and `->` is the canonical way:

```
my @arr  = (10, 20, 30);
my $aref = \@arr;

$arr[0]            # 10  - direct array access
$$aref[0]          # 10  - dereference, then subscript (old syntax)
$aref->[0]         # 10  - dereference and subscript in one move
$aref->[-1]        # 30  - negative indices work the same way
```

```
my %h    = (name => 'John');
my $href = \%h;

$h{name}           # 'John'
${$href}{name}     # 'John' - full deref + subscript
$href->{name}      # 'John' - same, idiomatic form
```

Coderef invocation

```
my $f = sub { $_[0] * 2 };

$f->(5)            # 10  - invoke the coderef with argument 5
&$f(5)             # same - older syntax, still works
```

The `->()` form is preferred in modern Perl. Empty argument list is `$f->()`.

Arrow elision between subscripts

When two subscripts directly follow each other on a deref chain, the **second arrow can be omitted**. The grammar implies it:

```
$grid[1]->[2]      # explicit
$grid[1][2]        # arrow elided - the only way most people write
↪this

$tree->{children}[0]{name}    # parses with arrows between every
↪pair
                              # - the chain is read as a sequence
                              # of subscripts on the same deref path
```

The first arrow (between a scalar variable and the *first* subscript) is **required**. Only arrows *between consecutive subscripts* may be elided.

```
$aref->[0]->[1]    # explicit
$aref->[0][1]      # elided second arrow - fine
$aref[0][1]        # WRONG - $aref[0] reads from @aref, not the
                   # arrayref in $aref. Different variable!
```

The first-arrow rule is the source of subtle bugs when refactoring `$aref->[0]` into a deeper expression - keep the leading arrow.

Method calls

When the right side of `->` is an *identifier* (or a parenthesised name expression), it is interpreted as a method call rather than a subscript:

```
$obj->print;                        # call $obj->print()
$obj->name;                         # accessor: returns $obj->name
$obj->set_name("Alice");            # mutator: sets and may return $obj
Foo::Bar->new(arg => 1);            # class method call

$obj->$method_name(@args);          # method name in a variable
$class->can('foo')                  # can() - does method exist?
```

Method calls dispatch through the package the object is blessed into. The exact resolution path is governed by `@ISA` and the chosen MRO; see the *OOP guide* for the full story.

Indirect-object syntax - don't use it

Perl 5 historically allowed `print $fh "foo"` and `new Foo @args` without an arrow. These forms still parse but are context-sensitive in ways that surprise - the indirect-object syntax can resolve the `new` (or whatever) at the wrong time. The modern guidance is to **always use `->`** for method calls:

```
my $obj = Foo->new(@args);                    # correct
my $obj = new Foo @args;                       # avoid (still␣
↪parses)

print $fh "hello\n";                           # OK - print is␣
↪special
$fh->print("hello\n");                         # also fine, prefer␣
↪this
                                               # for IO::File-style␣
↪handles
```

Precedence

`->` is row 2 of the *precedence* table - second only to terms and list operators. It binds tighter than every other operator, which is what you want: `$h->{a} + $h->{b}` parses as the sum of two hash lookups, not as a lookup of `{a + $h->{b}}`.

See also

- *Subscript* - direct (non-dereferenced) array/hash access; the arrow-elision rule.
- *References tutorial* - full treatment of references, including how `\` builds them and how the deref alternatives (`@{...}`, `%{...}`, `${...}`) work.
- *OOP guide* - `->method` and class dispatch in depth.

- `ref`, `bless` - perlfunc tools that pair with the arrow operator's two roles.
- *References* - what `\X` produces and how the deref shapes (`@{...}`, `%{...}`, `${...}`) compose.

Comma operators

Two operators, both spelled with a comma - one of them spelled "fat comma" with =>. They are syntactically the same in most positions but differ on the left-hand-side bareword rule.

Operator	Reads as	What it does
,	comma	list separator (in list ctx); discard-and-yield-right (in scalar ctx)
=>	fat comma	same as , , plus auto-quotes a bareword on the left

In list context - building lists

In list context the comma is the list-element separator:

```
my @items = (1, 2, 3, 4);                    # 4-element list
my @flat  = (1, 2, (3, 4), 5);               # (1,2,3,4,5) - lists␣
↪flatten
my @kv    = (a => 1, b => 2);                # 4-element list: a, 1,␣
↪b, 2
my %h     = (a => 1, b => 2);                # hash from key-value␣
↪pairs
```

Lists in Perl are *flat* - there is no nested-list value. A list embedded in another list disappears: its elements simply become elements of the outer list. To get nested structure you need references (see the *References tutorial*):

```
my @flat   = (1, 2, (3, 4), 5);              # (1,2,3,4,5) - 5␣
↪elements
my @nested = (1, 2, [3, 4], 5);              # (1, 2, ARRAYREF, 5) - 4␣
↪elements
```

In scalar context - comma is "discard, then return right"

When a comma expression is *forced* into scalar context - by an assignment to a scalar, by `scalar(...)`, by being the condition of an `if` - it evaluates each operand left-to-right for side-effects, then yields the value of the rightmost operand:

```
my $x = (1, 2, 3);                           # $x = 3 - but warn␣
↪under
                                             # `use warnings`:
↪"Useless use of
```

(continues on next page)

(continued from previous page)

```
                                                # a constant in void␣
↪context"
                                                # for 1 and 2

my $r = (do_a(), do_b(), final_value());        # runs all three,␣
↪returns
                                                # the last value
```

This usage is rare in modern Perl - the C-style `for(init, init; cond; step, step)` idiom is the main place you encounter it (and even there, parens around the comma sub-expression are advisable).

=> - the fat comma

=> does the same thing as , *plus* it auto-quotes the identifier-shaped bareword on its left:

```
my %h = (
    name  => "John",                            # 'name'  auto-quoted
    age   => 30,
    "key with spaces" => "...",                 # quote required for␣
↪non-bareword
);

# Equivalent without =>, requiring explicit quoting:
my %h = (
    'name', "John",
    'age',  30,
    'key with spaces', '...',
);
```

The auto-quote applies only when the left side is **a syntactically valid bareword identifier** - letters, digits, underscores, starting with a letter or underscore. Other shapes are not auto-quoted:

```
( name      => 1 )          # 'name' auto-quoted - fine
( "name"    => 1 )          # already a string - fine
( $variable => 1 )          # $variable evaluated - fine
( 0+1       => 1 )          # numeric expression - fine
( -name     => 1 )          # WRONG - looks like negative bareword;
                            # parses as -('name') which is the
                            # numeric negation of a string. Quote␣
↪it.
( CONST     => 1 )          # 'CONST' auto-quoted - does NOT call
                            # the constant CONST!
( CONST()   => 1 )          # explicit empty parens - calls CONST
```

The interaction with constants is the most common surprise - see also the *subscript* page on hash key autoquoting, which has the same pattern.

Common idioms

```
# Function arguments - the comma is just the argument separator
some_function($arg1, $arg2, $arg3);

# Hash construction - fat-comma communicates "this is a key"
my %config = (
    host => 'localhost',
    port => 8080,
    user => $ENV{USER},
);

# List literal - interchangeable
my @months = (qw/Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec/);

# Trailing comma - allowed and idiomatic for diff-friendly multi-
 ↪line literals
my @list = (
    "first",
    "second",
    "third",       # trailing comma is fine
);
```

The trailing comma is intentional: it lets you append, remove, or reorder list elements without touching surrounding lines, which shows up as cleaner diffs in version control.

Precedence

`,` and `=>` sit at row 20 - looser than assignment, tighter only than the word-form logicals (`and`/`or`/`not`/`xor`) and list operators in their right-side position. This is why a list of assignments works:

```
my $a = 1, my $b = 2;          # WRONG: parses as (my $a = 1), my $b
 ↪= 2
                               # - comma at outer level, not list
 ↪assignment.
                               # $a is 1, $b is 2, but the warning
 ↪is loud.

my ($a, $b) = (1, 2);          # the right way: list assignment
```

See also

- *Assignment* - list assignment relies on the comma's role in list context.
- *Subscript* - `$h{name}` autoquoting, the parallel case to `name => ...`.
- `use constant` - the bareword/constant collision is the canonical fat-comma gotcha.
- *Context* - the list-vs-scalar story behind why `,` means two different things.

Precedence

Two pieces, in this order: the full precedence table, and the parse rule for *named unary operators* - built-in functions whose syntax binds at operator precedence rather than as full list-operator calls.

The full table

Operators tighter at the top, looser at the bottom. Same row = same precedence; combining two same-row operators uses associativity. none = non-associative (combining two of them without parens is an error or a parse-time warning).

Row	Associativity	Operators
1	left	terms, list operators (leftward)
2	left	`->`
3	none	`++ --`
4	right	`**`
5	right	`! ~ \` unary + unary -
6	left	`=~ !~`
7	left	`* / % x`
8	left	`+ - .`
9	left	`<< >>`
10	none	named unary operators
11	none	`< <= > >= lt le gt ge`
12	none	`== != <=> eq ne cmp`
13	left	`&`
14	left	`| ^`
15	left	`&&`
16	left	`|| ^^ //`
17	none	`.. ...`
18	right	`?:`
19	right	`= += -= *= /=` etc.
20	left	`, =>`
21	none	list operators (rightward)
22	right	`not`
23	left	`and`
24	left	`or xor`

A few rules of thumb:

- **Comparison and logic.** Comparison (rows 11–12) is tighter than &&/||/// (rows 15–16), so `$a < $b && $c > $d` works without inner parens.
- **Bitwise vs comparison.** &, |, ^ (rows 13–14) are *looser* than comparison. `$x & 0xFF == 0` parses as `$x & (0xFF == 0)`. Almost always parenthesise around bitwise.
- **Symbolic vs word logic.** &&/||/! are tight (rows 5, 15, 16). and/or/not are very loose (rows 22–24). Same logical meaning; different binding. The *logical* chapter shows where each is appropriate.
- **= is loose.** Row 19 - almost everything binds tighter than assignment. This is why

`my $fh = open ... or die` works: `or` (row 24) is even looser than `=`, so the open finishes assigning before `or` decides.

- **List operators bookend the table.** Row 1 looking *leftward*, row 21 looking *rightward*. A list operator like `print` or `sort` slurps everything to its right, up to the first operator that binds looser than row 21 - which is just `not`/`and`/`or`/`xor`. The *named unary* section below contrasts this with the row-10 single-argument case.

When in doubt, write parentheses. They cost nothing and document intent.

Post and pre increment / decrement

Row 3 (`++` and `--`) is non-associative. The pre and post forms differ in *when* the side-effect happens:

- **`++$x`** - increment first, then yield the new value.
- **`$x++`** - yield the current value, then increment.

```
my $x = 5;
my $a = ++$x;          # $x is now 6, $a is 6
my $b = $x++;          # $a was 6, $x++ yields 6 then increments,
                       # so $b is 6 and $x is now 7
```

`++` on a string performs Perl's *magic increment*:

```
my $s = "aa";
$s++;                  # "ab"
$s++;                  # "ac"
# ...
$s = "az"; $s++;     # "ba"  - carries like base-26
$s = "Aa"; $s++;     # "Ab"  - case is preserved per-position
$s = "Az9"; $s++;    # "Ba0" - digits and letters carry␣
 ↪independently
```

The magic-increment is what makes `'a'..'zz'` produce all 702 two-letter combinations in *range*.

Named unary operators

Row 10 contains a category called *named unary operators*. These are not symbolic operators - they are **built-in functions** that take exactly one argument and parse with operator-style precedence (between bitshifts and comparison) rather than as full list operators.

The practical consequence is one parsing rule:

> A named unary operator grabs its **one** argument greedily, and anything that would need more than one argument is not part of it.

```
defined $x + 1             # parses as: ( defined($x) ) + 1
                           # - NOT  defined($x + 1)
length $s > 3              # parses as: ( length($s) ) > 3
```

(continues on next page)

(continued from previous page)

```
-e $f && -r _              # both file tests are named-unary;
                           # `&&` combines two boolean results
```

Compare with a list operator (`print`, `sort`, `push`, ...), which is at row 1 / 21 and slurps the rest of the expression:

```
print "n=", $n, "\n"       # print's argument list is "n=", $n, "\n"
                           # all three commas belong to print
```

If you want a named-unary built-in to take a more complex argument, parenthesise:

```
defined($x + 1)            # explicit: argument is the sum $x+1
length($s . "x")           # length of $s with "x" appended
```

The unambiguous form is always available. When in doubt, parenthesise.

The list of named unary built-ins

These built-ins parse at row-10 precedence. Each links to its `perlfunc` detail page - the parsing rule above is the only thing you need to know about them as *operators*; their *function* behaviour is in the perlfunc reference.

`defined`, `exists`, `ref`, `scalar`, `length`, `uc`, `lc`, `ucfirst`, `lcfirst`, `chr`, `ord`, `hex`, `oct`, `int`, `abs`, `sqrt`, `sin`, `cos`, `log`, `exp`, `rand`, `srand`, `alarm`, `sleep`, `caller`, `wantarray`, `chroot`, `readlink`, `umask`, `lstat`, `stat` (when given exactly one argument), plus all the **file test operators** (`-e`, `-r`, `-w`, `-x`, `-f`, `-d`, `-l`, `-s`, and the rest - see the *file test operators* section below).

The “operator” framing in classical Perl documentation is historical. In modern Perl terms, these are *functions whose argument-parsing places them between bitshift and comparison on the precedence table*. The naming is what it is; the behaviour is what matters.

File test operators

File tests are unary operators (named-unary precedence) that inspect a property of a file or filehandle. They begin with a hyphen and a single letter. All return boolean except `-s` (file size in bytes), `-M`/`-A`/`-C` (age in days, fractional), and the read-time tests `-T`/`-B` (boolean but slightly slower because they read the file).

Op	Test	Returns
`-e`	exists	boolean
`-r`	readable by effective UID	boolean
`-w`	writable by effective UID	boolean
`-x`	executable by effective UID	boolean
`-o`	owned by effective UID	boolean
`-R`	readable by real UID	boolean
`-W`	writable by real UID	boolean
`-X`	executable by real UID	boolean
`-O`	owned by real UID	boolean
`-z`	zero size	boolean
`-s`	non-zero size	size in bytes or false
`-f`	regular file	boolean
`-d`	directory	boolean
`-l`	symbolic link	boolean
`-p`	named pipe (FIFO)	boolean
`-S`	socket	boolean
`-b`	block special file	boolean
`-c`	character special file	boolean
`-t`	TTY (terminal)	boolean
`-u`	setuid bit	boolean
`-g`	setgid bit	boolean
`-k`	sticky bit	boolean
`-T`	text file (heuristic)	boolean
`-B`	binary file (heuristic)	boolean
`-M`	modification time (script-start age, days)	number
`-A`	access time (days)	number
`-C`	inode change time (days)	number

```
if (-e $file) { ... }                # exists
if (-f $path && -r $path) { ... }    # regular file and readable
my $size = -s $file;                 # size in bytes
if (-d $path) { ... }                # is a directory
```

Stacked file tests. `-f -w -r $file` is shorthand for `(-f $file) && (-w $file) && (-r $file)` - the chain re-uses the same target. The pseudo-handle `_` (single underscore) re-uses the *previous* file test's `stat` cache to avoid extra `stat()` syscalls:

```
if (-f $file && -r _) { ... }        # one stat() call, two checks
```

The trailing `_` is the standard idiom for any chain of file tests on the same path.

See also

- Every other page in this reference links into the rows of this table from its own *Precedence* section. The cross-reference is intentional - when you need to combine an operator with another, this table is the answer.
- *perlfunc index* - every name in the named-unary list is documented there.

- *Arithmetic* - `+ - * / % **` and the unary `+`/`-`.
- *String* - `.` (concatenation), `x` (repetition).
- *Numeric comparison* - `< <= == >= > != <=>`.
- *String comparison* - `lt le eq ge gt ne cmp` (lexicographic).
- *Logical* - `&& || // ! and or not xor` and the short-circuit / operand-return rule.
- *Bitwise* - `& | ^ ~ << >>`.
- *Range* - `..` and `...`, with their list/scalar dual semantics.
- *Binding* - `=~` and `!~` connect strings with regex.
- *Assignment* - `=` and the compound forms (`+= .= ||= //=` etc.).
- *Ternary* - `?:`.
- *Subscript* - `[]`, `{}`, slices, deref braces.
- *Arrow* - `->` (deref, method call).
- *Comma* - `,` and the fat comma `=>`.
- *Precedence* - the full precedence table and the named-unary parse rule that confuses everyone the first time.

Cross-cutting concepts

A few ideas appear in every chapter. Read these once and you will recognise them throughout the rest:

- **Context.** Operators evaluate their operands in a specific context (numeric, string, list, scalar, boolean). Many of the surprises in Perl arithmetic and comparison come from context coercion at the operand boundary, not from the operator itself.
- **Short-circuit.** `&&`, `||`, `//`, `and`, `or` evaluate left-to-right and stop as soon as the result is determined. The whole expression's value is the last thing actually evaluated - which is one of its operands, not a normalised true/false. See *logical* and the *boolean-logic tutorial*.
- **Precedence.** Tells you how operators bind when parentheses do not. The full table is in *precedence* and every page links the rows it cares about.
- **Lvalue.** Most operators produce *rvalues* (values you can read). A few - assignment, deref, ternary in some shapes - produce *lvalues* (locations you can write to).

See also

- *Boolean Logic for Perl Programmers* - the conceptual companion to the *logical* and *bitwise* pages.
- *Regular expressions guide* - paired with *binding*.
- *perlfunc* - built-in functions, including every name listed in the *precedence* page's *named unary operators* section.

5.1.4 Data types

Perl has three fundamental data types - **scalars**, **arrays**, and **hashes** - plus a small number of supporting kinds (references, typeglobs, code) that are technically scalars but behave differently enough to deserve their own treatment. Every value the language manipulates is one of these.

This reference is split into one page per topic. Topics that share context (numeric vs string conversion, references vs typeglobs, context vs interpolation) live in the same page; topics that don't share context get their own page.

Choose a topic

Scalars

A scalar is the unit value of Perl. It holds one of: a number (integer or float), a string, a reference, or `undef`. The sigil is `$`, in every position where the value is a single scalar - including when the value is one element of an array or hash:

```
$x                  # the scalar named x
$arr[0]             # the first element of @arr - also a scalar
$h{key}             # one value from %h - also a scalar
```

A scalar is *not* statically typed. The same variable can hold a number now and a string a moment later. Perl converts between the two whenever an operator demands one shape or the other; see *numbers and strings*.

Truth, definedness, and undef

Two related but distinct questions about a scalar:

```
defined $x          # is there any value at all?
$x ? T : F          # is the value true?
```

A scalar is **false** in boolean context if it is one of:

```
undef               # the undefined value
0                   # numeric zero
0.0                 # also numeric zero
""                  # empty string
"0"                 # the literal string "0"
```

Everything else is true. Note `"00"`, `"0.0"`, `"0 but true"`, and `"0E0"` are all *true* - they are non-empty strings that don't match the literal `"0"`. The last two are common idioms for "this function returned successfully but there was nothing to count":

```
sub touch_files {
    my $count = 0;
    $count++ for grep { -e $_ && utime undef, undef, $_ } @_;
    return $count || '0E0';     # true even on zero work, so callers
                                # can write `if (touch_files(...))`
}
```

`defined` is the only way to tell `undef` from the false-but-defined values. A common bug:

```
if ($cache{$key}) { ... }            # WRONG - false for empty/0/
↪undef
if (exists $cache{$key}) { ... }      # right - pure presence test
if (defined $cache{$key}) { ... }     # right - value-not-undef test
```

Use `defined` when the only thing you care about is "is there a value here at all"; `exists` for "is this key/index in the container" (the two differ for sparse arrays and `deleted` keys).

Dynamic typing

A scalar can change shape during execution:

```
my $x = 42;          # internally an integer
$x = "hello";        # now a string
$x = 3.14;           # now a float
$x = \@items;        # now a reference
```

Most code never needs to think about which internal shape a scalar currently has. The exceptions are JSON-serialisation libraries that care whether 1 was assigned as a number or a string, and the `Scalar::Util` introspection helpers (`looks_like_number`, `dualvar`).

"Looks like a number"

When Perl converts a string to a number it stops at the first character that isn't part of a valid numeric prefix:

```
"42"        + 1   # 43
"42abc"     + 1   # 43      - leading digits parsed
"abc42"     + 1   # 1       - no leading digits, treated as 0
" 42"       + 1   # 43      - leading whitespace allowed
"4_2"       + 1   # 5       - underscores in literals only, NOT in␣
↪strings
"0x1F"      + 1   # 1       - hex string is NOT auto-recognised;␣
↪hex() does
```

The mismatch between numeric *literals* (which honour `_` and `0x`) and numeric *strings* (which do not) is one of the most reported "why" questions in Perl. Use `hex` / `oct` when the input is a string in non-decimal form:

```
my $bytes = hex "1F";        # 31
my $perms = oct "0755";       # 493
my $perms = oct "0o755";      # 493 - explicit octal prefix
my $bytes = oct "0x1F";       # 31  - oct() accepts any prefix
```

Under `use warnings`, an arithmetic operation on a non-numeric string emits *Argument "…" isn't numeric in addition (+)*. The operation still returns 0 - the warning, not a fatal error, is the signal.

Dual values

Some scalars carry both a string and a number form simultaneously, chosen by context. The classic example is `$!`:

```
$!                # in string context: "Permission denied"
$! + 0            # in numeric context: 13
```

You can build your own with `Scalar::Util::dualvar`:

```
use Scalar::Util qw(dualvar);
my $err = dualvar 13, 'Permission denied';

print "err is $err";           # "err is Permission denied"
print "err is " . ($err + 0);  # "err is 13"
```

Dual scalars are a niche tool. The common case where one matters is the `$!` errno; the general escape hatch is the dualvar helper.

"0 but true"

The literal string `"0 but true"` is the canonical Perl idiom for "a function whose normal return is a count, but the count is zero and the call succeeded". It is true (non-empty, non-`"0"`) but numerically equals 0:

```
my $hits = "0 but true";

if ($hits)        { print "succeeded\n"; }    # prints
print $hits + 0;                              # 0
print "got $hits results\n";                  # got 0 but true␣
↪results
```

The compiler also recognises `"0 but true"` specifically and skips the *isn't numeric* warning that any other "0 followed by garbage" string would emit.

See also

- *Numbers and strings* - how integer, float, and string forms convert into one another and where the corners are.
- *References* - `\X` produces a scalar that points to another value.
- *Context* - when a scalar gets one value vs many.
- `defined`, `undef` - test for and produce the undefined value.
- `scalar` - force scalar context on a list or array.
- `$_` - the implicit scalar for many built-ins.

Arrays

An array is an ordered, integer-indexed sequence of scalars. The sigil is `@` for the whole array and for slices (which are lists); `$` for a single element (which is one scalar). The leading sigil tells you the *result type*, not the container.

```
my @days = ('Mon', 'Tue', 'Wed', 'Thu', 'Fri');

@days              # the whole array (5 elements)
$days[0]           # the first element - a single scalar
@days[1, 3]        # a slice - a list of two scalars
$#days             # the last valid index - here, 4
scalar @days       # the length - here, 5
```

Indices start at 0. Negative indices count from the end (`$days[-1]` is `'Fri'`).

Length: `$#arr` vs `scalar @arr`

Two related quantities, used for different jobs:

```
$#days             # last valid index            - 4
scalar @days       # number of elements          - 5
$#days + 1         # equivalent to scalar @days
```

Assigning to `$#arr` resizes the array. Shrinking destroys the elements past the new end; growing fills them with `undef`:

```
my @a = (1, 2, 3, 4, 5);
$#a = 2;              # @a is now (1, 2, 3)         - length 3
$#a = 5;              # @a is now (1, 2, 3, undef, undef, undef)
$#a = -1;             # @a is now ()                 - empty
```

The `@a = ()` form is the idiomatic empty-array assignment; `$#a = -1` is equivalent and occasionally clearer when the intent is "shrink to nothing without changing the variable's identity".

Element access

```
my $first = $days[0];             # 'Mon'
my $last  = $days[-1];            # 'Fri'
my $last  = $days[$#days];         # same; redundant but explicit
$days[2]  = 'Wednesday';          # set
$days[10] = 'Sunday';             # auto-extends; $days[5..9] become␣
↪undef
```

Reading past the end returns `undef` (no warning - out-of-range read is a non-event). Writing past the end *grows* the array; the intervening slots become `undef`.

```
my @a;
$a[5] = 'fifth';
# @a is now (undef, undef, undef, undef, undef, 'fifth')
print scalar @a;                   # 6
```

Reading a negative index further back than the start is a *fatal* error:

```
my @a = (1, 2, 3);
$a[-10];                            # FATAL: Modification of non-
 ↪creatable
                                    #        array value attempted
```

The whole-array operations

```
push    @arr, $value;               # append at end           -␣
 ↪returns new length
my $v = pop  @arr;                  # remove from end         -␣
 ↪returns removed element
my $v = shift @arr;                 # remove from start       -␣
 ↪returns removed element
unshift @arr, $value;               # prepend                 -␣
 ↪returns new length

splice  @arr, $offset, $len, @replacement;   # general remove/insert
```

`shift` and `unshift` are O(n) because they renumber every remaining element; `push` and `pop` are O(1) amortised. For queue-shaped workloads where this matters, prefer `push`/`shift` on the *short* end or use a deque from `List::Util` (which has reductions, not a deque).

`splice` is the swiss-army knife: remove a stretch of elements, insert a stretch in their place, return what was removed:

```
my @arr = (1, 2, 3, 4, 5);
my @gone = splice @arr, 1, 2;            # @gone = (2, 3); @arr =␣
 ↪(1, 4, 5)
splice @arr, 1, 0, ('a', 'b');            # insert at offset 1, no␣
 ↪removal
                                          # @arr = (1, 'a', 'b', 4,␣
 ↪5)
```

A negative `$offset` counts from the end. A negative `$len` keeps that many trailing elements:

```
splice @arr, -1;                    # remove just the last element
splice @arr, 0, -2;                 # remove all but the last two
```

Slices

A slice extracts (or assigns to) several elements at once. Indices come from a list:

```
my @arr = ('a', 'b', 'c', 'd', 'e');

my @subset = @arr[0, 2, 4];           # ('a', 'c', 'e')
my @middle = @arr[1..3];              # ('b', 'c', 'd')
```

(continues on next page)

(continued from previous page)

```
my @last3  = @arr[-3..-1];           # ('c', 'd', 'e')
my @rev    = @arr[reverse 0..$#arr];  # ('e', 'd', 'c', 'b', 'a')

@arr[1, 3] = ('B', 'D');              # set positions 1 and 3
@arr[1, 3] = @arr[3, 1];              # swap two elements (no temp)
```

A *key/value* slice with % returns index/value pairs (Perl 5.20+):

```
my @arr = ('a', 'b', 'c', 'd', 'e');
my %picked = %arr[1, 3];             # (1 => 'b', 3 => 'd')
```

This is the right shape when you need both the index and the value of selected elements. See *subscript* for the twelve-way matrix of slice forms.

List context vs scalar context

An array in list context yields its elements; in scalar context, it yields its length:

```
my @arr = (10, 20, 30);

my @copy   = @arr;                # list context - three elements
my $length = @arr;                # scalar context - 3
print "got @arr\n";               # list context inside ""    - "got␣
 ↪10 20 30"
print "got " . @arr . "\n";       # scalar context (concat)   - "got␣
 ↪3"
```

The distinction is the most common Perl trap. Two rules of thumb:

- *Inside double-quoted strings*, arrays interpolate as their elements joined by `$"` (a single space by default). See *interpolation*.
- *In a numeric or string operator context* (+, ., comparison), an array yields its length.

Force scalar context with `scalar` when the expression position is otherwise ambiguous:

```
print "got " . scalar(@arr) . " items\n";    # "got 3 items"
print "got @arr items\n";                    # "got 10 20 30 items"␣
 ↪- different!
```

List assignment and the count idiom

Assigning a list to an array makes the array exactly that long:

```
my @arr = (1, 2, 3);
@arr = (10, 20);                  # @arr is now (10, 20)
@arr = ();                        # @arr is now empty
```

Assigning a list to a *list* of scalars unpacks (any extras are discarded; missing ones become undef):

```
my ($x, $y, $z) = (1, 2, 3);              # 1, 2, 3
my ($x, $y, $z) = (1, 2);                 # 1, 2, undef
my ($x, $y)     = (1, 2, 3);              # 1, 2 - '3' silently␣
↪dropped
```

A list assignment in scalar context returns the number of elements on the *right* side. This is the source of the count idiom:

```
my $count = () = $string =~ /\d+/g;
# $string =~ /\d+/g  - match in list context, returns all matches
# () = LIST          - list assignment to empty list
# $count = SCALAR    - that assignment in scalar context = count of␣
↪RHS
```

The `() = LIST` middle is the trick. Without it, `$count = $s =~ /\d+/g` gives a boolean instead of a count.

Array references

Arrays are not first-class values in the sense that they can be passed by name through a chain of function calls without flattening into a list. To pass an array around without flattening, take a **reference** to it:

```
my @items = (1, 2, 3);

my $aref = \@items;                  # reference to the existing @items
my $anon = [1, 2, 3];                # anonymous array reference

$aref->[0]                            # access through the arrow - 1
@{$aref}                              # fully bracketed deref - list␣
↪(1, 2, 3)
@$aref                                # unbracketed deref     - same as␣
↪above
$#{$aref}                             # last index of dereffed array
scalar @$aref                         # length
```

See *references* for the full picture.

Real example: building a histogram of word lengths

```
my @words = qw(the quick brown fox jumps over the lazy dog);

my @hist;
$hist[length $_]++ for @words;

for my $len (1 .. $#hist) {
    next unless $hist[$len];
    printf "%2d-letter words: %d\n", $len, $hist[$len];
}
# 3-letter words: 4
```

(continues on next page)

(continued from previous page)

```
# 4-letter words: 2
# 5-letter words: 3
```

Note `$hist[length $_]++` autovivifies (creates) the slot when it's the first word of that length. Reading from `$hist[7]` later returns `undef`, which `++` happily increments - but `next unless $hist[$len]` skips the gap before printing.

See also

- *Hashes* - when the index is a string, not an integer.
- *References* - passing arrays around without flattening.
- *Context* - the list-vs-scalar story expanded.
- *Interpolation* - how arrays appear inside `""`.
- *Subscript and slice operators* - the bracket-and-brace family in detail.
- *Range operator* `..` - the most common source of slice index lists.
- `push`, `pop`, `shift`, `unshift`, `splice` - the whole-array operations.
- `scalar` - force scalar context (length).
- `reverse`, `sort` - whole-array transformations.

Hashes

A hash is an unordered collection of key-value pairs where keys are strings and values are scalars. The sigil is `%` for the whole hash, `@` for value slices and key/value slices, and `$` for one value:

```
my %user = (
    name  => 'John',
    age   => 30,
    email => 'john@example.com',
);

%user                # the whole hash (six elements: key1, val1, ...
 ↪)
$user{name}          # one value          - 'John'
@user{qw(name age)}  # value slice        - ('John', 30)
%user{qw(name age)}  # key/value slice    - (name => 'John', age =>␣
 ↪30)
keys %user           # the list of keys
```

Hash keys are always coerced to strings. Storing under the integer key `42` and reading under the string key `"42"` retrieve the same slot - they are the same key.

Initialisation: pairs, fat comma, and %h = LIST

A hash is initialised from a list of even length, in key/value order:

```
my %h = ('a', 1, 'b', 2);                    # legal but ugly
my %h = (a => 1, b => 2);                    # idiomatic
```

The fat comma => is a comma that *also* auto-quotes a left-hand bareword that looks like an identifier. So `a => 1` is exactly `'a', 1`. Auto-quoting requires the bareword to be a simple identifier - `2.0 => 'x'` is parsed as the number 2, not the string `"2.0"`:

```
my %h = (a => 1);                  # ('a', 1)          - auto-quoted
my %h = ('a' => 1);                # ('a', 1)          - same
my %h = (2.0 => 'x');              # (2, 'x')          - not ('2.0', 'x
 ↪')!
my %h = ("2.0" => 'x');            # ('2.0', 'x')      - explicit quote
```

If a key appears more than once in the initialiser list, the **last occurrence wins**. The standard idiom for "merge with overrides" exploits that:

```
my %config = (%defaults, %overrides);
# %config has every default key, with %overrides values where they␣
 ↪collide
```

Access, with the four sigil shapes

Reading and writing one slot is $-sigil:

```
my $name = $user{name};                  # read
$user{city} = 'NYC';                     # write
$user{age}++;                            # arithmetic on a hash value
delete $user{email};                     # remove the slot entirely
```

The other three subscript shapes match what you want back:

```
@user{qw(name age city)}                 # ('John', 31, 'NYC')          ␣
 ↪ - values
%user{qw(name age)}                       # (name => 'John', age => 31)␣
 ↪  - pairs
keys %user                                # ('name', 'age', 'city')      ␣
 ↪   - keys
values %user                              # ('John', 31, 'NYC')          ␣
 ↪   - values
```

See *subscript* for the whole twelve-way matrix and the bareword-auto-quote rule inside {}.

`exists` vs `defined` vs truthiness

Three distinct questions about a key:

```
exists  $h{key}      # is the slot present at all?
defined $h{key}      # is the slot present AND non-undef?
        $h{key}      # is the value true (non-empty, non-zero, non-
↪"0")?
```

These differ when:

```
$h{a} = undef;
$h{b} = 0;
delete $h{c};

exists  $h{a}        # TRUE  - slot exists, value is undef
defined $h{a}        # FALSE - value is undef
        $h{a}        # FALSE - undef is false

exists  $h{b}        # TRUE
defined $h{b}        # TRUE
        $h{b}        # FALSE - 0 is false

exists  $h{c}        # FALSE - never set, or deleted
defined $h{c}        # FALSE
        $h{c}        # FALSE
```

Pick by intent: `exists` for *presence*, `defined` for *value-not-undef*, truthiness for *meaningful value*. Reaching for `if ($h{key})` when `exists` was meant is a bug source - `0` and `""` are valid values that the test rejects.

Iteration: `keys`, `values`, `each`, `while (each)`

```
for my $k (keys %user) {
    print "$k = $user{$k}\n";
}

for my $v (values %user) { ... }            # values only

while (my ($k, $v) = each %user) {          # keys + values, one␣
↪pair per call
    print "$k = $v\n";
}
```

`each` carries an iterator state on the hash itself; calling `keys` (or `values`) on the hash *resets* that iterator. Mixing the two produces hard-to-debug looping bugs:

```
while (my ($k, $v) = each %h) {
    if (some_condition($k)) {
        print "size: ", scalar keys %h, "\n";   # resets each()␣
↪iterator!
```

(continues on next page)

(continued from previous page)

```
        # next iteration of while() starts over from the top
    }
}
```

Fix: either accumulate the list with `keys` once at the top, or avoid `each` entirely. Most hash iteration is best written with `keys`:

```
for my $k (sort keys %h) {            # bonus: defined ordering
    ...
}
```

Key ordering

Hash keys come out in **insertion-order-perturbed-by-hash-randomisation** order - i.e. there is no guaranteed order. Two runs of the same program with the same input may iterate keys in different orders. This is a security feature (it prevents algorithmic-complexity attacks against the hash function); it is also a recurring source of brittle tests:

```
my %h = (a => 1, b => 2, c => 3);
print "$_ " for keys %h;        # output order is undefined
```

If you need a stable order, sort:

```
print "$_ " for sort keys %h;          # alphabetical
print "$_ " for sort { $h{$a} <=> $h{$b} } keys %h;   # by value
```

Hash references

A hash, like an array, flattens when passed through a list. To pass a hash without flattening, take a **reference**:

```
my %h = (a => 1, b => 2);

my $href = \%h;                # reference to the existing %h
my $anon = { x => 1, y => 2 }; # anonymous hash reference

$href->{a}                     # access through the arrow      - 1
${$href}{a}                    # fully bracketed deref         - 1
keys %$href                    # deref then keys
%{$href}                       # deref to flat key/value list
```

See *references* for the full picture.

Real example: counting and grouping

A frequency count is the textbook small example for hashes:

```
my @words = qw(apple banana apple cherry apple banana);
my %count;
$count{$_}++ for @words;
# %count = (apple => 3, banana => 2, cherry => 1)
```

Grouping is the next step up - building a hash whose values are arrayrefs:

```
my @people = (
    { name => 'Alice',   dept => 'eng' },
    { name => 'Bob',     dept => 'sales' },
    { name => 'Carol',   dept => 'eng' },
    { name => 'Dan',     dept => 'sales' },
);

my %by_dept;
push @{ $by_dept{$_->{dept}} }, $_->{name} for @people;
# %by_dept = (eng => ['Alice', 'Carol'], sales => ['Bob', 'Dan'])
```

The `push @{ $by_dept{$_->{dept}} }, ...` line autovivifies: when the `eng` key isn't there yet, the hash slot is created and an anonymous array goes into it, then the value is pushed. See *references* for the autoviv story.

Hash slice as multi-value get/set

```
my %config = (host => 'localhost', port => 80, debug => 0);

my ($h, $p) = @config{'host', 'port'};       # ('localhost', 80)

@config{qw(host port debug)} = ('elsewhere', 8080, 1);   # set three
 ↪at once
```

Combine with the `%h{...}` slice for "extract a record":

```
my %fields = %config{qw(host port)};
# %fields = (host => 'localhost', port => 80)
```

This is the canonical way to project a hash down to a subset of its keys. Without `%h{...}` you'd hand-roll it with a loop or with `map`.

See also

- *Arrays* - when the index is an integer.
- *References* - autovivification, hash-of-arrays, hash-of-hashes; the `$h->{a}{b}` shape.
- *Subscript and slice operators* - the bareword-auto-quote rule and the four slice shapes.
- `exists`, `defined`, `delete` - the presence/value/removal trio.
- `keys`, `values`, `each` - whole-hash iteration.

- `sort` - for ordered iteration.
- `tie` - make a hash backed by something other than the in-memory hash table (DBM file, lazy database, ...).

Context

Almost every operator in Perl evaluates its operands in a specific **context** - list, scalar, void, or boolean - and many operators return *different things* depending on the context they were called in. The same expression can mean different things in different positions:

```
my @arr = (10, 20, 30);

my @copy = @arr;          # list context     - three elements
my $len  = @arr;          # scalar context   - 3
print @arr;               # list context     - "102030"
print "@arr";             # list context inside ""  - "10 20 30"
print "" . @arr;          # scalar context (concat) - "3"
@arr;                     # void context - discarded; warns under -w
```

Context is the single biggest difference between Perl and most other languages, and the single most common source of "why doesn't this do what I expect" surprises.

The four contexts

Context	What the operand sees	Imposed by
list	"give me all of your values"	`@a = ...`, `(...)`, function args, `print`
scalar	"give me one value"	`$s = ...`, `if/while`, `.`, `+`, comparison
void	"throw the result away"	a statement whose value isn't used
boolean	"give me a true/false"	`if`, `unless`, `while`, `&&`, `||`, `!`

Boolean context is technically a special case of scalar context - the operand is asked for one value, then that value is tested for truth. The two are not always the same: a `wantarray`-aware sub can detect boolean context separately (see below).

Imposing context

The left side of an assignment imposes context on the right:

```
my @x = func();           # func() is called in LIST context
my $x = func();           # func() is called in SCALAR context
my ($x, $y) = func();     # func() is called in LIST context
my ($x) = func();         # also LIST - note the parens
my  $x  = func();         # SCALAR - same chars, very different
```

The single most-tripped-over case is `my $x = func()` vs `my ($x) = func()`. Without parens, the assignment is a scalar; with them, it's a 1-element list assignment. If `func()` returns three things in list context and "the count" in scalar context, the difference is between getting `'first-thing'` and getting 3.

Operators impose context too. Numeric and string operators impose scalar context on their operands; `print` imposes list:

```
my @arr = (1, 2, 3);
my $n = @arr + 0;          # scalar context - 3 + 0 = 3
my $s = @arr . "";         # scalar context - "3"
print @arr;                # list context - prints "123"
```

And the boolean tests impose scalar (boolean) context:

```
if (@arr)        { ... }  # true if @arr is non-empty
unless (%hash)   { ... }  # true if %hash is empty
while (<$fh>)    { ... }  # context is BOOLEAN, but the diamond op␣
↪is
                          #   special: it reads ONE line and assigns␣
↪to $_
```

What an array yields in scalar context

An **array** in scalar context yields its **length**:

```
my @arr = (1, 2, 3);
my $n = @arr;             # 3
```

A **list literal** in scalar context yields its **last element** (the comma operator):

```
my $x = (1, 2, 3);        # 3   - comma operator, last element
my $x = ('a', 'b', 'c'); # 'c'
```

These look identical and behave differently. The rule is: arrays have a length, list literals don't. A bare comma-list is the C comma operator and produces its last operand; an array variable in scalar position is asking "how many":

```
my @a = ('x', 'y', 'z');
my $count   = @a;         # 3
my $last_el = $a[-1];     # 'z'
```

The `scalar` operator forces scalar context on whatever follows, which is most useful when the expression position is otherwise ambiguous:

```
print "got " . scalar(@arr) . " items\n";     # "got 3 items"
print "got " . @arr . " items\n";              # same - concat␣
↪imposes scalar
print "got @arr items\n";                      # "got 1 2 3 items" -␣
↪list inside ""
```

What a hash yields in scalar context

A non-empty hash in scalar context returns a true value (the count of keys, since Perl 5.25); an empty hash returns the false value 0. The shape that matters is the truth value:

```
if (%h) { ... }          # true if %h has any pairs
my $n = keys %h;         # explicit: count of keys
my $n = %h;              # also count of keys (Perl 5.25+); was a␣
↪debug ratio earlier
```

For older code targeting pre-5.25 perls you sometimes still see `scalar keys %h`; on PetaPerl (target perl 5.44) `scalar %h` returns the key count and is fine.

`wantarray` - write context-sensitive subs

A sub can ask "what context was I called in?" with `wantarray`:

```
sub items {
    return wantarray ? (1, 2, 3)        # list context
         : defined wantarray ? 3        # scalar context
         : do { warn "items() called in void context\n"; () };  #␣
↪void context
}

my @list = items();       # (1, 2, 3)
my $cnt  = items();       # 3
items();                  # void - warning fires
```

The three-way distinction is the only way to detect *void* from inside a sub. Use it sparingly: most subs return one shape and let the caller worry about context coercion. The places where `wantarray` earns its keep are:

- Functions that legitimately return either "the list" or "the count" - e.g. text-search routines.
- Diagnostic helpers that print a warning when their result was discarded.

A sub that returns different *meanings* (not just different shapes) based on `wantarray` is a maintenance hazard. Two named functions read better than one polymorphic one.

Pitfall: `scalar(@arr)` vs `@arr` in different positions

```
my @arr = (1, 2, 3);

# 1. Function arguments - LIST context
some_func(@arr);          # passes 1, 2, 3 as three arguments
some_func(scalar @arr);   # passes 3 as a single argument

# 2. String concatenation - SCALAR context
"got " . @arr             # "got 3"

# 3. Inside double quotes - array interpolation, NOT scalar!
"got @arr"                # "got 1 2 3"

# 4. Boolean test - SCALAR (boolean) context
if (@arr) { ... }         # true iff @arr non-empty
```

(continues on next page)

(continued from previous page)

```
# 5. Comparison - SCALAR context on both sides
@arr == 3                  # 3 == 3 is true
@arr eq "3"                # "3" eq "3" is true; this is rarely what␣
↪you want
```

The mismatch between case 2 (concat → scalar) and case 3 (inside "" → list) is the single most-asked Perl context question. Inside "", `@arr` is array interpolation; outside "", the context of the surrounding expression decides.

Pitfall: list-flattening in function arguments

A function receives `@_` as a single flat list of all its arguments - there is no way for the callee to tell where one array ended and the next began:

```
sub many {
    print "got ", scalar @_, " args\n";
}

my @a = (1, 2);
my @b = (3, 4, 5);

many(@a, @b);              # 5 args - both arrays flattened
many(\@a, \@b);            # 2 args - two array references
many(scalar @a, scalar @b); # 2 args - two integers (the lengths)
```

This is *the* reason references exist: to pass a non-flat shape through the list-flattening calling convention. Anything you want to keep its identity inside `@_` must arrive as a reference.

Real example: a sub that returns count or list

```
sub digits_of {
    my ($s) = @_;
    return $s =~ /\d/g;     # in list context: all digit chars
                            # in scalar context: TRUE/FALSE (last␣
↪match)
}

my @digits = digits_of("a1b2c3");      # ('1', '2', '3')   - three␣
↪matches
my $any    = digits_of("a1b2c3");      # 1                  -␣
↪boolean
```

To make `digits_of` return a *count* in scalar context, the right-hand side has to coerce explicitly:

```
sub digits_of {
    my ($s) = @_;
```

(continues on next page)

(continued from previous page)

```
    my @d = $s =~ /\d/g;
    return wantarray ? @d : scalar @d;
}

my @digits = digits_of("a1b2c3");      # ('1', '2', '3')
my $n      = digits_of("a1b2c3");      # 3
```

This is the canonical "return a list or its length" idiom. Note that without the `wantarray` switch, `return @d` would still give the count in scalar context (because of the array-in-scalar rule) - but it's clearer to be explicit.

See also

- *Arrays* - the array-vs-list-vs-scalar trichotomy expanded.
- *Hashes* - %h in scalar context.
- *Comma* - the operator that builds list literals and yields its last operand in scalar context.
- `wantarray` - the in-sub way to ask.
- `scalar` - force scalar context.
- `reverse` - context-sensitive: a list of scalars in list context, a single reversed string in scalar context.
- *References* - passing a non-flat shape through the flattening list-context arg convention.

Numbers and strings

A scalar can hold a number, a string, or both at once. Perl silently converts between the two whenever an operator demands one shape - + wants numbers, . wants strings, comparison operators have separate numeric and string forms. Understanding the conversion rules and where they bite is essential to writing predictable Perl.

Numeric literals

Integer literals come in five bases:

```
0                   # zero
12345               # decimal
4_294_967_296       # underscores for legibility (decimal 4.29␣
↪billion)
0xff                # hex      - 255
0Xdead_beef         # hex with underscores
0b01_1011           # binary   - 27
0377                # octal    - 255 (leading 0, NOT followed by␣
↪[bBxX])
0o12_345            # octal, alternative spelling (5.33.5+)
```

Float literals require either a dot or an exponent:

```
3.14
0.5                 # leading zero ok
.5                  # leading dot ok
1e3                 # 1000
1.5E-10             # exponent - uppercase or lowercase 'e'
```

Underscores are accepted between digits anywhere - `3.14_15_92`, `1_000_000`. Multiple consecutive underscores warn: `23__500` warns, `23_500` does not.

Hex and octal floats need the p (power-of-2) exponent:

```
0x1.999ap-4         # hex float
0b101.01p-1         # binary float
0o7.65p2            # octal float
```

The p exponent is *required* for non-decimal floats. Plain `0x1F.5` is a parse error.

String → number

When Perl converts a string to a number, it consumes the longest prefix that looks like a number and ignores the rest:

```
"42"        + 0     # 42
"42abc"     + 0     # 42      - leading digits parsed
"abc42"     + 0     # 0       - no leading digits
" \t  42"   + 0     # 42      - leading whitespace ok
"4_2"       + 0     # 4       - underscores in literals only, NOT␣
↪strings
"0xFF"      + 0     # 0       - hex prefix NOT auto-recognised
"3.14e2"    + 0     # 314     - scientific notation parsed
"Inf"       + 0     # Inf     - case-insensitive infinity
"NaN"       + 0     # NaN
```

The mismatch between numeric *literals* (which honour `_`, `0x`, `0b`, `0`) and numeric *strings* (which honour none of those) is the single most-tripped-over conversion rule. To convert a string that's in non-decimal form, use `hex` or `oct`:

```
my $bytes = hex "FF";          # 255
my $bytes = hex "0xFF";        # 255  - hex() ignores the 0x
my $perms = oct "0755";        # 493
my $perms = oct "0o755";       # 493  - explicit octal prefix
my $bytes = oct "0xFF";        # 255  - oct() recognises any prefix
```

Under `use warnings`, an arithmetic operation on a string that doesn't *start* with a number prefix emits *Argument "..." isn't numeric in addition (+)*. The result is still 0; the warning is the signal. Writing tolerant code that accepts user input often means filtering with `Scalar::Util::looks_like_number` or a regex first.

Number → string

Perl converts a number to a string using the same logic as `sprintf("%g", $n)` for floats and the obvious decimal representation for integers:

```
my $i = 42;              "$i"   # "42"
my $f = 3.14;            "$f"   # "3.14"
my $z = 0.0;             "$z"   # "0"        - not "0.0"
my $b = 1e10;            "$b"   # "10000000000"
my $s = 1e20;            "$s"   # "1e+20"    - switches to scientific
 ↪past ~15 digits
my $n = "Inf" + 0;       "$n"   # "Inf"
```

The default precision is enough digits to roundtrip *most* doubles. For exact control, format explicitly with `sprintf`:

```
sprintf "%.2f", 3.14159          # "3.14"
sprintf "%05d", 42               # "00042"
sprintf "%e", 1234567            # "1.234567e+06"
```

Concatenation forces strings; arithmetic forces numbers

```
"42" + "3"           # 45    - both coerced to numbers
42 . 3               # "423" - both coerced to strings
"3" x "5"            # "33333" - left operand string, right operand
 ↪number
```

The `.` operator is the explicit-stringify operator; `+` is the explicit-numerify operator. A scalar that holds a number you want as a string can be concatenated with `""`:

```
my $s = "" . $n;     # idiomatic "stringify"
my $s = "$n";        # same; usually clearer
```

A scalar that holds a string you want as a number adds 0:

```
my $n = $s + 0;      # idiomatic "numify"
```

Numeric vs string comparison

Two parallel families of comparison operators with **different semantics**:

Numeric	String	Meaning
`==`	`eq`	equal
`!=`	`ne`	not equal
`<`	`lt`	less
`<=`	`le`	less or equal
`>`	`gt`	greater
`>=`	`ge`	greater or equal
`<=>`	`cmp`	three-way compare

The sort order is dramatically different on values that look like numbers but are large:

```
"10" <  "9"          # FALSE - 10 < 9 numerically
"10" lt "9"          # TRUE  - '1' < '9' lexicographically
```

Sort numbers numerically:

```
my @sorted = sort { $a <=> $b } @numbers;       # numeric
my @sorted = sort                @strings;       # default - string
my @sorted = sort { $a cmp $b }  @strings;       # explicit string
```

Picking the wrong operator is the source of "my sort is wrong" bugs. See *numeric comparison* and *string comparison*.

Floating-point: precision, Inf, NaN

Most Perls use IEEE 754 double-precision floats - about 15–17 decimal digits of precision. The classic surprise:

```
0.1 + 0.2 == 0.3      # FALSE - neither side is exactly␣
↪representable
                      # 0.1 + 0.2 = 0.30000000000000004
```

For money or anything that *must* be exact, work in integers (cents instead of dollars) or use a bignum module:

```
use bignum;            # auto-promotes to arbitrary precision
0.1 + 0.2 == 0.3      # TRUE under bignum

use Math::BigInt;
my $big = Math::BigInt->new("10000000000000000000") + 1;
```

`Inf` and `NaN` are real values:

```
my $inf = 9 ** 9 ** 9;       # Inf - overflow
my $nan = $inf - $inf;       # NaN - undefined arithmetic
my $nan = "NaN" + 0;         # NaN - accepted from string

$nan == $nan                  # FALSE - NaN never equals anything
$nan != $nan                  # TRUE  - including itself
```

`Inf` and `NaN` are case-insensitive on input and have several spellings (`Infinity`, `1.#INF`, ...); on output Perl normalises to the short forms.

Note that perl5 generates *fatal* errors for `1/0` and `sqrt(-1)`; those don't produce `NaN`. They are exceptional, not arithmetic.

Locale-affected number parsing

Under `use locale` plus an active POSIX locale, the decimal point is whatever the locale says. This affects parsing and stringifying of floats:

```
use locale;
use POSIX qw(setlocale LC_NUMERIC);
setlocale LC_NUMERIC, 'de_DE.UTF-8';   # German uses ',' as decimal

my $x = 3.14;
print "$x\n";                            # "3,14"  - locale␣
 ↪formatting
"3,14" + 0                                # 3.14    - locale parsing
```

This is rarely what you want for data interchange. Most code that reads or writes structured data should explicitly *not* be in the `use locale` scope, so that `.` is always the decimal point. Use locale formatting only at the boundary where output is for a human reader.

"Looks like a number"

The rules for whether a string is "numeric enough" are not simple to inline; use `Scalar::Util::looks_like_number`:

```
use Scalar::Util qw(looks_like_number);

looks_like_number("42")         # true
looks_like_number("3.14e10")    # true
looks_like_number("Inf")        # true
looks_like_number("NaN")        # true
looks_like_number("0xFF")       # FALSE - hex strings are not
 ↪"numeric"
looks_like_number("")           # FALSE - empty string
looks_like_number(undef)        # FALSE
looks_like_number("3.14abc")    # FALSE - trailing junk
```

This is the validator function - strict, no leading garbage, no trailing garbage, no hex/oct prefixes. It matches the conditions under which Perl will use a number's cached numeric form rather than re-parsing on every arithmetic op.

See also

- *Scalars* - the dual-form / dualvar story; truth values.
- *Numeric comparison* - ==, <=>.
- *String comparison* - eq, cmp.
- *Arithmetic* - +, -, *, /, %, **.
- *String* - . (concat), x (repetition).
- `hex`, `oct` - convert non-decimal numeric strings.
- `int`, `abs`, `sprintf` - explicit numeric conversion and formatting.
- `Scalar::Util` - `looks_like_number`, `dualvar`.

References

A reference is a scalar that *points to* another value. References are how Perl builds nested data structures - arrays of arrays, hash of hashes, hash of arrays of hashes - because the basic container types (arrays, hashes) flatten when they appear in lists, and a reference is a single scalar that survives the flattening.

```
my @arr = (1, 2, 3);
my $ref = \@arr;              # take a reference to @arr
$$ref[0]                      # full deref form     - 1
$ref->[0]                     # arrow shorthand     - 1
@$ref                         # whole-array deref   - (1, 2, 3)
```

A reference is itself a scalar. It can be passed through any list-flattening calling convention without losing its identity, stored in arrays and hashes, returned from subs, weakened, serialised - anywhere a scalar can go.

Things you can reference

Seven things take a reference:

```
\$scalar              # SCALAR ref
\@array               # ARRAY ref
\%hash                # HASH ref
\&subroutine          # CODE ref
\*glob                # GLOB ref
\\$scalar             # REF (a reference to a reference)
\$file_handle         # IO ref (reference to a filehandle)
```

Plus the **anonymous** constructors, which build the value and return a reference in one step:

```
my $aref = [1, 2, 3];                  # anonymous array
my $href = { name => 'Alice', age => 30 };  # anonymous hash
my $cref = sub { $_[0] * 2 };          # anonymous sub
```

The bracket-vs-paren distinction is essential: `(1, 2, 3)` is a **list**, `[1, 2, 3]` is an **anonymous array reference**. The first flattens; the second does not.

Dereferencing

Four spellings, in increasing order of how often they appear in modern code:

```
${$sref}              # full sigil-deref form
$$sref                # short sigil-deref form (only if name is␣
↪simple)
$sref->{key}          # arrow form (postfix; for arrays and hashes)
$sref->[0]            # same, for arrays
```

The `->` arrow form is the idiomatic spelling for nested access:

```
$config->{database}{host}            # arrow elision between
 ↪subscripts
$config->{database}->{host}          # explicit; same meaning
${$config->{database}}{host}         # full form; the same again
```

Between two consecutive subscripts the `->` is **optional** (the "arrow elision" rule); at the head of a chain it is required:

```
$ref->[0]                        # required arrow at the head
$ref->[0][1]                     # second arrow elided
$ref->[0]->[1]                   # equivalent
[1,2,3]->[0]                     # required arrow on an anonymous ref
```

Whole-container deref:

```
@$aref                           # the whole array
%$href                           # the whole hash, as flat key/value
 ↪list
&$cref(@args)                    # call the sub
$cref->(@args)                   # idiomatic call form
```

Use the curly-braced form `@{ EXPR }` / `%{ EXPR }` when the deref target is an expression more complex than a bare scalar variable:

```
my @found = @{ $config->{users} };   # deref a chain - needs braces
```

See *arrow* for the operator and *subscript* for the bracket family.

Precedence of prefix deref versus subscript

The five prefix dereference sigils `$ @ * % &` bind **tighter** than the postfix subscript brackets `[]` and `{}`. So these two are exactly the same expression:

```
$$aref[2][2]                     # confusing, but valid
$aref->[2][2]                    # clear; identical meaning
```

`$$aref[2][2]` dereferences `$aref` *first*, then subscripts the result. It is **not** the C reading `*a[i]`, where the subscript would apply before the dereference. Perl has no construct that subscripts before dereferencing at that position; if you want "the thing the `$i`-th element points at", spell it out left to right with a postfix deref:

```
$aref->[$i]->$*                  # deref the i-th element explicitly
```

The arrow form is the readable choice for nested access; the leading-sigil form is a trap precisely because it reads as the C precedence and means the opposite. See *precedence* for the full operator table.

What `ref` returns

`ref` reports the kind of thing a reference points to:

```
ref \$scalar          # 'SCALAR'
ref \@array           # 'ARRAY'
ref \%hash            # 'HASH'
ref \&sub             # 'CODE'
ref \*FH              # 'GLOB'
ref \\$scalar         # 'REF'    - a ref-to-ref
ref \$io_handle       # 'IO'     - wrapper around a file handle

ref [1, 2, 3]         # 'ARRAY'
ref { a => 1 }        # 'HASH'
ref sub {}            # 'CODE'

ref bless {}, 'Foo'   # 'Foo'    - blessed; class name not 'HASH'
ref 42                # ''       - not a reference at all
ref undef             # ''
```

For blessed objects, `ref` returns the class. To distinguish a blessed hash-ref from a plain hash-ref, use `Scalar::Util::reftype` (which always returns the underlying container shape regardless of bless) or `isa` (the inheritance-aware check).

Postfix dereference (@*, %*, $*)

Perl 5.20+ added a postfix syntax that mirrors the arrow chain:

```
$ref->@*              # equivalent to @$ref
$ref->%*              # equivalent to %$ref
$ref->$*              # equivalent to $$ref

$ref->@[1, 3]         # array slice through ref - equivalent to @{
 ↪$ref}[1,3]
$ref->%[1, 3]         # key/value slice
$ref->@{qw(a b)}      # hash slice
```

The postfix forms read left-to-right with the rest of an arrow chain, which is often clearer when the deref target is itself a chain:

```
$config->{users}->@*          # all users - chains naturally
@{$config->{users}}           # same; reads right-to-left
```

Pick the form that reads better in context. Postfix is the clearer choice in chains; the leading-sigil form is shorter for plain scalar refs.

Autovivification

Reading or writing through a deref into a slot that doesn't exist yet auto-creates the intermediate structure:

```
my %h;
$h{a}{b}{c} = 1;
# %h is now (a => { b => { c => 1 } })  - three nested hashes built
```

The mechanism: when the assignment target is `$h{a}{b}{c}`, Perl sees `$h{a}` is `undef`, the next subscript is `{b}` (a hash subscript), so it makes `$h{a}` a fresh hash ref. Then `$h{a}{b}` is `undef`, next subscript is `{c}` (also hash), so make it a fresh hash ref. Then assign 1 to the leaf.

The same happens for array subscripts:

```
my %by_dept;
push @{ $by_dept{eng} }, 'Alice';
# $by_dept{eng} did not exist; the push autoviv'd it as an arrayref
```

Autoviv on the **left** of an assignment (or a `push`/`pop`/... mutator) is what makes this work. Autoviv on the **right** of an expression is more dangerous - *reading* `$h{a}{b}{c}` when `$h{a}` is `undef` *also* autovivifies, populating the hash as a side effect of inspection:

```
my %h;
exists $h{a}{b};          # accidentally creates $h{a} = {} as a␣
↪hash!
                           # %h is now (a => {})
```

The fix when you genuinely just want to test: chain `exists`:

```
exists $h{a} && exists $h{a}{b};        # short-circuits if $h{a}␣
↪absent
```

This is the most-bitten autoviv trap. Tests of nested existence must short-circuit, or they leave hash slots behind.

Hash of arrays - the canonical example

```
my @people = (
    { name => 'Alice', dept => 'eng'   },
    { name => 'Bob',   dept => 'sales' },
    { name => 'Carol', dept => 'eng'   },
    { name => 'Dan',   dept => 'sales' },
);

# Group by department:
my %by_dept;
for my $p (@people) {
    push @{ $by_dept{ $p->{dept} } }, $p->{name};
}
# %by_dept = (
#     eng   => ['Alice', 'Carol'],
#     sales => ['Bob', 'Dan'],
# )
```

(continues on next page)

(continued from previous page)

```
# Iterate:
for my $dept (sort keys %by_dept) {
    print "$dept: @{ $by_dept{$dept} }\n";
}
# eng: Alice Carol
# sales: Bob Dan
```

The `@{ $by_dept{...} }` is the deref-with-braces form. It's necessary because the deref target is an expression (`$by_dept{$dept}`), not a bare variable.

The same shape with the postfix form:

```
print "$dept: ", join(' ', $by_dept{$dept}->@*), "\n";
```

Pick whichever you find clearer; both are idiomatic in modern Perl.

Weak references and circular structures

A reference holds the referent alive (refcount-bumped) until the reference itself goes out of scope. This becomes a problem when two structures point at each other:

```
my $parent = { name => 'p' };
my $child  = { name => 'c', parent => $parent };
push @{ $parent->{children} }, $child;

# parent refers to child via the children arrayref
# child refers back to parent via the parent slot
# When both go out of scope, neither's refcount can reach 0 - leak.
```

The fix is a **weak reference** - a reference that doesn't bump the refcount. The `$child->{parent}` slot is the one that should be weakened (the back-pointer):

```
use Scalar::Util qw(weaken);

my $parent = { name => 'p' };
my $child  = { name => 'c', parent => $parent };
weaken $child->{parent};                # break the cycle
push @{ $parent->{children} }, $child;
```

After weakening, `$child->{parent}` reads as the parent while it exists, and silently becomes `undef` when the parent is freed. The canonical place for `weaken` is on every back-pointer in a tree structure where children point back at their parent.

The `Scalar::Util::isweak` predicate tests whether a given reference is weakened.

Reference equality

Two references are equal (numerically ==, stringwise eq) if and only if they refer to the **same** underlying value:

```
my @a = (1, 2, 3);
my $r1 = \@a;
my $r2 = \@a;
my $r3 = [1, 2, 3];      # different anonymous array

$r1 == $r2               # TRUE - same address
$r1 == $r3               # FALSE - different array, even though
 ↪contents match
```

Numeric comparison of refs effectively compares their machine address. Two refs with the same content but different identities compare unequal. Deep-equal needs a helper from `Test::More` or a hand-rolled walker.

A reference stringifies to something like `ARRAY(0x55ab12cd)` - class name plus hex address. Don't rely on the format; use it only as a debug print.

See also

- *Arrays*, *Hashes* - the two containers most references point at.
- *Typeglobs* - `*name`, the symbol-table entry that references-to-globs let you alias.
- *Arrow operator* - `->`, the deref shorthand.
- *Subscript* - `[]` and `{}` through a ref.
- `ref` - what kind of thing does this point at.
- `bless` - make a reference into an object.
- `Scalar::Util` - `weaken`, `isweak`, `reftype`, `blessed`, `looks_like_number`.
- *References tutorial* - step-by-step companion to this terse reference.
- *Weak refs tutorial* - the cycle problem in depth.

Typeglobs

A typeglob is a single name that holds *every* variable type with that name in a package - the scalar `$foo`, the array `@foo`, the hash `%foo`, the sub `&foo`, the filehandle `foo`, and the format `foo` all share one symbol-table entry, and `*foo` is the way to talk about that entry as a value. The sigil is `*`, deliberately chosen because the asterisk reads as "all".

```
*foo                # the typeglob - all of $foo, @foo, %foo, &foo,
 ↪etc.
*foo{SCALAR}        # ref to $foo
*foo{ARRAY}         # ref to @foo
*foo{HASH}          # ref to %foo
*foo{CODE}          # ref to &foo
*foo{IO}            # the filehandle, if foo is one
*foo{NAME}          # the name "foo" - a string
*foo{PACKAGE}       # the package name - a string
```

In modern Perl, typeglobs come up rarely. References cover most of what they used to be needed for. The remaining legitimate uses are package aliasing, filehandle creation in older idioms, and a few introspection hooks.

What a typeglob does

The single most useful sentence about typeglobs:

> Assigning to a typeglob makes one name mean another name.

```
*foo = *bar;        # $foo, @foo, %foo, &foo, FH foo all alias $bar,␣
↪@bar, %bar, &bar, FH bar
```

After that assignment, modifying `$foo` modifies `$bar`, calling `foo()` calls `bar()`, and so on for every type. The two names are now genuinely the same storage - not a copy.

Assigning a *reference* to a typeglob aliases only that one type:

```
our @items;
*shortcut = \@items;     # only @shortcut aliases @items; $shortcut␣
↪and %shortcut unchanged
```

This is how the `Exporter` module installs imported subs into the caller's package: `*caller_package::name = \&exporter_package::name`. Each imported function gets one slot of one glob assigned, so the rest of the destination package's variables under that name are untouched.

*name = \&sub - the standard import idiom

```
package My::Util;
sub greet { "Hello, $_[0]!" }

package main;
*greet = \&My::Util::greet;       # import greet into main::
print greet("world");              # "Hello, world!"
```

This is the mechanism that `use` invokes via `Exporter::import` - the importing package gets a glob assignment for each requested name. `Exporter` does it; you can do it by hand for cases `Exporter` doesn't fit.

Local glob aliases

`local *name = \$thing` is a temporary alias that auto-restores when the enclosing scope exits:

```
our $config = "default";

sub run_with_config {
    my ($alt_config, $code) = @_;
    local *config = \$alt_config;        # only inside this sub
    $code->();                            # any read of $config␣
```

(continues on next page)

(continued from previous page)

```
 ↪reads $alt_config
}

run_with_config("test", sub {
    print "config is $config\n";          # "config is test"
});

print "config is $config\n";              # "config is default" -␣
 ↪restored
```

This is a heavyweight mechanism that has largely been replaced by plain `local $config = $alt_config;` for scalars and explicit parameter-passing for everything else. The case where it still matters is when you need to alias a *filehandle* across the call of arbitrary user code (see below).

Filehandle typeglobs

Before Perl 5.6's "open `my $fh`" idiom, filehandles had to be named by package globals or passed as typeglobs:

```
# Old style - bareword filehandle in a glob slot
open FH, '<', $path or die $!;
while (<FH>) { ... }
close FH;

# Pass to a sub:
sub print_to { my $fh = shift; print $fh "stuff\n" }
print_to(\*FH);                           # ref-to-glob

# Modern style:
open my $fh, '<', $path or die $!;
while (<$fh>) { ... }
close $fh;
print_to($fh);                            # plain scalar ref to IO
```

The modern style is preferred everywhere new code is written. The typeglob-filehandle form survives in three places:

1. **The standard handles** - `STDIN`, `STDOUT`, `STDERR` are bareword filehandles backed by the typeglobs `*STDIN`, `*STDOUT`, `*STDERR`. You see these in error messages and reset code:

   ```
   print STDERR "error\n";
   *STDOUT = *STDERR;          # redirect all STDOUT to STDERR for␣
    ↪the rest of the program
   ```

2. **Returning newly-created filehandles** - before lexical filehandles, the way to return a fresh handle from a sub was to `local *FH` and return `*FH`:

```
sub new_handle {
    my $path = shift;
    local *FH;
    open FH, $path or return;
    return *FH;             # the glob outlives the local␣
↪because it was returned
}
```

Modern equivalent:

```
sub new_handle {
    my $path = shift;
    open my $fh, $path or return;
    return $fh;
}
```

3. **Selecting a different default output handle** - `select` returns and accepts a filehandle, in glob-or-string form:

```
my $old = select(STDERR);    # subsequent print without␣
↪explicit fh writes to STDERR
print "errors-only output\n";
select($old);                  # restore
```

*foo{THING} - the slot accessors

`*foo{THING}` returns a reference to one slot of the glob, or `undef` if that slot is empty:

```
*foo{SCALAR}        # \$foo  if $foo has been touched
*foo{ARRAY}         # \@foo  if @foo has been touched
*foo{HASH}          # \%foo  if %foo has been touched
*foo{CODE}          # \&foo  if &foo defined
*foo{IO}            # the IO handle, or undef
```

The `*foo{IO}` form is the one place you regularly see `THING` syntax in modern code. It's how you ask "is this scalar holding a filehandle":

```
sub is_handle {
    my $thing = shift;
    return defined ref($thing) && (
           ref($thing) eq 'GLOB'
        || (ref(\$thing) eq 'GLOB' && defined *$thing{IO})
    );
}
```

The standard introspection helper that wraps this is `Scalar::Util::openhandle`.

`*foo{SCALAR}` has a quirk: it always returns *something* - the package's `$foo` slot is created on demand. The other accessors return `undef` when the slot is empty.

What replaces typeglobs

For the patterns where typeglobs used to be the only answer, the modern alternatives are:

Old typeglob pattern	Modern replacement
Pass a filehandle to a sub	Lexical FH: `open my $fh, ...`
Return a filehandle from a sub	Lexical FH: `return $fh;`
Alias `@there` as `@here`	`our @here; *here = \@there;` (still typeglob, no replacement)
Pass an array by reference	`\@arr` - pass the reference
Install an imported sub	`*name = \&Source::name` (still typeglob) - or use `Exporter`
Localised global alias	`local $var = ...` for plain scalars

The `*here = \@there` package-aliasing pattern is the one that typeglobs still own outright - there's no "alias one package variable to another" mechanism that doesn't go through a glob. Inside `Exporter`-using modules that mostly happens automatically.

See also

- *References* - the modern alternative to globs for most "indirect access" patterns.
- *Scalars* - the dynamic-typing story. Globs are a separate type: `ref \*FH` returns `'GLOB'`.
- `local` - the safe way to install a temporary glob alias.
- `select` - the selected-output-handle function that takes glob or string handle names.
- `Scalar::Util` - `openhandle`, `reftype`, `blessed` for runtime introspection.
- `Symbol` - generate fresh anonymous globs and qualify package names. (No local reference page yet.)

Interpolation

Double-quoted strings, here-docs, regex patterns, the `s/.../.../` replacement, and `qq//` all *interpolate*: they expand variables and certain escape sequences in place. Single-quoted strings, `q//`, and the LHS pattern of `tr/.../.../` do not. Knowing which constructs interpolate and exactly which forms get expanded is one of the most useful pieces of Perl knowledge.

```
my $name = "Alice";
my @nums = (1, 2, 3);

"hello, $name"              # "hello, Alice"
'hello, $name'              # 'hello, $name' - single quotes don't␣
↪interpolate
```

(continues on next page)

(continued from previous page)

```
"@nums in a string"          # "1 2 3 in a string"
"\$name is $name"            # "$name is Alice"
```

What interpolates

Inside "" (and the other double-quotish contexts), Perl expands exactly two shapes of variable:

```
$name               # scalar - the simple form
@list               # array  - joined by $" (one space by default)
```

Plus the bracket-and-brace subscript forms that compose on top of those:

```
$arr[0]             # one element of @arr
$h{key}             # one value from %h
$ref->[0]           # via array ref
$ref->{key}         # via hash ref
@arr[1, 2]          # array slice
@h{qw(a b)}         # hash slice
```

Note that **whole hashes do not interpolate**:

```
"hash is %h"        # "hash is %h" - literal '%' is fine, '%h' is NOT␣
↪magical
```

If you want a hash dumped into a string, format it explicitly with `Data::Dumper` or a manual `join`.

What does *not* interpolate

```
&sub                # function calls don't interpolate
sub_with_args($x)   # neither do calls with arguments
$h->method          # method calls don't interpolate
$obj->{a} + 1       # arithmetic doesn't interpolate
$arr[0+1]           # this DOES interpolate as $arr[0+1] - subscripts␣
↪allow expressions

"\u\L$name"         # case-modifying escapes - these DO interpolate
"\x{2603}"          # numeric escapes - DO interpolate (a snowman)
```

The rule of thumb: **variables and their subscripts interpolate; function/method calls and arithmetic on top of them do not.** When you want an expression interpolated, see the `@{[ ... ]}` and `${\ ... }` constructs below.

The `${name}` braces

Curly braces around a variable name disambiguate where the name ends:

```
my $who = "Larry";

"${who}speak"          # "Larryspeak"
"$whospeak"             # "" with warning: $whospeak is␣
↪uninitialised
"${who}::sub"           # "Larry::sub"
"$who::sub"             # the variable $who::sub from package who -␣
↪bug
```

Without the braces, Perl is greedy: `$whospeak` reads as one identifier, `$who::sub` reads as a package-qualified name. The braces force the parser to stop at `}`.

This is not the same brace-form as a hash subscript:

```
"$h{key}"              # hash element                  - what most␣
↪people mean
"${h}{key}"            # hash element in a different parse path -␣
↪same result
"${h{key}}"            # also the same
"${\(\$h{key})}"        # the absurd "interpolate an expression"␣
↪form
```

The `${name}` braces are mostly used for the disambiguation case (name followed by alphanumerics that would otherwise be eaten), not for hash access.

The `@{...}` and `${...}` deref-with-braces forms

When you want to *dereference* something inside a string, the brace form lets the deref expression be arbitrary:

```
my $config = { users => ['Alice', 'Bob'] };

"users: @{ $config->{users} }"     # "users: Alice Bob"
"users: $config->{users}"          # "users: ARRAY(0x...)" - wrong;␣
↪whole arrayref stringified
"users: @$config->{users}"         # parse error / wrong
```

The `@{ EXPR }` form deref's `EXPR` as an array. The braces are **required** because the deref target is an expression, not a plain variable. `@$config->{users}` doesn't parse the way intended - `@$config` would deref `$config` as an array, and `->{users}` is a hash-element access on what's left.

The same for `${ EXPR }` (scalar deref) and `%{ EXPR }` (the hash deref, which returns a flat list, rarely useful in interpolation).

`@{[ expr ]}` - interpolate any list expression

To interpolate the result of an arbitrary expression, wrap it in `@{[ ... ]}`:

```
my $name = "World";
print "hello, @{[ uc $name ]}!\n";              # "hello, WORLD!"
```

(continues on next page)

(continued from previous page)

```
print "found @{[ scalar @hits ]} matches\n";      # interpolate a␣
↪count

print "users: @{[ join ', ', @users ]}\n";        # interpolate a␣
↪join
```

The mechanism: `[ ... ]` builds an anonymous array reference whose contents are the expression's list-context value; `@{ ... }` then deref's that arrayref back into a list, which interpolates as a list does (joined by `$"`).

This is the canonical workaround for "I want a function call inside a double-quoted string". The alternative is concatenation:

```
print "hello, " . uc($name) . "!\n";              # equivalent
```

The `@{[ ... ]}` form is denser when there are multiple interpolations; concatenation is clearer when there's just one. In templates with many fields, `printf` or `sprintf` is usually clearer still:

```
printf "found %d matches\n", scalar @hits;
```

Use `@{[ ... ]}` when you genuinely want a Perl expression in the middle of a string and the alternatives are uglier.

`${\ expr }` - interpolate a scalar expression

The scalar counterpart of `@{[ ... ]}`:

```
print "answer: ${\ 6 * 7 }\n";                    # "answer: 42"
print "ts: ${\ scalar localtime }\n";              # "ts: Mon Apr 27␣
↪..."
```

`\ EXPR` makes a reference to the scalar value of `EXPR`; `${ ... }` deref's it. The result is one value, interpolated as a scalar.

For most cases the `@{[ ... ]}` form is more uniform - it works for both scalar and list contexts. `${\ ... }` is occasionally clearer when you specifically want scalar context:

```
"got @{[ items() ]} items"      # items() called in LIST context
"got ${\ items() } items"        # items() called in SCALAR context
```

Hash-subscript ambiguity in patterns

Inside a regex pattern (which is double-quotish), the parser has to guess whether `$foo[bar]` means "interpolate `$foo[bar]` (one element of `@foo`)" or "interpolate `$foo` and then match a character class `[bar]`":

```
my @foo = ('hi');
$x =~ /$foo[abc]/                  # @foo exists - parsed as
```

(continues on next page)

(continued from previous page)

```
 ↪$foo[abc] (one element)
$x =~ /$foo[abc]/                  # @foo doesn't exist - parsed as
 ↪$foo + char class
```

The disambiguation is the brace form:

```
$x =~ /${foo}[abc]/                # $foo + character class [abc]
$x =~ /$foo[5]/                    # element 5 of @foo
$x =~ /\Q$foo\E[abc]/              # $foo (literal) + character class
```

The `\Q...\E` quoter disables metacharacter interpretation inside its scope, which is the cleanest fix when the variable is a user input.

Heredocs

Heredocs are double-quotish unless the marker is single-quoted:

```
my $msg = <<EOM;
Hello, $name!
The hash is %h.
EOM
# "Hello, Alice!\nThe hash is %h.\n" - $name interpolated, %h
 ↪literal

my $msg = <<'EOM';
Hello, $name!
EOM
# "Hello, $name!\n" - single-quoted marker disables interpolation
```

Indented heredocs (Perl 5.26+) ignore leading whitespace up to the shortest run on any line:

```
my $msg = <<~EOM;
    Hello, $name!
    EOM
# "Hello, Alice!\n" - leading "    " stripped from every line
```

The ~ form is preferred in modern code; the un-indented form is ugly inside loops and method bodies.

Real example: building HTML safely

```
my @items = ('Apples', 'Bananas', 'Cherries');

my $html = <<~HTML;
    <ul>
    @{[ map { "  <li>$_</li>" } @items ]}
    </ul>
    HTML
```

The `map` returns a list of `<li>...</li>` strings; `@{[ ... ]}` interpolates them; the implicit `$"` is a single space, giving:

```
<ul>
  <li>Apples</li>   <li>Bananas</li>   <li>Cherries</li>
</ul>
```

To get newlines between the items, set `$"` locally:

```
my $html = do {
    local $" = "\n";
    <<~HTML;
        <ul>
        @{[ map { "  <li>$_</li>" } @items ]}
        </ul>
        HTML
};
```

Or join explicitly:

```
my $li = join "\n", map { "  <li>$_</li>" } @items;
my $html = "<ul>\n$li\n</ul>\n";
```

The explicit `join` is usually clearer for non-trivial templates. The `@{[ ... ]}` form earns its place when the template is mostly literal text with one or two embedded computations.

What the special variable $" controls

The list separator in interpolation:

```
my @x = (1, 2, 3);
print "x = @x\n";                          # "x = 1 2 3" - default $"
↪is " "

local $" = ", ";
print "x = @x\n";                          # "x = 1, 2, 3"

local $" = "\n  ";
print "items:\n  @x\n";                    # "items:\n  1\n  2\n  3"
```

`$"` is the canonical way to control how arrays interpolate without rewriting the string. See *filehandle and printing state* for the family of separator variables.

See also

- *Scalars*, *Arrays*, *Hashes* - the shapes that interpolation expands.
- *References* - the `@{ EXPR }` deref form and what the `\` in `${\ ... }` does.
- `$"` - the list separator for array interpolation.
- `$,`, `$\`, `$/` - the related separator variables for `print` and input.

- `sprintf`, `printf` - the explicit format-string alternative when the template is mostly punctuation.
- `qq`, `q`, `qw` - the quote-like operators with custom delimiters.
- *Scalars* - `$x`. Numbers, strings, references; `undef`; truth and definedness; the dual numeric/ string nature of scalars.
- *Arrays* - `@arr`. Element access, `$#arr`, negative indices, slices, list assignment, the `@arr` / `$arr[i]` / `@arr[i,j]` sigil rule.
- *Hashes* - `%h`. Key-value pairs, the four flavours of subscript, `keys`/`values`/`each`, key ordering, hash slices and key/value slices.
- *Context* - list, scalar, void, boolean. How each operator and assignment imposes context, what `wantarray` reports, the `scalar()` operator, and the recurring traps with `,` versus list construction.
- *Numbers and strings* - automatic conversion both ways, integer vs float, `Inf` and `NaN`, numeric literal forms, "0 but true", `dualvar`, locale and precision pitfalls.
- *References* - `\X`, the seven things you can reference, the four ways to deref, autovivification, weak references, circular structures, `ref` and `Scalar::Util`.
- *Typeglobs* - `*name`. The symbol-table entry, aliasing, `*foo{THING}`, when modern Perl still needs them and when references are the right answer instead.
- *Interpolation* - what double-quoted strings, here-docs, and regex patterns interpolate; the `${name}` braces; the `@{[ expr ]}` and `${\ expr }` baroque constructs; ambiguities with `[` and `{` after a sigil.

Cross-cutting concepts

A handful of ideas appear on every page below. Read these once and the rest will feel familiar:

- **Sigils mark the *result*, not the container.** `@arr[1, 3]` is not "two arrays"; the `@` says "the result is a list". The variable named `arr` is the same in `@arr`, `$arr[0]`, `@arr[0..3]`, and `%arr[0..3]`.
- **Context coerces.** A scalar evaluated in list context becomes a one-element list; a list evaluated in scalar context yields its **last** element (comma operator), but an *array* in scalar context yields its **length**. The two cases are not the same; see *context*.
- **References are scalars.** A reference to anything fits in a scalar variable, which is what makes nested data structures possible. The deref syntax (`@$ref`, `$ref->[0]`, `${$ref}{key}`) is the inverse of `\`.
- **`undef` is a value, not the absence of one.** A scalar can hold the value `undef`; testing `defined $x` is how you ask. An array can have `undef` elements without being empty.
- **Strings and numbers convert silently.** `"42" + 1` is 43, `42 . "x"` is `"42x"`. The conversion is always defined; see *numbers and strings* for what counts as "looks like a number" and where the corners are.

See also

- perlop - the operators that read and produce these values (arithmetic, comparison, subscript, range, arrow, assignment).
- perlvar - the special variables Perl populates with state from the last operation, including $_, @_, the match variables, and the I/O-shaping $/, $\, $,, $".
- perlfunc - the built-in functions that operate on these types: ref, scalar, wantarray, defined, keys, values, each, exists, delete, bless.
- *References tutorial* - step-by-step introduction to the reference forms covered tersely in *references*.

5.1.5 Subroutines

A subroutine is a named (or anonymous) block of code that takes a list of arguments, runs in a fresh lexical scope, and returns a list of values. Subroutines are the unit of reuse in Perl, the unit of dispatch for object-oriented code, and - through closures - the unit of stateful encapsulation. Almost everything else in the language is built on top of them.

PetaPerl implements the full Perl 5.44 subroutine model: the classic @_-based calling convention, modern signatures, the prototype mechanism that controls call-site parsing, lexical and package scoping (my, our, local, state), closures, the attribute system (:lvalue, :method, :prototype(...)), recursion with goto &sub tail calls, and the special compile-time and run-time blocks (BEGIN, END, INIT, CHECK, UNITCHECK).

This reference is split into one page per topic. The @_ aliasing rule lives with arguments, the closure-over-loop-var pitfall lives with scoping, the parser-level effects of prototypes live with prototypes - so each page can be read and returned to without sweeping context.

Choose a topic

Declaration

Every sub is one of three things: a **named** sub bound to a package, an **anonymous** sub stored in a code reference, or a **lexical** sub bound to a my-like slot in the current scope. The declaration form picks one; everything else (prototype, signature, attributes, body) decorates it.

Named subs

```
sub greet {
    my ($name) = @_;
    return "Hello, $name!";
}

greet("Alice");                  # "Hello, Alice!"
```

The name greet is bound in the current package's symbol table under the ampersand & slot - the same slot the runtime looks up when you call greet(...) or \&greet. Named subs are visible to any code that runs after the sub statement is compiled; in practice

"compiled" means "the file is parsed up to the declaration", which is almost always what you want.

A named sub can also be declared in two parts - a forward declaration and a definition:

```
sub later;                          # promise: a sub named 'later' will
 ↪exist
sub later {                         # fulfilment
    return 42;
}
```

The forward declaration is mainly useful when prototypes need to take effect for recursive calls (the prototype must be visible *before* the recursive call is parsed) or when you want to suppress "Bareword not allowed" warnings for code that calls the sub before its definition.

Calling a sub by name

```
greet("Alice");                     # most common form
greet "Alice";                      # parens optional once predeclared
&greet("Alice");                    # legacy: disables prototype
 ↪checking
&greet;                             # legacy: pass current @_ as the
 ↪call's @_
```

The bare-& form (`&greet` with no parens) is a holdover from perl4 with one remaining legitimate use: passing the current `@_` straight through without copying. See *recursion* for `goto &greet`, the tail-call variant.

Anonymous subs

```
my $double = sub {
    my ($x) = @_;
    return $x * 2;
};

$double->(7);                       # 14
```

The `sub { ... }` expression evaluates to a code reference. It captures any lexical (`my`/`state`) variables from its enclosing scope - see *scoping* - and is invoked through `->()` or with the legacy `&$double()` / `&{$double}()` forms.

Anonymous subs are how every Perl callback works:

```
my @sorted = sort { $a <=> $b } @list;
my @big    = grep { $_ > 100   } @list;
my @str    = map  { "<$_>"     } @list;
```

They are also the building block for closures, dispatch tables, and mock objects. A dispatch table:

```
my %op = (
    add => sub { $_[0] + $_[1] },
    sub => sub { $_[0] - $_[1] },
    mul => sub { $_[0] * $_[1] },
);

my $result = $op{$cmd}->($x, $y);
```

Code references

A code reference is a first-class scalar that holds a sub. You get one from `sub { ... }`, from `\&named_sub`, or from a method lookup with `can`:

```
my $cref = \&greet;              # ref to named sub
my $cref = sub { ... };          # anonymous
my $cref = $obj->can('method');  # method lookup, returns code ref
 ↪or undef

$cref->("Alice");                # invocation
&$cref("Alice");                 # legacy invocation
```

Inside a code ref, the sub keeps its identity: `\&greet` is always the same ref for the lifetime of the package's `&greet`. Anonymous subs are distinct refs every time the `sub { ... }` expression is evaluated - that is what makes closure factories work.

Lexical subs

A lexical sub is declared with `my sub` (and the corresponding `state sub` and `our sub`):

```
sub outer {
    my sub helper {
        my ($x) = @_;
        return $x * 2;
    }

    return helper(21);           # 42
}

helper(1);                       # error: undefined subroutine
                                 # (helper isn't visible here)
```

`my sub` is most useful inside long functions where you want a private helper without polluting the package namespace. It is not heavily used in the wild, but it is the proper tool when you want true encapsulation. See *scoping* for `state` and `our` variants.

Predeclaration

A predeclaration is a `sub NAME;` statement (no body, no braces). Three reasons to write one:

```
# 1. Allow the sub to be called without parens before it's defined
sub later;
later;                          # parses as later()

# 2. Make a prototype visible before the body is reached
sub mypush (\@@);
sub mypush (\@@) { ... }

# 3. Document an exported API at the top of the file
sub greet;
sub farewell;
sub welcome;
# ... actual definitions further down
```

In modern code reasons (1) and (3) are mostly stylistic. Reason (2) is a real requirement when you use prototypes; see *prototypes*.

What you cannot do

- You cannot redefine a built-in by `sub print { ... }` and have Perl quietly use yours. Built-in overriding goes through `CORE::GLOBAL::print` and is documented under *attributes* and the `prototype` detail page.
- You cannot have two named subs with the same name in the same package without a "Subroutine NAME redefined" warning. Either rename or genuinely intend the redefinition (and silence the warning with `no warnings 'redefine';`).
- You cannot give a forward declaration and a body with *different* prototypes. The two declarations must agree.

See also

- *Arguments and* `@_` - what the body sees once the call is made.
- *Scoping* - what `my`, `our`, `local`, and `state` do to the variables inside the body.
- `sub` - the keyword's perlfunc page.
- `prototype` - runtime introspection of a sub's prototype, and the bridge to overriding built-ins.
- *Arrow operator* - `->()` invocation of code references.

Arguments and `@_`

When a sub is called the classic way (no signature), Perl flattens the entire argument list into a single array and exposes it inside the body as `@_`. The elements of that array are not copies - they are **aliases** to the caller's expressions. This single rule is the source of most of Perl's calling-convention surprises and most of its calling-convention power.

The aliasing rule

```
sub bump {
    $_[0]++;                        # mutates the caller's variable
}

my $x = 10;
bump($x);
say $x;                             # 11
```

Modifying `$_[0]` writes through the alias to whatever the caller passed. That is by design - it is how `chomp(@lines)` manages to strip newlines from every element of `@lines` rather than from a copy of `@lines`.

The aliasing extends to literals, with predictable consequences:

```
bump(10);                           # error: Modification of a read-
 ↪only
                                    # value attempted
```

A literal `10` is a constant. The alias points at it; the mutation attempt dies.

The unpacking idiom

Because the aliasing is rarely what the body wants, almost every classic-style sub starts with one of these two patterns:

```
# Named arguments (most common)
sub greet {
    my ($name, $greeting) = @_;  # COPY into private lexicals
    $greeting //= 'Hello';
    return "$greeting, $name!";
}

# Method dispatch
sub method_call {
    my $self = shift;            # COPY first arg, leave rest in @_
    my %args = @_;               # remaining args as a hash
    ...
}
```

Once unpacked, the body works with private copies and cannot accidentally mutate the caller's state. `shift` with no argument inside a sub defaults to `@_`, which is why `my $self = shift;` is everywhere in OO Perl.

Legitimate uses of aliasing

Aliasing is the right tool when you actually want to mutate caller state:

```
# in-place trim
sub trim_inplace {
```

(continues on next page)

(continued from previous page)

```
    for (@_) {
        s/^\s+//;
        s/\s+$//;
    }
}

my @data = ("  hello  ", "  world\n");
trim_inplace(@data);
# @data is now ("hello", "world")
```

The `for (@_)` loop aliases `$_` to each element of `@_`, which is itself an alias to each element of `@data`. The `s///` modifications go through both alias hops to the original strings. Without the aliasing, this same shape of code would be twice as much work.

The same mechanism powers `chomp(@lines)`, `chop(@buf)`, and any "work over a list and modify in place" function you write yourself. State the intent in the function's name (`_inplace`, `mutate_`, similar) so callers are not surprised.

Named arguments

Perl has no built-in named-parameter syntax in the classic form, but the `=>` (fat comma) plus a hash assignment gives the same effect:

```
sub configure {
    my %opt = @_;                # ('host' => 'localhost', 'port' =>␣
↪8080)
    $opt{host} //= 'localhost';
    $opt{port} //= 80;
    ...
}

configure( host => 'example.com', port => 8080 );
```

The fat comma is identical to a regular comma in semantics but quotes the bareword on its left, so `host => ...` is shorthand for `'host' => ...`. See `,` for the operator itself.

For mandatory-plus-optional named args, the standard shape is:

```
sub render {
    my ($template, %opt) = @_;   # one positional, rest as named
    $opt{escape} //= 1;
    ...
}

render('greeting.tt', escape => 0, locale => 'de');
```

For the modern named-parameter idiom built on signatures, see *signatures*.

`shift`, `pop`, `unshift`, `push` on `@_`

`@_` is a real array. You can `shift` from the front, pop from the back, slice it, modify it. None of these change the caller's state - only writing through `$_[N]` does.

```
sub describe {
    my $what = shift;            # remove first arg from @_
    my @rest = @_;               # copy what's left
    ...
}

sub method {
    my $self = shift;
    my @args = @_;
    $self->dispatch(@args);
}
```

A handy consequence: if you want to *re-pack* the remaining args to forward to another sub, just write `@_` - that is exactly what `goto &other` does without the explicit pass. See *recursion* for tail-call forwarding.

`wantarray` and the calling context

Inside a sub, `wantarray` tells you the context the call was made in:

`wantarray`	Context	Common shape
1 (true)	list	return a list
`0` (false)	scalar	return a single scalar
`undef`	void	return early; the result will be discarded

```
sub items {
    return unless defined wantarray;     # void: nothing to compute
    my @result = compute_items();
    return wantarray ? @result : scalar @result;
}
```

`@_` says *what* came in; `wantarray` says *what is expected back*. They are the two halves of the calling-convention picture. See *lvalue and context* for the full discussion.

Common pitfalls

- **`my ($x) = @_` vs `my $x = @_`.** The first is list context: `$x` gets the first argument. The second is scalar context: `$x` gets the *count* of arguments. The parentheses on the left are the difference; mistakes here are common.

```
sub one_arg {
    my ($x) = @_;            # $x = first arg
}
```

(continues on next page)

(continued from previous page)

```
sub count_args {
    my $n = @_;                 # $n = number of args
}
```

- **Modifying `$_[N]` "by accident".** Functions like `chomp`, `chop`, `tr///`, `s///` operate on their argument *in place*. `chomp($_[0])` mutates the caller's string. If that's not what you want, copy first.
- **Flattening of arrays and hashes.** Arrays and hashes passed as arguments lose their identity at the call site - they all become elements of the single flat `@_`. To pass an array without flattening, pass a reference:

```
takes_array(\@arr);             # one ref argument
sub takes_array {
    my ($aref) = @_;
    push @$aref, 'extra';
}
```

- **`@_` after `goto &sub`.** The current `@_` is passed through to the target. Anything you've already shifted off is gone from the perspective of the target. This is a feature, not a bug - see *recursion*.

See also

- `@_` - the variable's perlvar entry, with the aliasing rule and the loop-context discussion.
- `shift`, `pop` - default to `@_` inside a sub.
- *Signatures* - the modern alternative for named-and-positional parameter handling.
- *Return values* - what the body sends back.
- *Comma operator* - `=>` for named-arg construction.

Return values

A sub returns either by reaching its last statement, by hitting an explicit `return`, or by being `goto`-ed somewhere else (see *recursion*). The returned list is **flattened** the same way the argument list is - arrays and hashes lose their identity unless you return references - and is then evaluated in the caller's context.

Implicit return

If control falls off the end of the body, the value of the last *expression* statement is returned:

```
sub square {
    my ($x) = @_;
    $x * $x                     # implicit return
}
```

(continues on next page)

(continued from previous page)

```
say square(5);                    # 25
```

Implicit return is idiomatic for short, one-expression subs. It is *not* idiomatic for multi-branch subs, where readers benefit from seeing every exit explicitly marked with `return`.

If the last statement is a control-flow construct (a `for`, `while`, `if` without an `else`, ...), the returned value is unspecified. The empty sub returns the empty list.

Explicit return

```
sub max {
    my ($a, $b) = @_;
    return $a if $a >= $b;
    return $b;
}
```

Explicit `return` exits the sub immediately and supplies the return value. Multiple early returns are the standard shape for guarded code:

```
sub fetch {
    my ($id) = @_;
    return       unless defined $id;      # void/empty
    return undef if      $id eq '';       # explicit undef in scalar
    ...
}
```

Note the difference between `return;` and `return undef;`:

- `return;` returns the empty list in list context, `undef` in scalar context, and nothing in void context. It is the context-correct way to say "no result".
- `return undef;` returns the *list* `(undef)` in list context, which has length 1 - so a caller writing `if (my @r = my_sub()) { ... }` will see a one-element list and the `if` will fire. This is almost always wrong.

The rule of thumb: write `return;` to indicate failure or absence; write `return undef;` only when the caller is guaranteed to be in scalar context.

List flattening on return

The same flattening that applies to arguments applies to return values:

```
sub two_lists {
    my @a = (1, 2);
    my @b = (3, 4);
    return (@a, @b);              # returns (1, 2, 3, 4) - one flat␣
↪list
}
```

(continues on next page)

(continued from previous page)

```
my ($x, $y) = two_lists();        # $x = 1, $y = 2 ; rest discarded
my @all     = two_lists();        # @all = (1, 2, 3, 4)
my (@p, @q) = two_lists();        # @p = (1, 2, 3, 4), @q = ()
```

The last line is the classic surprise: `@p` greedily slurps the whole list because list-assignment to a flat array on the LHS has no way to know where one logical group ends and the next begins. To return two arrays distinctly, return references:

```
sub two_lists {
    return (\@a, \@b);
}

my ($pref, $qref) = two_lists();
```

Context-sensitive return

```
sub items {
    if (wantarray) {
        return (1, 2, 3);         # list context
    }
    elsif (defined wantarray) {
        return 3;                 # scalar context: count
    }
    else {
        return;                   # void: nothing to compute
    }
}

my @list = items();               # (1, 2, 3)
my $cnt  = items();               # 3
items();                          # discarded
```

The “void early return” branch is a real optimisation pattern - expensive computations should not run when the caller throws the result away.

For the full discussion of `wantarray` and how it composes with `:lvalue`, see *lvalue and context*.

Returning hashes and arrays

Returning a `@array` or `%hash` directly works only because of flattening:

```
sub config {
    return (host => 'localhost', port => 80);
}

my %c = config();                 # %c is the hash
my @c = config();                 # @c = ('host', 'localhost', 'port
 ↪', 80)
```

For larger structures, return references:

```
sub config {
    my %c = (host => 'localhost', port => 80);
    return \%c;                  # one ref, no flattening
}

my $c = config();
say $c->{host};
```

References are also the only way to return *multiple* aggregates without the caller having to know their sizes.

`return` inside an `eval`

`return` inside a `do {}` or `eval {}` block returns from the *enclosing sub*, not from the block:

```
sub safe {
    my $result = eval {
        return 'inside';         # this returns from `safe`, not
↪from eval
    };
    return $result;              # never reached
}
```

To exit just the `eval`, use a value (`'inside'` as the last expression of the block) - `eval` returns the value of its last expression like any other block.

Returning from `BEGIN`/`END`

`BEGIN`, `END`, `INIT`, `CHECK`, and `UNITCHECK` blocks are subs in the technical sense but their return value is discarded. Don't bother writing `return ...` inside them.

See also

- `return` - the keyword's perlfunc page.
- `wantarray` - what your caller wants back.
- *Lvalue subs and context* - the full picture for context-sensitive subs.
- *Arguments and* `@_` - the symmetric story for what comes in.
- `die`, `warn` - the alternatives to "return failure".

Scoping: `my`, `our`, `local`, `state`

Inside a sub, four declarators control where a variable lives, how long it lives, and who can see it. Pick the wrong one and the sub leaks state, fails to capture properly in a closure, or quietly clobbers something the caller relied on.

Declarator	Scope	Lifetime	Typical use
`my`	lexical (the block)	until the block ends	the default; private locals
`our`	lexical alias to package	as long as the package	shared package globals with a short name
`local`	dynamic (the call tree)	until the block ends	temporarily change a special variable
`state`	lexical (the block)	program lifetime	per-sub persistent state, init once

`my` - lexical locals

`my` is the everyday declarator:

```
sub greet {
    my ($name) = @_;
    my $message = "Hello, $name!";
    return $message;
}
```

The variables exist for the duration of the call, are private to the body, and are stored in the sub's *pad* (one slot per declared lexical). They are **not** visible to callers, callees, or `eval`'d code that doesn't textually nest inside the sub.

`my` variables are the only ones that close over correctly. Closures capture *cells* in the surrounding pad, and only `my` (and `state`) variables get pad cells.

`our` - package globals with a lexical alias

```
package Counter;

sub increment {
    our $value;                    # alias to $Counter::value
    $value++;
    return $value;
}
```

`our $x` declares a *lexical alias* in the current scope to the package variable `$Pkg::x`. The variable itself is global. If two packages both `our`-declare `$VERSION`, they each get their own - `our` prefixes with the current package name.

Use `our` when:

- You genuinely want a package-level shared variable.
- You want to silence `use strict 'vars';` complaints about a variable that *should* be a package global (`$VERSION`, `@ISA`, `@EXPORT`, `%ENV` aliases, ...).

Do **not** use `our` to share state between subs in the same file. That's what `my` at file scope is for:

```
package Counter;

my $value = 0;                    # private to this package's file
sub increment { ++$value }
sub get       { $value   }
```

`local` - dynamically-scoped temporary

`local` does *not* declare a new variable. It saves the current value of an existing (typically global) variable, sets a new value, and restores the old value when the enclosing block exits - even on exception:

```
sub slurp {
    my ($path) = @_;
    open my $fh, '<', $path or die "open $path: $!";
    local $/;                     # slurp mode for this sub only
    return <$fh>;
}
```

Without the `local`, the caller's `$/` would be permanently clobbered. With it, the change is confined to the dynamic extent of `slurp` - including any subs `slurp` happens to call.

`local` is the right tool for almost every special-variable override (`$/`, `$\`, `$,`, `$_`, `$@`, signal handlers in `%SIG`). It is rarely the right tool for ordinary user variables - those want `my`.

For the full register, see `local` and the *error-state* and *I/O* variable pages, which list which specials want `local`-isation in which patterns.

`state` - initialise once, keep forever

```
use feature 'state';

sub counter {
    state $n = 0;                 # initialised once, ever
    return ++$n;
}

counter();                        # 1
counter();                        # 2
counter();                        # 3
```

`state` variables are lexicals that survive between calls. The initialiser runs the first time control reaches the declaration; subsequent calls skip it.

`state` is the right tool for:

- Per-sub caches and counters where the cost of constructing the data dominates and you don't want a closure factory.
- One-time setup that depends on values not available at compile time:

```
sub compiled_re {
    my ($pattern) = @_;
    state %cache;
    $cache{$pattern} //= qr/$pattern/;
    return $cache{$pattern};
}
```

(Note: this caches the *first* pattern only with a literal `state` of a single regex; the hash form generalises it.)

Closures: capturing surrounding `my`

An anonymous sub captures every `my` (or `state`) variable from its surrounding scope that it textually references:

```
sub make_adder {
    my ($n) = @_;
    return sub {
        my ($x) = @_;
        return $x + $n;          # captures $n
    };
}

my $add5 = make_adder(5);
my $add9 = make_adder(9);

$add5->(3);                      # 8
$add9->(3);                      # 12
```

Each call to `make_adder` creates a fresh `$n`, and the returned closure references its own `$n`. Two calls produce two independent counters / adders / state machines.

The closure-over-loop-variable surprise

```
my @subs;
for my $i (1, 2, 3) {
    push @subs, sub { $i };
}

print $_->(), "\n" for @subs;
# 1
# 2
# 3   - fresh $i each iteration: each closure sees its own
```

This is the *correct* behaviour with `for my $i`: every iteration declares a fresh `$i`, each closure captures its own.

The buggy shape is when `$i` is declared *outside* the loop:

```
my $i;
my @subs;
for $i (1, 2, 3) {                # NO `my` - same $i every iteration
    push @subs, sub { $i };
}

print $_->(), "\n" for @subs;
# 3
# 3
# 3   - all three closures share the one $i, and its final value is␣
 ↪3
```

Always declare loop variables with `my`. Closure capture *follows the variable*, not the value, and "same variable" vs "different variable" is decided at the `my` site.

Capturing our and package globals

`our` aliases live for the whole package, so a closure capturing `$Counter::value` shares state with every other piece of code that touches `$Counter::value`. That's almost always not what you want - closures over package globals are a sign that the design wants a `my` instead.

Pad lifetime

Every active sub call has a **pad**, a slot per declared lexical. The pad is what `my` writes into and what closures hold a reference to. While a closure is alive, every pad it captures is alive - even if the originating call has long returned. This is how `make_adder`'s `$n` survives the call to `make_adder` itself.

See also

- `my`, `our`, `local`, `state` - perlfunc detail pages for each declarator.
- `@_` - the one variable that's *always* present in a sub regardless of declarators.
- *Declaration* - anonymous subs and the lexical `my sub` form.
- *Recursion* - closure-based trampolines and `__SUB__`.
- *Special variables* - almost every special wants `local`, not `my`.

Prototypes

A prototype is a piece of metadata attached to a sub that controls **how the parser parses calls to that sub**. It does not control the runtime - it does not check argument counts at runtime, it does not enforce argument types, it does not affect method calls. Its job is to make a user-defined sub *parse* like a built-in.

That narrow purpose is the one thing to remember about prototypes. Everything else follows.

Declaration

Two equivalent forms:

```
sub mypush (\@@) {                  # short form
    my $aref = shift;
    push @$aref, @_;
}

sub mypush :prototype(\@@) {        # attribute form
    ...
}
```

The attribute form is mandatory when *signatures* are enabled in the same scope, because the parenthesised list after the sub name is then claimed by the signature.

What prototypes actually do

A prototype rewrites the parser's expectations at the call site:

```
sub mygrep (&@) {
    my $code = shift;
    my @result;
    for (@_) { push @result, $_ if &$code }
    return @result;
}

mygrep { /foo/ } @items;            # parses correctly thanks to (&@)
                                    # without the prototype, this is
                                    # a syntax error
```

The `(&@)` prototype tells the parser: "the first argument is a block (no `sub` keyword needed, no comma after it), the rest is a list." That's exactly how built-in `grep`, `map`, `sort` parse.

This is the **only** place prototypes are useful: building DSL-shaped subs that look like built-ins.

Prototype characters

Char	Meaning
`$`	scalar - forces scalar context on the argument
`@`	list - eats all remaining arguments, list context
`%`	hash - like @ (even-length list)
`&`	block / code ref - first position lets you omit `sub`
`*`	bareword, scalar, typeglob, or globref
`\$`	scalar reference - caller must pass `$var` form (auto-\-applied)
`\@`	array reference - caller must pass `@arr` form
`\%`	hash reference - caller must pass %h form
`\&`	code reference - caller must pass `&fn` form
`\[$@%&*]`	reference to any of the bracketed kinds
`+`	scalar OR \@/\% (auto-detect array/hash literal)
`;`	separates mandatory from optional
`_`	scalar; if omitted, defaults to $_
`()`	no arguments - and triggers the constant-fold optimisation

The (\@) vs (@) difference

This is the whole reason backslash prototypes exist.

```
sub takes_array (\@) {              # backslash: pass-by-reference
    my ($aref) = @_;                # @_ holds [array_ref]
    push @$aref, 'extra';
}

sub takes_list (@) {                # no backslash: pass-by-value␣
↪(flattened)
    my @copy = @_;                  # @_ holds the flat list
    push @copy, 'extra';            # nothing in caller is affected
}

my @arr = (1, 2, 3);

takes_array(@arr);                  # caller writes @arr - parser␣
↪inserts \@arr
                                    # @arr is now (1, 2, 3, 'extra')

takes_list(@arr);                   # caller writes @arr - flattens to␣
↪a list
                                    # @arr is unchanged
```

The `(\@)` prototype lets you *write* the call as `takes_array(@arr)` - looking just like `push @arr, ...` - while the runtime actually receives an array reference. That's exactly what built-in `push`, `unshift`, `splice`, and `pop` look like to the user.

The `()` constant-fold prototype

A sub with the empty prototype `()` and a single-expression body that the optimiser can fold to a constant becomes inline-able:

```
sub PI ()  { 3.14159265358979 }
sub MAX () { 100 }

my $area = PI * $r ** 2;          # PI is inlined as 3.14159...
```

This is what `use constant` produces:

```
use constant {
    PI  => 3.14159265358979,
    MAX => 100,
};
```

Constant-folded subs are zero-cost at runtime. They do require the prototype to be `()` exactly (no arguments accepted), and the body must be a single expression that the optimiser can reduce. Adding `return` defeats the inlining:

```
sub PI () { return 3.14159; }    # NOT inlined: explicit return
```

What prototypes do *not* do

- They do **not** check argument counts at runtime. Calling a `($)` sub with three arguments is a *parse-time* error, but if you bypass the parser (via `&fn(...)`, code refs, `goto &fn`), there is no runtime check.
- They do **not** affect method calls. Method dispatch is resolved at runtime through `@ISA`, by which point parsing is long over. Prototypes on methods are silently ignored.
- They do **not** affect calls through code references. The call site sees a scalar, not a name; no name, no prototype lookup.
- They are **not** part of the sub's identity. `\&named_sub` loses prototype knowledge.

When you should use prototypes

You are writing a sub that looks like a built-in. Examples:

- A list-eating function with a block argument (`(&@)`): `mygrep`, `myfirst`, `try { ... } catch { ... }`.
- An "inplace" helper that mutates an array in place (`(\@)` or `(\@@)`).
- A constant-like sub (`()` for inlining).

That's the list. If you're not building a built-in look-alike, do not reach for prototypes - use a signature, or just unpack `@_`.

When you should not use prototypes

- For "type checking". Prototypes are not a type system. `sub f ($) { ... }` does not say "takes one scalar"; it says "takes one expression in scalar context". Calling `f(@arr)` passes the *count* of `@arr`, not the array itself, with no warning.
- For run-time argument-count enforcement. Use signatures (which *do* enforce arity at runtime) or write the check by hand.
- On methods. They are silently ignored.

Forward declarations and recursion

A recursive call sees the prototype only if the sub was predeclared:

```
sub fact ($);                       # forward declaration with␣
↪prototype
sub fact ($) {
    my ($n) = @_;
    return $n <= 1 ? 1 : $n * fact($n - 1);    # parsed with ($) in␣
↪mind
}
```

Without the forward declaration, the recursive `fact($n - 1)` inside the body is parsed before the `($)` prototype is finalised, so the prototype does not apply to that call.

Inspecting a prototype at runtime

`prototype` returns a sub's prototype string (or `undef` if there is none):

```
prototype(\&push);                  # '\@@' - yes, even built-ins have␣
↪one
prototype('CORE::push');            # '\@@'
prototype(\&my_sub);                # whatever you declared
```

See also

- `prototype` - runtime introspection.
- *Signatures* - the modern alternative for arity and named-parameter handling. They coexist with prototypes but have different goals.
- `sub` - the keyword's perlfunc page.
- *Declaration* - where prototypes attach in the declaration syntax.
- *Attributes* - `:prototype(...)` is the attribute form.

Signatures

A signature is a parenthesised parameter list that goes immediately after the sub name (and after any attributes), in place of - or in addition to - the body's manual `@_` unpacking. Signatures introduce **lexical** parameter variables, perform **arity checking** at call time, and support default values and slurpy parameters.

Signatures stabilised in Perl 5.36; in 5.44 they are a production feature available with one of:

```
use feature 'signatures';          # explicit feature import
use v5.36;                         # implicit (signatures + other 5.36
 ↪features)
```

PetaPerl supports the full 5.44 signature syntax.

Basic shape

```
use feature 'signatures';

sub add ($x, $y) {                 # two mandatory positional
 ↪parameters
    return $x + $y;
}

add(2, 3);                         # 5
add(2);                            # error: Too few arguments
add(2, 3, 4);                      # error: Too many arguments
```

Each `$name` in the signature declares a `my` lexical that receives the corresponding argument. Arity is checked at the call: the body never runs if the count is wrong.

This is the most concrete improvement over the classic form:

```
# Classic - silently accepts wrong arity
sub add {
    my ($x, $y) = @_;
    return $x + $y;
}

add(2);                            # $y is undef; later arithmetic
 ↪warns
add(2, 3, 4);                      # third arg silently ignored
```

Defaults

```
sub greet ($name, $greeting = 'Hello') {
    return "$greeting, $name!";
}

greet('Alice');                    # "Hello, Alice!"
greet('Alice', 'Hi');              # "Hi, Alice!"
```

The default is an arbitrary expression evaluated each time the parameter is omitted (so `state` and per-call computation both work). It can reference earlier parameters:

```
sub log_event ($message, $level = 'INFO', $prefix = "[$level]") {
    print "$prefix $message\n";
}

log_event('starting');                      # [INFO] starting
log_event('failure', 'ERROR');              # [ERROR] failure
log_event('done', 'OK', '>>');              # >> done
```

A default `= undef` is exactly equivalent to "no default" for arity (the parameter is still optional) but documents the intent. Use `//=`-style guards inside the body for "undef means use my fallback" semantics.

Optional vs mandatory

Parameters are mandatory by default. A parameter becomes optional by:

- Giving it a default value (`$x = 42`).
- Putting it after another optional parameter (transitive).
- Being followed by a slurpy `@arr` or `%hash`.

You cannot have a mandatory parameter after an optional one (except via slurpy):

```
sub bad ($x = 1, $y) { ... }     # error: Mandatory parameter
↪follows optional
```

Slurpy parameters

The last parameter may be a slurpy `@list` or `%hash`. It eats all remaining arguments:

```
sub join_all ($sep, @parts) {
    return join $sep, @parts;
}

join_all(', ', 'a', 'b', 'c');   # "a, b, c"

sub configure ($name, %opts) {
    $opts{verbose} //= 0;
    ...
}

configure('demo', verbose => 1, port => 8080);
```

The slurpy hash also fails if the remaining list has odd length:

```
configure('demo', 'verbose');    # error: Odd-sized list given
```

This is the single most useful pattern signatures unlock: a named-parameter API with one positional plus a `%opts` slurp.

Anonymous parameters

A $ (or @, %) with no name is a positional placeholder:

```
sub second ($, $x) { return $x }

second('ignored', 'kept');       # "kept"
```

This is mildly useful when adapting a callback to an interface that supplies more arguments than you care about.

Combining defaults and slurpy

```
sub render ($template, $cache = 1, %opt) {
    ...
}

render('greeting.tt');                          # cache=1, %opt empty
render('greeting.tt', 0);                       # cache=0, %opt empty
render('greeting.tt', 1, locale => 'de');       # all three
render('greeting.tt', locale => 'de');          # ERROR: 'locale'␣
↪becomes $cache
                                                # then 'de' is the lone␣
↪slurp arg
                                                # (odd-length hash) -␣
↪diagnose carefully
```

That last case is the gotcha: a positional optional parameter followed by a slurpy hash is ambiguous with named-arg-only calls. Either drop the optional positional and put everything in the hash, or always pass the optional positional explicitly.

Migration from @_ unpacking

The mechanical translation:

```
# Before
sub make_user {
    my ($name, $email, %extra) = @_;
    ...
}

# After
use feature 'signatures';
sub make_user ($name, $email, %extra) {
    ...
}
```

You get arity checking and named lexicals for free. The body is unchanged.

The places where you cannot mechanically translate:

- The body uses the *count* of arguments dynamically (`if (@_ >= 3)`) - signatures fix arity at parse time, so rework into an optional parameter with a default.

- The body modifies `$_[0]` for the aliasing effect - keep the classic form, since signatures introduce *copies*, breaking the alias.
- The body uses `goto &other` and wants to forward the original `@_` - `@_` is still populated under signatures in 5.44, but this is implementation-defined and may change. Keep the classic form for tail-call forwarding.

Signatures and prototypes are orthogonal

Signatures and *prototypes* solve different problems:

Mechanism	Affects	When checked	Useful for
signature	the body's view	runtime	arity, named params
prototype	how the parser parses	compile-time	DSL look-alikes

A sub can have both, with the prototype written as an attribute:

```
use feature 'signatures';
sub mygrep :prototype(&@) ($code, @list) {
    grep { $code->($_) } @list;
}

mygrep { /foo/ } @items;          # prototype lets us drop `sub`
                                  # signature gives us $code and
 ↪@list
```

The prototype shapes the call site; the signature shapes the body. They do not duplicate each other.

The single biggest reason to migrate

Catching wrong-arity calls at the moment they happen - not several lines later when you finally do `$y + 1` and trip a warning about an undef value. Signatures turn a class of subtle bugs into a clear error message.

See also

- *Arguments and* `@_` - the classic mechanism that signatures coexist with.
- *Prototypes* - different mechanism, often confused with signatures.
- `sub` - the keyword's perlfunc page.
- *Declaration* - where signatures sit in the declaration syntax.
- *Scoping* - signature parameters are `my`-style lexicals.

Recursion

Subs may call themselves, each other, or - with a bit of ceremony - replace their own call frame and pass control elsewhere without growing the stack. This page covers the four mechanisms: ordinary recursion, mutual recursion, `goto &fn` tail calls, and `__SUB__` for anonymous self-reference. It also covers the depth limit you will hit if you ignore them.

Plain recursion

```
sub factorial {
    my ($n) = @_;
    return 1 if $n <= 1;
    return $n * factorial($n - 1);
}

factorial(10);                    # 3628800
```

Each recursive call adds a new frame to Perl's call stack. The stack costs memory and a fixed per-call overhead; for shallow recursion this is irrelevant, for deep recursion it is the bottleneck.

Perl emits a "Deep recursion on subroutine X" warning when a sub recurses past 100 levels deep (compile-time threshold, `PERL_SUB_DEPTH_WARN`). The warning is in the `recursion` category - `no warnings 'recursion'` silences it. The only hard limit is the OS stack, typically 8 MiB by default, controllable with `ulimit -s`.

For deep, structurally-recursive workloads, prefer one of:

- An explicit work queue (iterative reformulation).
- Tail recursion via `goto &fn` (see below).
- A trampoline (see below).

Mutual recursion

Two subs that call each other:

```
sub is_even {
    my ($n) = @_;
    return 1 if $n == 0;
    return is_odd($n - 1);
}

sub is_odd {
    my ($n) = @_;
    return 0 if $n == 0;
    return is_even($n - 1);
}

is_even(10);                      # 1
```

Mutual recursion has no special treatment from the runtime - it is just two subs happening to call each other. The same depth limit applies.

If `is_even` is defined first and refers to `is_odd` before `is_odd` exists, you'll see a "Bareword not allowed" or "Called undefined subroutine" depending on the call shape. Predeclare to fix:

```
sub is_odd;                          # forward declaration
sub is_even { ... is_odd($n-1) }
sub is_odd  { ... is_even($n-1) }
```

`goto &fn` - tail-call forwarding

The `goto &SUBROUTINE` form **replaces the current frame** with a call to another sub. The current frame is gone - it is not on the stack - so the call does not grow the stack:

```
sub api_v2 {
    my @args = @_;
    push @args, deprecated => 1;
    @_ = @args;
    goto &api_v1;                # tail-call: api_v1 sees the␣
 ↪modified @_
}
```

This is the right tool when:

- A wrapper sub needs to forward to a real implementation without leaving a stack frame behind. Decorators, logging shims, AUTOLOAD dispatchers.
- A recursive sub can express its recursion as "do some work, then become this other call". Classic tail recursion:

  ```
  sub factorial_tail {
      my ($n, $acc) = @_;
      $acc //= 1;
      return $acc if $n <= 1;
      @_ = ($n - 1, $n * $acc);
      goto &factorial_tail;
  }

  factorial_tail(20);            # 2432902008176640000, no stack␣
   ↪growth
  ```

The mechanics: `@_` becomes the called sub's argument array, the current frame is unwound, control jumps. The caller of the *current* sub will see the result of the goto'd-to sub, never the one it called.

`goto &fn` is the **only** form of `goto` that performs a tail call. The label-based form (`goto LABEL`) is unrelated and far more restricted.

See `goto` for the full keyword description and the shape constraints.

__SUB__ - anonymous self-reference

```
use feature 'current_sub';                # or use v5.16

my $factorial = sub {
    my ($n) = @_;
    return 1 if $n <= 1;
    return $n * __SUB__->($n - 1);
};

$factorial->(5);                  # 120
```

__SUB__ evaluates to a code reference to the currently-running sub. It is the only way to recurse into an anonymous sub without storing it in a named variable first (and without leaking a strong reference to it from inside itself, the classic memory-leak shape).

For anonymous subs that do not need to recurse, prefer the plain anonymous form. __SUB__ exists for the recursion case specifically.

Trampolines

When you want bounded-stack recursion but `goto &fn` is awkward (because the recursion is mutual, or branches into many sub-cases), the trampoline pattern works:

```
sub trampoline {
    my $next = shift;
    while (ref $next eq 'CODE') {
        $next = $next->();
    }
    return $next;
}

sub even {
    my ($n) = @_;
    return $n == 0 ? 1 : sub { odd($n - 1)  };
}

sub odd {
    my ($n) = @_;
    return $n == 0 ? 0 : sub { even($n - 1) };
}

trampoline(even(10000));          # 1, no stack growth
```

Each "recursive" call returns a closure to be invoked next, rather than calling itself directly. The trampoline drives the closures in a loop. The stack stays at depth 2 forever.

Trampolines are heavier than `goto &fn` (one closure allocation per "call"), but they generalise to arbitrary control flow - including continuations and CPS-style code. For production Perl, `goto &fn` is the usual choice; trampolines appear when the structure of the recursion makes `goto` clumsy.

The depth limit, in practice

```
sub recurse {
    my ($n) = @_;
    return $n if $n <= 0;
    return recurse($n - 1);
}

recurse(2000);                  # warning: Deep recursion on
↪subroutine
                                # (default limit 1000)
```

The "Deep recursion" warning is informational, not fatal - it fires once per sub. You will run out of actual stack space well before any practical limit on most platforms. If you see this warning, the answer is almost always to refactor (iterative loop, `goto &fn`, or trampoline), not to raise the limit.

If you have already verified the recursion is bounded and you just want to suppress the noise, scope the warning category:

```
no warnings 'recursion';
```

This silences the warning entirely; it does not change the OS stack limit, which is what eventually terminates a runaway recursion regardless of warnings.

See also

- `goto` - the keyword behind `goto &fn`.
- `caller` - works through goto'd frames with the same numbering as ordinary frames.
- *Arguments and* `@_` - `@_` semantics under `goto &fn`.
- *Declaration* - anonymous subs, the substrate for `__SUB__` and trampolines.
- *Scoping* - closures, the substrate for trampolines.

Lvalue subs and context

Two related topics that come up wherever a sub needs to behave differently depending on how it is being used: **`:lvalue`** subs (those usable on the left-hand side of an assignment) and **`wantarray`**-driven dispatch on the caller's context.

Why context matters in subs

Every Perl expression evaluates in a context - list, scalar (with sub-flavours: numeric, string, boolean), or void - that's imposed by what surrounds it. A sub call inherits the context of its position:

```
my @r = my_sub();               # list context
my $r = my_sub();               # scalar context
my_sub();                       # void context
```

(continues on next page)

(continued from previous page)

```
print my_sub();                    # list context (print's args are
↪LIST)
my $n = () = my_sub();             # scalar of (a list assignment) -
↪element count
```

For most subs, this is invisible - they return the same data either way and Perl's automatic context conversion does the right thing. For some subs the work is genuinely different depending on context, and those subs benefit from explicitly inspecting `wantarray`.

wantarray: the three-way switch

```
sub items {
    if (wantarray) {
        return (1, 2, 3);                  # list context
    }
    elsif (defined wantarray) {
        return 3;                          # scalar context
    }
    else {
        return;                            # void context - caller
↪will
                                           # discard whatever we
↪return
    }
}
```

The three return values of `wantarray`:

Return value	Caller's context	Practical meaning
1 (true)	list	return all the things
`0` (false)	scalar	return one thing (e.g. count)
`undef`	void	nobody cares; consider returning early

`defined wantarray` is the test for "scalar or list" (anything non-void).

The void-context shortcut

When work is expensive and the caller threw the result away, return early:

```
sub expensive_query {
    return unless defined wantarray;       # nothing to compute
    my @rows = $db->select(...);           # actually do the work
    return wantarray ? @rows : scalar @rows;
}

expensive_query();                         # void: returns
↪immediately
```

(continues on next page)

(continued from previous page)

```
my @r = expensive_query();                    # list: full result
my $n = expensive_query();                    # scalar: count
```

This pattern is the single best reason to consult `wantarray`: turning a "always do the work" sub into a "do the work only if the caller wants it" sub.

When *not* to use `wantarray`

- For convenience overloading. A sub that returns a hashref in scalar context and a hash in list context is a maintenance hazard - readers can't tell from the call site which they're getting. Pick one.
- For subs that are obviously list-y or scalar-y. A sub named `count_users` should return a number; readers do not expect it to behave differently in list context.

`:lvalue` subs

An `:lvalue` sub is one whose return value is itself assignable. The body must end in an expression that has a memory location - typically a `my` lexical or an aggregate element:

```
sub editable :lvalue {
    my $self = shift;
    $self->{counter};                         # last expression is an␣
 ↪lvalue
}

my $obj = { counter => 0 };
bless $obj, 'Counter';

$obj->editable = 42;                          # writes through to $obj-
 ↪>{counter}
```

The body's last expression is what the assignment writes to. If that expression is not an lvalue (`return $x + 1`, `return scalar @list`), the assignment is illegal and you'll get a clear error.

`:lvalue` is rarely the right tool. The only places it pays for itself:

- Tied variables and tied-like wrappers, where the sub is mediating access to a real lvalue underneath.
- DSLs that genuinely want assignment-syntax for setting (eg. `$config->host = 'localhost';`).

For "set a property on an object", a plain setter is clearer:

```
sub set_counter {
    my ($self, $value) = @_;
    $self->{counter} = $value;
    return $self;                              # method chaining
}
```

(continues on next page)

(continued from previous page)

```
$obj->set_counter(42);                  # explicit, no surprises
```

`:lvalue` and the body's structure

Because the body's last expression is special, the rules are restrictive:

```
sub good :lvalue { $self->{x} }             # OK - last expr is
 ↪lvalue

sub also_good :lvalue {
    my @scratch = ...;                      # work
    $self->{x};                             # last expr is lvalue
}

sub bad :lvalue {
    return $self->{x};                      # NOT OK - explicit
 ↪return
                                            # disables :lvalue
}

sub also_bad :lvalue {
    if ($cond) { $self->{x} }
    else       { $self->{y} }               # an `if` block is not an
 ↪lvalue
}
```

The "no explicit `return`" rule is the same one that makes constant-fold prototypes work; both rely on the body being a single value-producing expression at the end.

For the modern alternative - a setter that returns `$self` for chaining - see the *Object-Oriented Programming guide*.

Combining `:lvalue` with context

An `:lvalue` sub cannot easily branch on `wantarray`, because the lvalue version of the call doesn't quite fit the "returns a value" model. If you want both behaviours from one sub, accept that you are in DSL territory and document the contract carefully - or split into two subs.

Context-affecting operators

A few operators force a specific context on their argument; the sub will see that context regardless of where the surrounding expression is going:

```
my @r = scalar my_sub();            # scalar() forces scalar context on
 ↪my_sub
my $n = () = my_sub();              # the inner () = ... is list
 ↪context;
```

(continues on next page)

(continued from previous page)

```
                                    # outer assignment to $n is scalar␣
↪of that
```

The `() = LIST` idiom is the standard way to force list context on a sub call and then read the count. See *assignment* for the operator-level discussion.

See also

- `wantarray` - the keyword.
- *Return values* - the rest of the return-value picture (without the lvalue twist).
- *Attributes* - `:lvalue` is one of several built-in attributes.
- *Assignment operator* - what the caller's = actually does on either side of an `:lvalue` call.
- `scalar` - explicit context coercion.

Attributes

A sub attribute is a colon-prefixed annotation between the sub's name (or signature) and its body. Attributes mark a sub as having some special property - being a method, being an lvalue, having a prototype, being constant, being subject to some third-party mechanism - that the runtime, the parser, or a user-written attribute handler can act on.

```
sub greet :method ($name)           { ... }
sub editable :lvalue                { ... }
sub mypush :prototype(\@@)          { ... }
sub PI :const                       { 3.14159 }
sub handler :Memoize                { ... }   # third-party␣
↪attribute
```

The four attribute steps

To follow how an attribute affects a sub, distinguish four phases:

1. **Parse.** The compiler reads `:foo(args)` and breaks it into a name (`foo`) and an optional parameter string (`args`).
2. **Register.** The compiler invokes either a built-in handler (for `:lvalue`, `:method`, `:prototype(...)`) or `MODIFY_CODE_ATTRIBUTES` from the relevant package (for user-defined attributes).
3. **Store.** Built-in attributes set a flag on the CV (the compiled sub object). User attributes are stored wherever the handler chooses.
4. **Query.** Anyone can later ask `attributes::get(\&sub)` for the attribute list. Built-in attributes are also queryable through more direct APIs (`prototype`, `is_lvalue`, ...).

The four phases are why attributes are heterogeneous in behaviour: built-in attributes are runtime concepts; user attributes are whatever code in `MODIFY_CODE_ATTRIBUTES` decides to do with them.

Built-in attributes

`:lvalue`

Marks the sub as usable on the left-hand side of an assignment. The body's last expression must be an lvalue. See *lvalue and context*.

```
sub editable :lvalue { $self->{x} }

$obj->editable = 42;            # writes through to $self->{x}
```

`:method`

Hint to the parser and to introspection that this sub is meant to be called as a method. The flag is mostly informational; it also suppresses prototype effects (since methods don't honour prototypes anyway):

```
package Counter;

sub increment :method {
    my $self = shift;
    $self->{n}++;
}
```

`:method` does **not** automatically `use parent` or set `@ISA`; it does not change dispatch. It is a documentation / intent marker that the runtime also uses to skip prototype processing.

`:prototype(...)`

The attribute form of a *prototype*, required when *signatures* are enabled in the same scope:

```
use feature 'signatures';
sub mygrep :prototype(&@) ($code, @list) {
    grep { $code->($_) } @list;
}
```

The body's parameter list is the signature; the attribute carries the prototype. Without the attribute form, the parser cannot tell which parenthesised list is which.

`:const`

Promises the runtime that the sub's body produces a constant result, enabling more aggressive inlining than the bare `()` prototype gives:

```
sub PI :const { 3.14159265358979 }
```

`:const` was introduced for use by `use constant` and similar tools. Most code should reach for `use constant` rather than applying `:const` directly.

Third-party attributes

Any package can register attribute handlers for code refs by implementing `MODIFY_CODE_ATTRIBUTES`. The classic example is `Attribute::Handlers`, which provides a more declarative API on top of the raw mechanism. A user attribute looks identical to a built-in at the call site:

```
use Some::Memoize;                # provides ':Memoize' attribute

sub expensive :Memoize {
    ...
}
```

The package providing the attribute decides what `:Memoize` means. Common third-party attributes:

- `:Memoize` - automatically cache return values (from `Memoize`).
- `:Logged` - wrap the sub with logging entry/exit (from various AOP-style modules).
- `:Test`, `:Benchmark` - testing-framework markers.

When you see a colon-attribute that isn't on the built-in list above, find the package supplying it.

How attributes are written

Several attributes can be chained. The ordering on disk is the ordering they're processed in:

```
sub fancy :method :prototype(\@) :Memoize {
    ...
}
```

Attributes attach to *both* a forward declaration and the definition; if you write both, the attribute lists must agree:

```
sub greet :method;                # forward declaration with :method
sub greet :method ($name) {       # body, same attribute
    ...
}
```

A signature, when present, comes after attributes:

```
sub greet :method ($name) {       # OK
    ...
}

sub greet ($name) :method {       # error: attributes must precede
↪signature
    ...
}
```

What attributes do *not* do

- They are not type annotations. `:Int` is not a built-in; attempts to read it that way will silently do nothing unless some package has registered a handler that means the same thing.
- They are not metadata for the test framework unless the test framework registered a handler for them. Sprinkling `:tested` on subs does nothing without code on the other side reading the annotation.
- They are not optimisation directives. `:fast` does nothing.

Querying a sub's attributes

```
use attributes;

sub demo :method :prototype($) :Memoize { ... }

my @attrs = attributes::get(\&demo);
# returns: ('method', 'prototype($)', 'Memoize')
```

This is the introspection equivalent of `prototype` - it answers "what annotations does this sub carry?" rather than "what does this sub look like to the parser?".

See also

- *Declaration* - where the `:attr` syntax fits in the declaration grammar.
- *Prototypes* - `:prototype(...)` is one attribute among several.
- *Lvalue and context* - the meaning of `:lvalue`.
- `prototype` - runtime introspection of one specific built-in attribute.
- *Object-Oriented Programming* - `:method` and the broader OO context.

Phasers: BEGIN, UNITCHECK, CHECK, INIT, END

Five specially named blocks let code hook the transitions a program makes between being compiled and being run. They look like subroutines but are not: you can write several of each, none has a callable name, and each one fires automatically at its own moment.

```
BEGIN     { ... }   # during compilation, as soon as defined
UNITCHECK { ... }   # right after this compilation unit is compiled
CHECK     { ... }   # after the whole program is compiled
INIT      { ... }   # just before the runtime begins
END       { ... }   # just before the interpreter exits
```

The sub prefix (`sub BEGIN { ... }`) is accepted but adds nothing and is discouraged. These are blocks, not named subs; you cannot call them, only let them fire.

When each one runs

A Perl program lives in two phases: a **compile phase** that parses the source into an internal form, and a **run phase** that executes it. The phasers mark the boundary.

- `BEGIN` runs the instant its closing brace is parsed, before the rest of the file is even read. This is how `use` works: a `use` is a `BEGIN`-wrapped `require` plus `import`, so the imported names exist for the code that follows. Once a `BEGIN` has run, it is undefined and its memory reclaimed.
- `UNITCHECK` runs as soon as the **compilation unit** that contains it has finished compiling. A unit is the main program, a module loaded by `require`, a string `eval`, a `do` FILE, or code compiled by the `(?{ })` regex construct. Because units finish compiling at different times, `UNITCHECK` is the only phaser whose timing tracks the individual file rather than the program as a whole.
- `CHECK` runs once, after the **initial** compile phase of the whole program ends and before the run phase starts.
- `INIT` runs once, just before the run phase begins, after all `CHECK` blocks.
- `END` runs once, as late as possible: after the program has finished (including after a `die`), just before the interpreter exits.

`BEGIN` and `UNITCHECK` are tied to *parsing*, not to the interpreter phase, so they can fire during any phase: a string `eval` at run time still compiles its body, and any `BEGIN` or `UNITCHECK` inside it runs then and there. `CHECK` and `INIT`, by contrast, mark the one-time compile/run boundary of the main program.

Multiplicity and execution order

You may write any number of each block. They all run, in a defined order:

- `BEGIN`: first in, first out (FIFO) as encountered during compilation. The first `BEGIN` in source order runs first.
- `UNITCHECK`: last in, first out (LIFO) within each unit, after that unit compiles.
- `CHECK`: LIFO, after all compilation.
- `INIT`: FIFO, just before the runtime.
- `END`: LIFO at exit. The last `END` defined is the first to run.

The instructive case is a file that mixes ordinary statements with all five block kinds. The blocks do not run where they are written; they run at their phase, in their per-phase order, while the ordinary statements run in between at run time. Conceptually, for a single file:

```
1. BEGIN blocks      -- FIFO, during compilation
2. UNITCHECK blocks  -- LIFO, after this unit compiles
3. CHECK blocks      -- LIFO, after all compilation
4. INIT blocks       -- FIFO, just before runtime
5. ordinary code     -- run phase, in source order
6. END blocks        -- LIFO, at exit
```

So BEGIN and UNITCHECK run before any CHECK; CHECK runs before INIT; INIT runs before the first line of ordinary code; and every END runs after the last. Within BEGIN and INIT, earlier-written blocks run earlier; within UNITCHECK, CHECK, and END, later-written blocks run earlier.

END blocks in detail

END is the cleanup phaser. It runs LIFO so that teardown unwinds in the reverse of setup, the way nested resources expect.

Inside an END block, `$?` holds the value the program is about to pass to `exit`. You may read it to learn how the program is ending, and you may assign to it to change the exit code:

```
END {
    $? = 0 if $? == 42;     # rewrite one exit code on the way out
}
```

Beware of clobbering $? by accident. Running anything via `system` inside an END block overwrites $? with the child's status, which then becomes the program's exit code unless you save and restore it.

END blocks do **not** run in several cases:

- Under the `-c` switch (compile-only syntax check). Main code does not run, and neither do END blocks.
- When compilation fails. A program that does not compile never reaches the point where END blocks would fire.
- When the process replaces itself with `exec`. The image is gone; there is nothing left to run the block.
- When the process is killed by a fatal signal you do not trap. Catch the signal yourself (via `%SIG`) if you need cleanup to run.

An END block created inside a string `eval` is **not** run when that `eval` finishes. It is registered like any other END block of its package and runs in LIFO order just before the interpreter exits.

${^GLOBAL_PHASE} and the phases

The read-only variable `${^GLOBAL_PHASE}` names the interpreter's current phase. Its values let code that runs in more than one phase find out where it is. During a CHECK block it reads CHECK; during INIT, INIT; during ordinary run-time code, RUN; during an END block, END. See `${^GLOBAL_PHASE}` for the full list of values and the phase timeline.

defer blocks

A `defer` block is END's small-scale cousin. It runs when the enclosing **block scope** exits, for any reason, rather than when the whole program exits. Reach for `defer` to pair acquisition with release inside a sub or loop; reach for END for process-wide teardown that must happen once, at the very end.

See also

- `BEGIN` *and* `use` - `use` is a `BEGIN`-wrapped load-and-import; the most common reason to care about compile-phase timing.
- `${^GLOBAL_PHASE}` - read the current phase from inside any block to learn where you are.
- `defer` - block-scoped cleanup, the right tool when `END`'s program-wide reach is too broad.
- `$?` - the exit status an `END` block can read and rewrite.
- `%SIG` - trap the signals that would otherwise bypass `END` blocks entirely.
- `exit` - what `END` blocks run just before; the value it carries is the `$?` an `END` block sees.
- *Declaration* - `sub NAME { ... }`, anonymous subs, forward declarations, code references, the `&` sigil and what predeclaration buys you.
- *Arguments and* `@_` - the aliasing rule, the unpacking idiom, named-arg conventions, `shift` defaults, the `chomp(@lines)` legitimate use of aliasing, and the `$_[0] = ...` trap.
- *Return values* - explicit `return`, implicit last-expression, list-flattening on return, void-context shortcuts, and the difference between `return;` and `return undef;`.
- *Scoping:* `my`, `our`, `local`, `state` - pad lifetime, closures, the closure-over-loop-variable surprise, when `local` is the right tool, and what `state` actually means.
- *Prototypes* - what they do (call-site parsing), what they don't (runtime checks), the `(\@)` vs `(@)` difference, the `()` constant-fold prototype, and why most modern code shouldn't reach for them.
- *Signatures* - the modern syntax from 5.20+ stable in 5.36, with positional, optional, and slurpy parameters; defaults; named parameters via slurpy hash; and how signatures interact with (and don't replace) prototypes.
- *Recursion* - depth limit, mutual recursion, `goto &fn` for tail calls, `__SUB__` for anonymous self-reference, trampolines.
- *Lvalue subs and context* - `:lvalue` semantics, `wantarray`, the void-scalar-list three-way switch, and the standard "return early in void context" pattern.
- *Attributes* - `:method`, `:lvalue`, `:prototype(...)`, `:const`, third-party attributes, and how the four attribute-related steps (parse, register, store, query) compose.
- *Phasers* - the `BEGIN`, `UNITCHECK`, `CHECK`, `INIT`, and `END` blocks: when each fires relative to the compile/run boundary, their FIFO/LIFO ordering, `$?` and `${^GLOBAL_PHASE}` inside `END`, and the `defer` block.

Defining a sub vs. calling a sub

The two halves of the subroutine system are mostly orthogonal:

- **Definition** is a declaration plus a body. The declaration fixes the name (or makes the sub anonymous), the optional prototype or signature, the optional attributes. The body is a block of statements with its own pad.
- **Call** turns a list of expressions into the new pad's `@_` (or signature variables), runs the body, and collects whatever the body returned in the caller's context.

Almost every "subtle" subroutine question is a question about *one* side or the other. Prototypes affect calls only, not definitions; signatures affect what the body sees, not how the caller writes the call; attributes affect how the *defined* sub is treated by the runtime.

The classic vs. modern split

Perl carries two parallel argument-passing mechanisms:

```
# Classic: @_ aliasing, manual unpacking
sub add {
    my ($a, $b) = @_;
    return $a + $b;
}

# Modern: signatures, named lexicals, arity checking
use feature 'signatures';
sub add ($a, $b) {
    return $a + $b;
}
```

Both are fully supported. Signatures are not a replacement for `@_`; they are an additional, more declarative entry into the same calling convention. Inside a signature-using sub, `@_` is still populated (this is implementation-defined and discouraged to rely on, but it is the case in 5.44).

The split between classic and modern is the *single most useful piece of orientation* when reading other people's Perl. Most older code uses the classic form; most newer code uses signatures; both will be in front of you on any non-trivial codebase.

See also

- `perlop` - operators that interact with subs: `->` for method dispatch and code-ref invocation, `,` (and the fat comma) for named-arg construction, `=` for the list-assignment used in the `my (...) = @_` idiom.
- `perlvar` - special variables a sub will read or write: `@_`, `$_`, `$@`, `$AUTOLOAD`.
- `perlfunc` - the verbs of the subroutine system: `sub`, `return`, `caller`, `wantarray`, `prototype`, `goto`, `my`, `our`, `state`, `local`.
- *Object-Oriented Programming* - subs as methods, the role of `bless`, dispatch via `@ISA`, the modern `use experimental 'class'` syntax.

5.1.6 Locale handling

A *locale* is the set of cultural conventions a program follows when it sorts text, changes case, formats numbers and money, names the days of the week, or writes an error message. The same array of words sorts one way under a Swedish locale and another under a German one; the number twelve-and-a-half prints as `12.5` in one country and `12,5` in another. Locale handling is how Perl lets a single program adapt to those conventions at runtime instead of hard-coding one country's rules.

By default PetaPerl ignores locale entirely: every built-in behaves the same regardless of the user's environment, using the C/POSIX conventions (ASCII collation, `.` as the decimal point, English month names). You opt in to locale-awareness, one lexical scope at a time, with the `use locale` pragma. This page covers what that pragma turns on, which operations it affects, how to set and inspect the active locale, and where locale and Unicode interact.

What a locale is

The system C library divides cultural conventions into independent *categories*, each named `LC_` followed by the area it governs. Sorting (`LC_COLLATE`) is separate from case rules (`LC_CTYPE`), which is separate from number formatting (`LC_NUMERIC`), and so on. A locale is a named bundle of settings - `de_DE.UTF-8`, `sv_SE.UTF-8`, `C` - and switching to it can change all categories at once or just the ones you select. The categories PetaPerl honours are listed in the category table below.

The `use locale` pragma

Locale-sensitive behaviour is **off by default** and **lexically scoped**. `use locale` enables it from that point to the end of the enclosing block (or file); `no locale` turns it back off for the remainder of its own scope. Nothing outside the lexical scope is affected - a module that does not say `use locale` keeps the default C-locale behaviour even when called from code that did.

```
use locale;                  # locale-aware from here down
my @sorted = sort @words;    # collates per LC_COLLATE
{
    no locale;               # back to codepoint order in this block
    my @plain = sort @words;
}
```

Without the pragma, `sort`, `lc`, `uc`, `printf` and the rest use fixed C-locale rules no matter what the environment says. This is a deliberate safety default: a program produces the same output on every machine until it explicitly asks to follow the local conventions.

The pragma sets a per-statement hint, exactly as upstream Perl does, so a single file can mix locale-aware and locale-blind regions freely. The setting travels with the lexical scope, not with the call stack.

Locale categories

Each category controls a distinct group of operations. `use locale` activates all of them at once; you choose *which* locale is active per category through `setlocale`. The "affected operations" column links to the reference pages where the behaviour is documented in full - that is where the locale-specific rules for each built-in live.

Category	Governs	Affected operations
`LC_COLLATE`	String sort order and comparison	`sort`, the `lt le gt ge cmp` string comparisons
`LC_CTYPE`	Character classification and case mapping	`lc`, `uc`, `lcfirst`, `ucfirst`, `\w \d \s` and `\b` in regex
`LC_NUMERIC`	Decimal point and digit grouping	`printf`, `sprintf`, *number-to-string conversion*
`LC_MONETARY`	Currency formatting conventions	`localeconv` (inspection only; no built-in formats money for you)
`LC_TIME`	Day and month names, date layout	`strftime` via `POSIX`, `langinfo`
`LC_MESSAGES`	Language of system error strings	`$!` text, `langinfo` message strings

`LC_ALL` is not a category of its own but a shorthand that sets every category at once.

Note one asymmetry that trips people up: `LC_NUMERIC` changes how numbers are *formatted for output*, but Perl's own source code and its numeric *parsing* always use `.` as the decimal separator. A script that reads `12,5` from a German-locale file still needs to translate the comma itself before treating it as a number - the locale does not make `"12,5" + 0` yield `12.5`. See *numbers and strings*.

Setting the locale

`use locale` says *follow the locale*; it does not say *which* one. That choice comes from the environment by default, and you can query or override it through the `POSIX` module.

`setlocale` selects the active locale for a category. The idiomatic call honours the user's environment (`LANG`, `LC_*`):

```
use POSIX qw(setlocale LC_ALL LC_COLLATE);
setlocale(LC_ALL, "");                # adopt the environment's␣
↪locale
my $cur = setlocale(LC_COLLATE);      # query the active collation
setlocale(LC_COLLATE, "sv_SE.UTF-8"); # force one explicitly
```

`localeconv` returns a hash describing the active `LC_NUMERIC` and `LC_MONETARY` conventions - the decimal point, thousands separator, currency symbol, and the rest - so a program can format values itself when no built-in does it directly.

`I18N::Langinfo` exposes `langinfo`, which reads individual items out of the current locale: the localized name of a weekday or month (`DAY_1`, `MON_1`), the date format string, the radix character, the currency symbol. It is the narrow, item-at-a-time companion to `localeconv`.

Locale and Unicode

Locale handling and Unicode are two different answers to "what is a character", and they interact in ways worth understanding before you combine them. The full treatment - how `use utf8`, UTF-8 I/O layers, and locale `LC_CTYPE` rules compose, and the traps that arise when they disagree - lives in the Unicode tutorial. Start with *Unicode strings and encodings* and the *pitfalls chapter*; the *regex-and-properties chapter* covers how `\w` and the character classes behave under each model.

Locale leaks and safety

A locale is process-global state in the C library underneath, so a `setlocale` call made anywhere changes the locale everywhere in the process - across module boundaries, not just the scope that called it. Two consequences follow:

- **`use locale` is safe to scope; `setlocale` is not.** The pragma is lexical and leaks nothing. A `setlocale` call is a global side effect - if a library sets `LC_NUMERIC` to a comma-decimal locale and forgets to restore it, every later `printf` in the program is affected. Save and restore around a temporary change:

```
my $saved = setlocale(LC_NUMERIC);
setlocale(LC_NUMERIC, "de_DE.UTF-8");
# ... locale-sensitive work ...
setlocale(LC_NUMERIC, $saved);
```

- **Untrusted locale names are untrusted input.** Passing an environment-supplied string straight to `setlocale` lets the environment steer your program's formatting. In a security-sensitive context, validate the name or pin the locale explicitly rather than calling `setlocale(LC_ALL, "")`.

Parallel execution and locale

PetaPerl auto-parallelizes some numeric loops across worker threads. This never affects locale-sensitive code, by construction. Operations whose results depend on the current locale - `sort` with the default collation, `LC_COLLATE` string comparison, `lc`/`uc`/`fc` case folding, and `LC_NUMERIC` number formatting - are never JIT-compiled or auto-parallelized; they always execute sequentially in the interpreter against the single process-wide locale. Auto-parallelization only ever runs numeric loop bodies, so no locale-sensitive computation is dispatched across worker threads. You do not need to guard locale state when writing parallelizable code: the parallel path simply never touches it.

Environment variables

When a program calls `setlocale(LC_ALL, "")`, the C library reads the locale from the environment in a fixed order of precedence: `LC_ALL` overrides every individual `LC_*` variable, each `LC_*` variable sets its own category, and `LANG` supplies the default for any category not otherwise named. These variables live in `%ENV` like any other environment value, so a program can inspect or adjust them before the `setlocale` call that consumes them.

Variable	Effect
`LC_ALL`	Overrides all categories
`LC_COLLATE`, `LC_CTYPE`, ...	Sets one named category
`LANG`	Default for any category not set above

Differences from upstream

- **Linux only, UTF-8 native.** PetaPerl runs only on Linux against the system C library's locales. The platform-specific locale machinery upstream Perl carries for VMS, Windows, and Android - the `Win32::` locale shims, the VMS category quirks, the "locales are not available on this platform" fallbacks - has no counterpart here and is simply absent. There is one locale model: the POSIX one your `glibc` provides.
- **`LC_NUMERIC` and source parsing.** As upstream, the decimal separator used when Perl *parses* numeric literals and string-to-number conversions is always `.`, regardless of locale; only *formatting* output respects `LC_NUMERIC`. PetaPerl makes no exception to this.
- **Auto-parallelization never touches locale-sensitive code.** PetaPerl auto-parallelizes some numeric loops across worker threads, a capability upstream Perl does not have. Because locale-dependent operations (collation, case folding, number formatting) are never JIT-compiled or parallelized, they always run sequentially against the single process-wide locale - so the feature introduces no locale-safety concern of its own. See *Parallel execution and locale* above.

See also

- `perlfunc` - the built-ins whose behaviour locale changes: `sort`, `lc`, `uc`, `lcfirst`, `ucfirst`, `sprintf`, `printf`.
- `perlop` - the string comparison operators (`lt`, `cmp`, ...) that collate under `LC_COLLATE`.
- `perlvar` - `%ENV` holds the `LANG`/`LC_*` variables; `$!` carries locale-translated error text.
- `POSIX` - `setlocale` and `localeconv`, the interface for choosing and inspecting the active locale.
- `I18N::Langinfo` - `langinfo`, for reading one localized item (month name, radix character) at a time.
- *Unicode strings and encodings* - how the Unicode character model relates to locale `LC_CTYPE`.

5.1.7 Diagnostic messages

When a program goes wrong, Perl tells you about it. Some diagnostics stop the program (`die`-style fatal errors); others print a line to `STDERR` and let it run on (warnings). This reference lists the diagnostics PetaPerl emits, grouped by where they come from, with the cause and the fix for each.

Every entry shows the **message verbatim** in a code span - the exact text the interpreter prints, with the placeholders (`%s`, `%d`, a named variable) shown the way they appear in the output. Search this page for the literal words you saw on `STDERR` and you will land on the entry.

Reading an entry

Each entry opens with the message text, then a short classification badge in parentheses, then the cause and the fix:

> `` `Illegal division by zero` `` **(F)** You divided by zero. Guard the divisor.

The badge follows the same letter code Perl has always used:

Code	Meaning
W	A warning - printed only when warnings are enabled. Optionally restricted to a category (shown as `(W category)`).
D	A deprecation - a warning that a feature is on the way out.
S	A severe warning - on by default, but still only a warning.
F	A fatal error - the program dies unless the error is trapped (`eval`, `try`).
P	An internal error (a panic) - should never reach a user. If you see one, it is a bug in pperl.
X	A very fatal error - the program exits immediately.
A	An alien error - produced by something other than Perl itself.

Enabling and silencing warnings

Warnings (the **W**, **D**, and **S** classes) are controlled by the `warnings` pragma. Without it, only the severe (**S**) warnings print:

```
use warnings;                       # turn on every warning
no warnings;                        # turn them all off (lexically)
no warnings 'uninitialized';        # turn off one category
use warnings FATAL => 'all';        # promote warnings to fatal errors
```

The category named in a `(W category)` badge is the string you pass to `no warnings` to silence just that diagnostic. A single `$SIG{__WARN__}` handler can intercept every warning at runtime; `$SIG{__DIE__}` does the same for fatal errors. See `%SIG`.

A note on portability

PetaPerl runs only on Linux and ships no XS layer. The platform diagnostic families a portable perl carries - VMS status codes, Win32 `GetLastError` text, "unimplemented on this architecture" notices, XS-loader failures - therefore do not arise here. Where upstream Perl documents a diagnostic that only fires on another operating system or through XS, PetaPerl simply never emits it.

The diagnostics

General fatal errors

Runtime dies that are neither regex-compilation nor compile-time syntax errors: method resolution, inheritance, subroutine calls, attributes, and module-level usage errors. All are fatal (**F**) and trappable with `eval` or `try`.

Subroutine calls

`` `Undefined subroutine &%s called` ``

(F) A call named a subroutine that does not exist at call time. The fully-qualified name is shown. Causes: a typo in the name, a missing `use` of the module that defines it, or calling before the `sub` is declared in a context where that matters. The runtime form `` `Undefined subroutine &NAME called at FILE line N.` `` adds the call site.

Method resolution and inheritance

`` `Can't use anonymous symbol table for method lookup` ``

(F) Method dispatch reached a package with no named symbol table (an anonymous stash). A blessed reference must point into a real, named package for method calls to resolve.

`` `Recursive inheritance detected in package '%s'` ``

(F) A class's `@ISA` chain leads back to itself - a cycle. Method resolution cannot proceed. Break the loop in the inheritance graph.

`` `Inconsistent hierarchy during C3 merge of class '%s'` ``

(F) Under the C3 method-resolution order, the linearization of this class's parents has no consistent ordering. Two parents disagree on the order of a shared ancestor. Restructure the inheritance so the parent orderings are compatible, or switch the class back to the default depth-first MRO.

`` `Invalid mro name: '%s'` ``

(F) A `use mro '...'` named an order that does not exist. The two valid names are `` `dfs` `` (the default depth-first order) and `` `c3` ``.

Attributes

`` `Unterminated attribute parameter in attribute list` ``

(F) An attribute with a parenthesized parameter - `:Foo(bar` - was not closed. Balance the parentheses in the attribute list.

Module usage errors

These come from native modules when called incorrectly. They are fatal because the call cannot proceed.

`Usage: POSIX::sprintf(pattern, args...)`

(F) A `POSIX` function was called with the wrong arguments. The message names the correct calling form.

`IO::Socket::%s`

(F) An `IO::Socket` operation failed; the message names the specific call.

See also

- `@ISA` - the inheritance array whose contents the MRO errors are about.
- `mro` - selecting and inspecting the method-resolution order.
- `bless` - associates a reference with the package whose methods the anonymous-symbol-table error is about.
- *Value and type errors* - "Not a subroutine reference" and the other what-kind-of-value diagnostics.

Regex compilation errors

Every diagnostic on this page comes from the pattern compiler. A regular expression is compiled before it runs, and a malformed pattern dies at that point - before the match is ever attempted. The errors are all fatal (**F**); trap them with `eval` if you build patterns from untrusted input:

```
my $re = eval { qr/$user_pattern/ };
die "bad pattern: $@" unless $re;
```

Upstream Perl appends a `marked by <-- HERE in m/.../` suffix that points at the offending position. PetaPerl reports the same errors; the exact position-marker formatting may differ.

Common structural errors

`Unmatched ) in regex`

(F) A `)` in the pattern has no matching `(`. Usually a typo or an unescaped literal parenthesis. To match a literal `)`, escape it as `\)` or put it in a character class `[)]`.

`Quantifier follows nothing in regex`

(F) A quantifier (`*`, `+`, `?`, `{n,m}`) appeared with nothing in front of it to repeat - for example `/*abc/` or `/(?:)+/` on an empty group. Quantify a literal `*` by escaping it: `/\*/`.

`Unterminated group`

(F) A `(` opened a group that the pattern never closes. Balance the parentheses.

`Unterminated character class`

(F) A `[` opened a character class with no closing `]`. To match a literal `]`, place it first in the class (`[]x]`) or escape it (`[\]x]`).

`Unterminated comment group`

(F) A `(?#...)` comment group was opened but not closed with `)`.

`Unterminated conditional pattern`

(F) A `(?(condition)yes|no)` construct was left unclosed.

`Unterminated group name`

(F) A named-group construct (`(?<name>...)`, `\k<name>`) ran to the end of the pattern without closing its name delimiter.

`Unrecognized group type`

(F) The characters after `(?` do not form a group type PetaPerl knows. Check the `(?...)` spelling against the regex reference.

`Unexpected end of character class`

(F) The pattern ended in the middle of a `[...]` class.

`Unexpected end of escape sequence`

(F) A backslash escape was cut off by the end of the pattern.

Character-class and range errors

`Invalid [] range "%s" in regex`

(F) A character range inside `[...]` runs backwards (high to low), as in `[z-a]`. Order the endpoints low to high, or escape the `-` to match it literally.

`Invalid range endpoint in character class`

(F) One end of a `-` range is not a single character (for example a multi-character escape used as an endpoint).

`POSIX class [:%s:] unknown`

(F) A `[:name:]` POSIX class names a class that does not exist. The valid names are `alpha`, `digit`, `alnum`, `space`, `upper`, `lower`, `punct`, `print`, `graph`, `cntrl`, `xdigit`, `blank`, `word`, and `ascii`. A bare `[:alpha:]` outside another `[...]` is also an error - POSIX classes only work *inside* a character class: `[[:alpha:]]`.

`Empty POSIX character class name`

(F) A `[::]` with nothing between the colons.

Control-character and numeric escapes

`Missing control char`

(F) A `\c` escape was not followed by the character it should control, as in a pattern ending in `\c`.

`Missing control character after \c`

(F) Same cause as above, from a different point in the compiler: `\c` needs one following character.

`Invalid control character '\c%s'`

(F) The character after `\c` is not one `\c` accepts.

`Invalid octal escape`

(F) A `\0`/`\o` octal escape is malformed.

`Invalid octal digit in \o{}`

(F) A digit inside `\o{...}` is not 0–7.

`Expected '{' after \o`

(F) `\o` must be followed by `{`, as in `\o{17}`.

`Expected '}' in \o{}` and `Invalid Unicode code point in \o{}`

(F) A `\o{...}` escape is unterminated or names a code point outside the valid range.

`Invalid hex escape` / `Invalid hex digit` / `Expected '}' in hex escape`

(F) A `\x` escape is malformed: a non-hex digit, or an unterminated `\x{...}`.

\g, \k, and group references

`\g0 is not allowed` and `\g{0} is not allowed`

(F) Group 0 is the whole match and cannot be back-referenced. Reference a real capture group (`\g1`, `\g{name}`).

`Invalid group number in \g` / `Invalid group number in \g{}`

(F) The number after `\g` is not a valid group reference.

`Expected digit or '{' after \g` / `Expected '}' after \g{`

(F) `\g` must be followed by a digit, a sign, or `{name}`.

`Reference to nonexistent group number %d`

(F) A `\g`/`\1`-style back-reference names a group number the pattern does not contain.

`Reference to nonexistent group (relative offset %d)`

(F) A relative back-reference (`\g{-2}`) points past the groups that exist.

`Reference to nonexistent named group '%s'`

(F) A `\k<name>` or `\g{name}` references a named group that was never defined.

`Reference to nonexistent or invalid group`

(F) A group reference could not be resolved to any group.

`Expected '<' or '\'' after \k` / `Expected '}' after \k{`

(F) A named back-reference `\k` must be written `\k<name>`, `\k'name'`, or `\k{name}`.

Named groups

`Empty group name`

(F) A named group `(?<>...)` or reference has empty name brackets.

`Group name must not start with a digit`

(F) Group names follow identifier rules; they may not begin with a digit.

\N and Unicode escapes

`\N in character class requires braces`

(F) Inside `[...]`, `\N` must be written `\N{...}`; the bare “any non-newline” form is not allowed in a class.

`\N{NAME} not supported in character class`

(F) A named-character `\N{NAME}` escape is not permitted inside a character class.

`Expected '}' after \N{` / `Empty \N{U+} escape`

(F) A `\N{...}` escape is unterminated or `\N{U+}` has no hex digits.

`Invalid hex in \N{U+...}` / `Invalid Unicode code point in \N{U+...}`

(F) The hex inside `\N{U+...}` is malformed or names a code point outside the Unicode range.

`Invalid Unicode code point`

(F) A code-point escape names a value above the Unicode maximum (`0x10FFFF`).

`Empty Unicode property name` / `Unknown Unicode property name`

(F) A `\p{...}` / `\P{...}` property escape is empty or names a property PetaPerl does not recognize.

`(?...)` extended constructs

`Invalid flag after (?^` / `Invalid flag character`

(F) A flag-setting group `(?flags)` or `(?^flags:...)` contains a character that is not a valid pattern modifier.

`Expected '{' after (??`

(F) A `(??{ code })` postponed-subpattern construct is malformed.

`Expected '<', '=', or '>' after (?P`

(F) The Python-style `(?P<name>...)` / `(?P=name)` / `(?P>name)` construct is malformed.

`Expected DEFINE after (?(` / `Expected ')' after DEFINE`

(F) A `(?(DEFINE)...)` block is malformed.

`Switch condition not recognized in regex` / `Invalid lookaround condition`

(F) The condition in a `(?(...)...)` conditional is not a recognized form (a group number, a lookaround, `R`, or `DEFINE`).

`Expected ')' after condition` family

(F) A conditional construct's condition was not closed with `)`. The variants - `Expected ')' after (?(R...)`, `Expected ')' after code block condition`, `Expected ')' after lookahead condition`, `Expected ')' after lookbehind condition`, `Expected '=' or '!' after (?(<` - all report the same shape of error at different condition types.

Quantifier braces

`Expected '}' after quantifier` / `Expected '}' or ',' in quantifier`

(F) A `{n,m}` quantifier is unterminated or contains an unexpected character.

Lookbehind

`Variable length lookbehind not implemented`

(F) A lookbehind `(?<=...)` / `(?<!...)` whose contents can match a variable number of characters. PetaPerl, like upstream Perl, requires fixed-length lookbehind. Rewrite the lookbehind to a fixed width, or use `\K` to mark a keep-point instead.

Verbs

`Unterminated verb`

(F) A control verb `(*VERB)` / `(*VERB:ARG)` was not closed.

Catch-all

`Regex compilation error at position %d: %s` / `Regular expression error`

(F) A position-tagged compile error for cases the more specific messages above do not cover. The trailing text names the underlying problem.

`Unterminated string in %s`

(F) A delimited construct inside the pattern ran to the end of input without its closing delimiter.

See also

- `qr` - precompile a pattern so a malformed one fails once, at compile time, instead of on every match.
- `m` - the match operator; the same compiler rejects a malformed pattern here.
- `s` - the substitution operator, whose pattern half is compiled by the same engine.
- *Warnings* - the non-fatal regex diagnostics (`Useless use of \E`, `Invalid character in group name`).

Compile-time syntax errors

Errors the parser raises before the program runs. A compile-time error stops the whole program: nothing executes, because the source could not be turned into runnable ops. Unlike runtime `die`s, these are not trappable with `eval BLOCK` - but a string `eval` of source that fails to compile reports them through `$@`:

```
my $code = '1 +';
eval $code;
warn "compile failed: $@" if $@;
```

Subroutine resolution at compile time

`Undefined subroutine &%s`

(F) The parser resolved a call to a named subroutine that does not exist at compile time. This is the compile-time companion to the runtime `Undefined subroutine &%s called` error: the difference is *when* the missing sub is noticed. Make sure the defining module is `use`d, and that the name is spelled correctly and fully qualified where the package matters.

See also

- *General fatal errors* - the runtime form of the undefined-subroutine error.
- *Subroutines* - declaration, prototypes, and the rules that decide when a call resolves.
- `use` - pulls in the module that defines the subroutine, at compile time.

Warnings

Non-fatal diagnostics. A warning prints one line to `STDERR` and the program runs on. The severe (**S**) warnings print even without the `warnings` pragma; the plain **W** warnings print only when warnings are enabled for their category:

```
use warnings;                    # all categories on
no warnings 'recursion';         # silence just deep-recursion
use warnings FATAL => 'utf8';    # make utf8 warnings die instead
```

The category in a `(W category)` badge is exactly the string you pass to `no warnings`.

Deep recursion

`Deep recursion on subroutine "%s"`

(W recursion) A subroutine has called itself (directly or through a chain) more than a hundred levels deep. This is usually a runaway recursion - a base case that never fires - rather than genuinely deep work. The message names the subroutine.

```
use warnings;
```

(continues on next page)

(continued from previous page)

```
sub countdown {
    my $n = shift;
    countdown($n + 1);            # bug: should be $n - 1, never␣
 ↪stops
}
countdown(1);
# Deep recursion on subroutine "main::countdown" at script line 5.
```

Fix the base case. If the recursion really is meant to run deep, silence the category for that scope:

```
no warnings 'recursion';
```

Wide characters

`Wide character in %s`

(S) A string containing a character above codepoint 255 was sent to a byte-oriented operation - most often `print` to a filehandle with no `:encoding` layer. Perl emits the bytes of the internal UTF-8 form and warns, because the output stream was never told it would carry Unicode.

```
use warnings;
print "café\x{2603}\n";          # Wide character in print at ...␣
 ↪line 2.
```

The fix is to tell the stream its encoding:

```
binmode STDOUT, ':encoding(UTF-8)';
print "café\x{2603}\n";          # no warning; correct UTF-8 bytes
```

This is a severe **(S)** warning: it prints even without `use warnings`, because silently emitting raw internal bytes is almost always a bug.

Useless constructs

`Useless use of \E`

(W regexp) An `\E` in a pattern or string closes a `\U`, `\L`, `\Q` case/quote span that was never opened. The `\E` has nothing to end. Remove it, or add the opening `\U`/`\L`/`\Q` it was meant to close.

Group names

`Invalid character in group name`

(W) A regex named group (`(?<name>...)`) contains a character that is not allowed in a group name. Group names follow identifier rules - letters, digits, and underscore, not starting with a digit. The pattern still compiles, treating the construct leniently, but the name is suspect.

Encoding

`Cannot find encoding "%s"`

(S) A `:encoding(...)` layer or an `Encode` call named an encoding that is not available. Check the spelling against the encodings `Encode` supports (`UTF-8`, `Latin-1`, `ASCII`, and the rest).

Catch-all

`Warning: something's wrong`

(W) A generic warning the codegen raises for a condition that has no more specific message. The surrounding context names what triggered it.

See also

- `warnings` - the pragma that enables, scopes, and categorizes every **W** warning.
- `$SIG{__WARN__}` - intercept warnings at runtime instead of letting them reach `STDERR`.
- *Regex compilation errors* - the fatal regex diagnostics, as opposed to the warning-level ones here.
- `Encode` - the encoding names behind the "Cannot find encoding" warning.

Value and type errors

These diagnostics fire when a value is not the kind of thing the operation needs: a plain scalar where a reference was expected, a zero where a divisor was expected, a read-only value where a writable one was expected. All are fatal (**F**) unless noted.

Arithmetic

`Illegal division by zero`

(F) The right-hand side of `/` or `%` evaluated to zero. Numeric division and modulus have no result at zero, so Perl dies rather than returning infinity or `NaN`. Guard the divisor:

```
my $rate = $count ? $total / $count : 0;
```

`Illegal modulus zero`

(F) The right operand of `%` was zero. Same cause and same fix as division by zero, for the modulus operator.

References

`Not a reference`

(F) A dereference (`@$x`, `%$x`, `$$x`, `$x->[0]`, `$x->{k}`) was attempted on a value that is not a reference. Most often `$x` is `undef` or a plain string. Check it first:

```
die "expected an arrayref" unless ref $x eq 'ARRAY';
```

`Not a subroutine reference`

(F) A value used where a code reference was required (for example the second argument to a higher-order function) is not a CODE ref. `set_subname: not a code reference` and `set_prototype: not a code reference` are the same error reported by Sub::Util, and `PadWalker: cv is not a code reference` is the PadWalker form.

Restricted hashes

A hash locked with Hash::Util's lock_keys rejects access to keys outside its allowed set.

`Attempt to access disallowed key '%s' in a restricted hash`

(F) You read or wrote a key that the restricted hash does not permit. Either add the key to the allowed set, or unlock the hash.

`Attempt to delete readonly key '%s' from a restricted hash`

(F) You tried to delete a key that the hash has locked against deletion.

Read-only values

`Modification of a read-only value attempted`

(F) An assignment or in-place operation targeted something that cannot be changed: a literal, a constant, a value passed by the caller and made read-only, or a Readonly/const value. A common surprise is modifying $_ inside a map/grep block when the source list elements are read-only.

Filehandles

`Bad filehandle: %s`

(F) An I/O operation was handed something that is not a usable filehandle - an undefined value, a closed handle, or a bareword that names no open handle. Open the handle (and check the result) before using it:

```
open my $fh, '<', $path or die "open $path: $!";
```

Memory

`Out of memory`

(F) An allocation failed. Usually the program asked for an implausibly large string, array, or hash - check the size you are requesting before assuming the machine is at fault.

See also

- *References* - what a reference is and how dereferencing works; the background for the "Not a reference" errors.
- `ref` - test what kind of reference a value holds before you dereference it.
- `Hash::Util` - `lock_keys`, `unlock_keys`, and the rest of the restricted-hash machinery.
- *General fatal errors* - method-resolution and inheritance errors that are also about what a value is.

Internal errors

A **P** diagnostic is a panic: a check that should never fail in correct operation did fail. These messages are not something your Perl code can cause through ordinary mistakes - they signal a bug in PetaPerl itself. If you see one, it is worth a bug report. See *Disclaimer & bug reports*.

There is nothing to fix on the Perl side. Capture the message, the input that triggered it, and the `pperl` version, and file it.

`Unknown error`

(P) A fallback message for an error path that produced no specific diagnostic - most often surfaced from a native module (such as the GMP bignum backend) when an underlying library returned a failure code with no text. Treat it as an internal error and report it with the operation that triggered it.

See also

- *Disclaimer & bug reports* - how to file a pperl bug, and what to include.
- *pperl-specific diagnostics* - other messages with no upstream Perl counterpart.

pperl-specific diagnostics

Messages PetaPerl emits that have no counterpart in upstream Perl. Most concern the daemon mode - pperl's persistent-process feature, which has no analogue in a stock perl - and a few native modules with pperl-specific behavior.

Daemon mode

`pperl` can run a script as a resident daemon and dispatch requests to it, which avoids paying interpreter startup on every call. The daemon diagnostics report problems with that channel.

`Daemon already running for this script (socket: %s)`

(F)

A `--daemon` start was requested for a script that already has a running daemon. The socket path is shown. Stop the existing daemon first, or connect to it instead of starting a new one.

`Cannot connect to daemon (%s): %s`

(F)

A client could not reach the daemon socket. The message continues with one of two hints depending on context - `Start with: pperl --daemon <script>` when no daemon appears to be running, or `Is the daemon running?` when the socket exists but the connection failed. Start the daemon, or check that the socket path matches.

Native modules

`ERRNO hash is read only!`

(F)

Code tried to assign to `%!` (the `Errno` symbolic-error hash) or to one of its keys. That hash is populated by the runtime and reflects the current `errno`; it is not writable. Read it (`$!{ENOENT}`) rather than assigning to it.

`Hash has key '%s' which is not in the new key set`

(F)

From `Hash::Util`: a `lock_keys` call supplied an explicit key set that does not include a key the hash already contains. Locking cannot proceed because that existing key would become inaccessible. Either include the existing key in the new set, or delete it before locking.

`Time::HiRes::usleep(%s): negative time not invented yet`

(F)

A `Time::HiRes` sleep function was given a negative duration. Time runs forward; pass a non-negative number of microseconds (or nanoseconds for `nanosleep`). The `nanosleep` form reports the same with its own name.

`%s version %s required--this is only version %s`

(F)

A `use Module VERSION` (or an explicit version check) demanded a newer version of a module than the one installed. The message names the module, the version requested, and the version present. Install or load the required version.

See also

- *Internal errors* - the panics that mean a pperl bug rather than a usage error.
- *Disclaimer & bug reports* - how pperl diverges from stock perl, and where to report problems.
- `Errno` - the module behind `%!`, the read-only symbolic-errno hash.
- `Hash::Util` - `lock_keys` and the restricted-hash machinery.
- *General fatal errors* - method resolution, references, arithmetic, restricted hashes, filehandles, and the rest of the runtime `die`s that are not regex- or parser-specific.
- *Regex compilation errors* - everything the pattern compiler rejects, from `Unmatched )` to malformed `\g`, `\N`, `\o`, and `(?...)` constructs.
- *Compile-time syntax errors* - errors the parser and codegen raise before the program runs.
- *Warnings* - the non-fatal diagnostics: deep recursion, wide characters, useless constructs.
- *Value and type errors* - diagnostics about what a value is and is not: not a reference, not a code reference, division by zero, restricted-hash access.
- *Internal errors* - the panics that signal a bug in pperl itself.
- *pperl-specific diagnostics* - messages PetaPerl emits that have no upstream Perl counterpart.

5.1.8 Security and taint mode

Security in a Perl program is mostly about not trusting data that came from outside the program: command-line arguments, the environment, file contents, network input. The dangerous step is not reading that data but *acting* on it. A filename that arrives from a web form is harmless until it reaches `open`; an environment variable is harmless until it steers a `system` call. This page describes the mechanism PetaPerl gives you to make that boundary explicit: taint mode. It explains what taint mode is, when it turns on, what it marks as untrusted, and how data is cleaned.

Taint mode

PetaPerl runs the perl5 5.44 core, so taint mode is the real perl5 taint machinery, not an approximation. Every check perl5 performs, PetaPerl performs.

Taint mode turns on automatically when PetaPerl detects that it is running with differing real and effective user or group IDs, which is how a setuid (Unix mode `04000`) or setgid (mode `02000`) script is run. You can also enable it explicitly with the `-T` command-line switch. `-T` is strongly recommended for any program run on behalf of someone else, a CGI script being the classic case. Once taint mode is on, it stays on for the rest of the program; it cannot be turned off from inside.

The switch must appear early enough that the interpreter sees it before it starts parsing your program. On the `#!` line of a script this is automatic. On the command line, put it among the first switches. The related `-t` switch enables the same checks but downgrades the fatal errors to warnings, which is useful when retrofitting taint mode onto

an existing program: you see every place that *would* fail without the program dying at the first one.

The read-only variable `${^TAINT}` reports the current state:

`${^TAINT}`	Meaning
`1`	Taint mode on, fatal (`-T`)
`-1`	Taint checks on, warn only (`-t`)
`0`	Taint mode off

What gets tainted

When taint mode is on, any value that came from outside the program is marked tainted, and that mark propagates through every expression the value touches. The tainted sources are:

- command-line arguments (`@ARGV`) and the environment (`%ENV`);
- all file input;
- the results of `readdir` and `readlink`, the variable read by `shmread`, the messages from `msgrcv`, and the password, gcos, and shell fields returned by the `getpw*` calls;
- locale data (see `perllocale`).

Tainted data may not be used, directly or indirectly, in any operation that starts a sub-shell or that modifies a file, directory, or process. Using it in such an operation is fatal, with a message like `Insecure dependency in %s` or `Insecure $ENV{PATH}`. There are three exceptions:

- arguments to `print` and `syswrite` are **not** checked;
- symbolic method calls (`$obj->$method(@args)`) and symbolic subroutine references (`$foo->(@args)`) are **not** checked, so validate those names yourself if they can come from outside;
- hash keys are **never** tainted.

For efficiency, taint is tracked conservatively: if any part of an expression is tainted, the whole expression is treated as tainted, even where the tainted value does not actually influence the result. Taint attaches to individual scalar values, so some elements of an array or hash can be tainted while others are not.

```
my $arg  = shift;           # tainted (from @ARGV)
my $hid  = $arg . 'bar';    # also tainted
my $line = <STDIN>;         # tainted (file input)
my $data = 'abc';           # not tainted

system "echo $arg";         # dies: Insecure dependency
system "/bin/echo", $arg;   # dies: tainted argument
system "echo $data";        # dies until PATH is cleaned (see
 ↪below)
```

The one structural exception to whole-expression tainting is the ternary `?:` operator. Because

```
my $result = $tainted ? "untainted" : "also untainted";
```

is just a conditional choice between two constants, `$result` is not tainted even though the condition is.

Laundering and detecting tainted data

The **only** way to clear the taint on a value is to extract a substring of it through a regular-expression capture. PetaPerl assumes that if you bother to write a pattern and pull out `$1`, you have thought about what you are willing to accept. Untaint by validating against a whitelist of acceptable characters, never by trying to strip out the bad ones:

```
if ($data =~ /^([-\@\w.]+)$/) {
    $data = $1;              # $data is now untainted
} else {
    die "rejected suspicious input";
}
```

This is reasonably safe because `\w`, a dot, a dash, and an at sign do not carry special meaning to a shell. A pattern like `/(.+)/` would "launder" the value while accepting anything, which defeats the purpose. Be deliberate about the pattern.

There is a locale caveat. Under `use locale` the set of characters that `\w` matches is itself drawn from the locale, which is external data PetaPerl does not trust, so a `\w` match under `use locale` does **not** untaint. If you need to launder with `\w` in a locale-aware program, put `no locale` ahead of the match in the same block.

To test whether a value is tainted, use `tainted` from the `Scalar::Util` module. Under `-T` it correctly reports the taint state of its argument.

Cleaning up %ENV and $ENV{PATH}

A tainted `PATH` is the most common surprise. Because PetaPerl cannot know that a program it runs will not itself consult `PATH`, it refuses to start a sub-process while `$ENV{PATH}` is tainted, with `Insecure $ENV{PATH}`. Every directory in the path must be absolute and writable only by its owner and group; a writable directory yields `Insecure directory in %s`. Note that an empty path component is treated as `.` (the current directory) and also triggers the message.

`PATH` is not the only offender. Because a shell may consult `IFS`, `CDPATH`, `ENV`, and `BASH_ENV`, PetaPerl requires those to be empty or untainted before it will start a sub-process, reporting `Insecure $ENV{%s}` otherwise. Set a known-good `PATH` and delete the shell variables early:

```
$ENV{PATH} = '/bin:/usr/bin';
delete @ENV{qw(IFS CDPATH ENV BASH_ENV)};
```

When you must open a file or pipe from a privileged (setuid or setgid) program, the safe pattern is to fork a child, have the child drop its privileges back to the real user and group, and let the unprivileged child do the work and pass the result back over a pipe:

```
use English;
die "can't fork: $!" unless defined(my $pid = open(my $kid, "-|"));
if ($pid) {                             # parent
    while (<$kid>) {
        # consume the child's output
    }
    close $kid;
} else {                                # child
    my @priv     = ($EUID, $EGID);
    my $real_uid = $UID;
    my $real_gid = $GID;
    $EUID = $UID;
    $EGID = $GID;
    $UID  = $real_uid;                  # drop privileges
    $GID  = $real_gid;
    ($EUID, $EGID) = @priv;             # make sure they are gone
    die "can't drop privileges"
        unless $UID == $EUID && $GID == $EGID;
    $ENV{PATH} = '/bin:/usr/bin';
    exec 'myprog', 'arg1', 'arg2'
        or die "can't exec myprog: $!";
}
```

Note that `exec` is called with a list, never a single string, so no shell is involved to re-expand the arguments. This is the safest way to invoke an external command. The same fork-and-drop strategy works for `glob` and backticks, which (unlike `system` and `exec`) offer no list-form calling convention of their own.

Taint mode and @INC

Under `-T`, PetaPerl ignores the `PERL5LIB`, `PERLLIB`, and `PERL_USE_UNSAFE_INC` environment variables. They are ignored because they are easy to set without a user noticing, which would let the environment inject library directories silently. You can still adjust the module search path with the visible `-I` switch or the `lib` pragma (`-Mlib=/path`), because those appear plainly on the command line.

The current directory `.` is **not** in the default `@INC`, taint mode or not. (`PERL_USE_UNSAFE_INC=1` would re-add it, but that variable is itself ignored under `-T`, so `.` stays out of `@INC` for any tainted program.)

Set-id scripts and the shebang race

PetaPerl detects the setuid/setgid condition (real user or group ID differing from the effective one) and enables taint mode accordingly; the “while running setuid” / “while running setgid” wording in the insecurity messages comes from that detection. PetaPerl does **not**, however, ship a setuid `pperl` install or any set-id helper. If you need a privileged Perl program, the supported approach on Linux is a small dedicated setuid wrapper that sanitises the environment and then execs the interpreter on a fixed, non-writable script path, rather than relying on a setuid script directly. The classic shebang race, where the kernel and the interpreter open the script file at two different

moments and an attacker swaps the file in between, is a property of kernel set-id-script handling, not of the interpreter, and the wrapper approach sidesteps it.

Algorithmic complexity attacks

Some internal algorithms can be driven to consume disproportionate time or memory by carefully chosen input, which is a denial-of-service vector independent of taint mode. The threat classes:

- **Hash collisions.** An attacker who can predict the hash seed can craft keys that all land in one bucket, degrading hash operations. pperl randomises the hash seed at process start and randomises the visible order of `keys`, `values`, and `each` per hash. Two environment variables override these defaults for debugging: `PERL_HASH_SEED` pins the seed (which controls storage, not the presented order) and `PERL_PERTURB_KEYS` controls traversal-order perturbation; the `Hash::Util` function `hash_traversal_mask` does the same per hash. These overrides weaken the protection and are not advised in production. PetaPerl has never guaranteed any hash key ordering; do not rely on it, and do not use the pseudo-random ordering as a shuffle.
- **Catastrophic regex backtracking.** The regular-expression engine is an NFA and can take time or space that grows steeply when a pattern can match an input in many ways. A hostile input matched against a vulnerable pattern can exhaust memory or CPU. Bound the input size and prefer patterns that cannot backtrack pathologically.
- **Sort.** `sort` uses mergesort, whose worst case is bounded, so it cannot be driven into quadratic behaviour by adversarial input.

Tied hashes may carry their own ordering and their own complexity characteristics.

Differences from upstream

- **Linux only.** PetaPerl runs only on Linux. The VMS, Windows, and other-platform set-id and security discussion that upstream Perl carries has no counterpart here and is simply absent.
- **No "protection" tooling.** PetaPerl has no bytecode compiler and no source-filter obfuscation machinery, so the upstream advice about shipping compiled or filtered source does not apply. Obscuring source was never a security measure in any case.

See also

- `perlvar` - `${^TAINT}`, `%ENV`, and the real/effective ID variables `$< $> $( $)` that decide whether taint mode auto-enables.
- `perllocale` - why locale data is untrusted and how `use locale` interacts with `\w` laundering.
- `perlop` - the `?:` operator and its exception to whole-expression tainting.
- *Command-line switches* - the `-T`, `-t`, and `-I` switches in context.
- *Command-line options* - how pperl parses its switches, including the runtime selector.

- `Scalar::Util` - `tainted`, for probing the taint state of a value.
- `Hash::Util` - `hash_traversal_mask`, for per-hash control of traversal-order randomisation.

5.2 Modules

Built-in modules, by top-level namespace. Badges show the feature tier and any extra Cargo features required - see *Disclaimer & bug reports* for how to check which features your pperl binary has.

- `AE` - simpler/faster/newer/cooler AnyEvent API
- `AnyDBM_File` - provide framework for multiple DBMs
- `AnyEvent` - the DBI of event loop programming
  :: [`DNS`, `Debug`, `Handle`, `IO`, `Impl`, `Log`, `Loop`, `Socket`, `Strict`, `TLS`, `Util`]
- `App` :: [`Cpan`, `Prove`]
- `Archive` :: [`Tar`]
- `Attribute` :: [`Handlers`]
- `AutoLoader` - load subroutines only on demand
- `AutoSplit` - split a package for autoloading
- `B` - Inspect a running Perl program's own optree, symbol table, and compiled subs.
  :: [`AV`, `BINOP`, `BM`, `COP`, `COW`, `CV`, `Concise`, `Deparse`, `FM`, `GV`, `HE`, `HV`, `IO`, `IV`, `LISTOP`, `LOGOP`, `LOOP`, `MAGIC`, `METHOP`, `NV`, `OP`, `Op_private`, `PADLIST`, `PADNAME`, `PADNAMELIST`, `PADOP`, `PMOP`, `PV`, `PVLV`, `PVMG`, `RHE`, `SV`, `SVOP`, `Showlex`, `Terse`, `UNOP`, `UNOP_AUX`, `Xref`]
- `Benchmark` - Benchmark running times of Perl code
- `CPAN` - query, download and build perl modules from CPAN sites
  :: [`Debug`, `Distroprefs`, `FirstTime`, `HandleConfig`, `Kwalify`, `Meta`, `Mirrors`, `Nox`, `Plugin`, `Queue`, `Tarzip`, `Version`]
- `Cairo` - Perl bindings for cairo, the 2D vector graphics library.
  :: [`Context`, `Font`, `FontFace`, `FontOptions`, `Format`, `FtFontFace`, `Gradient`, `ImageSurface`, `LinearGradient`, `Matrix`, `Pattern`, `PdfSurface`, `PsSurface`, `RadialGradient`, `RecordingSurface`, `Region`, `ScaledFont`, `SolidPattern`, `Surface`, `SurfacePattern`, `SvgSurface`, `ToyFontFace`]
- `Capture` :: [`Tiny`]
- `Carp` - alternative warn and die for modules
- `Class` :: [`Inspector`, `Load`, `MOP`, `Struct`, `XSAccessor`]
- `Clone` - Make a deep, independent copy of any Perl value - hashes, arrays, scalars, references, blessed objects, tied variables, and weak references.
- `Compress` :: [`Raw`, `Zlib`]
- `Config` - Look up the build-time configuration of the running Perl interpreter.
  :: [`Extensions`, `Perl`]

- Coro - the only real threads in perl
  :: [AIO, AnyEvent, BDB, Channel, Debug, EV, Event, Handle, LWP, RWLock, Select, Semaphore, SemaphoreSet, Signal, Socket, Specific, State, Storable, Timer, Util]
- Crypt :: [OpenSSL]
- Cwd - Report the current working directory and resolve paths to their absolute, symlink-free form.
- DB - programmatic interface to the Perl debugging API
- DBD :: [DBM, File, Gofer, Mem, Proxy, SQLite, Sponge]
- DBI - Database independent interface for Perl
  :: [Const, DBD, Gofer, Profile, ProfileData, ProfileDumper, ProfileSubs, ProxyServer, PurePerl, SQL, Util]
- DBM_Filter - Filter DBM keys/values
  :: [compress, encode, int32, null, utf8]
- DB_File - Tie a Perl hash or array to a database file: a hash table, a sorted B-tree, or an array of records kept in a plain text file.
- Data :: [Dumper, UUID]
- Devel :: [Cover, Peek, SelfStubber, Size]
- Digest - Modules that calculate message digests
  :: [MD5, SHA, SHA1, base, file]
- DirHandle - (obsolete) supply object methods for directory handles
- Dumpvalue - provides screen dump of Perl data.
- DynaLoader - Dynamically load C libraries into Perl code
- EV - A fast, full-featured event loop: call your code when a file handle is readable or writable, a timer expires, a signal arrives, a child process changes state or a file changes.
  :: [Async, Child, Embed, IO, Loop, Periodic, Signal, Stat, Timer, Watcher]
- Encode - Convert between Perl character strings and byte encodings.
  :: [Alias, Byte, CJKConstants, CN, Config, EBCDIC, Encoder, Encoding, GSM0338, Guess, JP, KR, MIME, Symbol, TW, Unicode, XS, utf8]
- English - use nice English (or awk) names for ugly punctuation variables
- Env - perl module that imports environment variables as scalars or arrays
- Errno - Symbolic names for the `errno(3)` values your system uses - EINTR, EAGAIN, ENOENT, EACCES, EEXIST, EIO, ENOTDIR, EISDIR, EBUSY, EPIPE, ECONNRESET, ECONNREFUSED, ETIMEDOUT, and every other errno name the host libc defines.
- Event - an event loop: watchers for timers, file descriptors,
  :: [Event, Watcher, generic, group, idle, io, signal, timer, var]
- Exporter - Implements default import method for modules
  :: [Heavy]

- ExtUtils :: [CBuilder, Command, Embed, HasCompiler, Install, Installed, Liblist, MM, MM_Any, MM_Unix, MY, MakeMaker, Manifest, Miniperl, PL2Bat, Packlist, testlib]
- FFI :: [Build, CheckLib, Platypus, Probe, Temp]
- Fatal - Replace functions with equivalents which succeed or die
- Fcntl - Symbolic constants for `fcntl(2)`, `flock(2)`, `lseek(2)`, and file-mode bits - everything you need to call `sysopen`, `fcntl`, `flock`, `seek`, and to interpret the mode word returned by `stat`.
- File :: [Basename, Compare, Copy, DosGlob, Fetch, Find, Glob, GlobMapper, Map, Path, ShareDir, Spec, Temp, Which, stat]
- FileCache - keep more files open than the system permits
- FileHandle - supply object methods for filehandles
- Filter :: [Simple, Util, cpp, decrypt, exec, m4, sh, tee]
- FindBin - Locate directory of original Perl script
- GD - Perl bindings for libgd, the GD graphics library.
  :: [Group, Image, Polygon, Polyline, Simple]
- GDBM_File - Tie a Perl hash to a database file with GNU dbm's interface: its open modes, error codes, options, dumps and recovery calls.
- Getopt :: [Long, Std]
- Guard - run a piece of code when a scope ends or an object goes away
- HTML :: [Entities, Filter, HeadParser, LinkExtor, Parser, PullParser, Tagset, TokeParser]
- HTTP :: [Parser, Tiny]
- Hash :: [Util]
- I18N :: [Collate, LangTags, Langinfo]
- IO - Load the core IO modules - `IO::Handle`, `IO::File`, `IO::Seekable`, `IO::Pipe`, `IO::Socket`, `IO::Dir` - in one `use`.
  :: [Compress, Dir, File, Handle, Pipe, Poll, Pty, Seekable, Select, Socket, Tty, Uncompress, Zlib]
- IPC :: [Cmd, Msg, Open2, Open3, Semaphore, SharedMem, SysV]
- Imager - Perl bindings for Imager, the raster graphics library.
  :: [Color, Expr, Fill, Font, Fountain, Matrix2d, Regops, Test, Transform, TrimColorList]
- JSON :: [PP, XS]
- LMDB_File - Tie a Perl hash to an LMDB database, or drive LMDB's environments, transactions and cursors directly.
- List :: [MoreUtils, Util]
- Locale :: [Maketext]
- MIME :: [Base64, Charset, QuotedPrint]

- Math :: [BigFloat, BigInt, BigRat, Complex, GMP, Random, Trig]
- Memoize - Make functions faster by trading space for time - cache a function's results by its arguments and return the cached result on a repeated call.
  :: [AnyDBM_File, Expire, NDBM_File, SDBM_File, Storable]
- Module :: [CoreList, Load, Loaded, Metadata]
- Moose - The compiled part of Moose and Class::MOP: the attribute readers of the metaclass objects, the method map cache, and a few helpers of Moose::Exporter, the type constraints and role application.
- NDBM_File - Tie a Perl hash to an ndbm database file, so keys and values persist between runs.
- NEXT - Provide a pseudo-class NEXT (et al) that allows method redispatch
- Net :: [Cmd, Config, Domain, FTP, NNTP, Netrc, POP3, Ping, SMTP, SSLeay, Time, hostent, netent, protoent, servent]
- O - Generic interface to Perl Compiler backends
- ODBM_File - Tie a Perl hash to an old-style dbm database file.
- Opcode - Name, group and mask the interpreter's opcodes - the machinery `Safe` compartments are built from.
- PDL - the Perl Data Language: fast, vectorised, multidimensional arrays for scientific and bulk numeric work
  :: [Bad, Basic, Complex, Compression, Constants, Core, Exporter, FFT, IO, Image2D, ImageND, ImageRGB, Lite, LiteF, Math, MatrixOps, Ops, Options, PP, Primitive, Slices, Trans, Transform, Types, Ufunc]
- POSIX - The POSIX (IEEE Std 1003.1) C standard library, exposed to Perl.
  :: [SigSet, Termios]
- Package :: [Stash]
- PadWalker - Reach into another subroutine's `my` and `our` variables - read them, change them, find out their names - from anywhere on the call stack.
- Params :: [Check, Util]
- Parse :: [CPAN]
- Perl :: [OSType]
- PerlIO - On demand loader for PerlIO layers and root of PerlIO::* name space
  :: [encoding, mmap, scalar, utf8_strict, via]
- Peta :: [Dict, FFI, ReadLine, Shell, WebGPU, XS]
- Pod :: [Checker, Escapes, Functions, Html, Man, ParseLink, Perldoc, Simple, Text, Usage]
- SDBM_File - Tie a Perl hash to an on-disk SDBM database so keys and values persist between runs.
- SDL2 - Perl bindings for SDL2 - windows, input, timers and hardware-accelerated 2D drawing.
  :: [Events, Image, Render, Surface, Video]

- Safe - Compile and execute code in restricted compartments
- Scalar :: [List, Util]
- Search :: [Dict]
- SelectSaver - save and restore selected file handle
- SelfLoader - load functions only on demand
- Socket - BSD socket constants and address pack/unpack helpers that feed Perl's built-in socket, bind, connect, accept, send, recv, setsockopt, and shutdown.
- Storable - Serialise arbitrary Perl data structures to bytes, reconstruct them later, or deep-clone them in memory.
- String :: [CRC32]
- Sub :: [Exporter, Identify, Name, Util]
- Symbol - manipulate Perl symbols and their names
- Sys :: [Hostname, Syslog]
- TAP :: [Base, Formatter, Harness, Object, Parser]
- Template :: [Stash]
- Term :: [ANSIColor, Cap, Complete, ReadKey, ReadLine, Table]
- Test - provides a simple framework for writing test scripts
  :: [Builder, Harness, LeakTrace, More, Simple, Tester, use]
- Test2 - Framework for writing test tools that all work together.
  :: [API, AsyncSubtest, Bundle, Compare, Env, Event, EventFacet, Formatter, Handle, Hub, IPC, Manual, Mock, Plugin, Require, Suite, Todo, Tools, Util, V0, V1, Workflow]
- Text :: [Abbrev, Balanced, CSV, CSV_PP, LevenshteinXS, LineFold, ParseWords, Tabs, Wrap]
- Thread - Manipulate threads in Perl (for old code only)
  :: [Queue, Semaphore]
- Tie :: [Array, File, Handle, Hash, Memoize, RefHash, Scalar, StdHandle, SubstrHash]
- Time :: [HiRes, Local, Piece, Seconds, gmtime, localtime, tm]
- Types :: [Serialiser]
- UNIVERSAL - base class for ALL classes (blessed references)
- URL :: [Encode]
- Unicode :: [Collate, GCString, LineBreak, Normalize, UCD]
- User :: [grent, pwent]
- XML :: [Parser]
- XSLoader - Start the native part of a module
- XString - The string quoting helpers of B, without loading B.

- YAML :: [LibYAML, PP, XS]
- _charnames - the Unicode character name table behind \N{NAME},
- attributes - The machinery behind sub name :attr and my $x :attr: applying the built-in attributes, handing the others to the package that defines them, and reading attributes back.
- autodie - Replace functions with ones that succeed or die with lexical scope :: [Scope, Util, exception, hints, skip]
- autouse - postpone load of modules until a function is used
- base - Establish an ISA relationship with base classes at compile time
- bigfloat - transparent big floating point number support for Perl
- bigint - transparent big integer support for Perl
- bignum - transparent big number support for Perl
- bigrat - transparent big rational number support for Perl
- blib - Use MakeMaker's uninstalled version of a package
- builtin - Perl pragma to import built-in utility functions
- bytes - Perl pragma to expose the individual bytes of characters
- charnames - access to Unicode character names and named character sequences; also define character names
- constant - Perl pragma to declare constants
- deprecate - Perl pragma for deprecating the inclusion of a module in core
- diagnostics - produce verbose warning diagnostics
- encoding - allows you to write your script in non-ASCII and non-UTF-8 :: [warnings]
- experimental - Experimental features made easy
- feature - Perl pragma to enable new features
- fields - compile-time class fields
- filetest - Perl pragma to control the filetest permission operators
- if - use a Perl module if a condition holds
- integer - Perl pragma to use integer arithmetic instead of floating point
- less - perl pragma to request less of something
- lib - manipulate @INC at compile time
- locale - Perl pragma to use or avoid POSIX locales for built-in operations
- mro - Method Resolution Order - how Perl decides which parent class's method a call inherits.
- ok - Alternative to Test::More::use_ok
- open - perl pragma to set default PerlIO layers for input and output

- ops - Perl pragma to restrict unsafe operations when compiling
- overload - Package for overloading Perl operations
- overloading - perl pragma to lexically control overloading
- parent - Establish an ISA relationship with base classes at compile time
- perlfaq - Frequently asked questions about Perl
- re - The re pragma and regular-expression introspection module.
- sigtrap - Perl pragma to enable simple signal handling
- sort - perl pragma to control sort() behaviour
- source :: [encoding]
- stable - Experimental features made easy, once we know they're stable
- strict - Perl pragma to restrict unsafe constructs
- subs - Perl pragma to predeclare subroutine names
- threads - Perl interpreter-based threads
  :: [shared]
- utf8 - Perl pragma to enable/disable UTF-8 (or UTF-EBCDIC) in source code
- vars - Perl pragma to predeclare global variable names
- version - Turn a version string into an object you can compare, print, and introspect.
- warnings - Perl pragma to control optional warnings
  :: [register]

5.2.1 AE

simpler/faster/newer/cooler AnyEvent API

SYNOPSIS

```
use AnyEvent; # not AE

# file handle or descriptor readable
my $w = AE::io $fh, 0, sub { ...  };

# one-shot or repeating timers
my $w = AE::timer $seconds,         0, sub { ... }; # once
my $w = AE::timer $seconds, $interval, sub { ... }; # repeated

print AE::now;  # prints current event loop time
print AE::time; # think Time::HiRes::time or simply CORE::time.

# POSIX signal
my $w = AE::signal TERM => sub { ... };
```

(continues on next page)

(continued from previous page)

```
# child process exit
my $w = AE::child $pid, sub {
   my ($pid, $status) = @_;
   ...
};

# called when event loop idle (if applicable)
my $w = AE::idle sub { ... };

my $cv = AE::cv; # stores whether a condition was flagged
$cv->send; # wake up current and all future recv's
$cv->recv; # enters "main loop" till $condvar gets ->send
# use a condvar in callback mode:
$cv->cb (sub { $_[0]->recv });
```

DESCRIPTION

This module documents the new simpler AnyEvent API.

The rationale for the new API is that experience with *EV* shows that this API actually "works", despite its lack of extensibility, leading to a shorter, easier and faster API.

The main differences from AnyEvent is that function calls are used instead of method calls, and that no named arguments are used.

This makes calls to watcher creation functions really short, which can make a program more readable despite the lack of named parameters. Function calls also allow more static type checking than method calls, so many mistakes are caught at compile-time with this API.

Also, some backends (Perl and EV) are so fast that the method call overhead is very noticeable (with EV it increases the execution time five- to six-fold, with Perl the method call overhead is about a factor of two).

Note that the AE API is an alternative to, not the future version of, the AnyEvent API. Both APIs can be used interchangeably and there are no plans to "switch", so if in doubt, feel free to use the *AnyEvent* API in new code.

As the AE API is complementary, not everything in the AnyEvent API is available, and you still need to use AnyEvent for the finer stuff. Also, you should not `use AE` directly, `use AnyEvent` will provide the AE namespace.

At the moment, these functions will become slower then their method-call counterparts when using *AnyEvent::Strict* or *AnyEvent::Debug*::wrap.

FUNCTIONS

This section briefly describes the alternative watcher constructors and other functions available inside the AE namespace. Semantics are not described here; please refer to the description of the function or method with the same name in the *AnyEvent* manpage for the details.

- $w = AE::io $fh_or_fd, $watch_write, $cb

Creates an I/O watcher that listens for read events (`$watch_write` false) or write events (`$watch_write` is true) on the file handle or file descriptor `$fh_or_fd`.

The callback `$cb` is invoked as soon and as long as I/O of the type specified by `$watch_write`) can be done on the file handle/descriptor.

Example: wait until STDIN becomes readable.

```
$stdin_ready = AE::io *STDIN, 0, sub { scalar <STDIN> };
```

Example: wait until STDOUT becomes writable and print something.

```
$stdout_ready = AE::io *STDOUT, 1, sub { print STDOUT "woaw\n" }
 ↪;
```

- $w = AE::timer $after, $interval, $cb

  Creates a timer watcher that invokes the callback `$cb` after at least `$after` second have passed (`$after` can be negative or `0`).

  If `$interval` is `0`, then the callback will only be invoked once, otherwise it must be a positive number of seconds that specifies the interval between successive invocations of the callback.

  Example: print “too late” after at least one second has passed.

  ```
  $timer_once = AE::timer 1, 0, sub { print "too late\n" };
  ```

  Example: print “blubb” once a second, starting as soon as possible.

  ```
  $timer_repeated = AE::timer 0, 1, sub { print "blubb\n" };
  ```

- $w = AE::signal $signame, $cb

  Invoke the callback `$cb` each time one or more occurrences of the named signal `$signame` are detected.

- $w = AE::child $pid, $cb

  Invokes the callback `$cb` when the child with the given `$pid` exits (or all children, when `$pid` is zero).

  The callback will get the actual pid and exit status as arguments.

- $w = AE::idle $cb

  Invoke the callback `$cb` each time the event loop is “idle” (has no events outstanding), but do not prevent the event loop from polling for more events.

- $cv = AE::cv

- $cv = AE::cv { BLOCK }

  Create a new condition variable. The first form is identical to `AnyEvent->condvar`, the second form additionally sets the callback (as if the `cb` method is called on the condition variable).

- AE::now

  Returns the current event loop time (may be cached by the event loop).

- AE::now_update

  Ensures that the current event loop time is up to date.

- AE::time

  Return the current time (not cached, always consults a hardware clock).

- AE::postpone { BLOCK }

  Exactly the same as `AnyEvent::postpone`.

- AE::log $level, $msg[, @args]

  Exactly the same as `AnyEvent::log` (or `AnyEvent::Log::log`).

AUTHOR

```
Marc Lehmann <schmorp@schmorp.de>
http://anyevent.schmorp.de
```

5.2.2 AnyDBM_File

provide framework for multiple DBMs

NDBM_File, DB_File, GDBM_File, SDBM_File, ODBM_File - various DBM implementations

SYNOPSIS

```
use AnyDBM_File;
```

DESCRIPTION

This module is a "pure virtual base class"–it has nothing of its own. It's just there to inherit from one of the various DBM packages. It prefers ndbm for compatibility reasons with Perl 4, then Berkeley DB (See *DB_File*), GDBM, SDBM (which is always there–it comes with Perl), and finally ODBM. This way old programs that used to use NDBM via dbmopen() can still do so, but new ones can reorder @ISA:

```
BEGIN { @AnyDBM_File::ISA = qw(DB_File GDBM_File NDBM_File) }
use AnyDBM_File;
```

Having multiple DBM implementations makes it trivial to copy database formats:

```
use Fcntl; use NDBM_File; use DB_File;
tie %newhash,  'DB_File', $new_filename, O_CREAT|O_RDWR;
tie %oldhash,  'NDBM_File', $old_filename, 1, 0;
%newhash = %oldhash;
```

DBM Comparisons

Here's a partial table of features the different packages offer:

```
                        odbm     ndbm     sdbm     gdbm     bsd-db
                        ----     ----     ----     ----     ------
Linkage comes w/ perl   yes      yes      yes      yes      yes
Src comes w/ perl       no       no       yes      no       no
Comes w/ many unix os   yes      yes[0]   no       no       no
Builds ok on !unix      ?        ?        yes      yes      ?
Code Size               ?        ?        small    big      big
Database Size           ?        ?        small    big?     ok[1]
Speed                   ?        ?        slow     ok       fast
FTPable                 no       no       yes      yes      yes
Easy to build          N/A      N/A       yes      yes      ok[2]
Size limits             1k       4k       1k[3]    none     none
Byte-order independent  no       no       no       no       yes
Licensing restrictions  ?        ?        no       yes      no
```

- [0]

  on mixed universe machines, may be in the bsd compat library, which is often shunned.

- [1]

  Can be trimmed if you compile for one access method.

- [2]

  See *DB_File*. Requires symbolic links.

- [3]

  By default, but can be redefined.

SEE ALSO

dbm(3), ndbm(3), DB_File(3), perldbmfilter

5.2.3 AnyEvent

the DBI of event loop programming

EV, Event, Glib, Tk, UV, Perl, Event::Lib, Irssi, rxvt-unicode, IO::Async, Qt, FLTK and POE are various supported event loops/environments.

SYNOPSIS

```
use AnyEvent;

# if you prefer function calls, look at the AE manpage for
# an alternative API.

```

(continues on next page)

(continued from previous page)

```
# file handle or descriptor readable
my $w = AnyEvent->io (fh => $fh, poll => "r", cb => sub { ...  });

# one-shot or repeating timers
my $w = AnyEvent->timer (after => $seconds, cb => sub { ...  });
my $w = AnyEvent->timer (after => $seconds, interval => $seconds, cb
 ↪=> ...);

print AnyEvent->now;  # prints current event loop time
print AnyEvent->time; # think Time::HiRes::time or simply
 ↪CORE::time.

# POSIX signal
my $w = AnyEvent->signal (signal => "TERM", cb => sub { ... });

# child process exit
my $w = AnyEvent->child (pid => $pid, cb => sub {
   my ($pid, $status) = @_;
   ...
});

# called when event loop idle (if applicable)
my $w = AnyEvent->idle (cb => sub { ... });

my $w = AnyEvent->condvar; # stores whether a condition was flagged
$w->send; # wake up current and all future recv's
$w->recv; # enters "main loop" till $condvar gets ->send
# use a condvar in callback mode:
$w->cb (sub { $_[0]->recv });
```

INTRODUCTION/TUTORIAL

This manpage is mainly a reference manual. If you are interested in a tutorial or some gentle introduction, have a look at the AnyEvent::Intro manpage.

SUPPORT

An FAQ document is available as AnyEvent::FAQ.

There also is a mailinglist for discussing all things AnyEvent, and an IRC channel, too.

See the AnyEvent project page at the **Schmorpforge Ta-Sa Software Repository**, at http://anyevent.schmorp.de, for more info.

WHY YOU SHOULD USE THIS MODULE (OR NOT)

Glib, POE, IO::Async, Event... CPAN offers event models by the dozen nowadays. So what is different about AnyEvent?

Executive Summary: AnyEvent is *compatible*, AnyEvent is *free of policy* and AnyEvent is *small and efficient*.

First and foremost, *AnyEvent is not an event model* itself, it only interfaces to whatever event model the main program happens to use, in a pragmatic way. For event models and certain classes of immortals alike, the statement "there can only be one" is a bitter reality: In general, only one event loop can be active at the same time in a process. AnyEvent cannot change this, but it can hide the differences between those event loops.

The goal of AnyEvent is to offer module authors the ability to do event programming (waiting for I/O or timer events) without subscribing to a religion, a way of living, and most importantly: without forcing your module users into the same thing by forcing them to use the same event model you use.

For modules like POE or IO::Async (which is a total misnomer as it is actually doing all I/O *synchronously*...), using them in your module is like joining a cult: After you join, you are dependent on them and you cannot use anything else, as they are simply incompatible to everything that isn't them. What's worse, all the potential users of your module are *also* forced to use the same event loop you use.

AnyEvent is different: AnyEvent + POE works fine. AnyEvent + Glib works fine. AnyEvent + Tk works fine etc. etc. but none of these work together with the rest: POE + EV? No go. Tk + Event? No go. Again: if your module uses one of those, every user of your module has to use it, too. But if your module uses AnyEvent, it works transparently with all event models it supports (including stuff like IO::Async, as long as those use one of the supported event loops. It is easy to add new event loops to AnyEvent, too, so it is future-proof).

In addition to being free of having to use *the one and only true event model*, AnyEvent also is free of bloat and policy: with POE or similar modules, you get an enormous amount of code and strict rules you have to follow. AnyEvent, on the other hand, is lean and to the point, by only offering the functionality that is necessary, in as thin as a wrapper as technically possible.

Of course, AnyEvent comes with a big (and fully optional!) toolbox of useful functionality, such as an asynchronous DNS resolver, 100% non-blocking connects (even with TLS/SSL, IPv6 and on broken platforms such as Windows) and lots of real-world knowledge and workarounds for platform bugs and differences.

Now, if you *do want* lots of policy (this can arguably be somewhat useful) and you want to force your users to use the one and only event model, you should *not* use this module.

DESCRIPTION

AnyEvent provides a uniform interface to various event loops. This allows module authors to use event loop functionality without forcing module users to use a specific event loop implementation (since more than one event loop cannot coexist peacefully).

The interface itself is vaguely similar, but not identical to the *Event* module.

During the first call of any watcher-creation method, the module tries to detect the currently loaded event loop by probing whether one of the following modules is already loaded: *EV*, *AnyEvent::Loop*, *Event*, Glib, Tk, Event::Lib, Qt, POE. The first one found is used. If none are detected, the module tries to load the first four modules in the order given; but note that if *EV* is not available, the pure-perl *AnyEvent::Loop* should always work, so the other two are not normally tried.

Because AnyEvent first checks for modules that are already loaded, loading an event model explicitly before first using AnyEvent will likely make that model the default. For

example:

```
use Tk;
use AnyEvent;

# .. AnyEvent will likely default to Tk
```

The *likely* means that, if any module loads another event model and starts using it, all bets are off - this case should be very rare though, as very few modules hardcode event loops without announcing this very loudly.

The pure-perl implementation of AnyEvent is called `AnyEvent::Loop`. Like other event modules you can load it explicitly and enjoy the high availability of that event loop :)

WATCHERS

AnyEvent has the central concept of a *watcher*, which is an object that stores relevant data for each kind of event you are waiting for, such as the callback to call, the file handle to watch, etc.

These watchers are normal Perl objects with normal Perl lifetime. After creating a watcher it will immediately "watch" for events and invoke the callback when the event occurs (of course, only when the event model is in control).

Note that **callbacks must not permanently change global variables** potentially in use by the event loop (such as `$_` or `$[`) and that **callbacks must not `die`**. The former is good programming practice in Perl and the latter stems from the fact that exception handling differs widely between event loops.

To disable a watcher you have to destroy it (e.g. by setting the variable you store it in to `undef` or otherwise deleting all references to it).

All watchers are created by calling a method on the `AnyEvent` class.

Many watchers either are used with "recursion" (repeating timers for example), or need to refer to their watcher object in other ways.

One way to achieve that is this pattern:

```
my $w; $w = AnyEvent->type (arg => value ..., cb => sub {
   # you can use $w here, for example to undef it
   undef $w;
});
```

Note that `my $w; $w =` combination. This is necessary because in Perl, my variables are only visible after the statement in which they are declared.

I/O WATCHERS

```
$w = AnyEvent->io (
   fh   => <filehandle_or_fileno>,
   poll => <"r" or "w">,
   cb   => <callback>,
);
```

You can create an I/O watcher by calling the `AnyEvent->io` method with the following mandatory key-value pairs as arguments:

`fh` is the Perl *file handle* (or a naked file descriptor) to watch for events (AnyEvent might or might not keep a reference to this file handle). Note that only file handles pointing to things for which non-blocking operation makes sense are allowed. This includes sockets, most character devices, pipes, fifos and so on, but not for example files or block devices.

`poll` must be a string that is either `r` or `w`, which creates a watcher waiting for "r"eadable or "w"ritable events, respectively.

`cb` is the callback to invoke each time the file handle becomes ready.

Although the callback might get passed parameters, their value and presence is undefined and you cannot rely on them. Portable AnyEvent callbacks cannot use arguments passed to I/O watcher callbacks.

The I/O watcher might use the underlying file descriptor or a copy of it. You must not close a file handle as long as any watcher is active on the underlying file descriptor.

Some event loops issue spurious readiness notifications, so you should always use non-blocking calls when reading/writing from/to your file handles.

Example: wait for readability of STDIN, then read a line and disable the watcher.

```
my $w; $w = AnyEvent->io (fh => \*STDIN, poll => 'r', cb => sub {
   chomp (my $input = <STDIN>);
   warn "read: $input\n";
   undef $w;
});
```

TIME WATCHERS

```
$w = AnyEvent->timer (after => <seconds>, cb => <callback>);

$w = AnyEvent->timer (
   after    => <fractional_seconds>,
   interval => <fractional_seconds>,
   cb       => <callback>,
);
```

You can create a time watcher by calling the `AnyEvent->timer` method with the following mandatory arguments:

`after` specifies after how many seconds (fractional values are supported) the callback should be invoked. `cb` is the callback to invoke in that case.

Although the callback might get passed parameters, their value and presence is undefined and you cannot rely on them. Portable AnyEvent callbacks cannot use arguments passed to time watcher callbacks.

The callback will normally be invoked only once. If you specify another parameter, `interval`, as a strictly positive number (> 0), then the callback will be invoked regularly at that interval (in fractional seconds) after the first invocation. If `interval` is specified with a false value, then it is treated as if it were not specified at all.

The callback will be rescheduled before invoking the callback, but no attempt is made to avoid timer drift in most backends, so the interval is only approximate.

Example: fire an event after 7.7 seconds.

```
my $w = AnyEvent->timer (after => 7.7, cb => sub {
   warn "timeout\n";
});

# to cancel the timer:
undef $w;
```

Example 2: fire an event after 0.5 seconds, then roughly every second.

```
my $w = AnyEvent->timer (after => 0.5, interval => 1, cb => sub {
   warn "timeout\n";
});
```

TIMING ISSUES

There are two ways to handle timers: based on real time (relative, “fire in 10 seconds”) and based on wallclock time (absolute, “fire at 12 o’clock”).

While most event loops expect timers to specified in a relative way, they use absolute time internally. This makes a difference when your clock “jumps”, for example, when ntp decides to set your clock backwards from the wrong date of 2014-01-01 to 2008-01-01, a watcher that is supposed to fire “after a second” might actually take six years to finally fire.

AnyEvent cannot compensate for this. The only event loop that is conscious of these issues is *EV*, which offers both relative (ev_timer, based on true relative time) and absolute (ev_periodic, based on wallclock time) timers.

AnyEvent always prefers relative timers, if available, matching the AnyEvent API.

AnyEvent has two additional methods that return the “current time”:

- AnyEvent->time

  This returns the “current wallclock time” as a fractional number of seconds since the Epoch (the same thing as `time` or `Time::HiRes::time` return, and the result is guaranteed to be compatible with those).

  It progresses independently of any event loop processing, i.e. each call will check the system clock, which usually gets updated frequently.

- AnyEvent->now

  This also returns the “current wallclock time”, but unlike `time`, above, this value might change only once per event loop iteration, depending on the event loop (most return the same time as `time`, above). This is the time that AnyEvent’s timers get scheduled against.

  In almost all cases (in all cases if you don’t care), this is the function to call when you want to know the current time.

This function is also often faster then `AnyEvent->time`, and thus the preferred method if you want some timestamp (for example, *AnyEvent::Handle* uses this to update its activity timeouts).

The rest of this section is only of relevance if you try to be very exact with your timing; you can skip it without a bad conscience.

For a practical example of when these times differ, consider Event::Lib and *EV* and the following set-up:

The event loop is running and has just invoked one of your callbacks at time=500 (assume no other callbacks delay processing). In your callback, you wait a second by executing `sleep 1` (blocking the process for a second) and then (at time=501) you create a relative timer that fires after three seconds.

With Event::Lib, `AnyEvent->time` and `AnyEvent->now` will both return `501`, because that is the current time, and the timer will be scheduled to fire at time=504 (`501` + 3).

With *EV*, `AnyEvent->time` returns `501` (as that is the current time), but `AnyEvent->now` returns `500`, as that is the time the last event processing phase started. With *EV*, your timer gets scheduled to run at time=503 (`500` + 3).

In one sense, Event::Lib is more exact, as it uses the current time regardless of any delays introduced by event processing. However, most callbacks do not expect large delays in processing, so this causes a higher drift (and a lot more system calls to get the current time).

In another sense, *EV* is more exact, as your timer will be scheduled at the same time, regardless of how long event processing actually took.

In either case, if you care (and in most cases, you don't), then you can get whatever behaviour you want with any event loop, by taking the difference between `AnyEvent->time` and `AnyEvent->now` into account.

- AnyEvent->now_update

  Some event loops (such as *EV* or *AnyEvent::Loop*) cache the current time for each loop iteration (see the discussion of AnyEvent->now, above).

  When a callback runs for a long time (or when the process sleeps), then this "current" time will differ substantially from the real time, which might affect timers and time-outs.

  When this is the case, you can call this method, which will update the event loop's idea of "current time".

  A typical example would be a script in a web server (e.g. `mod_perl`) - when mod_perl executes the script, then the event loop will have the wrong idea about the "current time" (being potentially far in the past, when the script ran the last time). In that case you should arrange a call to `AnyEvent->now_update` each time the web server process wakes up again (e.g. at the start of your script, or in a handler).

  Note that updating the time *might* cause some events to be handled.

SIGNAL WATCHERS

```
$w = AnyEvent->signal (signal => <uppercase_signal_name>, cb =>
↪<callback>);
```

You can watch for signals using a signal watcher, `signal` is the signal *name* in uppercase and without any `SIG` prefix, `cb` is the Perl callback to be invoked whenever a signal occurs.

Although the callback might get passed parameters, their value and presence is undefined and you cannot rely on them. Portable AnyEvent callbacks cannot use arguments passed to signal watcher callbacks.

Multiple signal occurrences can be clumped together into one callback invocation, and callback invocation will be synchronous. Synchronous means that it might take a while until the signal gets handled by the process, but it is guaranteed not to interrupt any other callbacks.

The main advantage of using these watchers is that you can share a signal between multiple watchers, and AnyEvent will ensure that signals will not interrupt your program at bad times.

This watcher might use `%SIG` (depending on the event loop used), so programs overwriting those signals directly will likely not work correctly.

Example: exit on SIGINT

```
my $w = AnyEvent->signal (signal => "INT", cb => sub { exit 1 });
```

Restart Behaviour

While restart behaviour is up to the event loop implementation, most will not restart syscalls (that includes Async::Interrupt and AnyEvent's pure perl implementation).

Safe/Unsafe Signals

Perl signals can be either "safe" (synchronous to opcode handling) or "unsafe" (asynchronous) - the former might delay signal delivery indefinitely, the latter might corrupt your memory.

AnyEvent signal handlers are, in addition, synchronous to the event loop, i.e. they will not interrupt your running perl program but will only be called as part of the normal event handling (just like timer, I/O etc. callbacks, too).

Signal Races, Delays and Workarounds

Many event loops (e.g. Glib, Tk, Qt, IO::Async) do not support attaching callbacks to signals in a generic way, which is a pity, as you cannot do race-free signal handling in perl, requiring C libraries for this. AnyEvent will try to do its best, which means in some cases, signals will be delayed. The maximum time a signal might be delayed is 10 seconds by default, but can be overriden via `$ENV{PERL_ANYEVENT_MAX_SIGNAL_LATENCY}` or `$AnyEvent::MAX_SIGNAL_LATENCY` - see the "ENVIRONMENT VARIABLES" section for details.

All these problems can be avoided by installing the optional Async::Interrupt module, which works with most event loops. It will not work with inherently broken event loops such as *Event* or Event::Lib (and not with POE currently). For those, you just have to suffer the delays.

CHILD PROCESS WATCHERS

```
$w = AnyEvent->child (pid => <process id>, cb => <callback>);
```

You can also watch for a child process exit and catch its exit status.

The child process is specified by the `pid` argument (on some backends, using `0` watches for any child process exit, on others this will croak). The watcher will be triggered only when the child process has finished and an exit status is available, not on any trace events (stopped/continued).

The callback will be called with the pid and exit status (as returned by waitpid), so unlike other watcher types, you *can* rely on child watcher callback arguments.

This watcher type works by installing a signal handler for `SIGCHLD`, and since it cannot be shared, nothing else should use SIGCHLD or reap random child processes (waiting for specific child processes, e.g. inside `system`, is just fine).

There is a slight catch to child watchers, however: you usually start them *after* the child process was created, and this means the process could have exited already (and no SIGCHLD will be sent anymore).

Not all event models handle this correctly (neither POE nor IO::Async do, see their AnyEvent::Impl manpages for details), but even for event models that *do* handle this correctly, they usually need to be loaded before the process exits (i.e. before you fork in the first place). AnyEvent's pure perl event loop handles all cases correctly regardless of when you start the watcher.

This means you cannot create a child watcher as the very first thing in an AnyEvent program, you *have* to create at least one watcher before you `fork` the child (alternatively, you can call `AnyEvent::detect`).

As most event loops do not support waiting for child events, they will be emulated by AnyEvent in most cases, in which case the latency and race problems mentioned in the description of signal watchers apply.

Example: fork a process and wait for it

```
my $done = AnyEvent->condvar;

# this forks and immediately calls exit in the child. this
# normally has all sorts of bad consequences for your parent,
# so take this as an example only. always fork and exec,
# or call POSIX::_exit, in real code.
my $pid = fork or exit 5;

my $w = AnyEvent->child (
   pid => $pid,
   cb  => sub {
```

(continues on next page)

(continued from previous page)

```
      my ($pid, $status) = @_;
      warn "pid $pid exited with status $status";
      $done->send;
   },
);

# do something else, then wait for process exit
$done->recv;
```

IDLE WATCHERS

```
$w = AnyEvent->idle (cb => <callback>);
```

This will repeatedly invoke the callback after the process becomes idle, until either the watcher is destroyed or new events have been detected.

Idle watchers are useful when there is a need to do something, but it is not so important (or wise) to do it instantly. The callback will be invoked only when there is "nothing better to do", which is usually defined as "all outstanding events have been handled and no new events have been detected". That means that idle watchers ideally get invoked when the event loop has just polled for new events but none have been detected. Instead of blocking to wait for more events, the idle watchers will be invoked.

Unfortunately, most event loops do not really support idle watchers (only EV, Event and Glib do it in a usable fashion) - for the rest, AnyEvent will simply call the callback "from time to time".

Example: read lines from STDIN, but only process them when the program is otherwise idle:

```
my @lines; # read data
my $idle_w;
my $io_w = AnyEvent->io (fh => \*STDIN, poll => 'r', cb => sub {
   push @lines, scalar <STDIN>;

   # start an idle watcher, if not already done
   $idle_w ||= AnyEvent->idle (cb => sub {
      # handle only one line, when there are lines left
      if (my $line = shift @lines) {
         print "handled when idle: $line";
      } else {
         # otherwise disable the idle watcher again
         undef $idle_w;
      }
   });
});
```

CONDITION VARIABLES

```
$cv = AnyEvent->condvar;

$cv->send (<list>);
my @res = $cv->recv;
```

If you are familiar with some event loops you will know that all of them require you to run some blocking "loop", "run" or similar function that will actively watch for new events and call your callbacks.

AnyEvent is slightly different: it expects somebody else to run the event loop and will only block when necessary (usually when told by the user).

The tool to do that is called a "condition variable", so called because they represent a condition that must become true.

Now is probably a good time to look at the examples further below.

Condition variables can be created by calling the `AnyEvent->condvar` method, usually without arguments. The only argument pair allowed is `cb`, which specifies a callback to be called when the condition variable becomes true, with the condition variable as the first argument (but not the results).

After creation, the condition variable is "false" until it becomes "true" by calling the `send` method (or calling the condition variable as if it were a callback, read about the caveats in the description for the `->send` method).

Since condition variables are the most complex part of the AnyEvent API, here are some different mental models of what they are - pick the ones you can connect to:

- Condition variables are like callbacks - you can call them (and pass them instead of callbacks). Unlike callbacks however, you can also wait for them to be called.
- Condition variables are signals - one side can emit or send them, the other side can wait for them, or install a handler that is called when the signal fires.
- Condition variables are like "Merge Points" - points in your program where you merge multiple independent results/control flows into one.
- Condition variables represent a transaction - functions that start some kind of transaction can return them, leaving the caller the choice between waiting in a blocking fashion, or setting a callback.
- Condition variables represent future values, or promises to deliver some result, long before the result is available.

Condition variables are very useful to signal that something has finished, for example, if you write a module that does asynchronous http requests, then a condition variable would be the ideal candidate to signal the availability of results. The user can either act when the callback is called or can synchronously `->recv` for the results.

You can also use them to simulate traditional event loops - for example, you can block your main program until an event occurs - for example, you could `->recv` in your main program until the user clicks the Quit button of your app, which would `->send` the "quit" event.

Note that condition variables recurse into the event loop - if you have two pieces of code that call `->recv` in a round-robin fashion, you lose. Therefore, condition variables are good to export to your caller, but you should avoid making a blocking wait yourself, at least in callbacks, as this asks for trouble.

Condition variables are represented by hash refs in perl, and the keys used by AnyEvent itself are all named `_ae_XXX` to make subclassing easy (it is often useful to build your own transaction class on top of AnyEvent). To subclass, use `AnyEvent::CondVar` as base class and call its `new` method in your own `new` method.

There are two "sides" to a condition variable - the "producer side" which eventually calls `-> send`, and the "consumer side", which waits for the send to occur.

Example: wait for a timer.

```
# condition: "wait till the timer is fired"
my $timer_fired = AnyEvent->condvar;

# create the timer - we could wait for, say
# a handle becomign ready, or even an
# AnyEvent::HTTP request to finish, but
# in this case, we simply use a timer:
my $w = AnyEvent->timer (
   after => 1,
   cb    => sub { $timer_fired->send },
);

# this "blocks" (while handling events) till the callback
# calls ->send
$timer_fired->recv;
```

Example: wait for a timer, but take advantage of the fact that condition variables are also callable directly.

```
my $done = AnyEvent->condvar;
my $delay = AnyEvent->timer (after => 5, cb => $done);
$done->recv;
```

Example: Imagine an API that returns a condvar and doesn't support callbacks. This is how you make a synchronous call, for example from the main program:

```
use AnyEvent::CouchDB;

...

my @info = $couchdb->info->recv;
```

And this is how you would just set a callback to be called whenever the results are available:

```
$couchdb->info->cb (sub {
   my @info = $_[0]->recv;
});
```

METHODS FOR PRODUCERS

These methods should only be used by the producing side, i.e. the code/module that eventually sends the signal. Note that it is also the producer side which creates the condvar in most cases, but it isn't uncommon for the consumer to create it as well.

- $cv->send (...)

  Flag the condition as ready - a running `->recv` and all further calls to `recv` will (eventually) return after this method has been called. If nobody is waiting the send will be remembered.

  If a callback has been set on the condition variable, it is called immediately from within send.

  Any arguments passed to the `send` call will be returned by all future `->recv` calls.

  Condition variables are overloaded so one can call them directly (as if they were a code reference). Calling them directly is the same as calling `send`.

- $cv->croak ($error)

  Similar to send, but causes all calls to `->recv` to invoke `Carp::croak` with the given error message/object/scalar.

  This can be used to signal any errors to the condition variable user/consumer. Doing it this way instead of calling `croak` directly delays the error detection, but has the overwhelming advantage that it diagnoses the error at the place where the result is expected, and not deep in some event callback with no connection to the actual code causing the problem.

- $cv->begin ([group callback])

- $cv->end

  These two methods can be used to combine many transactions/events into one. For example, a function that pings many hosts in parallel might want to use a condition variable for the whole process.

  Every call to `->begin` will increment a counter, and every call to `->end` will decrement it. If the counter reaches `0` in `->end`, the (last) callback passed to `begin` will be executed, passing the condvar as first argument. That callback is *supposed* to call `->send`, but that is not required. If no group callback was set, `send` will be called without any arguments.

  You can think of `$cv->send` giving you an OR condition (one call sends), while `$cv->begin` and `$cv->end` giving you an AND condition (all `begin` calls must be end'ed before the condvar sends).

  Let's start with a simple example: you have two I/O watchers (for example, STDOUT and STDERR for a program), and you want to wait for both streams to close before activating a condvar:

```
my $cv = AnyEvent->condvar;

$cv->begin; # first watcher
my $w1 = AnyEvent->io (fh => $fh1, cb => sub {
   defined sysread $fh1, my $buf, 4096
```

(continues on next page)

(continued from previous page)

```
      or $cv->end;
});

$cv->begin; # second watcher
my $w2 = AnyEvent->io (fh => $fh2, cb => sub {
   defined sysread $fh2, my $buf, 4096
      or $cv->end;
});

$cv->recv;
```

This works because for every event source (EOF on file handle), there is one call to `begin`, so the condvar waits for all calls to `end` before sending.

The ping example mentioned above is slightly more complicated, as the there are results to be passed back, and the number of tasks that are begun can potentially be zero:

```
my $cv = AnyEvent->condvar;

my %result;
$cv->begin (sub { shift->send (\%result) });

for my $host (@list_of_hosts) {
   $cv->begin;
   ping_host_then_call_callback $host, sub {
      $result{$host} = ...;
      $cv->end;
   };
}

$cv->end;

...

my $results = $cv->recv;
```

This code fragment supposedly pings a number of hosts and calls `send` after results for all then have have been gathered - in any order. To achieve this, the code issues a call to `begin` when it starts each ping request and calls `end` when it has received some result for it. Since `begin` and `end` only maintain a counter, the order in which results arrive is not relevant.

There is an additional bracketing call to `begin` and `end` outside the loop, which serves two important purposes: first, it sets the callback to be called once the counter reaches `0`, and second, it ensures that `send` is called even when no hosts are being pinged (the loop doesn't execute once).

This is the general pattern when you "fan out" into multiple (but potentially zero) subrequests: use an outer `begin`/`end` pair to set the callback and ensure `end` is called at least once, and then, for each subrequest you start, call `begin` and for each subrequest you finish, call `end`.

METHODS FOR CONSUMERS

These methods should only be used by the consuming side, i.e. the code awaits the condition.

- $cv->recv

  Wait (blocking if necessary) until the `->send` or `->croak` methods have been called on `$cv`, while servicing other watchers normally.

  You can only wait once on a condition - additional calls are valid but will return immediately.

  If an error condition has been set by calling `->croak`, then this function will call `croak`.

  In list context, all parameters passed to `send` will be returned, in scalar context only the first one will be returned.

  Note that doing a blocking wait in a callback is not supported by any event loop, that is, recursive invocation of a blocking `->recv` is not allowed and the `recv` call will `croak` if such a condition is detected. This requirement can be dropped by relying on *Coro::AnyEvent* , which allows you to do a blocking `->recv` from any thread that doesn't run the event loop itself. *Coro::AnyEvent* is loaded automatically when *Coro* is used with *AnyEvent*, so code does not need to do anything special to take advantage of that: any code that would normally block your program because it calls `recv`, be executed in an `async` thread instead without blocking other threads.

  Not all event models support a blocking wait - some die in that case (programs might want to do that to stay interactive), so *if you are using this from a module, never require a blocking wait*. Instead, let the caller decide whether the call will block or not (for example, by coupling condition variables with some kind of request results and supporting callbacks so the caller knows that getting the result will not block, while still supporting blocking waits if the caller so desires).

  You can ensure that `->recv` never blocks by setting a callback and only calling `->recv` from within that callback (or at a later time). This will work even when the event loop does not support blocking waits otherwise.

- $bool = $cv->ready

  Returns true when the condition is "true", i.e. whether `send` or `croak` have been called.

- $cb = $cv->cb ($cb->($cv))

  This is a mutator function that returns the callback set (or `undef` if not) and optionally replaces it before doing so.

  The callback will be called when the condition becomes "true", i.e. when `send` or `croak` are called, with the only argument being the condition variable itself. If the condition is already true, the callback is called immediately when it is set. Calling `recv` inside the callback or at any later time is guaranteed not to block.

  Additionally, when the callback is invoked, it is also removed from the condvar (reset to `undef`), so the condvar does not keep a reference to the callback after invocation.

SUPPORTED EVENT LOOPS/BACKENDS

The following backend classes are part of the AnyEvent distribution (every class has its own manpage):

- Backends that are autoprobed when no other event loop can be found.

  EV is the preferred backend when no other event loop seems to be in use. If EV is not installed, then AnyEvent will fall back to its own pure-perl implementation, which is available everywhere as it comes with AnyEvent itself.

```
AnyEvent::Impl::EV          based on EV (interface to libev, best␣
↪choice).
AnyEvent::Impl::Perl        pure-perl AnyEvent::Loop, fast and␣
↪portable.
```

- Backends that are transparently being picked up when they are used.

  These will be used if they are already loaded when the first watcher is created, in which case it is assumed that the application is using them. This means that AnyEvent will automatically pick the right backend when the main program loads an event module before anything starts to create watchers. Nothing special needs to be done by the main program.

```
AnyEvent::Impl::Event       based on Event, very stable, few␣
↪glitches.
AnyEvent::Impl::Glib        based on Glib, slow but very stable.
AnyEvent::Impl::Tk          based on Tk, very broken.
AnyEvent::Impl::UV          based on UV, innovated square wheels.
AnyEvent::Impl::EventLib    based on Event::Lib, leaks memory and␣
↪worse.
AnyEvent::Impl::POE         based on POE, very slow, some␣
↪limitations.
AnyEvent::Impl::Irssi       used when running within irssi.
AnyEvent::Impl::IOAsync     based on IO::Async.
AnyEvent::Impl::Cocoa       based on Cocoa::EventLoop.
AnyEvent::Impl::FLTK        based on FLTK (fltk 2 binding).
```

- Backends with special needs.

  Qt requires the Qt::Application to be instantiated first, but will otherwise be picked up automatically. As long as the main program instantiates the application before any AnyEvent watchers are created, everything should just work.

```
AnyEvent::Impl::Qt          based on Qt.
```

- Event loops that are indirectly supported via other backends.

  Some event loops can be supported via other modules:

  There is no direct support for WxWidgets (Wx) or Prima.

  WxWidgets has no support for watching file handles. However, you can use WxWidgets through the POE adaptor, as POE has a Wx backend that simply polls 20 times per second, which was considered to be too horrible to even consider for AnyEvent.

Prima is not supported as nobody seems to be using it, but it has a POE backend, so it can be supported through POE.

AnyEvent knows about both Prima and Wx, however, and will try to load POE when detecting them, in the hope that POE will pick them up, in which case everything will be automatic.

- Known event loops outside the AnyEvent distribution

  The following event loops or programs support AnyEvent by providing their own AnyEvent backend. They will be picked up automatically.

```
urxvt::anyevent           available to rxvt-unicode extensions
```

GLOBAL VARIABLES AND FUNCTIONS

These are not normally required to use AnyEvent, but can be useful to write AnyEvent extension modules.

- $AnyEvent::MODEL

  Contains `undef` until the first watcher is being created, before the backend has been autodetected.

  Afterwards it contains the event model that is being used, which is the name of the Perl class implementing the model. This class is usually one of the `AnyEvent::Impl::xxx` modules, but can be any other class in the case AnyEvent has been extended at runtime (e.g. in *rxvt-unicode* it will be `urxvt::anyevent`).

- AnyEvent::detect

  Returns `$AnyEvent::MODEL`, forcing autodetection of the event model if necessary. You should only call this function right before you would have created an AnyEvent watcher anyway, that is, as late as possible at runtime, and not e.g. during initialisation of your module.

  The effect of calling this function is as if a watcher had been created (specifically, actions that happen “when the first watcher is created” happen when calling detetc as well).

  If you need to do some initialisation before AnyEvent watchers are created, use `post_detect`.

- $guard = AnyEvent::post_detect { BLOCK }

  Arranges for the code block to be executed as soon as the event model is autodetected (or immediately if that has already happened).

  The block will be executed *after* the actual backend has been detected (`$AnyEvent::MODEL` is set), so it is possible to do some initialisation only when AnyEvent is actually initialised - see the sources of AnyEvent::AIO to see how this is used.

  The most common usage is to create some global watchers, without forcing event module detection too early. For example, AnyEvent::AIO creates and installs the global IO::AIO watcher in a `post_detect` block to avoid autodetecting the event module at load time.

If called in scalar or list context, then it creates and returns an object that automatically removes the callback again when it is destroyed (or `undef` when the hook was immediately executed). See AnyEvent::AIO for a case where this is useful.

Example: Create a watcher for the IO::AIO module and store it in `$WATCHER`, but do so only do so after the event loop is initialised.

```
our WATCHER;

my $guard = AnyEvent::post_detect {
   $WATCHER = AnyEvent->io (fh => IO::AIO::poll_fileno, poll =>
 ↪'r', cb => \&IO::AIO::poll_cb);
};

# the ||= is important in case post_detect immediately runs the␣
 ↪block,
# as to not clobber the newly-created watcher. assigning both␣
 ↪watcher and
# post_detect guard to the same variable has the advantage of␣
 ↪users being
# able to just C<undef $WATCHER> if the watcher causes them␣
 ↪grief.

$WATCHER ||= $guard;
```

- @AnyEvent::post_detect

  This is a lower level interface then `AnyEvent::post_detect` (the function). This variable is mainly useful for modules that can do something useful when AnyEvent is used and thus want to know when it is initialised, but do not need to even load it by default. This array provides the means to hook into AnyEvent passively, without loading it.

  Here is how it works: If there are any code references in this array (you can `push` to it before or after loading AnyEvent), then they will be called directly after the event loop has been chosen.

  You should check `$AnyEvent::MODEL` before adding to this array, though: if it is defined then the event loop has already been detected, and the array will be ignored.

  Best use `AnyEvent::post_detect { BLOCK }` when your application allows it, as it takes care of these details.

  Example: To load Coro::AnyEvent whenever Coro and AnyEvent are used together, you could put this into Coro (this is the actual code used by Coro to accomplish this):

```
if (defined $AnyEvent::MODEL) {
   # AnyEvent already initialised, so load Coro::AnyEvent
   require Coro::AnyEvent;
} else {
   # AnyEvent not yet initialised, so make sure to load␣
 ↪Coro::AnyEvent
   # as soon as it is
```

(continues on next page)

(continued from previous page)

```
   push @AnyEvent::post_detect, sub { require Coro::AnyEvent };
}
```

- AnyEvent::postpone { BLOCK }

  Arranges for the block to be executed as soon as possible, but not before the call itself returns. In practise, the block will be executed just before the event loop polls for new events, or shortly afterwards.

  This function never returns anything (to make the `return postpone { ... }` idiom more useful.

  To understand the usefulness of this function, consider a function that asynchronously does something for you and returns some transaction object or guard to let you cancel the operation. For example, `AnyEvent::Socket::tcp_connect`:

```
# start a connection attempt unless one is active
$self->{connect_guard} ||= AnyEvent::Socket::tcp_connect "www.
↪example.net", 80, sub {
   delete $self->{connect_guard};
   ...
};
```

  Imagine that this function could instantly call the callback, for example, because it detects an obvious error such as a negative port number. Invoking the callback before the function returns causes problems however: the callback will be called and will try to delete the guard object. But since the function hasn't returned yet, there is nothing to delete. When the function eventually returns it will assign the guard object to `$self->{connect_guard}`, where it will likely never be deleted, so the program thinks it is still trying to connect.

  This is where `AnyEvent::postpone` should be used. Instead of calling the callback directly on error:

```
$cb->(undef), return # signal error to callback, BAD!
   if $some_error_condition;
```

  It should use `postpone`:

```
AnyEvent::postpone { $cb->(undef) }, return # signal error to␣
↪callback, later
   if $some_error_condition;
```

- AnyEvent::log $level, $msg[, @args]

  Log the given `$msg` at the given `$level`.

  If *AnyEvent::Log* is not loaded then this function makes a simple test to see whether the message will be logged. If the test succeeds it will load AnyEvent::Log and call `AnyEvent::Log::log` - consequently, look at the *AnyEvent::Log* documentation for details.

  If the test fails it will simply return. Right now this happens when a numerical loglevel is used and it is larger than the level specified via

`$ENV{PERL_ANYEVENT_VERBOSE}`.

If you want to sprinkle loads of logging calls around your code, consider creating a logger callback with the `AnyEvent::Log::logger` function, which can reduce typing, codesize and can reduce the logging overhead enourmously.

- AnyEvent::fh_block $filehandle
- AnyEvent::fh_unblock $filehandle

  Sets blocking or non-blocking behaviour for the given filehandle.

WHAT TO DO IN A MODULE

As a module author, you should `use AnyEvent` and call AnyEvent methods freely, but you should not load a specific event module or rely on it.

Be careful when you create watchers in the module body - AnyEvent will decide which event module to use as soon as the first method is called, so by calling AnyEvent in your module body you force the user of your module to load the event module first.

Never call `->recv` on a condition variable unless you *know* that the `->send` method has been called on it already. This is because it will stall the whole program, and the whole point of using events is to stay interactive.

It is fine, however, to call `->recv` when the user of your module requests it (i.e. if you create a http request object ad have a method called `results` that returns the results, it may call `->recv` freely, as the user of your module knows what she is doing. Always).

WHAT TO DO IN THE MAIN PROGRAM

There will always be a single main program - the only place that should dictate which event model to use.

If the program is not event-based, it need not do anything special, even when it depends on a module that uses an AnyEvent. If the program itself uses AnyEvent, but does not care which event loop is used, all it needs to do is `use AnyEvent`. In either case, AnyEvent will choose the best available loop implementation.

If the main program relies on a specific event model - for example, in Gtk2 programs you have to rely on the Glib module - you should load the event module before loading AnyEvent or any module that uses it: generally speaking, you should load it as early as possible. The reason is that modules might create watchers when they are loaded, and AnyEvent will decide on the event model to use as soon as it creates watchers, and it might choose the wrong one unless you load the correct one yourself.

You can chose to use a pure-perl implementation by loading the `AnyEvent::Loop` module, which gives you similar behaviour everywhere, but letting AnyEvent chose the model is generally better.

MAINLOOP EMULATION

Sometimes (often for short test scripts, or even standalone programs who only want to use AnyEvent), you do not want to run a specific event loop.

In that case, you can use a condition variable like this:

```
AnyEvent->condvar->recv;
```

This has the effect of entering the event loop and looping forever.

Note that usually your program has some exit condition, in which case it is better to use the "traditional" approach of storing a condition variable somewhere, waiting for it, and sending it when the program should exit cleanly.

OTHER MODULES

The following is a non-exhaustive list of additional modules that use AnyEvent as a client and can therefore be mixed easily with other AnyEvent modules and other event loops in the same program. Some of the modules come as part of AnyEvent, the others are available via CPAN (see http://search.cpan.org/search?m=module&q=anyevent%3A%3A* for a longer non-exhaustive list), and the list is heavily biased towards modules of the AnyEvent author himself :)

- *AnyEvent::Util* (part of the AnyEvent distribution)

  Contains various utility functions that replace often-used blocking functions such as `inet_aton` with event/callback-based versions.

- *AnyEvent::Socket* (part of the AnyEvent distribution)

  Provides various utility functions for (internet protocol) sockets, addresses and name resolution. Also functions to create non-blocking tcp connections or tcp servers, with IPv6 and SRV record support and more.

- *AnyEvent::Handle* (part of the AnyEvent distribution)

  Provide read and write buffers, manages watchers for reads and writes, supports raw and formatted I/O, I/O queued and fully transparent and non-blocking SSL/TLS (via *AnyEvent::TLS*).

- *AnyEvent::DNS* (part of the AnyEvent distribution)

  Provides rich asynchronous DNS resolver capabilities.

- AnyEvent::HTTP, AnyEvent::IRC, AnyEvent::XMPP, AnyEvent::GPSD, AnyEvent::IGS, AnyEvent::FCP

  Implement event-based interfaces to the protocols of the same name (for the curious, IGS is the International Go Server and FCP is the Freenet Client Protocol).

- AnyEvent::AIO (part of the AnyEvent distribution)

  Truly asynchronous (as opposed to non-blocking) I/O, should be in the toolbox of every event programmer. AnyEvent::AIO transparently fuses IO::AIO and AnyEvent together, giving AnyEvent access to event-based file I/O, and much more.

- AnyEvent::Fork, AnyEvent::Fork::RPC, AnyEvent::Fork::Pool, AnyEvent::Fork::Remote

  These let you safely fork new subprocesses, either locally or remotely (e.g.v ia ssh), using some RPC protocol or not, without the limitations normally imposed by fork (AnyEvent works fine for example). Dynamically-resized worker pools are obviously included as well.

And they are quite tiny and fast as well - "abusing" AnyEvent::Fork just to exec external programs can easily beat using `fork` and `exec` (or even `system`) in most programs.

- AnyEvent::Filesys::Notify

  AnyEvent is good for non-blocking stuff, but it can't detect file or path changes (e.g. "watch this directory for new files", "watch this file for changes"). The AnyEvent::Filesys::Notify module promises to do just that in a portbale fashion, supporting inotify on GNU/Linux and some weird, without doubt broken, stuff on OS X to monitor files. It can fall back to blocking scans at regular intervals transparently on other platforms, so it's about as portable as it gets.

  (I haven't used it myself, but it seems the biggest problem with it is it quite bad performance).

- AnyEvent::DBI

  Executes *DBI* requests asynchronously in a proxy process for you, notifying you in an event-based way when the operation is finished.

- AnyEvent::FastPing

  The fastest ping in the west.

- *Coro*

  Has special support for AnyEvent via *Coro::AnyEvent*, which allows you to simply invert the flow control - don't call us, we will call you:

```
async {
   Coro::AnyEvent::sleep 5; # creates a 5s timer and waits for
↪it
   print "5 seconds later!\n";

   Coro::AnyEvent::readable *STDIN; # uses an I/O watcher
   my $line = <STDIN>; # works for ttys

   AnyEvent::HTTP::http_get "url", Coro::rouse_cb;
   my ($body, $hdr) = Coro::rouse_wait;
};
```

SIMPLIFIED AE API

Starting with version 5.0, AnyEvent officially supports a second, much simpler, API that is designed to reduce the calling, typing and memory overhead by using function call syntax and a fixed number of parameters.

See the *AE* manpage for details.

ERROR AND EXCEPTION HANDLING

In general, AnyEvent does not do any error handling - it relies on the caller to do that if required. The *AnyEvent::Strict* module (see also the `PERL_ANYEVENT_STRICT` environment variable, below) provides strict checking of all AnyEvent methods, however, which is highly useful during development.

As for exception handling (i.e. runtime errors and exceptions thrown while executing a callback), this is not only highly event-loop specific, but also not in any way wrapped by this module, as this is the job of the main program.

The pure perl event loop simply re-throws the exception (usually within `condvar->recv`), the *Event* and *EV* modules call `$Event/EV::DIED->()`, Glib uses `install_exception_handler` and so on.

ENVIRONMENT VARIABLES

AnyEvent supports a number of environment variables that tune the runtime behaviour. They are usually evaluated when AnyEvent is loaded, initialised, or a submodule that uses them is loaded. Many of them also cause AnyEvent to load additional modules - for example, `PERL_ANYEVENT_DEBUG_WRAP` causes the *AnyEvent::Debug* module to be loaded.

All the environment variables documented here start with `PERL_ANYEVENT_`, which is what AnyEvent considers its own namespace. Other modules are encouraged (but by no means required) to use `PERL_ANYEVENT_SUBMODULE` if they have registered the AnyEvent::Submodule namespace on CPAN, for any submodule. For example, AnyEvent::HTTP could be expected to use `PERL_ANYEVENT_HTTP_PROXY` (it should not access env variables starting with `AE_`, see below).

All variables can also be set via the `AE_` prefix, that is, instead of setting `PERL_ANYEVENT_VERBOSE` you can also set `AE_VERBOSE`. In case there is a clash btween anyevent and another program that uses `AE_something` you can set the corresponding `PERL_ANYEVENT_something` variable to the empty string, as those variables take precedence.

When AnyEvent is first loaded, it copies all `AE_xxx` env variables to their `PERL_ANYEVENT_xxx` counterpart unless that variable already exists. If taint mode is on, then AnyEvent will remove *all* environment variables starting with `PERL_ANYEVENT_` from `%ENV` (or replace them with `undef` or the empty string, if the corresaponding `AE_` variable is set).

The exact algorithm is currently:

```
1. if taint mode enabled, delete all PERL_ANYEVENT_xyz variables␣
↪from %ENV
2. copy over AE_xyz to PERL_ANYEVENT_xyz unless the latter alraedy␣
↪exists
3. if taint mode enabled, set all PERL_ANYEVENT_xyz variables to␣
↪undef.
```

This ensures that child processes will not see the `AE_` variables.

The following environment variables are currently known to AnyEvent:

- `PERL_ANYEVENT_VERBOSE`

  By default, AnyEvent will log messages with loglevel 4 (`error`) or higher (see *AnyEvent::Log*). You can set this environment variable to a numerical loglevel to make AnyEvent more (or less) talkative.

  If you want to do more than just set the global logging level you should have a look at `PERL_ANYEVENT_LOG`, which allows much more complex specifications.

When set to `0` (`off`), then no messages whatsoever will be logged with everything else at defaults.

When set to 5 or higher (`warn`), AnyEvent warns about unexpected conditions, such as not being able to load the event model specified by `PERL_ANYEVENT_MODEL`, or a guard callback throwing an exception - this is the minimum recommended level for use during development.

When set to 7 or higher (info), AnyEvent reports which event model it chooses.

When set to 8 or higher (debug), then AnyEvent will report extra information on which optional modules it loads and how it implements certain features.

- `PERL_ANYEVENT_LOG`

  Accepts rather complex logging specifications. For example, you could log all debug messages of some module to stderr, warnings and above to stderr, and errors and above to syslog, with:

```
PERL_ANYEVENT_LOG=Some::Module=debug,+log:filter=warn,+%syslog:
 ↪%syslog=error,syslog
```

  For the rather extensive details, see *AnyEvent::Log*.

  This variable is evaluated when AnyEvent (or *AnyEvent::Log*) is loaded, so will take effect even before AnyEvent has initialised itself.

  Note that specifying this environment variable causes the *AnyEvent::Log* module to be loaded, while `PERL_ANYEVENT_VERBOSE` does not, so only using the latter saves a few hundred kB of memory unless a module explicitly needs the extra features of AnyEvent::Log.

- `PERL_ANYEVENT_STRICT`

  AnyEvent does not do much argument checking by default, as thorough argument checking is very costly. Setting this variable to a true value will cause AnyEvent to load `AnyEvent::Strict` and then to thoroughly check the arguments passed to most method calls. If it finds any problems, it will croak.

  In other words, enables "strict" mode.

  Unlike `use strict` (or its modern cousin, `use [common::sense](https://metacpan.org/pod/common::sense)`, it is definitely recommended to keep it off in production. Keeping `PERL_ANYEVENT_STRICT=1` in your environment while developing programs can be very useful, however.

- `PERL_ANYEVENT_DEBUG_SHELL`

  If this env variable is nonempty, then its contents will be interpreted by `AnyEvent::Socket::parse_hostport` and `AnyEvent::Debug::shell` (after replacing every occurance of `$$` by the process pid). The shell object is saved in `$AnyEvent::Debug::SHELL`.

  This happens when the first watcher is created.

  For example, to bind a debug shell on a unix domain socket in `/tmp/debug<pid>.sock`, you could use this:

```
PERL_ANYEVENT_DEBUG_SHELL=/tmp/debug\$\$.sock perlprog
# connect with e.g.: socat readline /tmp/debug123.sock
```

Or to bind to tcp port 4545 on localhost:

```
PERL_ANYEVENT_DEBUG_SHELL=127.0.0.1:4545 perlprog
# connect with e.g.: telnet localhost 4545
```

Note that creating sockets in `/tmp` or on localhost is very unsafe on multiuser systems.

- `PERL_ANYEVENT_DEBUG_WRAP`

  Can be set to `0`, 1 or 2 and enables wrapping of all watchers for debugging purposes. See `AnyEvent::Debug::wrap` for details.

- `PERL_ANYEVENT_MODEL`

  This can be used to specify the event model to be used by AnyEvent, before auto detection and -probing kicks in.

  It normally is a string consisting entirely of ASCII letters (e.g. `EV` or `IOAsync`). The string `AnyEvent::Impl::` gets prepended and the resulting module name is loaded and - if the load was successful - used as event model backend. If it fails to load then AnyEvent will proceed with auto detection and -probing.

  If the string ends with `::` instead (e.g. `AnyEvent::Impl::EV::`) then nothing gets prepended and the module name is used as-is (hint: `::` at the end of a string designates a module name and quotes it appropriately).

  For example, to force the pure perl model (AnyEvent::Loop::Perl) you could start your program like this:

```
PERL_ANYEVENT_MODEL=Perl perl ...
```

- `PERL_ANYEVENT_IO_MODEL`

  The current file I/O model - see *AnyEvent::IO* for more info.

  At the moment, only `Perl` (small, pure-perl, synchronous) and `IOAIO` (truly asynchronous) are supported. The default is `IOAIO` if AnyEvent::AIO can be loaded, otherwise it is `Perl`.

- `PERL_ANYEVENT_PROTOCOLS`

  Used by both *AnyEvent::DNS* and *AnyEvent::Socket* to determine preferences for IPv4 or IPv6. The default is unspecified (and might change, or be the result of auto probing).

  Must be set to a comma-separated list of protocols or address families, current supported: `ipv4` and `ipv6`. Only protocols mentioned will be used, and preference will be given to protocols mentioned earlier in the list.

  This variable can effectively be used for denial-of-service attacks against local programs (e.g. when setuid), although the impact is likely small, as the program has to handle connection and other failures anyways.

Examples: `PERL_ANYEVENT_PROTOCOLS=ipv4,ipv6` - prefer IPv4 over IPv6, but support both and try to use both. `PERL_ANYEVENT_PROTOCOLS=ipv4` - only support IPv4, never try to resolve or contact IPv6 addresses. `PERL_ANYEVENT_PROTOCOLS=ipv6,ipv4` support either IPv4 or IPv6, but prefer IPv6 over IPv4.

- `PERL_ANYEVENT_HOSTS`

  This variable, if specified, overrides the `/etc/hosts` file used by *AnyEvent::Socket*`::resolve_sockaddr`, i.e. hosts aliases will be read from that file instead.

- `PERL_ANYEVENT_EDNS0`

  Used by *AnyEvent::DNS* to decide whether to use the EDNS0 extension for DNS. This extension is generally useful to reduce DNS traffic, especially when DNSSEC is involved, but some (broken) firewalls drop such DNS packets, which is why it is off by default.

  Setting this variable to 1 will cause *AnyEvent::DNS* to announce EDNS0 in its DNS requests.

- `PERL_ANYEVENT_MAX_FORKS`

  The maximum number of child processes that `AnyEvent::Util::fork_call` will create in parallel.

- `PERL_ANYEVENT_MAX_OUTSTANDING_DNS`

  The default value for the `max_outstanding` parameter for the default DNS resolver - this is the maximum number of parallel DNS requests that are sent to the DNS server.

- `PERL_ANYEVENT_MAX_SIGNAL_LATENCY`

  Perl has inherently racy signal handling (you can basically choose between losing signals and memory corruption) - pure perl event loops (including `AnyEvent::Loop`, when `Async::Interrupt` isn't available) therefore have to poll regularly to avoid losing signals.

  Some event loops are racy, but don't poll regularly, and some event loops are written in C but are still racy. For those event loops, AnyEvent installs a timer that regularly wakes up the event loop.

  By default, the interval for this timer is `10` seconds, but you can override this delay with this environment variable (or by setting the `$AnyEvent::MAX_SIGNAL_LATENCY` variable before creating signal watchers).

  Lower values increase CPU (and energy) usage, higher values can introduce long delays when reaping children or waiting for signals.

  The AnyEvent::Async module, if available, will be used to avoid this polling (with most event loops).

- `PERL_ANYEVENT_RESOLV_CONF`

  The absolute path to a `resolv.conf`-style file to use instead of `/etc/resolv.conf` (or the OS-specific configuration) in the default resolver, or the empty string to select the default configuration.

- `PERL_ANYEVENT_CA_FILE`, `PERL_ANYEVENT_CA_PATH`.

  When neither `ca_file` nor `ca_path` was specified during *AnyEvent::TLS* context creation, and either of these environment variables are nonempty, they will be used to specify CA certificate locations instead of a system-dependent default.

- `PERL_ANYEVENT_AVOID_GUARD` and `PERL_ANYEVENT_AVOID_ASYNC_INTERRUPT`

  When these are set to 1, then the respective modules are not loaded. Mostly good for testing AnyEvent itself.

SUPPLYING YOUR OWN EVENT MODEL INTERFACE

This is an advanced topic that you do not normally need to use AnyEvent in a module. This section is only of use to event loop authors who want to provide AnyEvent compatibility.

If you need to support another event library which isn't directly supported by AnyEvent, you can supply your own interface to it by pushing, before the first watcher gets created, the package name of the event module and the package name of the interface to use onto `@AnyEvent::REGISTRY`. You can do that before and even without loading AnyEvent, so it is reasonably cheap.

Example:

```
push @AnyEvent::REGISTRY, [urxvt => urxvt::anyevent::];
```

This tells AnyEvent to (literally) use the `urxvt::anyevent::` package/class when it finds the `urxvt` package/module is already loaded.

When AnyEvent is loaded and asked to find a suitable event model, it will first check for the presence of urxvt by trying to `use` the `urxvt::anyevent` module.

The class should provide implementations for all watcher types. See *AnyEvent::Impl::EV* (source code), *AnyEvent::Impl::Glib* (Source code) and so on for actual examples. Use `perldoc -m AnyEvent::Impl::Glib` to see the sources.

If you don't provide `signal` and `child` watchers than AnyEvent will provide suitable (hopefully) replacements.

The above example isn't fictitious, the *rxvt-unicode* (a.k.a. urxvt) terminal emulator uses the above line as-is. An interface isn't included in AnyEvent because it doesn't make sense outside the embedded interpreter inside *rxvt-unicode*, and it is updated and maintained as part of the *rxvt-unicode* distribution.

rxvt-unicode also cheats a bit by not providing blocking access to condition variables: code blocking while waiting for a condition will `die`. This still works with most modules/usages, and blocking calls must not be done in an interactive application, so it makes sense.

EXAMPLE PROGRAM

The following program uses an I/O watcher to read data from STDIN, a timer to display a message once per second, and a condition variable to quit the program when the user enters quit:

```
use AnyEvent;

my $cv = AnyEvent->condvar;

my $io_watcher = AnyEvent->io (
   fh   => \*STDIN,
   poll => 'r',
   cb   => sub {
      warn "io event <$_[0]>\n";   # will always output <r>
      chomp (my $input = <STDIN>); # read a line
      warn "read: $input\n";       # output what has been read
      $cv->send if $input =~ /^q/i; # quit program if /^q/i
   },
);

my $time_watcher = AnyEvent->timer (after => 1, interval => 1, cb =>
↪sub {
   warn "timeout\n"; # print 'timeout' at most every second
});

$cv->recv; # wait until user enters /^q/i
```

REAL-WORLD EXAMPLE

Consider the Net::FCP module. It features (among others) the following API calls, which are to freenet what HTTP GET requests are to http:

```
my $data = $fcp->client_get ($url); # blocks

my $transaction = $fcp->txn_client_get ($url); # does not block
$transaction->cb ( sub { ... } ); # set optional result callback
my $data = $transaction->result; # possibly blocks
```

The `client_get` method works like `LWP::Simple::get`: it requests the given URL and waits till the data has arrived. It is defined to be:

```
sub client_get { $_[0]->txn_client_get ($_[1])->result }
```

And in fact is automatically generated. This is the blocking API of Net::FCP, and it works as simple as in any other, similar, module.

More complicated is `txn_client_get`: It only creates a transaction (completion, result, ...) object and initiates the transaction.

```
my $txn = bless { }, Net::FCP::Txn::;
```

It also creates a condition variable that is used to signal the completion of the request:

```
$txn->{finished} = AnyAvent->condvar;
```

It then creates a socket in non-blocking mode.

```
socket $txn->{fh}, ...;
fcntl $txn->{fh}, F_SETFL, O_NONBLOCK;
connect $txn->{fh}, ...
   and !$!{EWOULDBLOCK}
   and !$!{EINPROGRESS}
   and Carp::croak "unable to connect: $!\n";
```

Then it creates a write-watcher which gets called whenever an error occurs or the connection succeeds:

```
$txn->{w} = AnyEvent->io (fh => $txn->{fh}, poll => 'w', cb => sub {
↪$txn->fh_ready_w });
```

And returns this transaction object. The `fh_ready_w` callback gets called as soon as the event loop detects that the socket is ready for writing.

The `fh_ready_w` method makes the socket blocking again, writes the request data and replaces the watcher by a read watcher (waiting for reply data). The actual code is more complicated, but that doesn't matter for this example:

```
fcntl $txn->{fh}, F_SETFL, 0;
syswrite $txn->{fh}, $txn->{request}
   or die "connection or write error";
$txn->{w} = AnyEvent->io (fh => $txn->{fh}, poll => 'r', cb => sub {
↪$txn->fh_ready_r });
```

Again, `fh_ready_r` waits till all data has arrived, and then stores the result and signals any possible waiters that the request has finished:

```
sysread $txn->{fh}, $txn->{buf}, length $txn->{$buf};

if (end-of-file or data complete) {
  $txn->{result} = $txn->{buf};
  $txn->{finished}->send;
  $txb->{cb}->($txn) of $txn->{cb}; # also call callback
}
```

The `result` method, finally, just waits for the finished signal (if the request was already finished, it doesn't wait, of course, and returns the data:

```
$txn->{finished}->recv;
return $txn->{result};
```

The actual code goes further and collects all errors (`die`s, exceptions) that occurred during request processing. The `result` method detects whether an exception as thrown (it is stored inside the $txn object) and just throws the exception, which means connection errors and other problems get reported to the code that tries to use the result, not in a random callback.

All of this enables the following usage styles:

1. Blocking:

```
my $data = $fcp->client_get ($url);
```

2. Blocking, but running in parallel:

```
my @datas = map $_->result,
               map $fcp->txn_client_get ($_),
                  @urls;
```

Both blocking examples work without the module user having to know anything about events.

3a. Event-based in a main program, using any supported event module:

```
use EV;

$fcp->txn_client_get ($url)->cb (sub {
   my $txn = shift;
   my $data = $txn->result;
   ...
});

EV::run;
```

3b. The module user could use AnyEvent, too:

```
use AnyEvent;

my $quit = AnyEvent->condvar;

$fcp->txn_client_get ($url)->cb (sub {
   ...
   $quit->send;
});

$quit->recv;
```

BENCHMARKS

To give you an idea of the performance and overheads that AnyEvent adds over the event loops themselves and to give you an impression of the speed of various event loops I prepared some benchmarks.

BENCHMARKING ANYEVENT OVERHEAD

Here is a benchmark of various supported event models used natively and through AnyEvent. The benchmark creates a lot of timers (with a zero timeout) and I/O watchers (watching STDOUT, a pty, to become writable, which it is), lets them fire exactly once and destroys them again.

Source code for this benchmark is found as `eg/bench` in the AnyEvent distribution. It uses the *AE* interface, which makes a real difference for the EV and Perl backends only.

Explanation of the columns

watcher is the number of event watchers created/destroyed. Since different event models feature vastly different performances, each event loop was given a number of watchers so that overall runtime is acceptable and similar between tested event loop (and keep them from crashing): Glib would probably take thousands of years if asked to process the same number of watchers as EV in this benchmark.

bytes is the number of bytes (as measured by the resident set size, RSS) consumed by each watcher. This method of measuring captures both C and Perl-based overheads.

create is the time, in microseconds (millionths of seconds), that it takes to create a single watcher. The callback is a closure shared between all watchers, to avoid adding memory overhead. That means closure creation and memory usage is not included in the figures.

invoke is the time, in microseconds, used to invoke a simple callback. The callback simply counts down a Perl variable and after it was invoked "watcher" times, it would `->send` a condvar once to signal the end of this phase.

destroy is the time, in microseconds, that it takes to destroy a single watcher.

Results

```
        name watchers bytes create invoke destroy comment
       EV/EV   100000   223   0.47   0.43    0.27 EV native␣
↪interface
      EV/Any   100000   223   0.48   0.42    0.26 EV + AnyEvent␣
↪watchers
Coro::EV/Any   100000   223   0.47   0.42    0.26 coroutines +␣
↪Coro::Signal
    Perl/Any   100000   431   2.70   0.74    0.92 pure perl␣
↪implementation
 Event/Event    16000   516  31.16  31.84    0.82 Event native␣
↪interface
   Event/Any    16000  1203  42.61  34.79    1.80 Event + AnyEvent␣
↪watchers
 IOAsync/Any    16000  1911  41.92  27.45   16.81 via␣
↪IO::Async::Loop::IO_Poll
 IOAsync/Any    16000  1726  40.69  26.37   15.25 via␣
↪IO::Async::Loop::Epoll
    Glib/Any    16000  1118  89.00  12.57   51.17 quadratic␣
↪behaviour
      Tk/Any     2000  1346  20.96  10.75    8.00 SEGV with >> 2000␣
↪watchers
     POE/Any     2000  6951 108.97 795.32   14.24 via␣
↪POE::Loop::Event
     POE/Any     2000  6648  94.79 774.40  575.51 via␣
↪POE::Loop::Select
```

Discussion

The benchmark does *not* measure scalability of the event loop very well. For example, a select-based event loop (such as the pure perl one) can never compete with an event loop that uses epoll when the number of file descriptors grows high. In this benchmark, all events become ready at the same time, so select/poll-based implementations get an unnatural speed boost.

Also, note that the number of watchers usually has a nonlinear effect on overall speed, that is, creating twice as many watchers doesn't take twice the time - usually it takes longer. This puts event loops tested with a higher number of watchers at a disadvantage.

To put the range of results into perspective, consider that on the benchmark machine, handling an event takes roughly 1600 CPU cycles with EV, 3100 CPU cycles with AnyEvent's pure perl loop and almost 3000000 CPU cycles with POE.

EV is the sole leader regarding speed and memory use, which are both maximal/minimal, respectively. When using the *AE* API there is zero overhead (when going through the AnyEvent API create is about 5-6 times slower, with other times being equal, so still uses far less memory than any other event loop and is still faster than Event natively).

The pure perl implementation is hit in a few sweet spots (both the constant timeout and the use of a single fd hit optimisations in the perl interpreter and the backend itself). Nevertheless this shows that it adds very little overhead in itself. Like any select-based backend its performance becomes really bad with lots of file descriptors (and few of them active), of course, but this was not subject of this benchmark.

The `Event` module has a relatively high setup and callback invocation cost, but overall scores in on the third place.

`IO::Async` performs admirably well, about on par with `Event`, even when using its pure perl backend.

`Glib`'s memory usage is quite a bit higher, but it features a faster callback invocation and overall ends up in the same class as `Event`. However, Glib scales extremely badly, doubling the number of watchers increases the processing time by more than a factor of four, making it completely unusable when using larger numbers of watchers (note that only a single file descriptor was used in the benchmark, so inefficiencies of `poll` do not account for this).

The `Tk` adaptor works relatively well. The fact that it crashes with more than 2000 watchers is a big setback, however, as correctness takes precedence over speed. Nevertheless, its performance is surprising, as the file descriptor is dup()ed for each watcher. This shows that the dup() employed by some adaptors is not a big performance issue (it does incur a hidden memory cost inside the kernel which is not reflected in the figures above).

POE, regardless of underlying event loop (whether using its pure perl select-based backend or the Event module, the POE-EV backend couldn't be tested because it wasn't working) shows abysmal performance and memory usage with AnyEvent: Watchers use almost 30 times as much memory as EV watchers, and 10 times as much memory as Event (the high memory requirements are caused by requiring a session for each watcher). Watcher invocation speed is almost 900 times slower than with AnyEvent's pure perl implementation.

The design of the POE adaptor class in AnyEvent can not really account for the performance issues, though, as session creation overhead is small compared to execution of the

state machine, which is coded pretty optimally within *AnyEvent::Impl::POE* (and while everybody agrees that using multiple sessions is not a good approach, especially regarding memory usage, even the author of POE could not come up with a faster design).

Summary

- Using EV through AnyEvent is faster than any other event loop (even when used without AnyEvent), but most event loops have acceptable performance with or without AnyEvent.
- The overhead AnyEvent adds is usually much smaller than the overhead of the actual event loop, only with extremely fast event loops such as EV does AnyEvent add significant overhead.
- You should avoid POE like the plague if you want performance or reasonable memory usage.

BENCHMARKING THE LARGE SERVER CASE

This benchmark actually benchmarks the event loop itself. It works by creating a number of "servers": each server consists of a socket pair, a timeout watcher that gets reset on activity (but never fires), and an I/O watcher waiting for input on one side of the socket. Each time the socket watcher reads a byte it will write that byte to a random other "server".

The effect is that there will be a lot of I/O watchers, only part of which are active at any one point (so there is a constant number of active fds for each loop iteration, but which fds these are is random). The timeout is reset each time something is read because that reflects how most timeouts work (and puts extra pressure on the event loops).

In this benchmark, we use 10000 socket pairs (20000 sockets), of which 100 (1%) are active. This mirrors the activity of large servers with many connections, most of which are idle at any one point in time.

Source code for this benchmark is found as `eg/bench2` in the AnyEvent distribution. It uses the *AE* interface, which makes a real difference for the EV and Perl backends only.

Explanation of the columns

sockets is the number of sockets, and twice the number of "servers" (as each server has a read and write socket end).

create is the time it takes to create a socket pair (which is nontrivial) and two watchers: an I/O watcher and a timeout watcher.

request, the most important value, is the time it takes to handle a single "request", that is, reading the token from the pipe and forwarding it to another server. This includes deleting the old timeout and creating a new one that moves the timeout into the future.

Results

```
   name sockets create  request
     EV   20000  62.66     7.99
   Perl   20000  68.32    32.64
IOAsync   20000 174.06   101.15 epoll
IOAsync   20000 174.67   610.84 poll
  Event   20000 202.69   242.91
   Glib   20000 557.01  1689.52
    POE   20000 341.54 12086.32 uses POE::Loop::Event
```

Discussion

This benchmark *does* measure scalability and overall performance of the particular event loop.

EV is again fastest. Since it is using epoll on my system, the setup time is relatively high, though.

Perl surprisingly comes second. It is much faster than the C-based event loops Event and Glib.

IO::Async performs very well when using its epoll backend, and still quite good compared to Glib when using its pure perl backend.

Event suffers from high setup time as well (look at its code and you will understand why). Callback invocation also has a high overhead compared to the `$_->() for ..` -style loop that the Perl event loop uses. Event uses select or poll in basically all documented configurations.

Glib is hit hard by its quadratic behaviour w.r.t. many watchers. It clearly fails to perform with many filehandles or in busy servers.

POE is still completely out of the picture, taking over 1000 times as long as EV, and over 100 times as long as the Perl implementation, even though it uses a C-based event loop in this case.

Summary

- The pure perl implementation performs extremely well.
- Avoid Glib or POE in large projects where performance matters.

BENCHMARKING SMALL SERVERS

While event loops should scale (and select-based ones do not...) even to large servers, most programs we (or I :) actually write have only a few I/O watchers.

In this benchmark, I use the same benchmark program as in the large server case, but it uses only eight “servers”, of which three are active at any one time. This should reflect performance for a small server relatively well.

The columns are identical to the previous table.

Results

```
 name sockets create request
   EV      16  20.00    6.54
 Perl      16  25.75   12.62
Event      16  81.27   35.86
 Glib      16  32.63   15.48
  POE      16 261.87  276.28 uses POE::Loop::Event
```

Discussion

The benchmark tries to test the performance of a typical small server. While knowing how various event loops perform is interesting, keep in mind that their overhead in this case is usually not as important, due to the small absolute number of watchers (that is, you need efficiency and speed most when you have lots of watchers, not when you only have a few of them).

EV is again fastest.

Perl again comes second. It is noticeably faster than the C-based event loops Event and Glib, although the difference is too small to really matter.

POE also performs much better in this case, but is is still far behind the others.

Summary

- C-based event loops perform very well with small number of watchers, as the management overhead dominates.

THE IO::Lambda BENCHMARK

Recently I was told about the benchmark in the IO::Lambda manpage, which could be misinterpreted to make AnyEvent look bad. In fact, the benchmark simply compares IO::Lambda with POE, and IO::Lambda looks better (which shouldn't come as a surprise to anybody). As such, the benchmark is fine, and mostly shows that the AnyEvent backend from IO::Lambda isn't very optimal. But how would AnyEvent compare when used without the extra baggage? To explore this, I wrote the equivalent benchmark for AnyEvent.

The benchmark itself creates an echo-server, and then, for 500 times, connects to the echo server, sends a line, waits for the reply, and then creates the next connection. This is a rather bad benchmark, as it doesn't test the efficiency of the framework or much non-blocking I/O, but it is a benchmark nevertheless.

```
name                   runtime
Lambda/select          0.330 sec
   + optimized         0.122 sec
Lambda/AnyEvent        0.327 sec
   + optimized         0.138 sec
Raw sockets/select     0.077 sec
POE/select, components 0.662 sec
POE/select, raw sockets 0.226 sec
```

(continues on next page)

(continued from previous page)

```
POE/select, optimized   0.404 sec

AnyEvent/select/nb      0.085 sec
AnyEvent/EV/nb          0.068 sec
   +state machine       0.134 sec
```

The benchmark is also a bit unfair (my fault): the IO::Lambda/POE benchmarks actually make blocking connects and use 100% blocking I/O, defeating the purpose of an event-based solution. All of the newly written AnyEvent benchmarks use 100% non-blocking connects (using AnyEvent::Socket::tcp_connect and the asynchronous pure perl DNS resolver), so AnyEvent is at a disadvantage here, as non-blocking connects generally require a lot more bookkeeping and event handling than blocking connects (which involve a single syscall only).

The last AnyEvent benchmark additionally uses *AnyEvent::Handle*, which offers similar expressive power as POE and IO::Lambda, using conventional Perl syntax. This means that both the echo server and the client are 100% non-blocking, further placing it at a disadvantage.

As you can see, the AnyEvent + EV combination even beats the hand-optimised "raw sockets benchmark", while AnyEvent + its pure perl backend easily beats IO::Lambda and POE.

And even the 100% non-blocking version written using the high-level (and slow :) *AnyEvent::Handle* abstraction beats both POE and IO::Lambda higher level ("unoptimised") abstractions by a large margin, even though it does all of DNS, tcp-connect and socket I/O in a non-blocking way.

The two AnyEvent benchmarks programs can be found as `eg/ae0.pl` and `eg/ae2.pl` in the AnyEvent distribution, the remaining benchmarks are part of the IO::Lambda distribution and were used without any changes.

SIGNALS

AnyEvent currently installs handlers for these signals:

- SIGCHLD

  A handler for `SIGCHLD` is installed by AnyEvent's child watcher emulation for event loops that do not support them natively. Also, some event loops install a similar handler.

  Additionally, when AnyEvent is loaded and SIGCHLD is set to IGNORE, then AnyEvent will reset it to default, to avoid losing child exit statuses.

- SIGPIPE

  A no-op handler is installed for `SIGPIPE` when `$SIG{PIPE}` is `undef` when AnyEvent gets loaded.

  The rationale for this is that AnyEvent users usually do not really depend on SIGPIPE delivery (which is purely an optimisation for shell use, or badly-written programs), but `SIGPIPE` can cause spurious and rare program exits as a lot of people do not expect `SIGPIPE` when writing to some random socket.

The rationale for installing a no-op handler as opposed to ignoring it is that this way, the handler will be restored to defaults on exec.

Feel free to install your own handler, or reset it to defaults.

RECOMMENDED/OPTIONAL MODULES

One of AnyEvent's main goals is to be 100% Pure-Perl(tm): only perl (and its built-in modules) are required to use it.

That does not mean that AnyEvent won't take advantage of some additional modules if they are installed.

This section explains which additional modules will be used, and how they affect AnyEvent's operation.

- Async::Interrupt

  This slightly arcane module is used to implement fast signal handling: To my knowledge, there is no way to do completely race-free and quick signal handling in pure perl. To ensure that signals still get delivered, AnyEvent will start an interval timer to wake up perl (and catch the signals) with some delay (default is 10 seconds, look for `$AnyEvent::MAX_SIGNAL_LATENCY`).

  If this module is available, then it will be used to implement signal catching, which means that signals will not be delayed, and the event loop will not be interrupted regularly, which is more efficient (and good for battery life on laptops).

  This affects not just the pure-perl event loop, but also other event loops that have no signal handling on their own (e.g. Glib, Tk, Qt).

  Some event loops (POE, Event, Event::Lib) offer signal watchers natively, and either employ their own workarounds (POE) or use AnyEvent's workaround (using `$AnyEvent::MAX_SIGNAL_LATENCY`). Installing Async::Interrupt does nothing for those backends.

- *EV*

  This module isn't really "optional", as it is simply one of the backend event loops that AnyEvent can use. However, it is simply the best event loop available in terms of features, speed and stability: It supports the AnyEvent API optimally, implements all the watcher types natively, does automatic timer adjustments even when no monotonic clock is available, can take avdantage of advanced kernel interfaces such as `epoll` and `kqueue`, and is the fastest backend *by far*. You can even embed Glib/Gtk2 in it (or vice versa, see EV::Glib and Glib::EV).

  If you only use backends that rely on another event loop (e.g. `Tk`), then this module will do nothing for you.

- *Guard*

  The guard module, when used, will be used to implement `AnyEvent::Util::guard`. This speeds up guards considerably (and uses a lot less memory), but otherwise doesn't affect guard operation much. It is purely used for performance.

- *JSON* and *JSON::XS*

One of these modules is required when you want to read or write JSON data via *AnyEvent::Handle*. *JSON* is also written in pure-perl, but can take advantage of the ultra-high-speed *JSON::XS* module when it is installed.

- *Net::SSLeay*

  Implementing TLS/SSL in Perl is certainly interesting, but not very worthwhile: If this module is installed, then *AnyEvent::Handle* (with the help of *AnyEvent::TLS*), gains the ability to do TLS/SSL.

- *Time::HiRes*

  This module is part of perl since release 5.008. It will be used when the chosen event library does not come with a timing source of its own. The pure-perl event loop (*AnyEvent::Loop*) will additionally load it to try to use a monotonic clock for timing stability.

- AnyEvent::AIO (and IO::AIO)

  The default implementation of *AnyEvent::IO* is to do I/O synchronously, stopping programs while they access the disk, which is fine for a lot of programs.

  Installing AnyEvent::AIO (and its IO::AIO dependency) makes it switch to a true asynchronous implementation, so event processing can continue even while waiting for disk I/O.

FORK

Most event libraries are not fork-safe. The ones who are usually are because they rely on inefficient but fork-safe `select` or `poll` calls - higher performance APIs such as BSD's kqueue or the dreaded Linux epoll are usually badly thought-out hacks that are incompatible with fork in one way or another. Only *EV* is fully fork-aware and ensures that you continue event-processing in both parent and child (or both, if you know what you are doing).

This means that, in general, you cannot fork and do event processing in the child if the event library was initialised before the fork (which usually happens when the first AnyEvent watcher is created, or the library is loaded).

If you have to fork, you must either do so *before* creating your first watcher OR you must not use AnyEvent at all in the child OR you must do something completely out of the scope of AnyEvent (see below).

The problem of doing event processing in the parent *and* the child is much more complicated: even for backends that *are* fork-aware or fork-safe, their behaviour is not usually what you want: fork clones all watchers, that means all timers, I/O watchers etc. are active in both parent and child, which is almost never what you want. Using `exec` to start worker children from some kind of manage prrocess is usually preferred, because it is much easier and cleaner, at the expense of having to have another binary.

In addition to logical problems with fork, there are also implementation problems. For example, on POSIX systems, you cannot fork at all in Perl code if a thread (I am talking of pthreads here) was ever created in the process, and this is just the tip of the iceberg. In general, using fork from Perl is difficult, and attempting to use fork without an exec to implement some kind of parallel processing is almost certainly doomed.

To safely fork and exec, you should use a module such as Proc::FastSpawn that let's you safely fork and exec new processes.

If you want to do multiprocessing using processes, you can look at the AnyEvent::Fork module (and some related modules such as AnyEvent::Fork::RPC, AnyEvent::Fork::Pool and AnyEvent::Fork::Remote). This module allows you to safely create subprocesses without any limitations - you can use X11 toolkits or AnyEvent in the children created by AnyEvent::Fork safely and without any special precautions.

SECURITY CONSIDERATIONS

AnyEvent can be forced to load any event model via $ENV{PERL_ANYEVENT_MODEL}. While this cannot (to my knowledge) be used to execute arbitrary code or directly gain access, it can easily be used to make the program hang or malfunction in subtle ways, as AnyEvent watchers will not be active when the program uses a different event model than specified in the variable.

You can make AnyEvent completely ignore this variable by deleting it before the first watcher gets created, e.g. with a `BEGIN` block:

```
BEGIN { delete $ENV{PERL_ANYEVENT_MODEL} }

use AnyEvent;
```

Similar considerations apply to $ENV{PERL_ANYEVENT_VERBOSE}, as that can be used to probe what backend is used and gain other information (which is probably even less useful to an attacker than PERL_ANYEVENT_MODEL), and $ENV{PERL_ANYEVENT_STRICT}.

Note that AnyEvent will remove *all* environment variables starting with `PERL_ANYEVENT_` from `%ENV` when it is loaded while taint mode is enabled.

BUGS

Perl 5.8 has numerous memleaks that sometimes hit this module and are hard to work around. If you suffer from memleaks, first upgrade to Perl 5.10 and check wether the leaks still show up. (Perl 5.10.0 has other annoying memleaks, such as leaking on `map` and `grep` but it is usually not as pronounced).

SEE ALSO

Tutorial/Introduction: AnyEvent::Intro.

FAQ: AnyEvent::FAQ.

Utility functions: *AnyEvent::Util* (misc. grab-bag), *AnyEvent::Log* (simply logging).

Development/Debugging: *AnyEvent::Strict* (stricter checking), *AnyEvent::Debug* (interactive shell, watcher tracing).

Supported event modules: *AnyEvent::Loop*, *EV*, EV::Glib, Glib::EV, *Event*, Glib::Event, Glib, Tk, Event::Lib, Qt, POE, FLTK, Cocoa::EventLoop, UV.

Implementations: *AnyEvent::Impl::EV*, *AnyEvent::Impl::Event*, *AnyEvent::Impl::Glib*, *AnyEvent::Impl::Tk*, *AnyEvent::Impl::Perl*, *AnyEvent::Impl::EventLib*, *AnyEvent::Impl::Qt*,

AnyEvent::Impl::POE, *AnyEvent::Impl::IOAsync*, *AnyEvent::Impl::Irssi*, *AnyEvent::Impl::FLTK*, *AnyEvent::Impl::Cocoa*, *AnyEvent::Impl::UV*.

Non-blocking handles, pipes, stream sockets, TCP clients and servers: *AnyEvent::Handle*, *AnyEvent::Socket*, *AnyEvent::TLS*.

Asynchronous File I/O: *AnyEvent::IO*.

Asynchronous DNS: *AnyEvent::DNS*.

Thread support: *Coro*, *Coro::AnyEvent*, *Coro::EV*, *Coro::Event*.

Nontrivial usage examples: AnyEvent::GPSD, AnyEvent::IRC, AnyEvent::HTTP.

AUTHOR

```
Marc Lehmann <schmorp@schmorp.de>
http://anyevent.schmorp.de
```

Modules

- `AnyEvent::DNS` - fully asynchronous DNS resolution
- `AnyEvent::Debug` - debugging utilities for AnyEvent
- `AnyEvent::Handle` - non-blocking I/O on streaming handles via AnyEvent
- `AnyEvent::IO` - the DBI of asynchronous I/O implementations
- `AnyEvent::Impl`
- `AnyEvent::Log` - simple logging “framework”
- `AnyEvent::Loop` - AnyEvent’s Pure-Perl event loop
- `AnyEvent::Socket` - useful IPv4 and IPv6 stuff.
- `AnyEvent::Strict` - force strict mode on for the whole process
- `AnyEvent::TLS` - SSLv2/SSLv3/TLSv1 contexts for use in AnyEvent::Handle
- `AnyEvent::Util` - various utility functions.

AnyEvent::DNS

fully asynchronous DNS resolution

SYNOPSIS

```
use AnyEvent::DNS;

my $cv = AnyEvent->condvar;
AnyEvent::DNS::a "www.google.de", $cv;
# ... later
my @addrs = $cv->recv;
```

DESCRIPTION

This module offers both a number of DNS convenience functions as well as a fully asynchronous and high-performance pure-perl stub resolver.

The stub resolver supports DNS over IPv4 and IPv6, UDP and TCP, optional EDNS0 support for up to 4kiB datagrams and automatically falls back to virtual circuit mode for large responses.

CONVENIENCE FUNCTIONS

- AnyEvent::DNS::a $domain, $cb->(@addrs)

  Tries to resolve the given domain to IPv4 address(es).

- AnyEvent::DNS::aaaa $domain, $cb->(@addrs)

  Tries to resolve the given domain to IPv6 address(es).

- AnyEvent::DNS::mx $domain, $cb->(@hostnames)

  Tries to resolve the given domain into a sorted (lower preference value first) list of domain names.

- AnyEvent::DNS::ns $domain, $cb->(@hostnames)

  Tries to resolve the given domain name into a list of name servers.

- AnyEvent::DNS::txt $domain, $cb->(@hostnames)

  Tries to resolve the given domain name into a list of text records. Only the first text string per record will be returned. If you want all strings, you need to call the resolver manually:

```
resolver->resolve ($domain => "txt", sub {
   for my $record (@_) {
      my (undef, undef, undef, @txt) = @$record;
      # strings now in @txt
   }
});
```

- AnyEvent::DNS::srv $service, $proto, $domain, $cb->(@srv_rr)

  Tries to resolve the given service, protocol and domain name into a list of service records.

  Each `$srv_rr` is an array reference with the following contents: `[$priority, $weight, $transport, $target]`.

  They will be sorted with lowest priority first, then randomly distributed by weight as per RFC 2782.

  Example:

```
AnyEvent::DNS::srv "sip", "udp", "schmorp.de", sub { ...
# @_ = ( [10, 10, 5060, "sip1.schmorp.de" ] )
```

- AnyEvent::DNS::any $domain, $cb->(@rrs)

Tries to resolve the given domain and passes all resource records found to the callback. Note that this uses a DNS ANY query, which, as of RFC 8482, are officially deprecated.

- AnyEvent::DNS::ptr $domain, $cb->(@hostnames)

  Tries to make a PTR lookup on the given domain. See `reverse_lookup` and `reverse_verify` if you want to resolve an IP address to a hostname instead.

- AnyEvent::DNS::reverse_lookup $ipv4_or_6, $cb->(@hostnames)

  Tries to reverse-resolve the given IPv4 or IPv6 address (in textual form) into its hostname(s). Handles V4MAPPED and V4COMPAT IPv6 addresses transparently.

- AnyEvent::DNS::reverse_verify $ipv4_or_6, $cb->(@hostnames)

  The same as `reverse_lookup`, but does forward-lookups to verify that the resolved hostnames indeed point to the address, which makes spoofing harder.

  If you want to resolve an address into a hostname, this is the preferred method: The DNS records could still change, but at least this function verified that the hostname, at one point in the past, pointed at the IP address you originally resolved.

  Example:

```
AnyEvent::DNS::reverse_verify "2001:500:2f::f", sub { print␣
↪shift };
# => f.root-servers.net
```

LOW-LEVEL DNS EN-/DECODING FUNCTIONS

- $AnyEvent::DNS::EDNS0

  This variable decides whether dns_pack automatically enables EDNS0 support. By default, this is disabled (`0`), unless overridden by `$ENV{PERL_ANYEVENT_EDNS0}`, but when set to 1, AnyEvent::DNS will use EDNS0 in all requests.

- $pkt = AnyEvent::DNS::dns_pack $dns

  Packs a perl data structure into a DNS packet. Reading RFC 1035 is strongly recommended, then everything will be totally clear. Or maybe not.

  Resource records are not yet encodable.

  Examples:

```
# very simple request, using lots of default values:
{ rd => 1, qd => [ [ "host.domain", "a"] ] }

# more complex example, showing how flags etc. are named:

{
   id => 10000,
   op => "query",
   rc => "nxdomain",

   # flags
```

(continues on next page)

(continued from previous page)

```
   qr => 1,
   aa => 0,
   tc => 0,
   rd => 0,
   ra => 0,
   ad => 0,
   cd => 0,

   qd => [@rr], # query section
   an => [@rr], # answer section
   ns => [@rr], # authority section
   ar => [@rr], # additional records section
}
```

- $dns = AnyEvent::DNS::dns_unpack $pkt

  Unpacks a DNS packet into a perl data structure.

  Examples:

```
# an unsuccessful reply
{
  'qd' => [
            [ 'ruth.plan9.de.mach.uni-karlsruhe.de', '*', 'in' ]
          ],
  'rc' => 'nxdomain',
  'ar' => [],
  'ns' => [
            [
              'uni-karlsruhe.de',
              'soa',
              'in',
              600,
              'netserv.rz.uni-karlsruhe.de',
              'hostmaster.rz.uni-karlsruhe.de',
              2008052201, 10800, 1800, 2592000, 86400
            ]
          ],
  'tc' => '',
  'ra' => 1,
  'qr' => 1,
  'id' => 45915,
  'aa' => '',
  'an' => [],
  'rd' => 1,
  'op' => 'query',
  '__' => '<original dns packet>',
}

# a successful reply
```

(continues on next page)

(continued from previous page)

```
{
  'qd' => [ [ 'www.google.de', 'a', 'in' ] ],
  'rc' => 0,
  'ar' => [
            [ 'a.l.google.com', 'a', 'in', 3600, '209.85.139.9'
↪],
            [ 'b.l.google.com', 'a', 'in', 3600, '64.233.179.9'
↪],
            [ 'c.l.google.com', 'a', 'in', 3600, '64.233.161.9'
↪],
          ],
  'ns' => [
            [ 'l.google.com', 'ns', 'in', 3600, 'a.l.google.com
↪' ],
            [ 'l.google.com', 'ns', 'in', 3600, 'b.l.google.com
↪' ],
          ],
  'tc' => '',
  'ra' => 1,
  'qr' => 1,
  'id' => 64265,
  'aa' => '',
  'an' => [
            [ 'www.google.de', 'cname', 'in', 3600, 'www.google.
↪com' ],
            [ 'www.google.com', 'cname', 'in', 3600, 'www.l.
↪google.com' ],
            [ 'www.l.google.com', 'a', 'in', 3600, '66.249.93.
↪104' ],
            [ 'www.l.google.com', 'a', 'in', 3600, '66.249.93.
↪147' ],
          ],
  'rd' => 1,
  'op' => 0,
  '__' => '<original dns packet>',
}
```

Extending DNS Encoder and Decoder

This section describes an *experimental* method to extend the DNS encoder and decoder with new opcode, rcode, class and type strings, as well as resource record decoders.

Since this is experimental, it can change, as anything can change, but this interface is expe ctedc to be relatively stable and was stable during the whole existance of `AnyEvent::DNS` so far.

Note that, since changing the decoder or encoder might break existing code, you should either be sure to control for this, or only temporarily change these values, e.g. like so:

```
my $decoded = do {
   local $AnyEvent::DNS::opcode_str{7} = "yxrrset";
   AnyEvent::DNS::dns_unpack $mypkt
};
```

- Two hashes that map lowercase opcode strings to numerical id's (For the encoder), or vice versa (for the decoder). Example: add a new opcode string `notzone`.

  ```
  $AnyEvent::DNS::opcode_id{notzone} = 10;
  $AnyEvent::DNS::opcode_str{10} = 'notzone';
  ```

- Same as above, for for rcode values.
- Same as above, but for resource record class names/values.
- Same as above, but for resource record type names/values.
- This hash maps resource record type values to code references. When decoding, they are called with `$_` set to the undecoded data portion and `$ofs` being the current byte offset. of the record. You should have a look at the existing implementations to understand how it works in detail, but here are two examples:

  Decode an A record. A records are simply four bytes with one byte per address component, so the decoder simply unpacks them and joins them with dots in between:

  ```
  $AnyEvent::DNS::dec_rr{1} = sub { join ".", unpack "C4", $_ };
  ```

  Decode a CNAME record, which contains a potentially compressed domain name.

  ```
  package AnyEvent::DNS; # for %dec_rr, $ofsd and &_dec_name
  $dec_rr{5} = sub { local $ofs = $ofs - length; _dec_name };
  ```

THE AnyEvent::DNS RESOLVER CLASS

This is the class which does the actual protocol work.

- AnyEvent::DNS::resolver

  This function creates and returns a resolver that is ready to use and should mimic the default resolver for your system as good as possible. It is used by AnyEvent itself as well.

  It only ever creates one resolver and returns this one on subsequent calls - see `$AnyEvent::DNS::RESOLVER`, below, for details.

  Unless you have special needs, prefer this function over creating your own resolver object.

  The resolver is created with the following parameters:

  ```
  untaint          enabled
  max_outstanding  $ENV{PERL_ANYEVENT_MAX_OUTSTANDING_DNS}␣
   ↪(default 10)
  ```

`os_config` will be used for OS-specific configuration, unless `$ENV{PERL_ANYEVENT_RESOLV_CONF}` is specified, in which case that file gets parsed.

- $AnyEvent::DNS::RESOLVER

  This variable stores the default resolver returned by `AnyEvent::DNS::resolver`, or `undef` when the default resolver hasn't been instantiated yet.

  One can provide a custom resolver (e.g. one with caching functionality) by storing it in this variable, causing all subsequent resolves done via `AnyEvent::DNS::resolver` to be done via the custom one.

- $resolver = new AnyEvent::DNS key => value...

  Creates and returns a new resolver.

  The following options are supported:

  - server => [...]

    A list of server addresses (default: `v127.0.0.1` or `::1`) in network format (i.e. as returned by `AnyEvent::Socket::parse_address` - both IPv4 and IPv6 are supported).

  - timeout => [...]

    A list of timeouts to use (also determines the number of retries). To make three retries with individual time-outs of 2, 5 and 5 seconds, use `[2, 5, 5]`, which is also the default.

  - search => [...]

    The default search list of suffixes to append to a domain name (default: none).

  - ndots => $integer

    The number of dots (default: `1`) that a name must have so that the resolver tries to resolve the name without any suffixes first.

  - max_outstanding => $integer

    Most name servers do not handle many parallel requests very well. This option limits the number of outstanding requests to `$integer` (default: `10`), that means if you request more than this many requests, then the additional requests will be queued until some other requests have been resolved.

  - reuse => $seconds

    The number of seconds (default: `300`) that a query id cannot be re-used after a timeout. If there was no time-out then query ids can be reused immediately.

  - untaint => $boolean

    When true, then the resolver will automatically untaint results, and might also ignore certain environment variables.

- $resolver->parse_resolv_conf ($string)

  Parses the given string as if it were a `resolv.conf` file. The following directives are supported (but not necessarily implemented).

#- and ;-style comments, `nameserver`, `domain`, `search`, `sortlist`, `options` (`timeout`, `attempts`, `ndots`).

Everything else is silently ignored.

- $resolver->os_config

  Tries so load and parse `/etc/resolv.conf` on portable operating systems. Tries various egregious hacks on windows to force the DNS servers and searchlist out of the system.

  This method must be called at most once before trying to resolve anything.

- $resolver->timeout ($timeout, ...)

  Sets the timeout values. See the `timeout` constructor argument (and note that this method expects the timeout values themselves, not an array-reference).

- $resolver->max_outstanding ($nrequests)

  Sets the maximum number of outstanding requests to `$nrequests`. See the `max_outstanding` constructor argument.

- $resolver->request ($req, $cb->($res))

  This is the main low-level workhorse for sending DNS requests.

  This function sends a single request (a hash-ref formated as specified for `dns_pack`) to the configured nameservers in turn until it gets a response. It handles timeouts, retries and automatically falls back to virtual circuit mode (TCP) when it receives a truncated reply. It does not handle anything else, such as the domain searchlist or relative names - use `->resolve` for that.

  Calls the callback with the decoded response packet if a reply was received, or no arguments in case none of the servers answered.

- $resolver->resolve ($qname, $qtype, %options, $cb->(@rr))

  Queries the DNS for the given domain name `$qname` of type `$qtype`.

  A `$qtype` is either a numerical query type (e.g. 1 for A records) or a lowercase name (you have to look at the source to see which aliases are supported, but all types from RFC 1035, `aaaa`, `srv`, `spf` and a few more are known to this module). A `$qtype` of "*" is supported and means "any" record type.

  The callback will be invoked with a list of matching result records or none on any error or if the name could not be found.

  CNAME chains (although illegal) are followed up to a length of 10.

  The callback will be invoked with arraryefs of the form `[$name, $type, $class, $ttl, @data]`, where `$name` is the domain name, `$type` a type string or number, `$class` a class name, `$ttl` is the remaining time-to-live and `@data` is resource-record-dependent data, in seconds. For `a` records, this will be the textual IPv4 addresses, for `ns` or `cname` records this will be a domain name, for `txt` records these are all the strings and so on.

  All types mentioned in RFC 1035, `aaaa`, `srv`, `naptr` and `spf` are decoded. All resource records not known to this module will have the raw `rdata` field as fifth array element.

Note that this resolver is just a stub resolver: it requires a name server supporting recursive queries, will not do any recursive queries itself and is not secure when used against an untrusted name server.

The following options are supported:

– search => [$suffix...]

  Use the given search list (which might be empty), by appending each one in turn to the `$qname`. If this option is missing then the configured `ndots` and `search` values define its value (depending on `ndots`, the empty suffix will be prepended or appended to that `search` value). If the `$qname` ends in a dot, then the searchlist will be ignored.

– accept => [$type...]

  Lists the acceptable result types: only result types in this set will be accepted and returned. The default includes the `$qtype` and nothing else. If this list includes `cname`, then CNAME-chains will not be followed (because you asked for the CNAME record).

– class => "class"

  Specify the query class ("in" for internet, "ch" for chaosnet and "hs" for hesiod are the only ones making sense). The default is "in", of course.

Examples:

```
# full example, you can paste this into perl:
use Data::Dumper;
use AnyEvent::DNS;
AnyEvent::DNS::resolver->resolve (
   "google.com", "*", my $cv = AnyEvent->condvar);
warn Dumper [$cv->recv];

# shortened result:
# [
#   [ 'google.com', 'soa', 'in', 3600, 'ns1.google.com', 'dns-
 ↪admin.google.com',
#     2008052701, 7200, 1800, 1209600, 300 ],
#   [
#     'google.com', 'txt', 'in', 3600,
#     'v=spf1 include:_netblocks.google.com ~all'
#   ],
#   [ 'google.com', 'a', 'in', 3600, '64.233.187.99' ],
#   [ 'google.com', 'mx', 'in', 3600, 10, 'smtp2.google.com' ],
#   [ 'google.com', 'ns', 'in', 3600, 'ns2.google.com' ],
# ]

# resolve a records:
$res->resolve ("ruth.plan9.de", "a", sub { warn Dumper [@_] });

# result:
# [
#   [ 'ruth.schmorp.de', 'a', 'in', 86400, '129.13.162.95' ]
```

(continues on next page)

(continued from previous page)

```
# ]

# resolve any records, but return only a and aaaa records:
$res->resolve ("test1.laendle", "*",
   accept => ["a", "aaaa"],
   sub {
      warn Dumper [@_];
   }
);

# result:
# [
#   [ 'test1.laendle', 'a', 'in', 86400, '10.0.0.255' ],
#   [ 'test1.laendle', 'aaaa', 'in', 60,
 ↪'3ffe:1900:4545:0002:0240:0000:0000:f7e1' ]
# ]
```

- $resolver->wait_for_slot ($cb->($resolver))

  Wait until a free request slot is available and call the callback with the resolver object.

  A request slot is used each time a request is actually sent to the nameservers: There are never more than `max_outstanding` of them.

  Although you can submit more requests (they will simply be queued until a request slot becomes available), sometimes, usually for rate-limiting purposes, it is useful to instead wait for a slot before generating the request (or simply to know when the request load is low enough so one can submit requests again).

  This is what this method does: The callback will be called when submitting a DNS request will not result in that request being queued. The callback may or may not generate any requests in response.

  Note that the callback will only be invoked when the request queue is empty, so this does not play well if somebody else keeps the request queue full at all times.

AUTHOR

```
Marc Lehmann <schmorp@schmorp.de>
http://anyevent.schmorp.de
```

AnyEvent::Debug

debugging utilities for AnyEvent

SYNOPSIS

```
use AnyEvent::Debug;

# create an interactive shell into the program
my $shell = AnyEvent::Debug::shell "unix/", "/home/schmorp/myshell";
# then on the shell: "socat readline /home/schmorp/myshell"
```

DESCRIPTION

This module provides functionality hopefully useful for debugging.

At the moment, "only" an interactive shell is implemented. This shell allows you to interactively "telnet into" your program and execute Perl code, e.g. to look at global variables.

FUNCTIONS

- $shell = AnyEvent::Debug::shell $host, $service

  This function binds on the given host and service port and returns a shell object, which determines the lifetime of the shell. Any number of connections are accepted on the port, and they will give you a very primitive shell that simply executes every line you enter.

  All commands will be executed "blockingly" with the socket `selected` for output. For a less "blocking" interface see *Coro::Debug*.

  The commands will be executed in the `AnyEvent::Debug::shell` package, which currently has "help" and a few other commands, and can be freely modified by all shells. Code is evaluated under `use strict 'subs'`.

  Every shell has a logging context (`$LOGGER`) that is attached to `$AnyEvent::Log::COLLECT`), which is especially useful to gether debug and trace messages.

  As a general programming guide, consider the beneficial aspects of using more global (`our`) variables than local ones (`my`) in package scope: Earlier all my modules tended to hide internal variables inside `my` variables, so users couldn't accidentally access them. Having interactive access to your programs changed that: having internal variables still in the global scope means you can debug them easier.

  As no authentication is done, in most cases it is best not to use a TCP port, but a unix domain socket, whcih can be put wherever you can access it, but not others:

  ```
  our $SHELL = AnyEvent::Debug::shell "unix/", "/home/schmorp/
  ↪shell";
  ```

  Then you can use a tool to connect to the shell, such as the ever versatile `socat`, which in addition can give you readline support:

  ```
  socat readline /home/schmorp/shell
  # or:
  cd /home/schmorp; socat readline unix:shell
  ```

Socat can even give you a persistent history:

```
socat readline,history=.anyevent-history unix:shell
```

Binding on `127.0.0.1` (or `::1`) might be a less secure but sitll not totally insecure (on single-user machines) alternative to let you use other tools, such as telnet:

```
our $SHELL = AnyEvent::Debug::shell "127.1", "1357";
```

And then:

```
telnet localhost 1357
```

- AnyEvent::Debug::wrap [$level]

  Sets the instrumenting/wrapping level of all watchers that are being created after this call. If no `$level` has been specified, then it toggles between `0` and `1`.

  The default wrap level is `0`, or whatever `$ENV{PERL_ANYEVENT_DEBUG_WRAP}` specifies.

  A level of `0` disables wrapping, i.e. AnyEvent works normally, and in its most efficient mode.

  A level of `1` or higher enables wrapping, which replaces all watchers by AnyEvent::Debug::Wrapped objects, stores the location where a watcher was created and wraps the callback to log all invocations at "trace" loglevel if tracing is enabled fore the watcher. The initial state of tracing when creating a watcher is taken from the global variable `$AnyEvent::Debug::TRACE`. The default value of that variable is `1`, but it can make sense to set it to `0` and then do `local $AnyEvent::Debug::TRACE = 1` in a block where you create "interesting" watchers. Tracing can also be enabled and disabled later by calling the watcher's `trace` method.

  The wrapper will also count how many times the callback was invoked and will record up to ten runtime errors with corresponding backtraces. It will also log runtime errors at "error" loglevel.

  To see the trace messages, you can invoke your program with `PERL_ANYEVENT_VERBOSE=9`, or you can use AnyEvent::Log to divert the trace messages in any way you like (the EXAMPLES section in *AnyEvent::Log* has some examples).

  A level of 2 does everything that level 1 does, but also stores a full backtrace of the location the watcher was created, which slows down watcher creation considerably.

  Every wrapped watcher will be linked into `%AnyEvent::Debug::Wrapped`, with its address as key. The `wl` command in the debug shell can be used to list watchers.

  Instrumenting can increase the size of each watcher multiple times, and, especially when backtraces are involved, also slows down watcher creation a lot.

  Also, enabling and disabling instrumentation will not recover the full performance that you had before wrapping (the AE::xxx functions will stay slower, for example).

  If you are developing your program, also consider using AnyEvent::Strict to check for common mistakes.

- AnyEvent::Debug::path2mod $path

  Tries to replace a path (e.g. the file name returned by caller) by a module name. Returns the path unchanged if it fails.

  Example:

```
print AnyEvent::Debug::path2mod "/usr/lib/perl5/AnyEvent/Debug.
 ↪pm";
# might print "AnyEvent::Debug"
```

- AnyEvent::Debug::cb2str $cb

  Using various gambits, tries to convert a callback (e.g. a code reference) into a more useful string.

  Very useful if you debug a program and have some callback, but you want to know where in the program the callback is actually defined.

- AnyEvent::Debug::backtrace [$skip]

  Creates a backtrace (actually an AnyEvent::Debug::Backtrace object that you can stringify), not unlike the Carp module would. Unlike the Carp module it resolves some references (such as callbacks) to more user-friendly strings, has a more succinct output format and most importantly: doesn't leak memory like hell.

  The reason it creates an object is to save time, as formatting can be done at a later time. Still, creating a backtrace is a relatively slow operation.

THE AnyEvent::Debug::Wrapped CLASS

All watchers created while the wrap level is non-zero will be wrapped inside an AnyEvent::Debug::Wrapped object. The address of the wrapped watcher will become its ID - every watcher will be stored in `$AnyEvent::Debug::Wrapped{$id}`.

These wrapper objects can be stringified and have some methods defined on them.

For debugging, of course, it can be helpful to look into these objects, which is why this is documented here, but this might change at any time in future versions.

Each object is a relatively standard hash with the following members:

```
type   => name of the method used ot create the watcher (e.g. C<io>,␣
 ↪C<timer>).
w      => the actual watcher
rfile  => reference to the filename of the file the watcher was␣
 ↪created in
line   => line number where it was created
sub    => function name (or a special string) which created the␣
 ↪watcher
cur    => if created inside another watcher callback, this is the␣
 ↪string rep of the other watcher
now    => the timestamp (AE::now) when the watcher was created
arg    => the arguments used to create the watcher (sans C<cb>)
cb     => the original callback used to create the watcher
called => the number of times the callback was called
```

Each object supports the following mehtods (warning: these are only available on wrapped watchers, so are best for interactive use via the debug shell).

- $w->id

  Returns the numerical id of the watcher, as used in the debug shell.

- $w->verbose

  Returns a multiline textual description of the watcher, including the first ten exceptions caught while executing the callback.

- $w->trace ($on)

  Enables ($on is true) or disables ($on is false) tracing on this watcher.

  To get tracing messages, both the global logging settings must have trace messages enabled for the context `AnyEvent::Debug` and tracing must be enabled for the wrapped watcher.

  To enable trace messages globally, the simplest way is to start the program with `PERL_ANYEVENT_VERBOSE=9` in the environment.

  Tracing for each individual watcher is enabled by default (unless `$AnyEvent::Debug::TRACE` has been set to false).

AUTHOR

```
Marc Lehmann <schmorp@schmorp.de>
http://anyevent.schmorp.de
```

AnyEvent::Handle

non-blocking I/O on streaming handles via AnyEvent

SYNOPSIS

```
use AnyEvent;
use AnyEvent::Handle;

my $cv = AnyEvent->condvar;

my $hdl; $hdl = new AnyEvent::Handle
   fh => \*STDIN,
   on_error => sub {
      my ($hdl, $fatal, $msg) = @_;
      AE::log error => $msg;
      $hdl->destroy;
      $cv->send;
   };

# send some request line
$hdl->push_write ("getinfo\015\012");
```

(continues on next page)

(continued from previous page)

```
# read the response line
$hdl->push_read (line => sub {
   my ($hdl, $line) = @_;
   say "got line <$line>";
   $cv->send;
});

$cv->recv;
```

DESCRIPTION

This is a helper module to make it easier to do event-based I/O on stream-based filehandles (sockets, pipes, and other stream things). Specifically, it doesn't work as expected on files, packet-based sockets or similar things.

The AnyEvent::Intro tutorial contains some well-documented AnyEvent::Handle examples.

In the following, where the documentation refers to "bytes", it means characters. As sysread and syswrite are used for all I/O, their treatment of characters applies to this module as well.

At the very minimum, you should specify `fh` or `connect`, and the `on_error` callback.

All callbacks will be invoked with the handle object as their first argument.

METHODS

- $handle = **new** AnyEvent::Handle fh => $filehandle, key => value...

  The constructor supports these arguments (all as `key => value` pairs).

  - fh => $filehandle [`fh` or `connect` MANDATORY]

    The filehandle this *AnyEvent::Handle* object will operate on. NOTE: The filehandle will be set to non-blocking mode (using `AnyEvent::fh_unblock`) by the constructor and needs to stay in that mode.

  - connect => [$host, $service] [`fh` or `connect` MANDATORY]

    Try to connect to the specified host and service (port), using `AnyEvent::Socket::tcp_connect`. The `$host` additionally becomes the default `peername`.

    You have to specify either this parameter, or `fh`, above.

    It is possible to push requests on the read and write queues, and modify properties of the stream, even while AnyEvent::Handle is connecting.

    When this parameter is specified, then the `on_prepare`, `on_connect_error` and `on_connect` callbacks will be called under the appropriate circumstances:

    * on_prepare => $cb->($handle)

This (rarely used) callback is called before a new connection is attempted, but after the file handle has been created (you can access that file handle via `$handle->{fh}`). It could be used to prepare the file handle with parameters required for the actual connect (as opposed to settings that can be changed when the connection is already established).

The return value of this callback should be the connect timeout value in seconds (or `0`, or `undef`, or the empty list, to indicate that the default timeout is to be used).

* on_connect => $cb->($handle, $host, $port, $retry->())

  This callback is called when a connection has been successfully established.

  The peer's numeric host and port (the socket peername) are passed as parameters, together with a retry callback. At the time it is called the read and write queues, EOF status, TLS status and similar properties of the handle will have been reset.

  If, for some reason, the handle is not acceptable, calling `$retry` will continue with the next connection target (in case of multi-homed hosts or SRV records there can be multiple connection endpoints). The `$retry` callback can be invoked after the connect callback returns, i.e. one can start a handshake and then decide to retry with the next host if the handshake fails.

  In most cases, you should ignore the `$retry` parameter.

* on_connect_error => $cb->($handle, $message)

  This callback is called when the connection could not be established. `$!` will contain the relevant error code, and `$message` a message describing it (usually the same as `"$!"`).

  If this callback isn't specified, then `on_error` will be called with a fatal error instead.

– on_error => $cb->($handle, $fatal, $message)

  This is the error callback, which is called when, well, some error occured, such as not being able to resolve the hostname, failure to connect, or a read error.

  Some errors are fatal (which is indicated by `$fatal` being true). On fatal errors the handle object will be destroyed (by a call to `-> destroy`) after invoking the error callback (which means you are free to examine the handle object). Examples of fatal errors are an EOF condition with active (but unsatisfiable) read watchers (`EPIPE`) or I/O errors. In cases where the other side can close the connection at will, it is often easiest to not report `EPIPE` errors in this callback.

  AnyEvent::Handle tries to find an appropriate error code for you to check against, but in some cases (TLS errors), this does not work well.

  If you report the error to the user, it is recommended to always output the `$message` argument in human-readable error messages (you don't need to report `"$!"` if you report `$message`).

  If you want to react programmatically to the error, then looking at `$!` and comparing it against some of the documented `Errno` values is usually better

than looking at the `$message`.

Non-fatal errors can be retried by returning, but it is recommended to simply ignore this parameter and instead abondon the handle object when this callback is invoked. Examples of non-fatal errors are timeouts `ETIMEDOUT`) or badly-formatted data (`EBADMSG`).

On entry to the callback, the value of `$!` contains the operating system error code (or `ENOSPC`, `EPIPE`, `ETIMEDOUT`, `EBADMSG` or `EPROTO`).

While not mandatory, it is *highly* recommended to set this callback, as you will not be notified of errors otherwise. The default just calls `croak`.

– on_read => $cb->($handle)

This sets the default read callback, which is called when data arrives and no read request is in the queue (unlike read queue callbacks, this callback will only be called when at least one octet of data is in the read buffer).

To access (and remove data from) the read buffer, use the `->rbuf` method or access the `$handle->{rbuf}` member directly. Note that you must not enlarge or modify the read buffer, you can only remove data at the beginning from it.

You can also call `->push_read (...)` or any other function that modifies the read queue. Or do both. Or ...

When an EOF condition is detected, AnyEvent::Handle will first try to feed all the remaining data to the queued callbacks and `on_read` before calling the `on_eof` callback. If no progress can be made, then a fatal error will be raised (with `$!` set to `EPIPE`).

Note that, unlike requests in the read queue, an `on_read` callback doesn't mean you *require* some data: if there is an EOF and there are outstanding read requests then an error will be flagged. With an `on_read` callback, the `on_eof` callback will be invoked.

– on_eof => $cb->($handle)

Set the callback to be called when an end-of-file condition is detected, i.e. in the case of a socket, when the other side has closed the connection cleanly, and there are no outstanding read requests in the queue (if there are read requests, then an EOF counts as an unexpected connection close and will be flagged as an error).

For sockets, this just means that the other side has stopped sending data, you can still try to write data, and, in fact, one can return from the EOF callback and continue writing data, as only the read part has been shut down.

If an EOF condition has been detected but no `on_eof` callback has been set, then a fatal error will be raised with `$!` set to <0>.

– on_drain => $cb->($handle)

This sets the callback that is called once when the write buffer becomes empty (and immediately when the handle object is created).

To append to the write buffer, use the `->push_write` method.

This callback is useful when you don't want to put all of your write data into the queue at once, for example, when you want to write the contents of some file to the socket you might not want to read the whole file into memory and push it into the queue, but instead only read more data from the file when the write queue becomes empty.

– timeout => $fractional_seconds

– rtimeout => $fractional_seconds

– wtimeout => $fractional_seconds

If non-zero, then these enables an "inactivity" timeout: whenever this many seconds pass without a successful read or write on the underlying file handle (or a call to `timeout_reset`), the `on_timeout` callback will be invoked (and if that one is missing, a non-fatal `ETIMEDOUT` error will be raised).

There are three variants of the timeouts that work independently of each other, for both read and write (triggered when nothing was read *OR* written), just read (triggered when nothing was read), and just write: `timeout`, `rtimeout` and `wtimeout`, with corresponding callbacks `on_timeout`, `on_rtimeout` and `on_wtimeout`, and reset functions `timeout_reset`, `rtimeout_reset`, and `wtimeout_reset`.

Note that timeout processing is active even when you do not have any outstanding read or write requests: If you plan to keep the connection idle then you should disable the timeout temporarily or ignore the timeout in the corresponding `on_timeout` callback, in which case AnyEvent::Handle will simply restart the timeout.

Zero (the default) disables the corresponding timeout.

– on_timeout => $cb->($handle)

– on_rtimeout => $cb->($handle)

– on_wtimeout => $cb->($handle)

Called whenever the inactivity timeout passes. If you return from this callback, then the timeout will be reset as if some activity had happened, so this condition is not fatal in any way.

– rbuf_max => <bytes>

If defined, then a fatal error will be raised (with `$!` set to `ENOSPC`) when the read buffer ever (strictly) exceeds this size. This is useful to avoid some forms of denial-of-service attacks.

For example, a server accepting connections from untrusted sources should be configured to accept only so-and-so much data that it cannot act on (for example, when expecting a line, an attacker could send an unlimited amount of data without a callback ever being called as long as the line isn't finished).

– wbuf_max => <bytes>

If defined, then a fatal error will be raised (with `$!` set to `ENOSPC`) when the write buffer ever (strictly) exceeds this size. This is useful to avoid some forms of denial-of-service attacks.

Although the units of this parameter is bytes, this is the *raw* number of bytes not yet accepted by the kernel. This can make a difference when you e.g. use TLS, as TLS typically makes your write data larger (but it can also make it smaller due to compression).

As an example of when this limit is useful, take a chat server that sends chat messages to a client. If the client does not read those in a timely manner then the send buffer in the server would grow unbounded.

– autocork => <boolean>

When disabled (the default), `push_write` will try to immediately write the data to the handle if possible. This avoids having to register a write watcher and wait for the next event loop iteration, but can be inefficient if you write multiple small chunks (on the wire, this disadvantage is usually avoided by your kernel's nagle algorithm, see `no_delay`, but this option can save costly syscalls).

When enabled, writes will always be queued till the next event loop iteration. This is efficient when you do many small writes per iteration, but less efficient when you do a single write only per iteration (or when the write buffer often is full). It also increases write latency.

– no_delay => <boolean>

When doing small writes on sockets, your operating system kernel might wait a bit for more data before actually sending it out. This is called the Nagle algorithm, and usually it is beneficial.

In some situations you want as low a delay as possible, which can be accomplishd by setting this option to a true value.

The default is your operating system's default behaviour (most likely enabled). This option explicitly enables or disables it, if possible.

– keepalive => <boolean>

Enables (default disable) the SO_KEEPALIVE option on the stream socket: normally, TCP connections have no time-out once established, so TCP connections, once established, can stay alive forever even when the other side has long gone. TCP keepalives are a cheap way to take down long-lived TCP connections when the other side becomes unreachable. While the default is OS-dependent, TCP keepalives usually kick in after around two hours, and, if the other side doesn't reply, take down the TCP connection some 10 to 15 minutes later.

It is harmless to specify this option for file handles that do not support keepalives, and enabling it on connections that are potentially long-lived is usually a good idea.

– oobinline => <boolean>

BSD majorly fucked up the implementation of TCP urgent data. The result is that almost no OS implements TCP according to the specs, and every OS implements it slightly differently.

If you want to handle TCP urgent data, then setting this flag (the default is enabled) gives you the most portable way of getting urgent data, by putting it into the stream.

Since BSD emulation of OOB data on top of TCP's urgent data can have security implications, AnyEvent::Handle sets this flag automatically unless explicitly specified. Note that setting this flag after establishing a connection *may* be a bit too late (data loss could already have occured on BSD systems), but at least it will protect you from most attacks.

– read_size => <bytes>

The initial read block size, the number of bytes this module will try to read during each loop iteration. Each handle object will consume at least this amount of memory for the read buffer as well, so when handling many connections watch out for memory requirements). See also `max_read_size`. Default: `2048`.

– max_read_size => <bytes>

The maximum read buffer size used by the dynamic adjustment algorithm: Each time AnyEvent::Handle can read `read_size` bytes in one go it will double `read_size` up to the maximum given by this option. Default: `131072` or `read_size`, whichever is higher.

– low_water_mark => <bytes>

Sets the number of bytes (default: `0`) that make up an "empty" write buffer: If the buffer reaches this size or gets even samller it is considered empty.

Sometimes it can be beneficial (for performance reasons) to add data to the write buffer before it is fully drained, but this is a rare case, as the operating system kernel usually buffers data as well, so the default is good in almost all cases.

– linger => <seconds>

If this is non-zero (default: `3600`), the destructor of the AnyEvent::Handle object will check whether there is still outstanding write data and will install a watcher that will write this data to the socket. No errors will be reported (this mostly matches how the operating system treats outstanding data at socket close time).

This will not work for partial TLS data that could not be encoded yet. This data will be lost. Calling the `stoptls` method in time might help.

– peername => $string

A string used to identify the remote site - usually the DNS hostname (*not* IDN!) used to create the connection, rarely the IP address.

Apart from being useful in error messages, this string is also used in TLS peername verification (see `verify_peername` in *AnyEvent::TLS*). This verification will be skipped when `peername` is not specified or is `undef`.

– tls => "accept" | "connect" | Net::SSLeay::SSL object

When this parameter is given, it enables TLS (SSL) mode, that means AnyEvent will start a TLS handshake as soon as the connection has been established and will transparently encrypt/decrypt data afterwards.

All TLS protocol errors will be signalled as `EPROTO`, with an appropriate error message.

TLS mode requires Net::SSLeay to be installed (it will be loaded automatically when you try to create a TLS handle): this module doesn't have a dependency on that module, so if your module requires it, you have to add the dependency yourself. If Net::SSLeay cannot be loaded or is too old, you get an EPROTO error.

Unlike TCP, TLS has a server and client side: for the TLS server side, use `accept`, and for the TLS client side of a connection, use `connect` mode.

You can also provide your own TLS connection object, but you have to make sure that you call either `Net::SSLeay::set_connect_state` or `Net::SSLeay::set_accept_state` on it before you pass it to AnyEvent::Handle. Also, this module will take ownership of this connection object.

At some future point, AnyEvent::Handle might switch to another TLS implementation, then the option to use your own session object will go away.

IMPORTANT: since Net::SSLeay "objects" are really only integers, passing in the wrong integer will lead to certain crash. This most often happens when one uses a stylish `tls => 1` and is surprised about the segmentation fault.

Use the `->starttls` method if you need to start TLS negotiation later.

– tls_ctx => $anyevent_tls

Use the given `AnyEvent::TLS` object to create the new TLS connection (unless a connection object was specified directly). If this parameter is missing (or `undef`), then AnyEvent::Handle will use `AnyEvent::Handle::TLS_CTX`.

Instead of an object, you can also specify a hash reference with `key => value` pairs. Those will be passed to *AnyEvent::TLS* to create a new TLS context object.

– on_starttls => $cb->($handle, $success[, $error_message])

This callback will be invoked when the TLS/SSL handshake has finished. If `$success` is true, then the TLS handshake succeeded, otherwise it failed (`on_stoptls` will not be called in this case).

The session in `$handle->{tls}` can still be examined in this callback, even when the handshake was not successful.

TLS handshake failures will not cause `on_error` to be invoked when this callback is in effect, instead, the error message will be passed to `on_starttls`.

Without this callback, handshake failures lead to `on_error` being called as usual.

Note that you cannot just call `starttls` again in this callback. If you need to do that, start an zero-second timer instead whose callback can then call `->starttls` again.

– on_stoptls => $cb->($handle)

When a SSLv3/TLS shutdown/close notify/EOF is detected and this callback is set, then it will be invoked after freeing the TLS session. If it is not, then a TLS shutdown condition will be treated like a normal EOF condition on the handle.

The session in `$handle->{tls}` can still be examined in this callback.

This callback will only be called on TLS shutdowns, not when the underlying handle signals EOF.

– json => *JSON*, *JSON::PP* or *JSON::XS* object

This is the json coder object used by the `json` read and write types.

If you don't supply it, then AnyEvent::Handle will create and use a suitable one (on demand), which will write and expect UTF-8 encoded JSON texts (either using *JSON::XS* or *JSON*). The written texts are guaranteed not to contain any newline character.

For security reasons, this encoder will likely *not* handle numbers and strings, only arrays and objects/hashes. The reason is that originally JSON was self-delimited, but Dougles Crockford thought it was a splendid idea to redefine JSON incompatibly, so this is no longer true.

For protocols that used back-to-back JSON texts, this might lead to run-ins, where two or more JSON texts will be interpreted as one JSON text.

For this reason, if the default encoder uses *JSON::XS*, it will default to not allowing anything but arrays and objects/hashes, at least for the forseeable future (it will change at some point). This might or might not be true for the *JSON* module, so this might cause a security issue.

If you depend on either behaviour, you should create your own json object and pass it in explicitly.

– cbor => CBOR::XS object

This is the cbor coder object used by the `cbor` read and write types.

If you don't supply it, then AnyEvent::Handle will create and use a suitable one (on demand), which will write CBOR without using extensions, if possible.

Note that you are responsible to depend on the CBOR::XS module if you want to use this functionality, as AnyEvent does not have a dependency on it itself.

- $fh = $handle->fh

  This method returns the file handle used to create the *AnyEvent::Handle* object.

- $handle->on_error ($cb)

  Replace the current `on_error` callback (see the `on_error` constructor argument).

- $handle->on_eof ($cb)

  Replace the current `on_eof` callback (see the `on_eof` constructor argument).

- $handle->on_timeout ($cb)

- $handle->on_rtimeout ($cb)

- $handle->on_wtimeout ($cb)

  Replace the current `on_timeout`, `on_rtimeout` or `on_wtimeout` callback, or disables the callback (but not the timeout) if `$cb` = `undef`. See the `timeout` constructor argument and method.

- $handle->autocork ($boolean)

Enables or disables the current autocork behaviour (see `autocork` constructor argument). Changes will only take effect on the next write.

- $handle->no_delay ($boolean)

  Enables or disables the `no_delay` setting (see constructor argument of the same name for details).

- $handle->keepalive ($boolean)

  Enables or disables the `keepalive` setting (see constructor argument of the same name for details).

- $handle->oobinline ($boolean)

  Enables or disables the `oobinline` setting (see constructor argument of the same name for details).

- $handle->on_starttls ($cb)

  Replace the current `on_starttls` callback (see the `on_starttls` constructor argument).

- $handle->on_stoptls ($cb)

  Replace the current `on_stoptls` callback (see the `on_stoptls` constructor argument).

- $handle->rbuf_max ($max_octets)

  Configures the `rbuf_max` setting (`undef` disables it).

- $handle->wbuf_max ($max_octets)

  Configures the `wbuf_max` setting (`undef` disables it).

- $handle->timeout ($seconds)
- $handle->rtimeout ($seconds)
- $handle->wtimeout ($seconds)

  Configures (or disables) the inactivity timeout.

  The timeout will be checked instantly, so this method might destroy the handle before it returns.

- $handle->timeout_reset
- $handle->rtimeout_reset
- $handle->wtimeout_reset

  Reset the activity timeout, as if data was received or sent.

  These methods are cheap to call.

WRITE QUEUE

AnyEvent::Handle manages two queues per handle, one for writing and one for reading.

The write queue is very simple: you can add data to its end, and AnyEvent::Handle will automatically try to get rid of it for you.

When data could be written and the write buffer is shorter then the low water mark, the `on_drain` callback will be invoked once.

- $handle->on_drain ($cb)

  Sets the `on_drain` callback or clears it (see the description of `on_drain` in the constructor).

  This method may invoke callbacks (and therefore the handle might be destroyed after it returns).

- $handle->push_write ($data)

  Queues the given scalar to be written. You can push as much data as you want (only limited by the available memory and `wbuf_max`), as `AnyEvent::Handle` buffers it independently of the kernel.

  This method may invoke callbacks (and therefore the handle might be destroyed after it returns).

- $handle->push_write (type => @args)

  Instead of formatting your data yourself, you can also let this module do the job by specifying a type and type-specific arguments. You can also specify the (fully qualified) name of a package, in which case AnyEvent tries to load the package and then expects to find the `anyevent_write_type` function inside (see "custom write types", below).

  Predefined types are (if you have ideas for additional types, feel free to drop by and tell us):

  – netstring => $string

    Formats the given value as netstring (http://cr.yp.to/proto/netstrings.txt, this is not a recommendation to use them).

  – packstring => $format, $data

    An octet string prefixed with an encoded length. The encoding `$format` uses the same format as a Perl `pack` format, but must specify a single integer only (only one of `cCsSlLqQiInNvVjJw` is allowed, plus an optional `!`, < or > modifier).

  – json => $array_or_hashref

    Encodes the given hash or array reference into a JSON object. Unless you provide your own JSON object, this means it will be encoded to JSON text in UTF-8.

    The default encoder might or might not handle every type of JSON value - it might be limited to arrays and objects for security reasons. See the `json` constructor attribute for more details.

    JSON objects (and arrays) are self-delimiting, so if you only use arrays and hashes, you can write JSON at one end of a handle and read them at the other end without using any additional framing.

    The JSON text generated by the default encoder is guaranteed not to contain any newlines: While this module doesn't need delimiters after or between JSON texts to be able to read them, many other languages depend on them.

A simple RPC protocol that interoperates easily with other languages is to send JSON arrays (or objects, although arrays are usually the better choice as they mimic how function argument passing works) and a newline after each JSON text:

```
$handle->push_write (json => ["method", "arg1", "arg2"]); #␣
↪whatever
$handle->push_write ("\012");
```

An AnyEvent::Handle receiver would simply use the `json` read type and rely on the fact that the newline will be skipped as leading whitespace:

```
$handle->push_read (json => sub { my $array = $_[1]; ... });
```

Other languages could read single lines terminated by a newline and pass this line into their JSON decoder of choice.

– cbor => $perl_scalar

Encodes the given scalar into a CBOR value. Unless you provide your own CBOR::XS object, this means it will be encoded to a CBOR string not using any extensions, if possible.

CBOR values are self-delimiting, so you can write CBOR at one end of a handle and read them at the other end without using any additional framing.

A simple nd very very fast RPC protocol that interoperates with other languages is to send CBOR and receive CBOR values (arrays are recommended):

```
$handle->push_write (cbor => ["method", "arg1", "arg2"]); #␣
↪whatever
```

An AnyEvent::Handle receiver would simply use the `cbor` read type:

```
$handle->push_read (cbor => sub { my $array = $_[1]; ... });
```

– storable => $reference

Freezes the given reference using *Storable* and writes it to the handle. Uses the `nfreeze` format.

- $handle->push_shutdown

Sometimes you know you want to close the socket after writing your data before it was actually written. One way to do that is to replace your `on_drain` handler by a callback that shuts down the socket (and set `low_water_mark` to `0`). This method is a shorthand for just that, and replaces the `on_drain` callback with:

```
sub { shutdown $_[0]{fh}, 1 }
```

This simply shuts down the write side and signals an EOF condition to the the peer.

You can rely on the normal read queue and `on_eof` handling afterwards. This is the cleanest way to close a connection.

This method may invoke callbacks (and therefore the handle might be destroyed after it returns).

- custom write types - Package::anyevent_write_type $handle, @args

  Instead of one of the predefined types, you can also specify the name of a package. AnyEvent will try to load the package and then expects to find a function named `anyevent_write_type` inside. If it isn't found, it progressively tries to load the parent package until it either finds the function (good) or runs out of packages (bad).

  Whenever the given `type` is used, `push_write` will the function with the handle object and the remaining arguments.

  The function is supposed to return a single octet string that will be appended to the write buffer, so you can mentally treat this function as a "arguments to on-the-wire-format" converter.

  Example: implement a custom write type `join` that joins the remaining arguments using the first one.

```
$handle->push_write (My::Type => " ", 1,2,3);

# uses the following package, which can be defined in the
 ↪"My::Type" or in
# the "My" modules to be auto-loaded, or just about anywhere␣
 ↪when the
# My::Type::anyevent_write_type is defined before invoking it.

package My::Type;

sub anyevent_write_type {
   my ($handle, $delim, @args) = @_;

   join $delim, @args
}
```

READ QUEUE

AnyEvent::Handle manages two queues per handle, one for writing and one for reading.

The read queue is more complex than the write queue. It can be used in two ways, the "simple" way, using only `on_read` and the "complex" way, using a queue.

In the simple case, you just install an `on_read` callback and whenever new data arrives, it will be called. You can then remove some data (if enough is there) from the read buffer (`$handle->rbuf`). Or you can leave the data there if you want to accumulate more (e.g. when only a partial message has been received so far), or change the read queue with e.g. `push_read`.

In the more complex case, you want to queue multiple callbacks. In this case, AnyEvent::Handle will call the first queued callback each time new data arrives (also the first time it is queued) and remove it when it has done its job (see `push_read`, below).

This way you can, for example, push three line-reads, followed by reading a chunk of data, and AnyEvent::Handle will execute them in order.

Example 1: EPP protocol parser. EPP sends 4 byte length info, followed by the specified number of bytes which give an XML datagram.

```
# in the default state, expect some header bytes
$handle->on_read (sub {
   # some data is here, now queue the length-header-read (4 octets)
   shift->unshift_read (chunk => 4, sub {
      # header arrived, decode
      my $len = unpack "N", $_[1];

      # now read the payload
      shift->unshift_read (chunk => $len, sub {
         my $xml = $_[1];
         # handle xml
      });
   });
});
```

Example 2: Implement a client for a protocol that replies either with “OK” and another line or “ERROR” for the first request that is sent, and 64 bytes for the second request. Due to the availability of a queue, we can just pipeline sending both requests and manipulate the queue as necessary in the callbacks.

When the first callback is called and sees an “OK” response, it will `unshift` another line-read. This line-read will be queued *before* the 64-byte chunk callback.

```
# request one, returns either "OK + extra line" or "ERROR"
$handle->push_write ("request 1\015\012");

# we expect "ERROR" or "OK" as response, so push a line read
$handle->push_read (line => sub {
   # if we got an "OK", we have to _prepend_ another line,
   # so it will be read before the second request reads its 64 bytes
   # which are already in the queue when this callback is called
   # we don't do this in case we got an error
   if ($_[1] eq "OK") {
      $_[0]->unshift_read (line => sub {
         my $response = $_[1];
         ...
      });
   }
});

# request two, simply returns 64 octets
$handle->push_write ("request 2\015\012");

# simply read 64 bytes, always
$handle->push_read (chunk => 64, sub {
   my $response = $_[1];
   ...
});
```

- $handle->on_read ($cb)

  This replaces the currently set `on_read` callback, or clears it (when the new callback is `undef`). See the description of `on_read` in the constructor.

  This method may invoke callbacks (and therefore the handle might be destroyed after it returns).

- $handle->rbuf

  Returns the read buffer (as a modifiable lvalue). You can also access the read buffer directly as the `->{rbuf}` member, if you want (this is much faster, and no less clean).

  The only operation allowed on the read buffer (apart from looking at it) is removing data from its beginning. Otherwise modifying or appending to it is not allowed and will lead to hard-to-track-down bugs.

  NOTE: The read buffer should only be used or modified in the `on_read` callback or when `push_read` or `unshift_read` are used with a single callback (i.e. untyped). Typed `push_read` and `unshift_read` methods will manage the read buffer on their own.

- $handle->push_read ($cb)

- $handle->unshift_read ($cb)

  Append the given callback to the end of the queue (`push_read`) or prepend it (`unshift_read`).

  The callback is called each time some additional read data arrives.

  It must check whether enough data is in the read buffer already.

  If not enough data is available, it must return the empty list or a false value, in which case it will be called repeatedly until enough data is available (or an error condition is detected).

  If enough data was available, then the callback must remove all data it is interested in (which can be none at all) and return a true value. After returning true, it will be removed from the queue.

  These methods may invoke callbacks (and therefore the handle might be destroyed after it returns).

- $handle->push_read (type => @args, $cb)

- $handle->unshift_read (type => @args, $cb)

  Instead of providing a callback that parses the data itself you can chose between a number of predefined parsing formats, for chunks of data, lines etc. You can also specify the (fully qualified) name of a package, in which case AnyEvent tries to load the package and then expects to find the `anyevent_read_type` function inside (see "custom read types", below).

  Predefined types are (if you have ideas for additional types, feel free to drop by and tell us):

  – chunk => $octets, $cb->($handle, $data)

Invoke the callback only once `$octets` bytes have been read. Pass the data read to the callback. The callback will never be called with less data.

Example: read 2 bytes.

```
$handle->push_read (chunk => 2, sub {
   say "yay " . unpack "H*", $_[1];
});
```

– line => [$eol,]$cb->($handle, $line, $eol)

The callback will be called only once a full line (including the end of line marker, `$eol`) has been read. This line (excluding the end of line marker) will be passed to the callback as second argument (`$line`), and the end of line marker as the third argument (`$eol`).

The end of line marker, `$eol`, can be either a string, in which case it will be interpreted as a fixed record end marker, or it can be a regex object (e.g. created by `qr`), in which case it is interpreted as a regular expression.

The end of line marker argument `$eol` is optional, if it is missing (NOT undef), then `qr|\015?\012|` is used (which is good for most internet protocols).

Partial lines at the end of the stream will never be returned, as they are not marked by the end of line marker.

– regex => $accept[, $reject[, $skip], $cb->($handle, $data)

Makes a regex match against the regex object `$accept` and returns everything up to and including the match. All the usual regex variables ($1, %+ etc.) from the regex match are available in the callback.

Example: read a single line terminated by '\n'.

```
$handle->push_read (regex => qr<\n>, sub { ... });
```

If `$reject` is given and not undef, then it determines when the data is to be rejected: it is matched against the data when the `$accept` regex does not match and generates an `EBADMSG` error when it matches. This is useful to quickly reject wrong data (to avoid waiting for a timeout or a receive buffer overflow).

Example: expect a single decimal number followed by whitespace, reject anything else (not the use of an anchor).

```
$handle->push_read (regex => qr<^[0-9]+\s>, qr<[^0-9]>, sub
↪{ ... });
```

If `$skip` is given and not `undef`, then it will be matched against the receive buffer when neither `$accept` nor `$reject` match, and everything preceding and including the match will be accepted unconditionally. This is useful to skip large amounts of data that you know cannot be matched, so that the `$accept` or `$reject` regex do not have to start matching from the beginning. This is purely an optimisation and is usually worth it only when you expect more than a few kilobytes.

Example: expect a http header, which ends at `\015\012\015\012`. Since we expect the header to be very large (it isn't in practice, but...), we use a skip

regex to skip initial portions. The skip regex is tricky in that it only accepts something not ending in either \015 or \012, as these are required for the accept regex.

```
$handle->push_read (regex =>
   qr<\015\012\015\012>,
   undef, # no reject
   qr<^.*[^\015\012]>,
   sub { ... });
```

– netstring => $cb->($handle, $string)

A netstring (http://cr.yp.to/proto/netstrings.txt, this is not an endorsement).

Throws an error with `$!` set to EBADMSG on format violations.

– packstring => $format, $cb->($handle, $string)

An octet string prefixed with an encoded length. The encoding `$format` uses the same format as a Perl `pack` format, but must specify a single integer only (only one of `cCsSlLqQiInNvVjJw` is allowed, plus an optional `!`, < or > modifier).

For example, DNS over TCP uses a prefix of `n` (2 octet network order), EPP uses a prefix of `N` (4 octtes).

Example: read a block of data prefixed by its length in BER-encoded format (very efficient).

```
$handle->push_read (packstring => "w", sub {
   my ($handle, $data) = @_;
});
```

– json => $cb->($handle, $hash_or_arrayref)

Reads a JSON object or array, decodes it and passes it to the callback. When a parse error occurs, an `EBADMSG` error will be raised.

If a `json` object was passed to the constructor, then that will be used for the final decode, otherwise it will create a *JSON::XS* or *JSON::PP* coder object expecting UTF-8.

This read type uses the incremental parser available with JSON version 2.09 (and JSON::XS version 2.2) and above.

Since JSON texts are fully self-delimiting, the `json` read and write types are an ideal simple RPC protocol: just exchange JSON datagrams. See the `json` write type description, above, for an actual example.

– cbor => $cb->($handle, $scalar)

Reads a CBOR value, decodes it and passes it to the callback. When a parse error occurs, an `EBADMSG` error will be raised.

If a CBOR::XS object was passed to the constructor, then that will be used for the final decode, otherwise it will create a CBOR coder without enabling any options.

You have to provide a dependency to CBOR::XS on your own: this module will load the CBOR::XS module, but AnyEvent does not depend on it itself.

Since CBOR values are fully self-delimiting, the `cbor` read and write types are an ideal simple RPC protocol: just exchange CBOR datagrams. See the `cbor` write type description, above, for an actual example.

– storable => $cb->($handle, $ref)

Deserialises a *Storable* frozen representation as written by the `storable` write type (BER-encoded length prefix followed by nfreeze'd data).

Raises `EBADMSG` error if the data could not be decoded.

– tls_detect => $cb->($handle, $detect, $major, $minor)

Checks the input stream for a valid SSL or TLS handshake TLSPaintext record without consuming anything. Only SSL version 3 or higher is handled, up to the fictituous protocol 4.x (but both SSL3+ and SSL2-compatible framing is supported).

If it detects that the input data is likely TLS, it calls the callback with a true value for `$detect` and the (on-wire) TLS version as second and third argument (`$major` is 3, and `$minor` is 0..4 for SSL 3.0, TLS 1.0, 1.1, 1.2 and 1.3, respectively). If it detects the input to be definitely not TLS, it calls the callback with a false value for `$detect`.

The callback could use this information to decide whether or not to start TLS negotiation.

In all cases the data read so far is passed to the following read handlers.

Usually you want to use the `tls_autostart` read type instead.

If you want to design a protocol that works in the presence of TLS dtection, make sure that any non-TLS data doesn't start with the octet 22 (ASCII SYN, 16 hex) or 128-255 (i.e. highest bit set). The checks this read type does are a bit more strict, but might losen in the future to accomodate protocol changes.

This read type does not rely on *AnyEvent::TLS* (and thus, not on *Net::SSLeay*).

– tls_autostart => [$tls_ctx,]$tls

Tries to detect a valid SSL or TLS handshake. If one is detected, it tries to start tls by calling `starttls` with the given arguments.

In practise, `$tls` must be `accept`, or a Net::SSLeay context that has been configured to accept, as servers do not normally send a handshake on their own and ths cannot be detected in this way.

See `tls_detect` above for more details.

Example: give the client a chance to start TLS before accepting a text line.

```
$hdl->push_read (tls_autostart => "accept");
$hdl->push_read (line => sub {
   print "received ", ($_[0]{tls} ? "encrypted" : "cleartext
↪"), " <$_[1]>\n";
});
```

- custom read types - Package::anyevent_read_type $handle, $cb, @args

  Instead of one of the predefined types, you can also specify the name of a package. AnyEvent will try to load the package and then expects to find a function named `anyevent_read_type` inside. If it isn't found, it progressively tries to load the parent package until it either finds the function (good) or runs out of packages (bad).

  Whenever this type is used, `push_read` will invoke the function with the handle object, the original callback and the remaining arguments.

  The function is supposed to return a callback (usually a closure) that works as a plain read callback (see `->push_read ($cb)`), so you can mentally treat the function as a "configurable read type to read callback" converter.

  It should invoke the original callback when it is done reading (remember to pass `$handle` as first argument as all other callbacks do that, although there is no strict requirement on this).

  For examples, see the source of this module (`perldoc -m AnyEvent::Handle`, search for `register_read_type`)).

- $handle->stop_read

- $handle->start_read

  In rare cases you actually do not want to read anything from the socket. In this case you can call `stop_read`. Neither `on_read` nor any queued callbacks will be executed then. To start reading again, call `start_read`.

  Note that AnyEvent::Handle will automatically `start_read` for you when you change the `on_read` callback or push/unshift a read callback, and it will automatically `stop_read` for you when neither `on_read` is set nor there are any read requests in the queue.

  In older versions of this module (<= 5.3), these methods had no effect, as TLS does not support half-duplex connections. In current versions they work as expected, as this behaviour is required to avoid certain resource attacks, where the program would be forced to read (and buffer) arbitrary amounts of data before being able to send some data. The drawback is that some readings of the the SSL/TLS specifications basically require this attack to be working, as SSL/TLS implementations might stall sending data during a rehandshake.

  As a guideline, during the initial handshake, you should not stop reading, and as a client, it might cause problems, depending on your application.

- $handle->starttls ($tls[, $tls_ctx])

  Instead of starting TLS negotiation immediately when the AnyEvent::Handle object is created, you can also do that at a later time by calling `starttls`. See the `tls` constructor argument for general info.

  Starting TLS is currently an asynchronous operation - when you push some write data and then call `->starttls` then TLS negotiation will start immediately, after which the queued write data is then sent. This might change in future versions, so best make sure you have no outstanding write data when calling this method.

  The first argument is the same as the `tls` constructor argument (either `"connect"`, `"accept"` or an existing Net::SSLeay object).

The second argument is the optional `AnyEvent::TLS` object that is used when AnyEvent::Handle has to create its own TLS connection object, or a hash reference with `key => value` pairs that will be used to construct a new context.

The TLS connection object will end up in `$handle->{tls}`, the TLS context in `$handle->{tls_ctx}` after this call and can be used or changed to your liking. Note that the handshake might have already started when this function returns.

Due to bugs in OpenSSL, it might or might not be possible to do multiple handshakes on the same stream. It is best to not attempt to use the stream after stopping TLS.

This method may invoke callbacks (and therefore the handle might be destroyed after it returns).

- $handle->stoptls

  Shuts down the SSL connection - this makes a proper EOF handshake by sending a close notify to the other side, but since OpenSSL doesn't support non-blocking shut downs, it is not guaranteed that you can re-use the stream afterwards.

  This method may invoke callbacks (and therefore the handle might be destroyed after it returns).

- $handle->resettls

  This rarely-used method simply resets and TLS state on the handle, usually causing data loss.

  One case where it may be useful is when you want to skip over the data in the stream but you are not interested in interpreting it, so data loss is no concern.

- $handle->destroy

  Shuts down the handle object as much as possible - this call ensures that no further callbacks will be invoked and as many resources as possible will be freed. Any method you will call on the handle object after destroying it in this way will be silently ignored (and it will return the empty list).

  Normally, you can just "forget" any references to an AnyEvent::Handle object and it will simply shut down. This works in fatal error and EOF callbacks, as well as code outside. It does *NOT* work in a read or write callback, so when you want to destroy the AnyEvent::Handle object from within such an callback. You *MUST* call `->destroy` explicitly in that case.

  Destroying the handle object in this way has the advantage that callbacks will be removed as well, so if those are the only reference holders (as is common), then one doesn't need to do anything special to break any reference cycles.

  The handle might still linger in the background and write out remaining data, as specified by the `linger` option, however.

- $handle->destroyed

  Returns false as long as the handle hasn't been destroyed by a call to `->destroy`, true otherwise.

  Can be useful to decide whether the handle is still valid after some callback possibly destroyed the handle. For example, `->push_write`, `->starttls` and other methods can call user callbacks, which in turn can destroy the handle, so work can be avoided by checking sometimes:

```
$hdl->starttls ("accept");
return if $hdl->destroyed;
$hdl->push_write (...
```

Note that the call to `push_write` will silently be ignored if the handle has been destroyed, so often you can just ignore the possibility of the handle being destroyed.

- AnyEvent::Handle::TLS_CTX

  This function creates and returns the AnyEvent::TLS object used by default for TLS mode.

  The context is created by calling *AnyEvent::TLS* without any arguments.

NONFREQUENTLY ASKED QUESTIONS

- I `undef` the AnyEvent::Handle reference inside my callback and still get further invocations!

  That's because AnyEvent::Handle keeps a reference to itself when handling read or write callbacks.

  It is only safe to "forget" the reference inside EOF or error callbacks, from within all other callbacks, you need to explicitly call the `->destroy` method.

- Why is my `on_eof` callback never called?

  Probably because your `on_error` callback is being called instead: When you have outstanding requests in your read queue, then an EOF is considered an error as you clearly expected some data.

  To avoid this, make sure you have an empty read queue whenever your handle is supposed to be "idle" (i.e. connection closes are O.K.). You can set an `on_read` handler that simply pushes the first read requests in the queue.

  See also the next question, which explains this in a bit more detail.

- How can I serve requests in a loop?

  Most protocols consist of some setup phase (authentication for example) followed by a request handling phase, where the server waits for requests and handles them, in a loop.

  There are two important variants: The first (traditional, better) variant handles requests until the server gets some QUIT command, causing it to close the connection first (highly desirable for a busy TCP server). A client dropping the connection is an error, which means this variant can detect an unexpected detection close.

  To handle this case, always make sure you have a non-empty read queue, by pushing the "read request start" handler on it:

```
# we assume a request starts with a single line
my @start_request; @start_request = (line => sub {
   my ($hdl, $line) = @_;

   ... handle request
```

(continues on next page)

(continued from previous page)

```
   # push next request read, possibly from a nested callback
   $hdl->push_read (@start_request);
});

# auth done, now go into request handling loop
# now push the first @start_request
$hdl->push_read (@start_request);
```

By always having an outstanding `push_read`, the handle always expects some data and raises the `EPIPE` error when the connction is dropped unexpectedly.

The second variant is a protocol where the client can drop the connection at any time. For TCP, this means that the server machine may run out of sockets easier, and in general, it means you cannot distinguish a protocl failure/client crash from a normal connection close. Nevertheless, these kinds of protocols are common (and sometimes even the best solution to the problem).

Having an outstanding read request at all times is possible if you ignore `EPIPE` errors, but this doesn't help with when the client drops the connection during a request, which would still be an error.

A better solution is to push the initial request read in an `on_read` callback. This avoids an error, as when the server doesn't expect data (i.e. is idly waiting for the next request, an EOF will not raise an error, but simply result in an `on_eof` callback. It is also a bit slower and simpler:

```
# auth done, now go into request handling loop
$hdl->on_read (sub {
   my ($hdl) = @_;

   # called each time we receive data but the read queue is␣
↪empty
   # simply start read the request

   $hdl->push_read (line => sub {
      my ($hdl, $line) = @_;

      ... handle request

      # do nothing special when the request has been handled,␣
↪just
      # let the request queue go empty.
   });
});
```

- I get different callback invocations in TLS mode/Why can't I pause reading?

  Unlike, say, TCP, TLS connections do not consist of two independent communication channels, one for each direction. Or put differently, the read and write directions are not independent of each other: you cannot write data unless you are also prepared to read, and vice versa.

This means that, in TLS mode, you might get `on_error` or `on_eof` callback invocations when you are not expecting any read data - the reason is that AnyEvent::Handle always reads in TLS mode.

During the connection, you have to make sure that you always have a non-empty read-queue, or an `on_read` watcher. At the end of the connection (or when you no longer want to use it) you can call the `destroy` method.

- How do I read data until the other side closes the connection?

  If you just want to read your data into a perl scalar, the easiest way to achieve this is by setting an `on_read` callback that does nothing, clearing the `on_eof` callback and in the `on_error` callback, the data will be in `$_[0]{rbuf}`:

```
$handle->on_read (sub { });
$handle->on_eof (undef);
$handle->on_error (sub {
   my $data = delete $_[0]{rbuf};
});
```

  Note that this example removes the `rbuf` member from the handle object, which is not normally allowed by the API. It is expressly permitted in this case only, as the handle object needs to be destroyed afterwards.

  The reason to use `on_error` is that TCP connections, due to latencies and packets loss, might get closed quite violently with an error, when in fact all data has been received.

  It is usually better to use acknowledgements when transferring data, to make sure the other side hasn't just died and you got the data intact. This is also one reason why so many internet protocols have an explicit QUIT command.

- I don't want to destroy the handle too early - how do I wait until all data has been written?

  After writing your last bits of data, set the `on_drain` callback and destroy the handle in there - with the default setting of `low_water_mark` this will be called precisely when all data has been written to the socket:

```
$handle->push_write (...);
$handle->on_drain (sub {
   AE::log debug => "All data submitted to the kernel.";
   undef $handle;
});
```

  If you just want to queue some data and then signal EOF to the other side, consider using `->push_shutdown` instead.

- I want to contact a TLS/SSL server, I don't care about security.

  If your TLS server is a pure TLS server (e.g. HTTPS) that only speaks TLS, connect to it and then create the AnyEvent::Handle with the `tls` parameter:

```
tcp_connect $host, $port, sub {
   my ($fh) = @_;
```

(continues on next page)

(continued from previous page)

```
   my $handle = new AnyEvent::Handle
      fh  => $fh,
      tls => "connect",
      on_error => sub { ... };

   $handle->push_write (...);
};
```

- I want to contact a TLS/SSL server, I do care about security.

  Then you should additionally enable certificate verification, including peername verification, if the protocol you use supports it (see *AnyEvent::TLS*, `verify_peername`).

  E.g. for HTTPS:

```
tcp_connect $host, $port, sub {
   my ($fh) = @_;

    my $handle = new AnyEvent::Handle
       fh       => $fh,
       peername => $host,
       tls      => "connect",
       tls_ctx  => { verify => 1, verify_peername => "https" },
       ...
```

  Note that you must specify the hostname you connected to (or whatever "peername" the protocol needs) as the `peername` argument, otherwise no peername verification will be done.

  The above will use the system-dependent default set of trusted CA certificates. If you want to check against a specific CA, add the `ca_file` (or `ca_cert`) arguments to `tls_ctx`:

```
tls_ctx  => {
   verify          => 1,
   verify_peername => "https",
   ca_file         => "my-ca-cert.pem",
},
```

- I want to create a TLS/SSL server, how do I do that?

  Well, you first need to get a server certificate and key. You have three options: a) ask a CA (buy one, use cacert.org etc.) b) create a self-signed certificate (cheap. check the search engine of your choice, there are many tutorials on the net) or c) make your own CA (tinyca2 is a nice program for that purpose).

  Then create a file with your private key (in PEM format, see *AnyEvent::TLS*), followed by the certificate (also in PEM format). The file should then look like this:

```
-----BEGIN RSA PRIVATE KEY-----
...header data
```

(continues on next page)

(continued from previous page)

```
... lots of base64'y-stuff
-----END RSA PRIVATE KEY-----

-----BEGIN CERTIFICATE-----
... lots of base64'y-stuff
-----END CERTIFICATE-----
```

The important bits are the “PRIVATE KEY” and “CERTIFICATE” parts. Then specify this file as `cert_file`:

```
tcp_server undef, $port, sub {
   my ($fh) = @_;

   my $handle = new AnyEvent::Handle
      fh       => $fh,
      tls      => "accept",
      tls_ctx  => { cert_file => "my-server-keycert.pem" },
      ...
```

When you have intermediate CA certificates that your clients might not know about, just append them to the `cert_file`.

SUBCLASSING AnyEvent::Handle

In many cases, you might want to subclass AnyEvent::Handle.

To make this easier, a given version of AnyEvent::Handle uses these conventions:

- all constructor arguments become object members.

  At least initially, when you pass a `tls`-argument to the constructor it will end up in `$handle->{tls}`. Those members might be changed or mutated later on (for example `tls` will hold the TLS connection object).

- other object member names are prefixed with an _.

  All object members not explicitly documented (internal use) are prefixed with an underscore character, so the remaining non-_-namespace is free for use for subclasses.

- all members not documented here and not prefixed with an underscore are free to use in subclasses.

  Of course, new versions of AnyEvent::Handle may introduce more “public” member variables, but that’s just life. At least it is documented.

AUTHOR

Robin Redeker `<elmex at ta-sa.org>`, Marc Lehmann schmorp@schmorp.de.

AnyEvent::IO

the DBI of asynchronous I/O implementations

SYNOPSIS

```
use AnyEvent::IO;

# load /etc/passwd, call callback with the file data when done.
aio_load "/etc/passwd", sub {
   my ($data) = @_
      or return AE::log error => "/etc/passwd: $!";

   warn "/etc/passwd contains ", ($data =~ y/:/:/) , " colons.\n";
};

# the rest of the SYNOPSIS does the same, but with individual I/O
↪calls

# also import O_XXX flags
use AnyEvent::IO qw(:DEFAULT :flags);

my $filedata = AE::cv;

# first open the file
aio_open "/etc/passwd", O_RDONLY, 0, sub {
   my ($fh) = @_
      or return AE::log error => "/etc/passwd: $!";

   # now stat the file to get the size
   aio_stat $fh, sub {
      @_
         or return AE::log error => "/etc/passwd: $!";

      my $size = -s _;

      # now read all the file data
      aio_read $fh, $size, sub {
         my ($data) = @_
            or return AE::log error => "/etc/passwd: $!";

         $size == length $data
            or return AE::log error => "/etc/passwd: short read,
↪file changed?";

         # mostly the same as aio_load, above - $data contains
         # the file contents now.
         $filedata->($data);
      };
   };
```

(continues on next page)

(continued from previous page)

```
};

my $passwd = $filedata->recv;
warn length $passwd, " octets.\n";
```

DESCRIPTION

This module provides functions that do I/O in an asynchronous fashion. It is to I/O the same as *AnyEvent* is to event libraries - it only *interfaces* to other implementations or to a portable pure-perl implementation (which does not, however, do asynchronous I/O).

The only other implementation that is supported (or even known to the author) is IO::AIO, which is used automatically when it can be loaded (via AnyEvent::AIO, which also needs to be installed). If it is not available, then *AnyEvent::IO* falls back to its synchronous pure-perl implementation.

Unlike *AnyEvent*, which model to use is currently decided at module load time, not at first use. Future releases might change this.

RATIONALE

While disk I/O often seems "instant" compared to, say, socket I/O, there are many situations where your program can block for extended time periods when doing disk I/O. For example, you access a disk on an NFS server and it is gone - can take ages to respond again, if ever. Or your system is extremely busy because it creates or restores a backup - reading data from disk can then take seconds. Or you use Linux, which for so many years has a close-to-broken VM/IO subsystem that can often induce minutes or more of delay for disk I/O, even under what I would consider light I/O loads.

Whatever the situation, some programs just can't afford to block for long times (say, half a second or more), because they need to respond as fast as possible.

For those cases, you need asynchronous I/O.

The problem is, AnyEvent itself sometimes reads disk files (for example, when looking at `/etc/hosts`), and under the above situations, this can bring your program to a complete halt even if your program otherwise takes care to only use asynchronous I/O for everything (e.g. by using IO::AIO).

On the other hand, requiring IO::AIO for AnyEvent is clearly impossible, as AnyEvent promises to stay pure-perl, and the overhead of IO::AIO for small programs would be immense, especially when asynchronous I/O isn't even needed.

Clearly, this calls for an abstraction layer, and that is what you are looking at right now :-)

ASYNCHRONOUS VS. NON-BLOCKING

Many people are continuously confused on what the difference is between asynchronous I/O and non-blocking I/O. In fact, those two terms are not well defined, which often makes it hard to even talk about the difference. Here is a short guideline that should leave you less confused. It only talks about read operations, but the reasoning works with other I/O operations as well.

Non-blocking I/O means that data is delivered by some external means, automatically - that is, something *pushes* data towards your file handle, without you having to do anything. Non-blocking means that if your operating system currently has no data (or EOF, or some error) available for you, it will not wait ("block") as it would normally do, but immediately return with an error (e.g. `EWOULDBLOCK` - "I would have blocked, but you forbid it").

Your program can then wait for data to arrive by other means, for example, an I/O watcher which tells you when to re-attempt the read, after which it can try to read again, and so on.

Often, you would expect this to work for disk files as well - if the data isn't already in memory, one might want to wait for it and then re-attempt the read for example. While this is sound reasoning, the POSIX API does not support this, because disk drives and file systems do not send data "on their own", and more so, the OS already knows that data is there, it doesn't need to "wait" until it arrives from some external entity, it only needs to transfer the data from disk to your memory buffer.

So basically, while the concept is sound, the existing OS APIs do not support this. Therefore, it makes no sense to switch a disk file handle into non-blocking mode - it will behave exactly the same as in blocking mode, namely it will block until the data has been read from the disk.

The alternative to non-blocking I/O that actually works with disk files is usually called *asynchronous I/O*. Asynchronous, because the actual I/O is done while your program does something else: there is no need to call the read function to see if data is there, you only order the read once, and it will notify you when the read has finished and the data is your buffer - all the work is done in the background.

This works with disk files, and even with sockets and other sources. It is, however, not very efficient when used with sources that could be driven in a non-blocking way, because it usually has higher overhead in the OS than non-blocking I/O, because it ties memory buffers for a potentially unlimited time and often only a limited number of operations can be done in parallel.

That's why asynchronous I/O makes most sense when confronted with disk files, and non-blocking I/O only makes sense with sockets, pipes and similar streaming sources.

IMPORT TAGS

By default, this module exports all `aio_xxx` functions. In addition, the following import tags can be used:

```
:aio       all aio_* functions, same as :DEFAULT
:flags     the fcntl open flags (O_CREAT, O_RDONLY, ...)
```

API NOTES

The functions in this module are not meant to be the most versatile or the highest-performers (they are not very slow either, of course). They are primarily meant to give users of your code the option to do the I/O asynchronously (by installing IO::AIO and AnyEvent::AIO), without adding a dependency on those modules.

NAMING

All the functions in this module implement an I/O operation, usually with the same or similar name as the Perl built-in that they mimic, but with an `aio_` prefix. If you like you can think of the `aio_xxx` functions as "AnyEvent I/O" or "Asynchronous I/O" variants of Perl built-ins.

CALLING CONVENTIONS AND ERROR REPORTING

Each function expects a callback as their last argument. The callback is usually called with the result data or result code. An error is usually signalled by passing no arguments to the callback, which is then free to look at `$!` for the error code.

This makes all of the following forms of error checking valid:

```
aio_open ...., sub {
   my $fh = shift   # scalar assignment - will assign undef on error
      or return AE::log error => "...";

   my ($fh) = @_    # list assignment - will be 0 elements on error
      or return AE::log error => "...";

   @_               # check the number of elements directly
      or return AE::log error => "...";
```

CAVEAT: RELATIVE PATHS

When a path is specified, this path *must be an absolute* path, unless you make certain that nothing in your process calls `chdir` or an equivalent function while the request executes.

CAVEAT: OTHER SHARED STATE

Changing the `umask` while any requests execute that create files (or otherwise rely on the current umask) results in undefined behaviour - likewise changing anything else that would change the outcome, such as your effective user or group ID.

CALLBACKS MIGHT BE CALLED BEFORE FUNCTION RETURNS TO CALLER

Unlike other functions in the AnyEvent module family, these functions *may* call your callback instantly, before returning. This should not be a real problem, as these functions never return anything useful.

BEHAVIOUR AT PROGRAM EXIT

Both *AnyEvent::IO::Perl* and *AnyEvent::IO::IOAIO* implementations make sure that operations that have started will be finished on a clean programs exit. That makes programs work that start some I/O operations and then exit. For example this complete program:

```
use AnyEvent::IO;

aio_stat "path1", sub {
   aio_stat "path2", sub {
      warn "both stats done\n";
   };
};
```

Starts a `stat` operation and then exits by "falling off the end" of the program. Nevertheless, *both* `stat` operations will be executed, as AnyEvent::IO waits for all outstanding requests to finish and you can start new requests from request callbacks.

In fact, since *AnyEvent::IO::Perl* is currently synchronous, the program will do both stats before falling off the end, but with *AnyEvent::IO::IOAIO*, the program first falls of the end, then the stats are executed.

While not guaranteed, this behaviour will be present in future versions, if reasonably possible (which is extreemly likely :).

GLOBAL VARIABLES AND FUNCTIONS

- $AnyEvent::IO::MODEL

  Contains the package name of the backend I/O model in use - at the moment, this is usually `AnyEvent::IO::Perl` or `AnyEvent::IO::IOAIO`.

- aio_load $path, $cb->($data)

  Tries to open `$path` and read its contents into memory (obviously, should only be used on files that are "small enough"), then passes them to the callback as a string.

  Example: load `/etc/hosts`.

  ```
  aio_load "/etc/hosts", sub {
     my ($hosts) = @_
        or return AE::log error => "/etc/hosts: $!";

     AE::log info => "/etc/hosts contains ", ($hosts =~ y/\n/), "␣
  ↪lines\n";
  };
  ```

- aio_open $path, $flags, $mode, $cb->($fh)

  Tries to open the file specified by `$path` with the O_XXX-flags `$flags` (from the Fcntl module, or see below) and the mode `$mode` (a good value is 0666 for `O_CREAT`, and `0` otherwise).

  The (normal, standard, perl) file handle associated with the opened file is then passed to the callback.

  This works very much like Perl's `sysopen` function.

  Changing the `umask` while this request executes results in undefined behaviour - likewise changing anything else that would change the outcome, such as your effective user or group ID.

To avoid having to load *Fcntl*, this module provides constants for `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_CREAT`, `O_EXCL`, `O_TRUNC` and `O_APPEND` - you can either access them directly (`AnyEvent::IO::O_RDONLY`) or import them by specifying the `:flags` import tag (see SYNOPSIS).

Example: securely open a file in `/var/tmp`, fail if it exists or is a symlink.

```
use AnyEvent::IO qw(:flags);

aio_open "/var/tmp/mytmp$$", O_CREAT | O_EXCL | O_RDWR, 0600,␣
↪sub {
   my ($fh) = @_
      or return AE::log error => "$! - denial of service attack?
↪";

   # now we have $fh
};
```

- aio_close $fh, $cb->($success)

  Closes the file handle (yes, close can block your process indefinitely) and passes a true value to the callback on success.

  Due to idiosyncrasies in perl, instead of calling `close`, the file handle might get closed by dup2'ing another file descriptor over it, that is, the `$fh` might still be open, but can be closed safely afterwards and must not be used for anything.

  Example: close a file handle, and dirty as we are, do not even bother to check for errors.

```
aio_close $fh, sub { };
```

- aio_read $fh, $length, $cb->($data)

  Tries to read `$length` octets from the current position from `$fh` and passes these bytes to `$cb`. Otherwise the semantics are very much like those of Perl's `sysread`.

  If less than `$length` octets have been read, `$data` will contain only those bytes actually read. At EOF, `$data` will be a zero-length string. If an error occurs, then nothing is passed to the callback.

  Obviously, multiple `aio_read`'s or `aio_write`'s at the same time on file handles sharing the underlying open file description results in undefined behaviour, due to sharing of the current file offset (and less obviously so, because OS X is not thread safe and corrupts data when you try).

  Example: read 128 octets from a file.

```
aio_read $fh, 128, sub {
   my ($data) = @_
      or return AE::log error "read from fh: $!";

   if (length $data) {
      print "read ", length $data, " octets.\n";
   } else {
```

(continues on next page)

(continued from previous page)

```
      print "EOF\n";
   }
};
```

- aio_seek $fh, $offset, $whence, $callback->($offs)

  Seeks the filehandle to the new `$offset`, similarly to Perl's `sysseek`. The `$whence` are the traditional values (`0` to count from start, 1 to count from the current position and 2 to count from the end).

  The resulting absolute offset will be passed to the callback on success.

  Example: measure the size of the file in the old-fashioned way using seek.

```
aio_seek $fh, 0, 2, sub {
   my ($size) = @_
      or return AE::log error => "seek to end failed: $!";

   # maybe we need to seek to the beginning again?
   aio_seek $fh, 0, 0, sub {
      # now we are hopefully at the beginning
   };
};
```

- aio_write $fh, $data, $cb->($length)

  Tries to write the octets in `$data` to the current position of `$fh` and passes the actual number of bytes written to the `$cb`. Otherwise the semantics are very much like those of Perl's `syswrite`.

  If less than `length $data` octets have been written, `$length` will reflect that. If an error occurs, then nothing is passed to the callback.

  Obviously, multiple `aio_read`'s or `aio_write`'s at the same time on file handles sharing the underlying open file description results in undefined behaviour, due to sharing of the current file offset (and less obviously so, because OS X is not thread safe and corrupts data when you try).

- aio_truncate $fh_or_path, $new_length, $cb->($success)

  Calls `truncate` on the path or perl file handle and passes a true value to the callback on success.

  Example: truncate `/etc/passwd` to zero length - this only works on systems that support `truncate`, should not be tried out for obvious reasons and debian will probably open yte another security bug about this example.

```
aio_truncate "/etc/passwd", sub {
   @_
      or return AE::log error => "/etc/passwd: $! - are you␣
↪root enough?";
};
```

- aio_utime $fh_or_path, $atime, $mtime, $cb->($success)

Calls `utime` on the path or perl file handle and passes a true value to the callback on success.

The special case of both `$atime` and `$mtime` being `undef` sets the times to the current time, on systems that support this.

Example: try to touch `file`.

```
aio_utime "file", undef, undef, sub { };
```

- aio_chown $fh_or_path, $uid, $gid, $cb->($success)

  Calls `chown` on the path or perl file handle and passes a true value to the callback on success.

  If `$uid` or `$gid` can be specified as `undef`, in which case the uid or gid of the file is not changed. This differs from Perl's `chown` built-in, which wants `-1` for this.

  Example: update the group of `file` to 0 (root), but leave the owner alone.

```
aio_chown "file", undef, 0, sub {
   @_
      or return AE::log error => "chown 'file': $!";
};
```

- aio_chmod $fh_or_path, $perms, $cb->($success)

  Calls `chmod` on the path or perl file handle and passes a true value to the callback on success.

  Example: change `file` to be user/group/world-readable, but leave the other flags alone.

```
aio_stat "file", sub {
   @_
      or return AE::log error => "file: $!";

   aio_chmod "file", (stat _)[2] & 07777 | 00444, sub { };
};
```

- aio_stat $fh_or_path, $cb->($success)
- aio_lstat $path, $cb->($success)

  Calls `stat` or `lstat` on the path or perl file handle and passes a true value to the callback on success.

  The stat data will be available by `stat`'ing the _ file handle (e.g. `-x _`, `stat _` and so on).

  Example: see if we can find the number of subdirectories of `/etc`.

```
aio_stat "/etc", sub {
   @_
      or return AE::log error => "/etc: $!";

   (stat _)[3] >= 2
```

(continues on next page)

(continued from previous page)

```
      or return AE::log warn => "/etc has low link count - non-
↪POSIX filesystem?";

   print "/etc has ", (stat _)[3] - 2, " subdirectories.\n";
};
```

- aio_link $oldpath, $newpath, $cb->($success)

  Calls `link` on the paths and passes a true value to the callback on success.

  Example: link "`file` to `file.bak`, then rename `file.new` over `file`, to atomically replace it.

```
aio_link "file", "file.bak", sub {
   @_
      or return AE::log error => "file: $!";

   aio_rename "file.new", "file", sub {
      @_
         or return AE::log error => "file.new: $!";

      print "file atomically replaced by file.new, backup file.
↪bak\n";
   };
};
```

- aio_symlink $oldpath, $newpath, $cb->($success)

  Calls `symlink` on the paths and passes a true value to the callback on success.

  Example: create a symlink "`slink` containing "random data".

```
aio_symlink "random data", "slink", sub {
   @_
      or return AE::log error => "slink: $!";
};
```

- aio_readlink $path, $cb->($target)

  Calls `readlink` on the paths and passes the link target string to the callback.

  Example: read the symlink called Fyslink> and verify that it contains "random data".

```
aio_readlink "slink", sub {
   my ($target) = @_
      or return AE::log error => "slink: $!";

   $target eq "random data"
      or AE::log critical => "omg, the world will end!";
};
```

- aio_rename $oldpath, $newpath, $cb->($success)

Calls `rename` on the paths and passes a true value to the callback on success.

See `aio_link` for an example.

- aio_unlink $path, $cb->($success)

  Tries to unlink the object at `$path` and passes a true value to the callback on success.

  Example: try to delete the file `tmpfile.dat~`.

```
aio_unlink "tmpfile.dat~", sub { };
```

- aio_mkdir $path, $perms, $cb->($success)

  Calls `mkdir` on the path with the given permissions `$perms` (when in doubt, `0777` is a good value) and passes a true value to the callback on success.

  Example: try to create the directory `subdir` and leave it to whoeveer comes after us to check whether it worked.

```
aio_mkdir "subdir", 0777, sub { };
```

- aio_rmdir $path, $cb->($success)

  Tries to remove the directory at `$path` and passes a true value to the callback on success.

  Example: try to remove the directory `subdir` and don't give a damn if that fails.

```
aio_rmdir "subdir", sub { };
```

- aio_readdir $path, $cb->(\@names)

  Reads all filenames from the directory specified by `$path` and passes them to the callback, as an array reference with the names (without a path prefix). The `.` and `..` names will be filtered out first.

  The ordering of the file names is undefined - backends that are capable of it (e.g. IO::AIO) will return the ordering that most likely is fastest to `stat` through, and furthermore put entries that likely are directories first in the array.

  If you need best performance in recursive directory traversal or when looking at really big directories, you are advised to use IO::AIO directly, specifically the `aio_readdirx` and `aio_scandir` functions, which have more options to tune performance.

  Example: recursively scan a directory hierarchy, silently skip diretcories we couldn't read and print all others.

```
sub scan($); # visibility-in-next statement is not so useful␣
↪these days
sub scan($) {
   my ($path) = @_;

   aio_readdir $path, sub {
      my ($names) = @_
```

(continues on next page)

(continued from previous page)

```
         or return;

      print "$path\n";

      for my $name (@$names) {
         aio_lstat "$path/$name", sub {
            scan "$path/$name"
               if -d _;
         };
      }
   };
}

scan "/etc";
```

ENVIRONMENT VARIABLES

See the description of `PERL_ANYEVENT_IO_MODEL` in the *AnyEvent* manpage.

AUTHOR

```
Marc Lehmann <schmorp@schmorp.de>
http://anyevent.schmorp.de
```

Modules

- `AnyEvent::IO::IOAIO` - AnyEvent::IO backend based on IO::AIO
- `AnyEvent::IO::Perl` - pure perl backend for AnyEvent::IO

AnyEvent::IO::IOAIO

AnyEvent::IO backend based on IO::AIO

SYNOPSIS

```
use AnyEvent::IO;
```

DESCRIPTION

This is the IO::AIO-based backend of *AnyEvent::IO* (via AnyEvent::AIO). All I/O operations it implements are done asynchronously.

FUNCTIONS

SEE ALSO

AnyEvent::IO, *AnyEvent*.

AUTHOR

```
Marc Lehmann <schmorp@schmorp.de>
http://anyevent.schmorp.de
```

AnyEvent::IO::Perl

pure perl backend for AnyEvent::IO

SYNOPSIS

```
use AnyEvent::IO;
```

DESCRIPTION

This is the pure-perl backend of *AnyEvent::IO* - it is always available, but does not actually implement any I/O operation asynchronously - everything is synchronous.

For simple programs that can wait for I/O, this is likely the most efficient implementation.

SEE ALSO

AnyEvent::IO, *AnyEvent*.

AUTHOR

```
Marc Lehmann <schmorp@schmorp.de>
http://anyevent.schmorp.de
```

AnyEvent::Impl

Namespace containing the following modules:

- `AnyEvent::Impl::Cocoa` - AnyEvent adaptor for Cocoa::EventLoop
- `AnyEvent::Impl::EV` - AnyEvent adaptor for EV
- `AnyEvent::Impl::Event` - AnyEvent adaptor for Event
- `AnyEvent::Impl::EventLib` - AnyEvent adaptor for Event::Lib
- `AnyEvent::Impl::FLTK` - AnyEvent adaptor for FLTK (Fast Light Toolkit version two)
- `AnyEvent::Impl::Glib` - AnyEvent adaptor for Glib

- `AnyEvent::Impl::IOAsync` - AnyEvent adaptor for IO::Async
- `AnyEvent::Impl::Irssi` - AnyEvent adaptor for Irssi
- `AnyEvent::Impl::POE` - AnyEvent adaptor for POE
- `AnyEvent::Impl::Perl` - AnyEvent adaptor for AnyEvent's pure perl AnyEvent::Loop
- `AnyEvent::Impl::Qt` - AnyEvent adaptor for Qt
- `AnyEvent::Impl::Tk` - AnyEvent adaptor for Tk
- `AnyEvent::Impl::UV` - AnyEvent adaptor for UV

AnyEvent::Impl::Cocoa

AnyEvent adaptor for Cocoa::EventLoop

SYNOPSIS

```
use AnyEvent;
use Cocoa::EventLoop;

# do something
```

DESCRIPTION

This module provides NSRunLoop support to AnyEvent.

NSRunLoop is an event loop for Cocoa applications, wrapped by Cocoa::EventLoop. By using this module, you can use Cocoa based API in your AnyEvent application, or AnyEvent within Cocoa applications.

It does not support blocking waits.

BUGS

Right now, Cocoa::EventLoop (and this module) are in an early development phase and has some shortcomings and likely bugs.

For example, there seems to be no way to just handle a single event with Cocoa (is there nothing they can implement properly?), so this module currently wakes up at least ten times a second when waiting for events. Also, events caused by timers might get delayed by up to 0.1 seconds.

AUTHORS

Daisuke Murase typester@cpan.org, Marc Lehmann schmorp@schmorp.de.

COPYRIGHTS

```
Copyright (c) 2009 by KAYAC Inc.
Copyright (c) 2010,2011 by Marc Lehmann <schmorp@schmorp.de>
```

AnyEvent::Impl::EV

AnyEvent adaptor for EV

SYNOPSIS

```
use AnyEvent;
use EV;

# this module gets loaded automatically as required
```

DESCRIPTION

This module provides transparent support for AnyEvent. You don't have to do anything to make EV work with AnyEvent except by loading EV before creating the first AnyEvent watcher.

EV is the fastest event library for perl, and best supported by AnyEvent. Most functions from the *AE* API are implemented as direct aliases to EV functions, so using EV via AE is as fast as using EV directly.

SEE ALSO

AnyEvent, *EV*.

AUTHOR

```
Marc Lehmann <schmorp@schmorp.de>
http://anyevent.schmorp.de
```

AnyEvent::Impl::Event

AnyEvent adaptor for Event

SYNOPSIS

```
use AnyEvent;
use Event;

# this module gets loaded automatically as required
```

DESCRIPTION

This module provides transparent support for AnyEvent. You don't have to do anything to make Event work with AnyEvent except by loading Event before creating the first AnyEvent watcher.

The event module is reasonably efficient and generally works correctly even with many watchers, except that its signal handling is inherently racy and requires the wake-up-frequently workaround.

SEE ALSO

AnyEvent, *Event*.

AUTHOR

Marc Lehmann <schmorp@schmorp.de>
http://anyevent.schmorp.de

AnyEvent::Impl::EventLib

AnyEvent adaptor for Event::Lib

SYNOPSIS

```
use AnyEvent;
use Event::Lib;

# this module gets loaded automatically as required
```

DESCRIPTION

This module provides transparent support for AnyEvent. You don't have to do anything to make Event work with AnyEvent except by loading Event::Lib before creating the first AnyEvent watcher.

Note: the AnyEvent author has not found recent releases of Event::Lib to be even remotely working (not even the examples from the manpage or the testsuite work), so this event backend should be avoided (or somebody should step up and maintain it, hint, hint).

The Event::Lib module suffers from the same limitations and bugs as libevent, most notably it kills already-installed watchers on a file descriptor and it is unable to support fork. These are not fatal issues, and are worked-around by this module, but the Event::Lib perl module itself has many additional bugs such as taking references to file handles and callbacks instead of making a copy or freeing still-allocated scalars, causing memory corruption and random crashes. Only Tk rivals it in its brokenness.

This adaptor module employs the same workaround around the watcher problems as Tk and should therefore be avoided. (This was done for simplicity, one could in theory work around the problems with lower overhead by managing our own watchers).

Event::Lib also leaks file handles and memory and tends to just exit on problems.

It also doesn't work around the Windows bug of not signalling TCP connection failures.

It also doesn't work with many special devices on Linux (`/dev/random` works, `/dev/urandom` fails, `/dev/tty` works, `/dev/null` fails and so on).

Event::Lib does not support idle watchers. They could be emulated using low-priority timers but as the priority range (and availability) is not queryable nor guaranteed, and the default priority is likely the lowest one, this module cannot use them.

Avoid Event::Lib if you can.

SEE ALSO

AnyEvent, Event::Lib.

AUTHOR

```
Marc Lehmann <schmorp@schmorp.de>
http://anyevent.schmorp.de
```

AnyEvent::Impl::FLTK

AnyEvent adaptor for FLTK (Fast Light Toolkit version two)

SYNOPSIS

```
use AnyEvent;
use FLTK;

# this module gets loaded automatically as required
```

DESCRIPTION

This module provides transparent support for AnyEvent. You don't have to do anything to make FLTK work with AnyEvent except by loading FLTK before creating the first AnyEvent watcher.

This implementation is not to be confused with AnyEvent::Impl::FLTK by Sanko Robinson. That implementation is completely broken, and the author is apparently unreachable.

In any case, FLTK suffers from typical GUI-ToolKit diseases, such as O(n) or worse for every operation (adding a timer, destroying a timer etc.), the typical Not-Well-Tested Perl Interface disases such as non-random memory corruption and the typical Event-Loop-as-an-Afterthrough issues, such as multiple watchers on the same fd silently overwriting the others.

It doesn't have native idle, signal or child watchers, so all of these are emulated.

SEE ALSO

AnyEvent, FLTK.

AUTHOR

Marc Lehmann <schmorp@schmorp.de>
http://anyevent.schmorp.de

AnyEvent::Impl::Glib

AnyEvent adaptor for Glib

SYNOPSIS

```
use AnyEvent;
use Glib;

# this module gets loaded automatically as required
```

DESCRIPTION

This module provides transparent support for AnyEvent. You don't have to do anything to make Glib work with AnyEvent except by loading Glib before creating the first AnyEvent watcher.

Glib is probably the most inefficient event loop that has ever seen the light of the world: Glib not only scans all its watchers (really, ALL of them, whether I/O-related, timer-related or what not) during each loop iteration, it also does so multiple times and rebuilds the poll list for the kernel each time again, dynamically even. Newer versions of libglib fortunately do not call malloc/free on every single watcher invocation, though.

Glib also enforces certain undocumented behaviours, for example, you cannot always remove active child watchers, and the conditions on when it is valid to do so are not documented. Of course, if you get it wrong, you get "GLib-CRITICAL" messages. This makes it extremely hard to write "correct" glib programs, as you have to study the source code to get it right, and hope future versions don't change any internals.

AnyEvent implements the necessary workarounds, at a small performance cost.

On the positive side, and most importantly, when it works, Glib generally works correctly, no quarrels there.

If you create many watchers (as in: more than two), you might consider one of the Glib::EV, EV::Glib or Glib::Event modules that map Glib to other, more efficient, event loops.

This module uses the default Glib main context for all its watchers.

SEE ALSO

AnyEvent, Glib.

AUTHOR

```
Marc Lehmann <schmorp@schmorp.de>
http://anyevent.schmorp.de
```

AnyEvent::Impl::IOAsync

AnyEvent adaptor for IO::Async

SYNOPSIS

```
use AnyEvent;
use IO::Async::Loop;

# optionally set another event loop
use AnyEvent::Impl::IOAsync;
my $loop = new IO::Async::Loop;
AnyEvent::Impl::IOAsync::set_loop $loop;
```

DESCRIPTION

This module provides support for IO::Async as AnyEvent backend. It supports I/O, timers, signals and child process watchers. Idle watchers are emulated. I/O watchers need to dup their fh because IO::Async only supports IO handles, not plain file descriptors.

FUNCTIONS AND VARIABLES

The only user-servicible part in this module is the `set_loop` function and `$LOOP` variable:

- AnyEvent::Impl::IOAsync::set_loop $new_loop

  Unfortunately, IO::Async has no concept of a default loop. Modules using IO::Async must be told by their caller which loop to use, which makes it impossible to transparently use IO::Async from a module.

  This module is no exception. It creates a new IO::Async::Loop object when it is loaded. This might not be the right loop object, though, and thus you can replace it by a call to this function with the loop object of your choice.

  Note that switching loops while watchers are already initialised can have unexpected effects, and is not supported unless you can live witht he consequences.

- $AnyEvent::Impl::IOAsync::LOOP

  This variable always contains the IO::Async::Loop object used by this AnyEvent backend. See above for more info.

Storing the "default" loop makes this module a possible arbiter for other modules that want to use IO::Async transparently. It's advised to directly refer to this variable each time you want to use it, without making a local copy.

PROBLEMS WITH IO::Async

This section had a long list of problems and shortcomings that made it almost impossible to support IO::Async. With version 0.33 of IO::Async, however, most of these have been fixed, so IO::Async can now be used as easily as many other loops.

There are a few remaining problems that require emulation or workarounds:

- No support for multiple watchers per event

  In most (all? documentation?) cases you cannot have multiple watchers for the same event (what's the point of having all these fancy notifier classes when you cannot have multiple notifiers for the same event? That's like only allowing one timer per second or so...).

  For I/O watchers, AnyEvent has to dup() every file handle, as IO::Async fails to support the same or different file handles pointing to the same fd (the good thing is that it is documented, but why not fix it instead?).

Apart from these fatal flaws, there are a number of unpleasent properties that just need some mentioning:

- Confusing and misleading names

  Another rather negative point about this module family is its name, which is deeply confusing: Despite the "async" in the name, IO::Async only does *synchronous* I/O, there is nothing "asynchronous" about it whatsoever (when I first heard about it, I thought, "wow, a second async I/O module, what does it do compared to IO::AIO", and was somehow set back when I learned that the only "async" aspect of it is the name).

- Inconsistent, incomplete and convoluted API

  Implementing AnyEvent's rather simple timers on top of IO::Async's timers was a nightmare (try implementing a timer with configurable interval and delay value...).

  The method naming is chaotic: `watch_child` creates a child watcher, but `watch_io` is an internal method; `detach_signal` removes a signal watcher, but `detach_child` forks a subprocess and so on).

- Unpleasant surprises on GNU/Linux

  When you develop your program on FreeBSD and run it on GNU/Linux, you might have unpleasant surprises, as IO::Async::Loop will by default use IO::Async::Loop::Epoll, which is incompatible with `fork`, so your network server will run into spurious and very hard to debug problems under heavy load, as IO::Async forks a lot of processes, e.g. for DNS resolution. It would be better if IO::Async would only load "safe" backends by default (or fix the epoll backend to work in the presence of fork, which admittedly is hard - EV does it for you, and also does not use unsafe backends by default).

On the positive side, performance with IO::Async is quite good even in my very demanding eyes.

SEE ALSO

AnyEvent, IO::Async.

AUTHOR

```
Marc Lehmann <schmorp@schmorp.de>
http://anyevent.schmorp.de

Paul Evans <leonerd@leonerd.org.uk>
Rewrote the backend for IO::Async version 0.33.
```

AnyEvent::Impl::Irssi

AnyEvent adaptor for Irssi

SYNOPSIS

```
use AnyEvent;

# this module gets loaded automatically when running under irssi
```

DESCRIPTION

This module provides transparent support for AnyEvent. You don't have to do anything to make Irssi scripts work with AnyEvent.

Limitations of this backend and implementation details:

- This backend does not support blocking waits.

  That means you must set a callback on any condvars, or otherwise make sure to never call `recv` on a condvar that hasn't been signalled yet.

- Child exits will be handled by AnyEvent.

  AnyEvent will take over child handling, as Irssi only polls for children once/second and cannot handle unspecific child watchers.

  This *should* have no negative effect, as AnyEvent will emit a pidwait signal just like irssi itself would.

- Artificial timer delays.

  Irssi artificially enforces timers to have at least a 10ms delay (by croaking, even).

  This means that some applications will be limited to a rate of 100Hz (for example, *Coro::AnyEvent* thread scheduling).

- Irssi leaks memory like hell.

  Yeah.

Apart from that, documentation is notoriously wrong (e.g. file handles are not supported by `input_add`, contrary to documentation), hooking into irssi has to be done in... weird... ways, but otherwise, Irssi is surprisingly full-featured (for basically being a hack).

SEE ALSO

AnyEvent, Irssi.

AUTHOR

```
Marc Lehmann <schmorp@schmorp.de>
http://anyevent.schmorp.de
```

AnyEvent::Impl::POE

AnyEvent adaptor for POE

SYNOPSIS

```
use AnyEvent;
use POE;

# this module gets loaded automatically as required
```

DESCRIPTION

This module provides transparent support for AnyEvent. You don't have to do anything to make POE work with AnyEvent except by loading POE before creating the first AnyEvent watcher. There are some cases where POE will issue spurious (and non-suppressible) warnings. These can be avoided by loading AnyEvent::Impl::POE before loading any other modules using POE and AnyEvent, i.e. in your main program.

AnyEvent::Impl::POE will output some spurious message how to work around POE's spurious messages when it detects these cases.

Unfortunately, POE isn't generic enough to implement a fully working AnyEvent backend: POE is too badly designed, too badly documented and too badly implemented.

Here are the details, and what it means to you if you want to be interoperable with POE:

- Weird messages

  If you only use `run_one_timeslice` (as AnyEvent has to for its condition variables), POE will print an ugly, unsuppressible, message at program exit:

  ```
  Sessions were started, but POE::Kernel's run() method was never.
  ↪..
  ```

  The message is correct, the question is why POE prints it in the first place in a correct program (this is not a singular case though).

AnyEvent consequently patches the POE kernel so it thinks it already ran. Other workarounds, even the one cited in the POE documentation itself, have serious side effects, such as throwing away events.

The author of POE verified that this is indeed true, and has no plans to change this.

POE has other weird messages, and sometimes weird behaviour, for example, it doesn't support overloaded code references as callbacks for no apparent reason.

- One POE session per Event

  AnyEvent has to create one POE::Session per event watcher, which is immensely slow and makes watchers very large. The reason for this is lacking lifetime management (mostly undocumented, too). Without one session/watcher it is not possible to easily keep the kernel from running endlessly.

  This is not just a problem with the way AnyEvent has to interact with POE, but is a principal issue with POEs lifetime management (namely that stopping the kernel stops sessions, but AnyEvent has no control over who and when the kernel starts or stops w.r.t. AnyEvent watcher creation/destruction).

  From benchmark data it is not clear that session creation is that costly, though - the real inefficiencies with POE seem to come from other sources, such as event handling.

- One watcher per fd/event combo

  POE, of course, suffers from the same bug as Tk and some other badly designed event models in that it doesn't support multiple watchers per fd/poll combo. The workaround is the same as with Tk: AnyEvent::Impl::POE creates a separate file descriptor to hand to POE, which isn't fast and certainly not nice to your resources.

  Of course, without the workaround, POE also prints ugly messages again that say the program *might* be buggy.

  While this is not good to performance, at least regarding speed, with a modern Linux kernel, the overhead is actually quite small.

- Timing deficiencies

  POE manages to not have a function that returns the current time. This is extremely problematic, as POE can use different time functions, which can differ by more than a second - and user code is left guessing which one is used.

  In addition, most timer functions in POE want an absolute timestamp, which is hard to create if all you have is a relative time and no function to return the "current time".

  And of course POE doesn't handle time jumps at all (not even when using an event loop that happens to do that, such as *EV*, as it does its own unoptimised timer management).

  AnyEvent works around the unavailability of the current time using relative timers exclusively, in the hope that POE gets it right at least internally.

- Lack of defined event ordering

  POE cannot guarantee the order of callback invocation for timers, and usually gets it wrong. That is, if you have two timers, one timing out after another (all else being equal), the callbacks might be called in reverse order.

How one manages to even implement stuff that way escapes me.

- Child watchers

  POE offers child watchers - which is a laudable thing, as few event loops do. Unfortunately, they cannot even implement AnyEvent's simple child watchers: they are not generic enough (the POE implementation isn't even generic enough to let properly designed back-end use their native child watcher instead - it insist on doing it itself the broken way).

  Unfortunately, POE's child handling is inherently racy: if the child exits before the handler is created (because e.g. it crashes or simply is quick about it), then current versions of POE (1.352) will *never* invoke the child watcher, and there is nothing that can be done about it. Older versions of POE only delayed in this case. The reason is that POE first checks if the child has already exited, and *then* installs the signal handler - aa classical race.

  Your only hope is for the fork'ed process to not exit too quickly, in which case everything happens to work.

  Of course, whenever POE reaps an unrelated child it will also output a message for it that you cannot suppress (which shouldn't be too surprising at this point). Very professional.

  As a workaround, AnyEvent::Impl::POE will take advantage of undocumented behaviour in POE::Kernel to catch the status of all child processes, but it cannot guarantee delivery.

  How one manages to have such a glaring bug in an event loop after ten years of development escapes me.

  (There are more annoying bugs, for example, POE runs `waitpid` unconditionally at finaliser time, so your program will hang until all child processes have exited.)

- Documentation quality

  At the time of this writing, POE was in its tenth year. Still, its documentation is extremely lacking, making it impossible to implement stuff as trivial as AnyEvent watchers without having to resort to undocumented behaviour or features.

  For example, the POE::Kernel manpage has nine occurrences of the word TODO with an explanation of whats missing. In general, the POE man pages are littered with comments like "section not yet written".

  Some other gems:

```
This allows many object methods to also be package methods.
```

  This is nice, but since it doesn't document *which* methods these are, this is utterly useless information.

```
Terminal signals will kill sessions if they are not handled by a
"sig_handled"() call. The OS signals that usually kill or dump a
process are considered terminal in POE, but they never trigger a
coredump. These are: HUP, INT, QUIT and TERM.
```

  Although AnyEvent calls `sig_handled`, removing it has no apparent effects on POE handling SIGINT.

```
refcount_increment SESSION_ID, COUNTER_NAME
```

Nowhere is explained which COUNTER_NAMEs are valid and which aren't - not all scalars (or even strings) are valid counter names. Take your guess, failure is of course completely silent. I found this out the hard way, as the first name I came up with was silently ignored.

```
get_next_event_time() returns the time the next event is due,␣
↪in a form
compatible with the UNIX time() function.
```

And surely, one would hope that POE supports sub-second accuracy as documented elsewhere, unlike the explanation above implies. Yet:

```
POE::Kernel timers support subsecond accuracy, but don't expect␣
↪too
much here. Perl is not the right language for realtime␣
↪programming.
```

... of course, Perl is not the right language to expect sub-second accuracy - the manpage author must hate Perl to spread so much FUD in so little space. The Deliantra game server logs with 100µs-accuracy because Perl is fast enough to require this, and is still able to deliver map updates with little jitter at exactly the right time. It does not, however, use POE.

```
Furthermore, since the Kernel keeps track of everything␣
↪sessions do, it
knows when a session has run out of tasks to perform.
```

This is impossible - how does the kernel know that a session is no longer watching for some (external) event (e.g. by some other session)? It cannot, and therefore this is wrong - but you would be hard pressed to find out how to work around this and tell the kernel manually about such events.

It gets worse, though - the notion of "task" or "resource", although used throughout the documentation, is not defined in a usable way. For example, waiting for a timeout is considered to be a task, waiting for a signal is not (a session that only waits for a signal is considered finished and gets removed). The user is left guessing when waiting for an event counts as task and when not (in fact, the issue with signals is mentioned in passing in a section about child watchers and directly contradicts earlier parts in that document).

One could go on endlessly - ten years, no usable documentation.

It is likely that differences between documentation, or the one or two things I had to guess, cause unanticipated problems with this adaptor.

- Fragile and inconsistent API

  The POE API is extremely inconsistent - sometimes you have to pass a session argument, sometimes it gets ignored, sometimes a session-specific method must not use a session argument.

  Error handling is sub-standard as well: even for programming mistakes, POE does

not `croak` but, in most cases, just sets `$!` or simply does nothing at all, leading to fragile programs.

Sometimes registering a handler uses the "eventname, parameter" ordering (timeouts), sometimes it is "parameter, eventname" (signals). There is little consistency overall.

- Lack of knowledge

```
The IO::Poll event loop provides an alternative that␣
↪theoretically
scales better than select().
```

The IO::Poll "event loop" (who in his right mind would call that an event loop) of course scales about identically (sometimes it is a bit faster, sometimes a bit slower) to select in theory, and also in practise, of course, as both are O(n) in the number of file descriptors, which is rather bad.

This is just one place where it gets obvious how little the author of the POE manpage understands.

- No idle events

The POE-recommended workaround to this is apparently to use `fork`. Consequently, idle watchers will have to be emulated by AnyEvent.

- Questionable maintainer behaviour

The author of POE is known to fabricate statements and post these to public mailinglists - apparently, spreading FUD about competing (in his eyes) projects or their maintainers is acceptable to him.

This has (I believe) zero effects on the quality or usefulness of his code, but it does completely undermine his trustworthyness - so don't blindly believe anything he says, he might have just made it up to suit his needs (benchmark results, the names of my ten wifes, the length of my penis, etc. etc.). When in doubt, double-check - not just him, anybody actually.

Example: http://www.nntp.perl.org/group/perl.perl5.porters/2012/01/msg182141.html. I challenged him in that thread to provide evidence for his statement by giving at least two examples, but of course since he just made it up, he couldn't provide any evidence.

On the good side, AnyEvent allows you to write your modules in a 100% POE-compatible way (bug-for-bug compatible even), without forcing your module to use POE - it is still open to better event models, of which there are plenty.

Oh, and one other positive thing:

```
RUNNING_IN_HELL
```

POE knows about the nature of the beast!

SEE ALSO

AnyEvent, POE.

AUTHOR

```
Marc Lehmann <schmorp@schmorp.de>
http://anyevent.schmorp.de
```

AnyEvent::Impl::Perl

AnyEvent adaptor for AnyEvent's pure perl AnyEvent::Loop

SYNOPSIS

```
use AnyEvent;
use AnyEvent::Loop;
# this module gets loaded automatically as required
```

DESCRIPTION

This module provides transparent support for AnyEvent in case no other event loop could be found or loaded.

If you want to use this module instead of autoloading another event loop you can simply load *AnyEvent::Loop* before creating the first watcher.

Naturally, it supports all features of AnyEvent.

See *AnyEvent::Loop* for more details on performance characteristics.

SEE ALSO

AnyEvent.

AUTHOR

```
Marc Lehmann <schmorp@schmorp.de>
http://anyevent.schmorp.de
```

AnyEvent::Impl::Qt

AnyEvent adaptor for Qt

SYNOPSIS

```
use AnyEvent;
use Qt;

my $app = Qt::Application \@ARGV; # REQUIRED!

# this module gets loaded automatically as required
```

DESCRIPTION

This module provides transparent support for AnyEvent. You don't have to do anything to make Qt work with AnyEvent except by loading Qt before creating the first AnyEvent watcher *and instantiating the Qt::Application object*. Failure to do so will result in segfaults, which is why this model doesn't work as a default model and will not be autoprobed (but it will be autodetected when the main program uses Qt).

Qt suffers from the same limitations as Event::Lib and Tk, the workaround is also the same (duplicating file descriptors).

Qt doesn't support idle events, so they are being emulated.

Avoid Qt if you can.

SEE ALSO

AnyEvent, Qt.

AUTHOR

```
Marc Lehmann <schmorp@schmorp.de>
http://anyevent.schmorp.de
```

AnyEvent::Impl::Tk

AnyEvent adaptor for Tk

SYNOPSIS

```
use AnyEvent;
use Tk;

# this module gets loaded automatically as required
```

DESCRIPTION

This module provides transparent support for AnyEvent. You don't have to do anything to make Tk work with AnyEvent except by loading Tk before creating the first AnyEvent watcher.

Tk is buggy. Tk is extremely buggy. Tk is so unbelievably buggy that for each bug reported and fixed, you get one new bug followed by reintroduction of the old bug in a

later revision. It is also basically unmaintained: the maintainers are not even interested in improving the situation - reporting bugs is considered rude, and fixing bugs is considered changing holy code, so it's apparently better to leave it broken.

I regularly run out of words to describe how bad it really is.

To work around some of the many, many bugs in Tk that don't get fixed, this adaptor dup()'s all filehandles that get passed into its I/O watchers, so if you register a read and a write watcher for one fh, AnyEvent will create two additional file descriptors (and handles).

This creates a high overhead and is slow, but seems to work around most known bugs in Tk::fileevent on 32 bit architectures (Tk seems to be terminally broken on 64 bit, do not expect more than 10 or so watchers to work on 64 bit machines).

Do not expect these workarounds to avoid segfaults and crashes inside Tk.

Note also that Tk event ids wrap around after 2**32 or so events, which on my machine can happen within less than 12 hours, after which Tk will stomp on random other events and kill them. So don't run Tk programs for more than an hour or so.

To be able to access the Tk event loop, this module creates a main window and withdraws it immediately. This might cause flickering on some platforms, but Tk perversely requires a window to be able to wait for file handle readyness notifications. This window is always created (in this version of AnyEvent) and can be accessed as `$AnyEvent::Impl::Tk::mw`.

SEE ALSO

AnyEvent, Tk.

AUTHOR

```
Marc Lehmann <schmorp@schmorp.de>
http://anyevent.schmorp.de
```

AnyEvent::Impl::UV

AnyEvent adaptor for UV

SYNOPSIS

```
use AnyEvent;
use UV;

# this module gets loaded automatically as required
```

DESCRIPTION

This module provides transparent support for AnyEvent. You don't have to do anything to make UV work with AnyEvent except by loading UV before creating the first AnyEvent watcher.

SEE ALSO

AnyEvent, UV.

AUTHOR

```
Mike Lowell <mikedotlowell@gmail.com>
```

AnyEvent::Log

simple logging "framework"

SYNOPSIS

Simple uses:

```
use AnyEvent;

AE::log fatal => "No config found, cannot continue!"; # never␣
 ↪returns
AE::log alert => "The battery died!";
AE::log crit  => "The battery is too hot!";
AE::log error => "Division by zero attempted.";
AE::log warn  => "Couldn't delete the file.";
AE::log note  => "Attempted to create config, but config already␣
 ↪exists.";
AE::log info  => "File soandso successfully deleted.";
AE::log debug => "the function returned 3";
AE::log trace => "going to call function abc";
```

Log level overview:

```
LVL  NAME      SYSLOG   PERL  NOTE
 1   fatal     emerg    exit  system unusable, aborts program!
 2   alert                    failure in primary system
 3   critical  crit             failure in backup system
 4   error     err      die   non-urgent program errors, a bug
 5   warn      warning          possible problem, not necessarily error
 6   note      notice           unusual conditions
 7   info                       normal messages, no action required
 8   debug                      debugging messages for development
 9   trace                      copious tracing output
```

"Complex" uses (for speed sensitive code, e.g. trace/debug messages):

```
use AnyEvent::Log;

my $tracer = AnyEvent::Log::logger trace => \my $trace;

$tracer->("i am here") if $trace;
$tracer->(sub { "lots of data: " . Dumper $self }) if $trace;
```

Configuration (also look at the EXAMPLES section):

```
# set default logging level to suppress anything below "notice"
# i.e. enable logging at "notice" or above - the default is to
# to not log anything at all.
$AnyEvent::Log::FILTER->level ("notice");

# set logging for the current package to errors and higher only
AnyEvent::Log::ctx->level ("error");

# enable logging for the current package, regardless of global␣
 ↪logging level
AnyEvent::Log::ctx->attach ($AnyEvent::Log::LOG);

# enable debug logging for module some::mod and enable logging by␣
 ↪default
(AnyEvent::Log::ctx "some::mod")->level ("debug");
(AnyEvent::Log::ctx "some::mod")->attach ($AnyEvent::Log::LOG);

# send all critical and higher priority messages to syslog,
# regardless of (most) other settings
$AnyEvent::Log::COLLECT->attach (new AnyEvent::Log::Ctx
   level         => "critical",
   log_to_syslog => "user",
);
```

DESCRIPTION

This module implements a relatively simple “logging framework”. It doesn’t attempt to be “the” logging solution or even “a” logging solution for AnyEvent - AnyEvent simply creates logging messages internally, and this module more or less exposes the mechanism, with some extra spiff to allow using it from other modules as well.

Remember that the default verbosity level is 4 (`error`), so only errors and more important messages will be logged, unless you set `PERL_ANYEVENT_VERBOSE` to a higher number before starting your program (`AE_VERBOSE=5` is recommended during development), or change the logging level at runtime with something like:

```
use AnyEvent::Log;
$AnyEvent::Log::FILTER->level ("info");
```

The design goal behind this module was to keep it simple (and small), but make it powerful enough to be potentially useful for any module, and extensive enough for the most common tasks, such as logging to multiple targets, or being able to log into a database.

The module is also usable before AnyEvent itself is initialised, in which case some of the functionality might be reduced.

The amount of documentation might indicate otherwise, but the runtime part of the module is still just below 300 lines of code.

LOGGING LEVELS

Logging levels in this module range from 1 (highest priority) to 9 (lowest priority). Note that the lowest numerical value is the highest priority, so when this document says "higher priority" it means "lower numerical value".

Instead of specifying levels by name you can also specify them by aliases:

```
LVL NAME      SYSLOG   PERL  NOTE
 1  fatal     emerg    exit  system unusable, aborts program!
 2  alert                    failure in primary system
 3  critical  crit             failure in backup system
 4  error     err      die   non-urgent program errors, a bug
 5  warn      warning          possible problem, not necessarily error
 6  note      notice           unusual conditions
 7  info                       normal messages, no action required
 8  debug                      debugging messages for development
 9  trace                      copious tracing output
```

As you can see, some logging levels have multiple aliases - the first one is the "official" name, the second one the "syslog" name (if it differs) and the third one the "perl" name, suggesting (only!) that you log `die` messages at `error` priority. The NOTE column tries to provide some rationale on how to chose a logging level.

As a rough guideline, levels 1..3 are primarily meant for users of the program (admins, staff), and are the only ones logged to STDERR by default. Levels 4..6 are meant for users and developers alike, while levels 7..9 are usually meant for developers.

You can normally only log a message once at highest priority level (`1`, `fatal`), because logging a fatal message will also quit the program - so use it sparingly :)

For example, a program that finds an unknown switch on the commandline might well use a fatal logging level to tell users about it - the "system" in this case would be the program, or module.

Some methods also offer some extra levels, such as `0`, `off`, none or `all` - these are only valid for the methods that documented them.

LOGGING FUNCTIONS

The following functions allow you to log messages. They always use the caller's package as a "logging context". Also, the main logging function, `log`, is aliased to `AnyEvent::log` and `AE::log` when the `AnyEvent` module is loaded.

- AnyEvent::Log::log $level, $msg[, @args]

  Requests logging of the given `$msg` with the given log level, and returns true if the message was logged *somewhere*.

  For loglevel `fatal`, the program will abort.

If only a `$msg` is given, it is logged as-is. With extra `@args`, the `$msg` is interpreted as an sprintf format string.

The `$msg` should not end with `\n`, but may if that is convenient for you. Also, multiline messages are handled properly.

Last not least, `$msg` might be a code reference, in which case it is supposed to return the message. It will be called only then the message actually gets logged, which is useful if it is costly to create the message in the first place.

This function takes care of saving and restoring `$!` and `$@`, so you don't have to.

Whether the given message will be logged depends on the maximum log level and the caller's package. The return value can be used to ensure that messages or not "lost" - for example, when *AnyEvent::Debug* detects a runtime error it tries to log it at `die` level, but if that message is lost it simply uses warn.

Note that you can (and should) call this function as `AnyEvent::log` or `AE::log`, without `use`-ing this module if possible (i.e. you don't need any additional functionality), as those functions will load the logging module on demand only. They are also much shorter to write.

Also, if you optionally generate a lot of debug messages (such as when tracing some code), you should look into using a logger callback and a boolean enabler (see `logger`, below).

Example: log something at error level.

```
AE::log error => "something";
```

Example: use printf-formatting.

```
AE::log info => "%5d %-10.10s %s", $index, $category, $msg;
```

Example: only generate a costly dump when the message is actually being logged.

```
AE::log debug => sub { require Data::Dump; Data::Dump::dump \
 ↪%cache };
```

- $logger = AnyEvent::Log::logger $level[, \$enabled]

  Creates a code reference that, when called, acts as if the `AnyEvent::Log::log` function was called at this point with the given level. `$logger` is passed a `$msg` and optional `@args`, just as with the `AnyEvent::Log::log` function:

  ```
  my $debug_log = AnyEvent::Log::logger "debug";

  $debug_log->("debug here");
  $debug_log->("%06d emails processed", 12345);
  $debug_log->(sub { $obj->as_string });
  ```

  The idea behind this function is to decide whether to log before actually logging - when the `logger` function is called once, but the returned logger callback often, then this can be a tremendous speed win.

  Despite this speed advantage, changes in logging configuration will still be reflected by the logger callback, even if configuration changes *after* it was created.

To further speed up logging, you can bind a scalar variable to the logger, which contains true if the logger should be called or not - if it is false, calling the logger can be safely skipped. This variable will be updated as long as `$logger` is alive.

Full example:

```
# near the init section
use AnyEvent::Log;

my $debug_log = AnyEvent::Log::logger debug => \my $debug;

# and later in your program
$debug_log->("yo, stuff here") if $debug;

$debug and $debug_log->("123");
```

- AnyEvent::Log::exact_time $on

  By default, `AnyEvent::Log` will use `AE::now`, i.e. the cached eventloop time, for the log timestamps. After calling this function with a true value it will instead resort to `AE::time`, i.e. fetch the current time on each log message. This only makes a difference for event loops that actually cache the time (such as *EV* or *AnyEvent::Loop*).

  This setting can be changed at any time by calling this function.

  Since `AnyEvent::Log` has to work even before the *AnyEvent* has been initialised, this switch will also decide whether to use `CORE::time` or `Time::HiRes::time` when logging a message before *AnyEvent* becomes available.

- AnyEvent::Log::format_time $timestamp

  Formats a timestamp as returned by `AnyEvent->now` or `AnyEvent->time` or many other functions in the same way as `AnyEvent::Log` does.

  In your main program (as opposed to in your module) you can override the default timestamp display format by loading this module and then redefining this function.

  Most commonly, this function can be used in formatting callbacks.

- AnyEvent::Log::default_format $time, $ctx, $level, $msg

  Format a log message using the given timestamp, logging context, log level and log message.

  This is the formatting function used to format messages when no custom function is provided.

  In your main program (as opposed to in your module) you can override the default message format by loading this module and then redefining this function.

- AnyEvent::Log::fatal_exit()

  This is the function that is called after logging a `fatal` log message. It must not return.

  The default implementation simply calls `exit 1`.

  In your main program (as opposed to in your module) you can override the fatal exit function by loading this module and then redefining this function. Make sure

you don't return.

LOGGING CONTEXTS

This module associates every log message with a so-called *logging context*, based on the package of the caller. Every perl package has its own logging context.

A logging context has three major responsibilities: filtering, logging and propagating the message.

For the first purpose, filtering, each context has a set of logging levels, called the log level mask. Messages not in the set will be ignored by this context (masked).

For logging, the context stores a formatting callback (which takes the timestamp, context, level and string message and formats it in the way it should be logged) and a logging callback (which is responsible for actually logging the formatted message and telling `AnyEvent::Log` whether it has consumed the message, or whether it should be propagated).

For propagation, a context can have any number of attached *slave contexts*. Any message that is neither masked by the logging mask nor masked by the logging callback returning true will be passed to all slave contexts.

Each call to a logging function will log the message at most once per context, so it does not matter (much) if there are cycles or if the message can arrive at the same context via multiple paths.

DEFAULTS

By default, all logging contexts have an full set of log levels ("all"), a disabled logging callback and the default formatting callback.

Package contexts have the package name as logging title by default.

They have exactly one slave - the context of the "parent" package. The parent package is simply defined to be the package name without the last component, i.e. `AnyEvent::Debug::Wrapped` becomes `AnyEvent::Debug`, and `AnyEvent` becomes ... `$AnyEvent::Log::COLLECT` which is the exception of the rule - just like the "parent" of any single-component package name in Perl is `main`, the default slave of any top-level package context is `$AnyEvent::Log::COLLECT`.

Since perl packages form only an approximate hierarchy, this slave context can of course be removed.

All other (anonymous) contexts have no slaves and an empty title by default.

When the module is loaded it creates the `$AnyEvent::Log::LOG` logging context that simply logs everything via `warn`, without propagating anything anywhere by default. The purpose of this context is to provide a convenient place to override the global logging target or to attach additional log targets. It's not meant for filtering.

It then creates the `$AnyEvent::Log::FILTER` context whose purpose is to suppress all messages with priority higher than `$ENV{PERL_ANYEVENT_VERBOSE}`. It then attached the `$AnyEvent::Log::LOG` context to it. The purpose of the filter context is to simply provide filtering according to some global log level.

Finally it creates the top-level package context `$AnyEvent::Log::COLLECT` and attaches the `$AnyEvent::Log::FILTER` context to it, but otherwise leaves it at default config. Its purpose is simply to collect all log messages system-wide.

The hierarchy is then:

```
any package, eventually -> $COLLECT -> $FILTER -> $LOG
```

The effect of all this is that log messages, by default, wander up to the `$AnyEvent::Log::COLLECT` context where all messages normally end up, from there to `$AnyEvent::Log::FILTER` where log messages with lower priority then `$ENV{PERL_ANYEVENT_VERBOSE}` will be filtered out and then to the `$AnyEvent::Log::LOG` context to be passed to `warn`.

This makes it easy to set a global logging level (by modifying $FILTER), but still allow other contexts to send, for example, their debug and trace messages to the $LOG target despite the global logging level, or to attach additional log targets that log messages, regardless of the global logging level.

It also makes it easy to modify the default warn-logger ($LOG) to something that logs to a file, or to attach additional logging targets (such as loggign to a file) by attaching it to $FILTER.

CREATING/FINDING/DESTROYING CONTEXTS

- $ctx = AnyEvent::Log::ctx [$pkg]

  This function creates or returns a logging context (which is an object).

  If a package name is given, then the context for that package is returned. If it is called without any arguments, then the context for the callers package is returned (i.e. the same context as a `AE::log` call would use).

  If `undef` is given, then it creates a new anonymous context that is not tied to any package and is destroyed when no longer referenced.

- AnyEvent::Log::reset

  Resets all package contexts and recreates the default hierarchy if necessary, i.e. resets the logging subsystem to defaults, as much as possible. This process keeps references to contexts held by other parts of the program intact.

  This can be used to implement config-file (re-)loading: before loading a configuration, reset all contexts.

- $ctx = new AnyEvent::Log::Ctx methodname => param...

  This is a convenience constructor that makes it simpler to construct anonymous logging contexts.

  Each key-value pair results in an invocation of the method of the same name as the key with the value as parameter, unless the value is an arrayref, in which case it calls the method with the contents of the array. The methods are called in the same order as specified.

  Example: create a new logging context and set both the default logging level, some slave contexts and a logging callback.

```
$ctx = new AnyEvent::Log::Ctx
   title    => "dubious messages",
   level    => "error",
   log_cb   => sub { print STDOUT shift; 0 },
   slaves   => [$ctx1, $ctx, $ctx2],
;
```

CONFIGURING A LOG CONTEXT

The following methods can be used to configure the logging context.

- $ctx->title ([$new_title])

  Returns the title of the logging context - this is the package name, for package contexts, and a user defined string for all others.

  If `$new_title` is given, then it replaces the package name or title.

LOGGING LEVELS

The following methods deal with the logging level set associated with the log context.

The most common method to use is probably `$ctx->level ($level)`, which configures the specified and any higher priority levels.

All functions which accept a list of levels also accept the special string `all` which expands to all logging levels.

- $ctx->levels ($level[, $level...)

  Enables logging for the given levels and disables it for all others.

- $ctx->level ($level)

  Enables logging for the given level and all lower level (higher priority) ones. In addition to normal logging levels, specifying a level of `0` or `off` disables all logging for this level.

  Example: log warnings, errors and higher priority messages.

  ```
  $ctx->level ("warn");
  $ctx->level (5); # same thing, just numeric
  ```

- $ctx->enable ($level[, $level...])

  Enables logging for the given levels, leaving all others unchanged.

- $ctx->disable ($level[, $level...])

  Disables logging for the given levels, leaving all others unchanged.

- $ctx->cap ($level)

  Caps the maximum priority to the given level, for all messages logged to, or passing through, this context. That is, while this doesn't affect whether a message is logged or passed on, the maximum priority of messages will be limited to the specified level - messages with a higher priority will be set to the specified priority.

Another way to view this is that `->level` filters out messages with a too low priority, while `->cap` modifies messages with a too high priority.

This is useful when different log targets have different interpretations of priority. For example, for a specific command line program, a wrong command line switch might well result in a `fatal` log message, while the same message, logged to syslog, is likely *not* fatal to the system or syslog facility as a whole, but more likely a mere `error`.

This can be modeled by having a stderr logger that logs messages "as-is" and a syslog logger that logs messages with a level cap of, say, `error`, or, for truly system-critical components, actually `critical`.

SLAVE CONTEXTS

The following methods attach and detach another logging context to a logging context.

Log messages are propagated to all slave contexts, unless the logging callback consumes the message.

- $ctx->attach ($ctx2[, $ctx3...])

  Attaches the given contexts as slaves to this context. It is not an error to add a context twice (the second add will be ignored).

  A context can be specified either as package name or as a context object.

- $ctx->detach ($ctx2[, $ctx3...])

  Removes the given slaves from this context - it's not an error to attempt to remove a context that hasn't been added.

  A context can be specified either as package name or as a context object.

- $ctx->slaves ($ctx2[, $ctx3...])

  Replaces all slaves attached to this context by the ones given.

LOG TARGETS

The following methods configure how the logging context actually does the logging (which consists of formatting the message and printing it or whatever it wants to do with it).

- $ctx->log_cb ($cb->($str))

  Replaces the logging callback on the context (`undef` disables the logging callback).

  The logging callback is responsible for handling formatted log messages (see `fmt_cb` below) - normally simple text strings that end with a newline (and are possibly multiline themselves).

  It also has to return true iff it has consumed the log message, and false if it hasn't. Consuming a message means that it will not be sent to any slave context. When in doubt, return `0` from your logging callback.

  Example: a very simple logging callback, simply dump the message to STDOUT and do not consume it.

```
$ctx->log_cb (sub { print STDERR shift; 0 });
```

You can filter messages by having a log callback that simply returns 1 and does not do anything with the message, but this counts as "message being logged" and might not be very efficient.

Example: propagate all messages except for log levels "debug" and "trace". The messages will still be generated, though, which can slow down your program.

```
$ctx->levels ("debug", "trace");
$ctx->log_cb (sub { 1 }); # do not log, but eat debug and trace
 ↪messages
```

- $ctx->fmt_cb ($fmt_cb->($timestamp, $orig_ctx, $level, $message))

  Replaces the formatting callback on the context (`undef` restores the default formatter).

  The callback is passed the (possibly fractional) timestamp, the original logging context (object, not title), the (numeric) logging level and the raw message string and needs to return a formatted log message. In most cases this will be a string, but it could just as well be an array reference that just stores the values.

  If, for some reason, you want to use `caller` to find out more about the logger then you should walk up the call stack until you are no longer inside the `AnyEvent::Log` package.

  To implement your own logging callback, you might find the `AnyEvent::Log::format_time` and `AnyEvent::Log::default_format` functions useful.

  Example: format the message just as AnyEvent::Log would, by letting AnyEvent::Log do the work. This is a good basis to design a formatting callback that only changes minor aspects of the formatting.

```
$ctx->fmt_cb (sub {
   my ($time, $ctx, $lvl, $msg) = @_;

   AnyEvent::Log::default_format $time, $ctx, $lvl, $msg
});
```

  Example: format just the raw message, with numeric log level in angle brackets.

```
$ctx->fmt_cb (sub {
   my ($time, $ctx, $lvl, $msg) = @_;

   "<$lvl>$msg\n"
});
```

  Example: return an array reference with just the log values, and use `PApp::SQL::sql_exec` to store the message in a database.

```
$ctx->fmt_cb (sub { \@_ });
$ctx->log_cb (sub {
```

(continues on next page)

(continued from previous page)

```
   my ($msg) = @_;

   sql_exec "insert into log (when, subsys, prio, msg) values (?
↪, ?, ?, ?)",
            $msg->[0] + 0,
            "$msg->[1]",
            $msg->[2] + 0,
            "$msg->[3]";

   0
});
```

- $ctx->log_to_warn

  Sets the `log_cb` to simply use `CORE::warn` to report any messages (usually this logs to STDERR).

- $ctx->log_to_file ($path)

  Sets the `log_cb` to log to a file (by appending), unbuffered. The function might return before the log file has been opened or created.

- $ctx->log_to_path ($path)

  Same as `->log_to_file`, but opens the file for each message. This is much slower, but allows you to change/move/rename/delete the file at basically any time.

  Needless(?) to say, if you do not want to be bitten by some evil person calling `chdir`, the path should be absolute. Doesn't help with `chroot`, but hey...

- $ctx->log_to_syslog ([$facility])

  Logs all messages via *Sys::Syslog*, mapping `trace` to debug and all the others in the obvious way. If specified, then the `$facility` is used as the facility (`user`, `auth`, `local0` and so on). The default facility is `user`.

  Note that this function also sets a `fmt_cb` - the logging part requires an array reference with [$level, $str] as input.

MESSAGE LOGGING

These methods allow you to log messages directly to a context, without going via your package context.

- $ctx->log ($level, $msg[, @params])

  Same as `AnyEvent::Log::log`, but uses the given context as log context.

  Example: log a message in the context of another package.

```
(AnyEvent::Log::ctx "Other::Package")->log (warn => "heely bo");
```

- $logger = $ctx->logger ($level[, \$enabled])

  Same as `AnyEvent::Log::logger`, but uses the given context as log context.

CONFIGURATION VIA $ENV{PERL_ANYEVENT_LOG}

Logging can also be configured by setting the environment variable `PERL_ANYEVENT_LOG` (or `AE_LOG`).

The value consists of one or more logging context specifications separated by : or whitespace. Each logging specification in turn starts with a context name, followed by =, followed by zero or more comma-separated configuration directives, here are some examples:

```
# set default logging level
filter=warn

# log to file instead of to stderr
log=file=/tmp/mylog

# log to file in addition to stderr
log=+%file:%file=file=/tmp/mylog

# enable debug log messages, log warnings and above to syslog
filter=debug:log=+%warnings:%warnings=warn,syslog=LOG_LOCAL0

# log trace messages (only) from AnyEvent::Debug to file
AnyEvent::Debug=+%trace:%trace=only,trace,file=/tmp/tracelog
```

A context name in the log specification can be any of the following:

- `collect`, `filter`, `log`

  Correspond to the three predefined `$AnyEvent::Log::COLLECT`, `AnyEvent::Log::FILTER` and `$AnyEvent::Log::LOG` contexts.

- `%name`

  Context names starting with a % are anonymous contexts created when the name is first mentioned. The difference to package contexts is that by default they have no attached slaves.

  This makes it possible to create new log contexts that can be refered to multiple times by name within the same log specification.

- a perl package name

  Any other string references the logging context associated with the given Perl `package`. In the unlikely case where you want to specify a package context that matches on of the other context name forms, you can add a `::` to the package name to force interpretation as a package.

The configuration specifications can be any number of the following:

- `stderr`

  Configures the context to use Perl's `warn` function (which typically logs to `STDERR`). Works like `log_to_warn`.

- `file=`*path*

  Configures the context to log to a file with the given path. Works like `log_to_file`.

- `path=`*path*

  Configures the context to log to a file with the given path. Works like `log_to_path`.

- `syslog` or `syslog=`*expr*

  Configures the context to log to syslog. If *expr* is given, then it is evaluated in the *Sys::Syslog* package, so you could use:

```
log=syslog=LOG_LOCAL0
```

- `nolog`

  Configures the context to not log anything by itself, which is the default. Same as `$ctx->log_cb (undef)`.

- `cap=`*level*

  Caps logging messages entering this context at the given level, i.e. reduces the priority of messages with higher priority than this level. The default is `0` (or `off`), meaning the priority will not be touched.

- `0` or `off`

  Sets the logging level of the context to `0`, i.e. all messages will be filtered out.

- `all`

  Enables all logging levels, i.e. filtering will effectively be switched off (the default).

- `only`

  Disables all logging levels, and changes the interpretation of following level specifications to enable the specified level only.

  Example: only enable debug messages for a context.

```
context=only,debug
```

- `except`

  Enables all logging levels, and changes the interpretation of following level specifications to disable that level. Rarely used.

  Example: enable all logging levels except fatal and trace (this is rather nonsensical).

```
filter=exept,fatal,trace
```

- `level`

  Enables all logging levels, and changes the interpretation of following level specifications to be “that level or any higher priority message”. This is the default.

  Example: log anything at or above warn level.

```
filter=warn

# or, more verbose
filter=only,level,warn
```

- 1..9 or a logging level name (`error`, debug etc.)

  A numeric loglevel or the name of a loglevel will be interpreted according to the most recent `only`, `except` or `level` directive. By default, specifying a logging level enables that and any higher priority messages.

- *+context*

  Attaches the named context as slave to the context.

- +

  A lone + detaches all contexts, i.e. clears the slave list from the context. Anonymous (`%name`) contexts have no attached slaves by default, but package contexts have the parent context as slave by default.

  Example: log messages from My::Module to a file, do not send them to the default log collector.

  ```
  My::Module=+,file=/tmp/mymodulelog
  ```

Any character can be escaped by prefixing it with a `\` (backslash), as usual, so to log to a file containing a comma, colon, backslash and some spaces in the filename, you would do this:

```
PERL_ANYEVENT_LOG='log=file=/some\ \:file\ with\,\ \\-escapes'
```

Since whitespace (which includes newlines) is allowed, it is fine to specify multiple lines in `PERL_ANYEVENT_LOG`, e.g.:

```
PERL_ANYEVENT_LOG="
   filter=warn
   AnyEvent::Debug=+%trace
   %trace=only,trace,+log
" myprog
```

Also, in the unlikely case when you want to concatenate specifications, use whitespace as separator, as `::` will be interpreted as part of a module name, an empty spec with two separators:

```
PERL_ANYEVENT_LOG="$PERL_ANYEVENT_LOG MyMod=debug"
```

EXAMPLES

This section shows some common configurations, both as code, and as `PERL_ANYEVENT_LOG` string.

- Setting the global logging level.

  Either put `PERL_ANYEVENT_VERBOSE=<number>` into your environment before running your program, use `PERL_ANYEVENT_LOG` or modify the log level of the root context at runtime:

  ```
  PERL_ANYEVENT_VERBOSE=5 ./myprog
  ```

(continues on next page)

(continued from previous page)

```
PERL_ANYEVENT_LOG=log=warn

$AnyEvent::Log::FILTER->level ("warn");
```

- Append all messages to a file instead of sending them to STDERR.

  This is affected by the global logging level.

```
$AnyEvent::Log::LOG->log_to_file ($path);

PERL_ANYEVENT_LOG=log=file=/some/path
```

- Write all messages with priority `error` and higher to a file.

  This writes them only when the global logging level allows it, because it is attached to the default context which is invoked *after* global filtering.

```
$AnyEvent::Log::FILTER->attach (
   new AnyEvent::Log::Ctx log_to_file => $path);

PERL_ANYEVENT_LOG=filter=+%filelogger:%filelogger=file=/some/
 ↪path
```

  This writes them regardless of the global logging level, because it is attached to the toplevel context, which receives all messages *before* the global filtering.

```
$AnyEvent::Log::COLLECT->attach (
   new AnyEvent::Log::Ctx log_to_file => $path);

PERL_ANYEVENT_LOG=%filelogger=file=/some/path:collect=+
 ↪%filelogger
```

  In both cases, messages are still written to STDERR.

- Additionally log all messages with `warn` and higher priority to `syslog`, but cap at `error`.

  This logs all messages to the default log target, but also logs messages with priority `warn` or higher (and not filtered otherwise) to syslog facility `user`. Messages with priority higher than `error` will be logged with level `error`.

```
$AnyEvent::Log::LOG->attach (
   new AnyEvent::Log::Ctx
      level  => "warn",
      cap    => "error",
      syslog => "user",
);

PERL_ANYEVENT_LOG=log=+%syslog:%syslog=warn,cap=error,syslog
```

- Write trace messages (only) from *AnyEvent::Debug* to the default logging target(s).

  Attach the `$AnyEvent::Log::LOG` context to the `AnyEvent::Debug` context - this simply circumvents the global filtering for trace messages.

```
my $debug = AnyEvent::Debug->AnyEvent::Log::ctx;
$debug->attach ($AnyEvent::Log::LOG);

PERL_ANYEVENT_LOG=AnyEvent::Debug=+log
```

This of course works for any package, not just *AnyEvent::Debug*, but assumes the log level for AnyEvent::Debug hasn't been changed from the default.

ASYNCHRONOUS DISK I/O

This module uses *AnyEvent::IO* to actually write log messages (in `log_to_file` and `log_to_path`), so it doesn't block your program when the disk is busy and a non-blocking *AnyEvent::IO* backend is available.

AUTHOR

```
Marc Lehmann <schmorp@schmorp.de>
http://anyevent.schmorp.de
```

AnyEvent::Loop

AnyEvent's Pure-Perl event loop

SYNOPSIS

```
use AnyEvent;
# use AnyEvent::Loop;

# this module gets loaded automatically when no other loop can be
 ↪found

# Explicit use:
use AnyEvent::Loop;
use AnyEvent;

...

AnyEvent::Loop::run; # run the event loop
```

DESCRIPTION

This module provides an event loop for AnyEvent in case no other event loop could be found or loaded. You don't have to do anything to make it work with AnyEvent except by possibly loading it before creating the first AnyEvent watcher.

This module is *not* some loop abstracion used by AnyEvent, but just another event loop like EV or Glib, just written in pure perl and delivered with AnyEvent, so AnyEvent always works, even in the absence of any other backend.

If you want to use this module instead of autoloading a potentially better event loop you can simply load it (and no other event loops) before creating the first watcher.

As for performance, this module is on par with (and usually faster than) most select/poll-based C event modules such as Event or Glib (it does not even come close to EV, though), with respect to I/O watchers. Timers are handled less optimally, but for many common tasks, it is still on par with event loops written in C.

This event loop has been optimised for the following use cases:

- monotonic clock is available

  This module will use the POSIX monotonic clock option (if it can be detected at runtime) or the POSIX `times` function (if the resolution is at least 100Hz), in which case it will not suffer adversely from time jumps.

  If no monotonic clock is available, this module will not attempt to correct for time jumps in any way.

  The clock chosen will be reported if the environment variable `$PERL_ANYEVENT_VERBOSE` is set to 8 or higher.

- any number of watchers on one fd

  Supporting a large number of watchers per fd is purely a dirty benchmark optimisation not relevant in practise. The more common case of having one watcher per fd/poll combo is special-cased, however, and therefore fast, too.

- relatively few active fds per `select` call

  This module expects that only a tiny amount of fds is active at any one time. This is relatively typical of larger servers (but not the case where `select` traditionally is fast), at the expense of the “dense activity case” where most of the fds are active (which suits `select`).

  The optimal implementation of the “dense” case is not much faster, though, so the module should behave very well in most cases, subject to the bad scalability of `select` in the presence of a large number of inactive file descriptors.

- lots of timer changes/iteration, or none at all

  This module sorts the timer list using perl’s `sort`, even though a total ordering is not required for timers internally.

  This sorting is expensive, but means sorting can be avoided unless the timer list has changed in a way that requires a new sort.

  This means that adding lots of timers is very efficient, as well as not changing the timers. Advancing timers (e.g. recreating a timeout watcher on activity) is also relatively efficient, for example, if you have a large number of timeout watchers that time out after 10 seconds, then the timer list will be sorted only once every 10 seconds.

  This should not have much of an impact unless you have hundreds or thousands of timers, though, or your timers have very small timeouts.

FUNCTIONS

The only user-visible functions provided by this module loop related - watchers are created via the normal AnyEvent mechanisms.

- AnyEvent::Loop::run

  Run the event loop, usually the last thing done in the main program when you want to use the pure-perl backend.

- AnyEvent::Loop::one_event

  Blocks until at least one new event has been received by the operating system, whether or not it was AnyEvent-related.

SEE ALSO

AnyEvent.

AUTHOR

```
Marc Lehmann <schmorp@schmorp.de>
http://anyevent.schmorp.de
```

AnyEvent::Socket

useful IPv4 and IPv6 stuff. also unix domain sockets. and stuff.

SYNOPSIS

```
use AnyEvent::Socket;

tcp_connect "gameserver.deliantra.net", 13327, sub {
   my ($fh) = @_
      or die "gameserver.deliantra.net connect failed: $!";

   # enjoy your filehandle
};

# a simple tcp server
tcp_server undef, 8888, sub {
   my ($fh, $host, $port) = @_;

   syswrite $fh, "The internet is full, $host:$port. Go away!\015\
 ↪012";
};
```

DESCRIPTION

This module implements various utility functions for handling internet protocol addresses and sockets, in an as transparent and simple way as possible.

All functions documented without `AnyEvent::Socket::` prefix are exported by default.

- $ipn = parse_ipv4 $dotted_quad

  Tries to parse the given dotted quad IPv4 address and return it in octet form (or undef when it isn't in a parsable format). Supports all forms specified by POSIX (e.g. `10.0.0.1`, `10.1`, `10.0x020304`, `0x12345678` or `0377.0377.0377.0377`).

- $ipn = parse_ipv6 $textual_ipv6_address

  Tries to parse the given IPv6 address and return it in octet form (or undef when it isn't in a parsable format).

  Should support all forms specified by RFC 2373 (and additionally all IPv4 forms supported by parse_ipv4). Note that scope-id's are not supported (and will not parse).

  This function works similarly to `inet_pton AF_INET6, ...`.

  Example:

```
print unpack "H*", parse_ipv6 "2002:5345::10.0.0.1";
# => 2002534500000000000000000a000001

print unpack "H*", parse_ipv6 "192.89.98.1";
# => 00000000000000000000ffffc0596201
```

- $token = parse_unix $hostname

  This function exists mainly for symmetry to the other `parse_protocol` functions - it takes a hostname and, if it is `unix/`, it returns a special address token, otherwise `undef`.

  The only use for this function is probably to detect whether a hostname matches whatever AnyEvent uses for unix domain sockets.

- $ipn = parse_address $ip

  Combines `parse_ipv4`, `parse_ipv6` and `parse_unix` in one function. The address here refers to the host address (not socket address) in network form (binary).

  If the `$text` is `unix/`, then this function returns a special token recognised by the other functions in this module to mean "UNIX domain socket".

  If the `$text` to parse is a plain IPv4 or mapped IPv4 in IPv6 address (:ffff::<ipv4>), then it will be treated as an IPv4 address and four octets will be returned. If you don't want that, you have to call `parse_ipv4` and/or `parse_ipv6` manually (the latter always returning a 16 octet IPv6 address for mapped IPv4 addresses).

  Example:

```
print unpack "H*", parse_address "10.1.2.3";
# => 0a010203
```

- $ipn = AnyEvent::Socket::aton $ip

  Same as `parse_address`, but not exported (think `Socket::inet_aton` but *without* name resolution).

- ($name, $aliases, $proto) = getprotobyname $name

  Works like the builtin function of the same name, except it tries hard to work even on broken platforms (well, that's windows), where getprotobyname is traditionally very unreliable.

  Example: get the protocol number for TCP (usually 6)

```
my $proto = getprotobyname "tcp";
```

- ($host, $service) = parse_hostport $string[, $default_service]

  Splitting a string of the form `hostname:port` is a common problem. Unfortunately, just splitting on the colon makes it hard to specify IPv6 addresses and doesn't support the less common but well standardised `[ip literal]` syntax.

  This function tries to do this job in a better way, it supports (at least) the following formats, where `port` can be a numerical port number of a service name, or a `name=port` string, and the `port` and `:port` parts are optional. Also, everywhere where an IP address is supported a hostname or unix domain socket address is also supported (see `parse_unix`), and strings starting with `/` will also be interpreted as unix domain sockets.

```
hostname:port    e.g. "www.linux.org", "www.x.de:443", "www.x.
↪de:https=443",
ipv4:port        e.g. "198.182.196.56", "127.1:22"
ipv6             e.g. "::1", "affe::1"
[ipv4or6]:port   e.g. "[::1]", "[10.0.1]:80"
[ipv4or6] port   e.g. "[127.0.0.1]", "[www.x.org] 17"
ipv4or6 port     e.g. "::1 443", "10.0.0.1 smtp"
unix/:path       e.g. "unix/:/path/to/socket"
/path            e.g. "/path/to/socket"
```

  It also supports defaulting the service name in a simple way by using `$default_service` if no service was detected. If neither a service was detected nor a default was specified, then this function returns the empty list. The same happens when a parse error was detected, such as a hostname with a colon in it (the function is rather forgiving, though).

  Example:

```
print join ",", parse_hostport "localhost:443";
# => "localhost,443"

print join ",", parse_hostport "localhost", "https";
# => "localhost,https"

print join ",", parse_hostport "[::1]";
# => "," (empty list)
```

(continues on next page)

(continued from previous page)

```
print join ",", parse_hostport "/tmp/debug.sock";
# => "unix/", "/tmp/debug.sock"
```

- $string = format_hostport $host, $port

  Takes a host (in textual form) and a port and formats in unambigiously in a way that `parse_hostport` can parse it again. `$port` can be `undef`.

- $sa_family = address_family $ipn

  Returns the address family/protocol-family (AF_xxx/PF_xxx, in one value :) of the given host address in network format.

- $text = format_ipv4 $ipn

  Expects a four octet string representing a binary IPv4 address and returns its textual format. Rarely used, see `format_address` for a nicer interface.

- $text = format_ipv6 $ipn

  Expects a sixteen octet string representing a binary IPv6 address and returns its textual format. Rarely used, see `format_address` for a nicer interface.

- $text = format_address $ipn

  Covnvert a host address in network format (e.g. 4 octets for IPv4 or 16 octets for IPv6) and convert it into textual form.

  Returns `unix/` for UNIX domain sockets.

  This function works similarly to `inet_ntop AF_INET || AF_INET6, ...`, except it automatically detects the address type.

  Returns `undef` if it cannot detect the type.

  If the `$ipn` is a mapped IPv4 in IPv6 address (:ffff::<ipv4>), then just the contained IPv4 address will be returned. If you do not want that, you have to call `format_ipv6` manually.

  Example:

```
print format_address "\x01\x02\x03\x05";
=> 1.2.3.5
```

- $text = AnyEvent::Socket::ntoa $ipn

  Same as format_address, but not exported (think `inet_ntoa`).

- inet_aton $name_or_address, $cb->(@addresses)

  Works similarly to its Socket counterpart, except that it uses a callback. Use the length to distinguish between ipv4 and ipv6 (4 octets for IPv4, 16 for IPv6), or use `format_address` to convert it to a more readable format.

  Note that `resolve_sockaddr`, while initially a more complex interface, resolves host addresses, IDNs, service names and SRV records and gives you an ordered list of socket addresses to try and should be preferred over `inet_aton`.

  Example.

```
inet_aton "www.google.com", my $cv = AE::cv;
say unpack "H*", $_
   for $cv->recv;
# => d155e363
# => d155e367 etc.

inet_aton "ipv6.google.com", my $cv = AE::cv;
say unpack "H*", $_
   for $cv->recv;
# => 20014860a003000000000000000000068
```

- $sa = AnyEvent::Socket::pack_sockaddr $service, $host

  Pack the given port/host combination into a binary sockaddr structure. Handles both IPv4 and IPv6 host addresses, as well as UNIX domain sockets ($host == unix/ and $service == absolute pathname).

  Example:

```
my $bind = AnyEvent::Socket::pack_sockaddr 43, v195.234.53.120;
bind $socket, $bind
   or die "bind: $!";
```

- ($service, $host) = AnyEvent::Socket::unpack_sockaddr $sa

  Unpack the given binary sockaddr structure (as used by bind, getpeername etc.) into a $service, $host combination.

  For IPv4 and IPv6, $service is the port number and $host the host address in network format (binary).

  For UNIX domain sockets, $service is the absolute pathname and $host is a special token that is understood by the other functions in this module (format_address converts it to unix/).

- AnyEvent::Socket::resolve_sockaddr $node, $service, $proto, $family, $type, $cb->([$family, $type, $proto, $sockaddr], ...)

  Tries to resolve the given nodename and service name into protocol families and sockaddr structures usable to connect to this node and service in a protocol-independent way. It works remotely similar to the getaddrinfo posix function.

  For internet addresses, $node is either an IPv4 or IPv6 address, an internet hostname (DNS domain name or IDN), and $service is either a service name (port name from /etc/services) or a numerical port number. If both $node and $service are names, then SRV records will be consulted to find the real service, otherwise they will be used as-is. If you know that the service name is not in your services database, then you can specify the service in the format name=port (e.g. http=80).

  If a host cannot be found via DNS, then it will be looked up in /etc/hosts (or the file specified via $ENV{PERL_ANYEVENT_HOSTS}). If they are found, the addresses there will be used. The effect is as if entries from /etc/hosts would yield A and AAAA records for the host name unless DNS already had records for them.

For UNIX domain sockets, $node must be the string unix/ and $service must be the absolute pathname of the socket. In this case, $proto will be ignored.

$proto must be a protocol name, currently tcp, udp or sctp. The default is currently tcp, but in the future, this function might try to use other protocols such as sctp, depending on the socket type and any SRV records it might find.

$family must be either 0 (meaning any protocol is OK), 4 (use only IPv4) or 6 (use only IPv6). The default is influenced by $ENV{PERL_ANYEVENT_PROTOCOLS}.

$type must be SOCK_STREAM, SOCK_DGRAM or SOCK_SEQPACKET (or undef in which case it gets automatically chosen to be SOCK_STREAM unless $proto is udp).

The callback will receive zero or more array references that contain $family, $type, $proto for use in socket and a binary $sockaddr for use in connect (or bind).

The application should try these in the order given.

Example:

```
resolve_sockaddr "google.com", "http", 0, undef, undef, sub { ..
 ↪. };
```

- $guard = tcp_connect $host, $service, $connect_cb[, $prepare_cb]

  This is a convenience function that creates a TCP socket and makes a 100% non-blocking connect to the given $host (which can be a DNS/IDN hostname or a textual IP address, or the string unix/ for UNIX domain sockets) and $service (which can be a numeric port number or a service name, or a servicename=portnumber string, or the pathname to a UNIX domain socket).

  If both $host and $port are names, then this function will use SRV records to locate the real target(s).

  In either case, it will create a list of target hosts (e.g. for multihomed hosts or hosts with both IPv4 and IPv6 addresses) and try to connect to each in turn.

  After the connection is established, then the $connect_cb will be invoked with the socket file handle (in non-blocking mode) as first, and the peer host (as a textual IP address) and peer port as second and third arguments, respectively. The fourth argument is a code reference that you can call if, for some reason, you don't like this connection, which will cause tcp_connect to try the next one (or call your callback without any arguments if there are no more connections). In most cases, you can simply ignore this argument.

  ```
  $cb->($filehandle, $host, $port, $retry)
  ```

  If the connect is unsuccessful, then the $connect_cb will be invoked without any arguments and $! will be set appropriately (with ENXIO indicating a DNS resolution failure).

  The callback will *never* be invoked before tcp_connect returns, even if tcp_connect was able to connect immediately (e.g. on unix domain sockets).

  The file handle is perfect for being plugged into *AnyEvent::Handle*, but can be used as a normal perl file handle as well.

Unless called in void context, `tcp_connect` returns a guard object that will automatically cancel the connection attempt when it gets destroyed - in which case the callback will not be invoked. Destroying it does not do anything to the socket after the connect was successful - you cannot "uncall" a callback that has been invoked already.

Sometimes you need to "prepare" the socket before connecting, for example, to `bind` it to some port, or you want a specific connect timeout that is lower than your kernel's default timeout. In this case you can specify a second callback, `$prepare_cb`. It will be called with the file handle in not-yet-connected state as only argument and must return the connection timeout value (or `0`, `undef` or the empty list to indicate the default timeout is to be used).

Note to the poor Microsoft Windows users: Windows (of course) doesn't correctly signal connection errors, so unless your event library works around this, failed connections will simply hang. The only event libraries that handle this condition correctly are *EV* and Glib. Additionally, AnyEvent works around this bug with *Event* and in its pure-perl backend. All other libraries cannot correctly handle this condition. To lessen the impact of this windows bug, a default timeout of 30 seconds will be imposed on windows. Cygwin is not affected.

Simple Example: connect to localhost on port 22.

```
tcp_connect localhost => 22, sub {
   my $fh = shift
      or die "unable to connect: $!";
   # do something
};
```

Complex Example: connect to www.google.com on port 80 and make a simple GET request without much error handling. Also limit the connection timeout to 15 seconds.

```
tcp_connect "www.google.com", "http",
   sub {
      my ($fh) = @_
         or die "unable to connect: $!";

      my $handle; # avoid direct assignment so on_eof has it in
↪scope.
      $handle = new AnyEvent::Handle
         fh     => $fh,
         on_error => sub {
            AE::log error => $_[2];
            $_[0]->destroy;
         },
         on_eof => sub {
            $handle->destroy; # destroy handle
            AE::log info => "Done.";
         };

      $handle->push_write ("GET / HTTP/1.0\015\012\015\012");
```

(continues on next page)

(continued from previous page)

```
      $handle->push_read (line => "\015\012\015\012", sub {
         my ($handle, $line) = @_;

         # print response header
         print "HEADER\n$line\n\nBODY\n";

         $handle->on_read (sub {
            # print response body
            print $_[0]->rbuf;
            $_[0]->rbuf = "";
         });
      });
   }, sub {
      my ($fh) = @_;
      # could call $fh->bind etc. here

      15
   };
```

Example: connect to a UNIX domain socket.

```
tcp_connect "unix/", "/tmp/.X11-unix/X0", sub {
   ...
}
```

- $guard = tcp_server $host, $service, $accept_cb[, $prepare_cb]

  Create and bind a stream socket to the given host address and port, set the SO_REUSEADDR flag (if applicable) and call `listen`. Unlike the name implies, this function can also bind on UNIX domain sockets.

  For internet sockets, `$host` must be an IPv4 or IPv6 address (or `undef`, in which case it binds either to `0` or to `::`, depending on whether IPv4 or IPv6 is the preferred protocol, and maybe to both in future versions, as applicable).

  To bind to the IPv4 wildcard address, use `0`, to bind to the IPv6 wildcard address, use `::`.

  The port is specified by `$service`, which must be either a service name or a numeric port number (or `0` or `undef`, in which case an ephemeral port will be used).

  For UNIX domain sockets, `$host` must be `unix/` and `$service` must be the absolute pathname of the socket. This function will try to `unlink` the socket before it tries to bind to it, and will try to unlink it after it stops using it. See SECURITY CONSIDERATIONS, below.

  For each new connection that could be `accepted`, call the `$accept_cb->($fh, $host, $port)` with the file handle (in non-blocking mode) as first, and the peer host and port as second and third arguments (see `tcp_connect` for details).

  Croaks on any errors it can detect before the listen.

  In non-void context, this function returns a guard object whose lifetime it tied to

the TCP server: If the object gets destroyed, the server will be stopped and the listening socket will be cleaned up/unlinked (already accepted connections will not be affected).

When called in void-context, AnyEvent will keep the listening socket alive internally. In this case, there is no guarantee that the listening socket will be cleaned up or unlinked.

In all cases, when the function returns to the caller, the socket is bound and in listening state.

If you need more control over the listening socket, you can provide a `$prepare_cb->($fh, $host, $port)`, which is called just before the `listen ()` call, with the listen file handle as first argument, and IP address and port number of the local socket endpoint as second and third arguments.

It should return the length of the listen queue (or `0` for the default).

Note to IPv6 users: RFC-compliant behaviour for IPv6 sockets listening on `::` is to bind to both IPv6 and IPv4 addresses by default on dual-stack hosts. Unfortunately, only GNU/Linux seems to implement this properly, so if you want both IPv4 and IPv6 listening sockets you should create the IPv6 socket first and then attempt to bind on the IPv4 socket, but ignore any `EADDRINUSE` errors.

Example: bind on some TCP port on the local machine and tell each client to go away.

```
tcp_server undef, undef, sub {
   my ($fh, $host, $port) = @_;

   syswrite $fh, "The internet is full, $host:$port. Go away!\
↪015\012";
}, sub {
   my ($fh, $thishost, $thisport) = @_;
   AE::log info => "Bound to $thishost, port $thisport.";
};
```

Example: bind a server on a unix domain socket.

```
tcp_server "unix/", "/tmp/mydir/mysocket", sub {
   my ($fh) = @_;
};
```

- $guard = AnyEvent::Socket::tcp_bind $host, $service, $done_cb[, $prepare_cb]

  Same as `tcp_server`, except it doesn't call `accept` in a loop for you but simply passes the listen socket to the `$done_cb`. This is useful when you want to have a convenient set up for your listen socket, but want to do the `accept`'ing yourself, for example, in another process.

  In case of an error, `tcp_bind` either croaks, or passes `undef` to the `$done_cb`.

  In non-void context, a guard will be returned. It will clean up/unlink the listening socket when destroyed. In void context, no automatic clean up might be performed.

- tcp_nodelay $fh, $enable

Enables (or disables) the TCP_NODELAY socket option (also known as Nagle's algorithm). Returns false on error, true otherwise.

- tcp_congestion $fh, $algorithm

  Sets the tcp congestion avoidance algorithm (via the TCP_CONGESTION socket option). The default is OS-specific, but is usually reno. Typical other available choices include cubic, lp, bic, highspeed, htcp, hybla, illinois, scalable, vegas, veno, westwood and yeah.

SECURITY CONSIDERATIONS

This module is quite powerful, with with power comes the ability to abuse as well: If you accept "hostnames" and ports from untrusted sources, then note that this can be abused to delete files (host=unix/). This is not really a problem with this module, however, as blindly accepting any address and protocol and trying to bind a server or connect to it is harmful in general.

AUTHOR

```
Marc Lehmann <schmorp@schmorp.de>
http://anyevent.schmorp.de
```

AnyEvent::Strict

force strict mode on for the whole process

SYNOPSIS

```
use AnyEvent::Strict;
# strict mode now switched on
```

DESCRIPTION

This module implements AnyEvent's strict mode.

Loading it makes AnyEvent check all arguments to AnyEvent-methods, at the expense of being slower (often the argument checking takes longer than the actual function). It also wraps all callbacks to check for modifications of $_, which indicates a programming bug inside the watcher callback.

Normally, you don't load this module yourself but instead use it indirectly via the PERL_ANYEVENT_STRICT environment variable (see *AnyEvent*). However, this module can be loaded manually at any time.

AUTHOR

```
Marc Lehmann <schmorp@schmorp.de>
http://anyevent.schmorp.de
```

AnyEvent::TLS

SSLv2/SSLv3/TLSv1 contexts for use in AnyEvent::Handle

SYNOPSIS

```
# via AnyEvent::Handle

use AnyEvent;
use AnyEvent::Handle;
use AnyEvent::Socket;

# simple https-client
my $handle = new AnyEvent::Handle
   connect  => [$host, $port],
   tls      => "connect",
   tls_ctx  => { verify => 1, verify_peername => "https" },
   ...

# simple ssl-server
tcp_server undef, $port, sub {
   my ($fh) = @_;

   my $handle = new AnyEvent::Handle
      fh       => $fh,
      tls      => "accept",
      tls_ctx  => { cert_file => "my-server-keycert.pem" },
      ...

# directly

my $tls = new AnyEvent::TLS
   verify => 1,
   verify_peername => "ldaps",
   ca_file => "/etc/cacertificates.pem";
```

DESCRIPTION

This module is a helper module that implements TLS/SSL (Transport Layer Security/Secure Sockets Layer) contexts. A TLS context is a common set of configuration values for use in establishing TLS connections.

For some quick facts about SSL/TLS, see the section of the same name near the end of the document.

A single TLS context can be used for any number of TLS connections that wish to use the same certificates, policies etc.

Note that this module is inherently tied to *Net::SSLeay*, as this library is used to implement it. Since that perl module is rather ugly, and OpenSSL has a rather ugly license, AnyEvent might switch TLS providers at some future point, at which this API

will change dramatically, at least in the Net::SSLeay-specific parts (most constructor arguments should still work, though).

Although this module does not require a specific version of Net::SSLeay, many features will gradually stop working, or bugs will be introduced with old versions (verification might succeed when it shouldn't - this is a real security issue). Version 1.35 is recommended, 1.33 should work, 1.32 might, and older versions are yours to keep.

USAGE EXAMPLES

See the *AnyEvent::Handle* manpage, NONFREQUENTLY ASKED QUESTIONS, for some actual usage examples.

PUBLIC METHODS AND FUNCTIONS

- $tls = new AnyEvent::TLS key => value...

  The constructor supports these arguments (all as key => value pairs).

  – method => "SSLv2" | "SSLv3" | "TLSv1" | "TLSv1_1" | "TLSv1_2" | "any"

    The protocol parser to use. `SSLv2`, `SSLv3`, `TLSv1`, `TLSv1_1` and `TLSv1_2` will use a parser for those protocols only (so will *not* accept or create connections with/to other protocol versions), while `any` (the default) uses a parser capable of all three protocols.

    The default is to use `"any"` but disable SSLv2. This has the effect of sending a SSLv2 hello, indicating the support for SSLv3 and TLSv1, but not actually negotiating an (insecure) SSLv2 connection.

    Specifying a specific version is almost always wrong to use for a server speaking to a wide variety of clients (e.g. web browsers), and often wrong for a client. If you only want to allow a specific protocol version, use the `sslv2`, `sslv3`, `tlsv1`, `tlsv1_1` or `tlsv1_2` arguments instead.

    For new services it is usually a good idea to enforce a `TLSv1` method from the beginning.

    `TLSv1_1` and `TLSv1_2` require *Net::SSLeay* >= 1.55 and OpenSSL >= 1.0.1. Check the *Net::SSLeay* and OpenSSL documentations for more details.

  – sslv2 => $enabled

    Enable or disable SSLv2 (normally *disabled*).

  – sslv3 => $enabled

    Enable or disable SSLv3 (normally *enabled*).

  – tlsv1 => $enabled

    Enable or disable TLSv1 (normally *enabled*).

  – tlsv1_1 => $enabled

    Enable or disable TLSv1_1 (normally *enabled*).

    This requires *Net::SSLeay* >= 1.55 and OpenSSL >= 1.0.1. Check the *Net::SSLeay* and OpenSSL documentations for more details.

– tlsv1_2 => $enabled

  Enable or disable TLSv1_2 (normally *enabled*).

  This requires *Net::SSLeay* >= 1.55 and OpenSSL >= 1.0.1. Check the *Net::SSLeay* and OpenSSL documentations for more details.

– verify => $enable

  Enable or disable peer certificate checking (default is *disabled*, which is *not recommended*).

  This is the "master switch" for all verify-related parameters and functions.

  If it is disabled, then no peer certificate verification will be done - the connection will be encrypted, but the peer certificate won't be verified against any known CAs, or whether it is still valid or not. No peername verification or custom verification will be done either.

  If enabled, then the peer certificate (required in client mode, optional in server mode, see `verify_require_client_cert`) will be checked against its CA certificate chain - that means there must be a signing chain from the peer certificate to any of the CA certificates you trust locally, as specified by the `ca_file` and/or `ca_path` and/or `ca_cert` parameters (or the system default CA repository, if all of those parameters are missing - see also the *AnyEvent* manpage for the description of PERL_ANYEVENT_CA_FILE).

  Other basic checks, such as checking the validity period, will also be done, as well as optional peername/hostname/common name verification `verify_peername`.

  An optional `verify_cb` callback can also be set, which will be invoked with the verification results, and which can override the decision.

– verify_require_client_cert => $enable

  Enable or disable mandatory client certificates (default is *disabled*). When this mode is enabled, then a client certificate will be required in server mode (a server certificate is mandatory, so in client mode, this switch has no effect).

– verify_peername => $scheme | $callback->($tls, $cert, $peername)

  TLS only protects the data that is sent - it cannot automatically verify that you are really talking to the right peer. The reason is that certificates contain a "common name" (and a set of possible alternative "names") that need to be checked against the peername (usually, but not always, the DNS name of the server) in a protocol-dependent way.

  This can be implemented by specifying a callback that has to verify that the actual `$peername` matches the given certificate in `$cert`.

  Since this can be rather hard to implement, AnyEvent::TLS offers a variety of predefined "schemes" (lifted from *IO::Socket::SSL*) that are named like the protocols that use them:

  * ldap (rfc4513), pop3,imap,acap (rfc2995), nntp (rfc4642)

    Simple wildcards in subjectAltNames are possible, e.g. *.example.org matches www.example.org but not lala.www.example.org. If nothing

from subjectAltNames matches, it checks against the common name, but there are no wildcards allowed.

* http (rfc2818)

  Extended wildcards in subjectAltNames are possible, e.g. *.example.org or even www*.example.org. Wildcards in the common name are not allowed. The common name will be only checked if no host names are given in subjectAltNames.

* smtp (rfc3207)

  This RFC isn't very useful in determining how to do verification so it just assumes that subjectAltNames are possible, but no wildcards are possible anywhere.

* [$wildcards_in_alt, $wildcards_in_cn, $check_cn]

  You can also specify a scheme yourself by using an array reference with three integers.

  `$wildcards_in_alt` and `$wildcards_in_cn` specify whether and where wildcards (`*`) are allowed in subjectAltNames and the common name, respectively. `0` means no wildcards are allowed, `1` means they are allowed only as the first component (`*.example.org`), and `2` means they can be used anywhere (`www*.example.org`), except that very dangerous matches will not be allowed (`*.org` or `*`).

  `$check_cn` specifies if and how the common name field is checked: `0` means it will be completely ignored, `1` means it will only be used if no host names have been found in the subjectAltNames, and `2` means the common name will always be checked against the peername.

You can specify either the name of the parent protocol (recommended, e.g. `http`, `ldap`), the protocol name as usually used in URIs (e.g. `https`, `ldaps`) or the RFC (not recommended, e.g. `rfc2995`, `rfc3920`).

This verification will only be done when verification is enabled (`verify => 1`).

– verify_cb => $callback->($tls, $ref, $cn, $depth, $preverify_ok, $x509_store_ctx, $cert)

Provide a custom peer verification callback used by TLS sessions, which is called with the result of any other verification (`verify`, `verify_peername`).

This callback will only be called when verification is enabled (`verify = > 1`).

`$tls` is the `AnyEvent::TLS` object associated with the session, while `$ref` is whatever the user associated with the session (usually an *AnyEvent::Handle* object when used by AnyEvent::Handle).

`$depth` is the current verification depth - `$depth = 0` means the certificate to verify is the peer certificate, higher levels are its CA certificate and so on. In most cases, you can just return `$preverify_ok` if the `$depth` is non-zero:

```
verify_cb => sub {
   my ($tls, $ref, $cn, $depth, $preverify_ok, $x509_store_
↪ctx, $cert) = @_;

   return $preverify_ok
      if $depth;

   # more verification
},
```

`$preverify_ok` is true iff the basic verification of the certificates was successful (a valid CA chain must exist, the certificate has passed basic validity checks, peername verification succeeded).

`$x509_store_ctx` is the Net::SSLeay::X509_CTX> object.

`$cert` is the `Net::SSLeay::X509` object representing the peer certificate, or zero if there was an error. You can call `AnyEvent::TLS::certname $cert` to get a nice user-readable string to identify the certificate.

The callback must return either `0` to indicate failure, or `1` to indicate success.

– verify_client_once => $enable

Enable or disable skipping the client certificate verification on renegotiations (default is *disabled*, the certificate will always be checked). Only makes sense in server mode.

– ca_file => $path

If this parameter is specified and non-empty, it will be the path to a file with (server) CA certificates in PEM format that will be loaded. Each certificate will look like:

```
-----BEGIN CERTIFICATE-----
... (CA certificate in base64 encoding) ...
-----END CERTIFICATE-----
```

You have to enable verify mode (`verify => 1`) for this parameter to have any effect.

– ca_path => $path

If this parameter is specified and non-empty, it will be the path to a directory with hashed CA certificate files in PEM format. When the ca certificate is being verified, the certificate will be hashed and looked up in that directory (see http://www.openssl.org/docs/ssl/SSL_CTX_load_verify_locations.html for details)

The certificates specified via `ca_file` take precedence over the ones found in `ca_path`.

You have to enable verify mode (`verify => 1`) for this parameter to have any effect.

– ca_cert => $string

In addition or instead of using `ca_file` and/or `ca_path`, you can also use `ca_cert` to directly specify the CA certificates (there can be multiple) in PEM format, in a string.

– check_crl => $enable

Enable or disable certificate revocation list checking. If enabled, then peer certificates will be checked against a list of revoked certificates issued by the CA. The revocation lists will be expected in the `ca_path` directory.

certificate verification will fail if this is enabled but no revocation list was found.

This requires OpenSSL >= 0.9.7b. Check the OpenSSL documentation for more details.

– key_file => $path

Path to the local private key file in PEM format (might be a combined certificate/private key file).

The local certificate is used to authenticate against the peer - servers mandatorily need a certificate and key, clients can use a certificate and key optionally to authenticate, e.g. for log-in purposes.

The key in the file should look similar this:

```
-----BEGIN RSA PRIVATE KEY-----
...header data
... (key data in base64 encoding) ...
-----END RSA PRIVATE KEY-----
```

– key => $string

The private key string in PEM format (see `key_file`, only one of `key_file` or `key` can be specified).

The idea behind being able to specify a string is to avoid blocking in I/O. Unfortunately, Net::SSLeay fails to implement any interface to the needed OpenSSL functionality, this is currently implemented by writing to a temporary file.

– cert_file => $path

The path to the local certificate file in PEM format (might be a combined certificate/private key file, including chained certificates).

The local certificate (and key) are used to authenticate against the peer - servers mandatorily need a certificate and key, clients can use certificate and key optionally to authenticate, e.g. for log-in purposes.

The certificate in the file should look like this:

```
-----BEGIN CERTIFICATE-----
... (certificate in base64 encoding) ...
-----END CERTIFICATE-----
```

If the certificate file or string contain both the certificate and private key, then there is no need to specify a separate `key_file` or `key`.

Additional signing certifiates to send to the peer (in SSLv3 and newer) can be specified by appending them to the certificate proper: the order must be from issuer certificate over any intermediate CA certificates to the root CA.

So the recommended ordering for a combined key/cert/chain file, specified via `cert_file` or `cert` looks like this:

```
certificate private key
client/server certificate
ca 1, signing client/server certficate
ca 2, signing ca 1
...
```

– cert => $string

The local certificate in PEM format (might be a combined certificate/private key file). See `cert_file`.

The idea behind being able to specify a string is to avoid blocking in I/O. Unfortunately, Net::SSLeay fails to implement any interface to the needed OpenSSL functionality, this is currently implemented by writing to a temporary file.

– cert_password => $string | $callback->($tls)

The certificate password - if the certificate is password-protected, then you can specify its password here.

Instead of providing a password directly (which is not so recommended), you can also provide a password-query callback. The callback will be called whenever a password is required to decode a local certificate, and is supposed to return the password.

– dh_file => $path

Path to a file containing Diffie-Hellman parameters in PEM format, for use in servers. See also dh on how to specify them directly, or use a pre-generated set.

Diffie-Hellman key exchange generates temporary encryption keys that are not transferred over the connection, which means that even if the certificate key(s) are made public at a later time and a full dump of the connection exists, the key still cannot be deduced.

These ciphers are only available with SSLv3 and later (which is the default with AnyEvent::TLS), and are only used in server/accept mode. Anonymous DH protocols are usually disabled by default, and usually not even compiled into the underlying library, as they provide no direct protection against man-in-the-middle attacks. The same is true for the common practise of self-signed certificates that you have to accept first, of course.

– dh => $string

Specify the Diffie-Hellman parameters in PEM format directly as a string (see `dh_file`), the default is `ffdhe3072` unless `dh_file` was specified.

AnyEvent::TLS supports supports a number of precomputed DH parameters, since computing them is expensive. They are:

```
# from RFC 7919 - recommended
ffdhe2048, ffdhe3072, ffdhe4096, ffdhe6144, ffdhe8192

# from "Assigned Number for SKIP Protocols"
skip512, skip1024, skip2048, skip4096

# from schmorp
schmorp1024, schmorp1539, schmorp2048, schmorp4096,␣
↪schmorp8192
```

It is said that 2048 bit DH parameters are safe till 2030, and DH parameters shorter than 900 bits are totally insecure.

To disable DH protocols completely, specify `undef` as dh parameter.

– dh_single_use => $enable

Enables or disables “use only once” mode when using Diffie-Hellman key exchange. When enabled (default), each time a new key is exchanged a new Diffie-Hellman key is generated, which improves security as each key is only used once. When disabled, the key will be created as soon as the AnyEvent::TLS object is created and will be reused.

All the DH parameters supplied with AnyEvent::TLS should be safe with `dh_single_use` switched off, but YMMV.

– cipher_list => $string

The list of ciphers to use, as a string (example: `AES:ALL:!aNULL:!eNULL:+RC4:@STRENGTH`). The format of this string and its default value is documented at http://www.openssl.org/docs/apps/ciphers.html#CIPHER_STRINGS.

– session_ticket => $enable

Enables or disables RC5077 support (Session Resumption without Server-Side State). The default is disabled for clients, as many (buggy) TLS/SSL servers choke on it, but enabled for servers.

When enabled and supported by the server, a session ticket will be provided to the client, which allows fast resuming of connections.

– prepare => $coderef->($tls)

If this argument is present, then it will be called with the new AnyEvent::TLS object after any other initialisation has bee done, in case you wish to fine-tune something...

- $tls = new_from_ssleay AnyEvent::TLS $ctx

  This constructor takes an existing *Net::SSLeay* SSL_CTX object (which is just an integer) and converts it into an `AnyEvent::TLS` object. This only works because AnyEvent::TLS is currently implemented using Net::SSLeay. As this is such a horrible perl module and OpenSSL has such an annoying license, this might change in the future, in which case this method might vanish.

- $ctx = $tls->ctx

Returns the actual Net::SSLeay::CTX object (just an integer).

- AnyEvent::TLS::init

  AnyEvent::TLS does on-demand initialisation, and normally there is no need to call an initialise function.

  As initialisation might take some time (to read e.g. `/dev/urandom`), this could be annoying in some highly interactive programs. In that case, you can call `AnyEvent::TLS::init` to make sure there will be no costly initialisation later. It is harmless to call `AnyEvent::TLS::init` multiple times.

- $certname = AnyEvent::TLS::certname $x509

  Utility function that returns a user-readable string identifying the X509 certificate object.

SSL/TLS QUICK FACTS

Here are some quick facts about TLS/SSL that might help you:

- A certificate is the public key part, a key is the private key part.

  While not strictly true, certificates are the things you can hand around publicly as a kind of identity, while keys should really be kept private, as proving that you have the private key is usually interpreted as being the entity behind the certificate.

- A certificate is signed by a CA (Certificate Authority).

  By signing, the CA basically claims that the certificate it signs really belongs to the identity named in it, verified according to the CA policies. For e.g. HTTPS, the CA usually makes some checks that the hostname mentioned in the certificate really belongs to the company/person that requested the signing and owns the domain.

- CAs can be certified by other CAs.

  Or by themselves - a certificate that is signed by a CA that is itself is called a self-signed certificate, a trust chain of length zero. When you find a certificate signed by another CA, which is in turn signed by another CA you trust, you have a trust chain of depth two.

- “Trusting” a CA means trusting all certificates it has signed.

  If you “trust” a CA certificate, then all certificates signed by it are automatically considered trusted as well.

- A successfully verified certificate means that you can be reasonably sure that whoever you are talking with really is who he claims he is.

  By verifying certificates against a number of CAs that you trust (meaning it is signed directly or indirectly by such a CA), you can find out that the other side really is whoever he claims, according to the CA policies, and your belief in the integrity of the CA.

- Verifying the certificate signature is not everything.

  Even when the certificate is correct, it might belong to somebody else: if www.attacker.com can make your computer believe that it is really called www.mybank.com (by making your DNS server believe this for example), then it could send you the certificate for www.attacker.com that your software trusts because

it is signed by a CA you trust, and intercept all your traffic that you think goes to www.mybank.com. This works because your software sees that the certificate is correctly signed (for www.attacker.com) and you think you are talking to your bank.

To thwart this attack vector, peername verification should be used, which basically checks that the certificate (for www.attacker.com) really belongs to the host you are trying to talk to (www.mybank.com), which in this example is not the case, as www.attacker.com (from the certificate) doesn't match www.mybank.com (the hostname used to create the connection).

So peername verification is almost as important as checking the CA signing. Unfortunately, every protocol implements this differently, if at all...

- Switching off verification is sometimes reasonable.

  You can switch off verification. You still get an encrypted connection that is protected against eavesdropping and injection - you just lose protection against man in the middle attacks, i.e. somebody else with enough abilities to intercept all traffic can masquerade herself as the other side.

  For many applications, switching off verification is entirely reasonable. Downloading random stuff from websites using HTTPS for no reason is such an application. Talking to your bank and entering TANs is not such an application.

- A SSL/TLS server always needs a certificate/key pair to operate, for clients this is optional.

  Apart from (usually disabled) anonymous cipher suites, a server always needs a certificate/key pair to operate.

  Clients almost never use certificates, but if they do, they can be used to authenticate the client, just as server certificates can be used to authenticate the server.

- SSL version 2 is very insecure.

  SSL version 2 is old and not only has it some security issues, SSLv2-only implementations are usually buggy, too, due to their age.

- Sometimes, even losing your "private" key might not expose all your data.

  With Diffie-Hellman ephemeral key exchange, you can lose the DH parameters (the "keys"), but all your connections are still protected. Diffie-Hellman needs special set-up (done by default by AnyEvent::TLS).

SECURITY CONSIDERATIONS

When you use any of the options that pass in keys or certificates as strings (e.g. `ca_cert`), then, due to serious shortcomings in *Net::SSLeay*, this module creates a temporary file to store the string - see *File::Temp* and possibly its `safe_level` setting for more details on what to watch out for.

BUGS

Due to the abysmal code quality of Net::SSLeay, this module will leak small amounts of memory per TLS connection (currently at least one perl scalar).

AUTHORS

Marc Lehmann schmorp@schmorp.de.

Some of the API, documentation and implementation (verify_hostname), and a lot of ideas/workarounds/knowledge have been taken from the *IO::Socket::SSL* module. Care has been taken to keep the API similar to that and other modules, to the extent possible while providing a sensible API for AnyEvent.

AnyEvent::Util

various utility functions.

SYNOPSIS

```
use AnyEvent::Util;
```

DESCRIPTION

This module implements various utility functions, mostly replacing well-known functions by event-ised counterparts.

All functions documented without `AnyEvent::Util::` prefix are exported by default.

- ($r, $w) = portable_pipe

  Calling `pipe` in Perl is portable - except it doesn't really work on sucky windows platforms (at least not with most perls - cygwin's perl notably works fine): On windows, you actually get two file handles you cannot use select on.

  This function gives you a pipe that actually works even on the broken windows platform (by creating a pair of TCP sockets on windows, so do not expect any speed from that) and using `pipe` everywhere else.

  See `portable_socketpair`, below, for a bidirectional "pipe".

  Returns the empty list on any errors.

- ($fh1, $fh2) = portable_socketpair

  Just like `portable_pipe`, above, but returns a bidirectional pipe (usually by calling `socketpair` to create a local loopback socket pair, except on windows, where it again returns two interconnected TCP sockets).

  Returns the empty list on any errors.

- fork_call { CODE } @args, $cb->(@res)

  Executes the given code block asynchronously, by forking. Everything the block returns will be transferred to the calling process (by serialising and deserialising via *Storable*).

If there are any errors, then the `$cb` will be called without any arguments. In that case, either `$@` contains the exception (and `$!` is irrelevant), or `$!` contains an error number. In all other cases, `$@` will be undefined.

The code block must not ever call an event-polling function or use event-based programming that might cause any callbacks registered in the parent to run.

Win32 spoilers: Due to the endlessly sucky and broken native windows perls (there is no way to cleanly exit a child process on that platform that doesn't also kill the parent), you have to make sure that your main program doesn't exit as long as any `fork_calls` are still in progress, otherwise the program won't exit. Also, on most windows platforms some memory will leak for every invocation.

Note that forking can be expensive in large programs (RSS 200MB+). On windows, it is abysmally slow, do not expect more than 5..20 forks/s on that sucky platform (note this uses perl's pseudo-threads, so avoid those like the plague).

Example: poor man's async disk I/O (better use *AnyEvent::IO* together with IO::AIO).

```
fork_call {
   open my $fh, "</etc/passwd"
      or die "passwd: $!";
   local $/;
   <$fh>
} sub {
   my ($passwd) = @_;
   ...
};
```

- $AnyEvent::Util::MAX_FORKS [default: 10]

  The maximum number of child processes that `fork_call` will fork in parallel. Any additional requests will be queued until a slot becomes free again.

  The environment variable `PERL_ANYEVENT_MAX_FORKS` is used to initialise this value.

- fh_nonblocking $fh, $nonblocking

  Sets the blocking state of the given filehandle (true == nonblocking, false == blocking). Uses fcntl on anything sensible and ioctl FIONBIO on broken (i.e. windows) platforms.

  Instead of using this function, you could use `AnyEvent::fh_block` or `AnyEvent::fh_unblock`.

- $guard = guard { CODE }

  This function creates a special object that, when destroyed, will execute the code block.

  This is often handy in continuation-passing style code to clean up some resource regardless of where you break out of a process.

  The *Guard* module will be used to implement this function, if it is available. Otherwise a pure-perl implementation is used.

While the code is allowed to throw exceptions in unusual conditions, it is not defined whether this exception will be reported (at the moment, the Guard module and AnyEvent's pure-perl implementation both try to report the error and continue).

You can call one method on the returned object:

- $guard->cancel

  This simply causes the code block not to be invoked: it "cancels" the guard.

- AnyEvent::Util::close_all_fds_except @fds

  This rarely-used function simply closes all file descriptors (or tries to) of the current process except the ones given as arguments.

  When you want to start a long-running background server, then it is often beneficial to do this, as too many C-libraries are too stupid to mark their internal fd's as close-on-exec.

  The function expects to be called shortly before an `exec` call.

  Example: close all fds except 0, 1, 2.

```
close_all_fds_except 0, 2, 1;
```

- $cv = run_cmd $cmd, key => value...

  Run a given external command, potentially redirecting file descriptors and return a condition variable that gets sent the exit status (like `$?`) when the program exits *and* all redirected file descriptors have been exhausted.

  The `$cmd` is either a single string, which is then passed to a shell, or an arrayref, which is passed to the `execvp` function (the first array element is used both for the executable name and argv[0]).

  The key-value pairs can be:

  – ">" => $filename

    Redirects program standard output into the specified filename, similar to `>filename` in the shell.

  – ">" => \$data

    Appends program standard output to the referenced scalar. The condvar will not be signalled before EOF or an error is signalled.

    Specifying the same scalar in multiple ">" pairs is allowed, e.g. to redirect both stdout and stderr into the same scalar:

```
">"  => \$output,
"2>" => \$output,
```

  – ">" => $filehandle

    Redirects program standard output to the given filehandle (or actually its underlying file descriptor).

  – ">" => $callback->($data)

Calls the given callback each time standard output receives some data, passing it the data received. On EOF or error, the callback will be invoked once without any arguments.

The condvar will not be signalled before EOF or an error is signalled.

– "fd>" => $see_above

Like ">", but redirects the specified fd number instead.

– "<" => $see_above

The same, but redirects the program's standard input instead. The same forms as for ">" are allowed.

In the callback form, the callback is supposed to return data to be written, or the empty list or `undef` or a zero-length scalar to signal EOF.

Similarly, either the write data must be exhausted or an error is to be signalled before the condvar is signalled, for both string-reference and callback forms.

– "fd<" => $see_above

Like "<", but redirects the specified file descriptor instead.

– on_prepare => $cb

Specify a callback that is executed just before the command is `exec`'ed, in the child process. Be careful not to use any event handling or other services not available in the child.

This can be useful to set up the environment in special ways, such as changing the priority of the command or manipulating signal handlers (e.g. setting `SIGINT` to `IGNORE`).

– close_all => $boolean

When `close_all` is enabled (default is disabled), then all extra file descriptors will be closed, except the ones that were redirected and `0`, `1` and `2`.

See `close_all_fds_except` for more details.

– '$$' => \$pid

A reference to a scalar which will receive the PID of the newly-created subprocess after `run_cmd` returns.

Note the the PID might already have been recycled and used by an unrelated process at the time `run_cmd` returns, so it's not useful to send signals, use as a unique key in data structures and so on.

Example: run `rm -rf /`, redirecting standard input, output and error to `/dev/null`.

```
my $cv = run_cmd [qw(rm -rf /)],
   "<", "/dev/null",
   ">", "/dev/null",
   "2>", "/dev/null";
$cv->recv and die "d'oh! something survived!"
```

Example: run `openssl` and create a self-signed certificate and key, storing them in `$cert` and `$key`. When finished, check the exit status in the callback and print key and certificate.

```
my $cv = run_cmd [qw(openssl req
                     -new -nodes -x509 -days 3650
                     -newkey rsa:2048 -keyout /dev/fd/3
                     -batch -subj /CN=AnyEvent
                    )],
   "<", "/dev/null",
   ">" , \my $cert,
   "3>", \my $key,
   "2>", "/dev/null";

$cv->cb (sub {
   shift->recv and die "openssl failed";

   print "$key\n$cert\n";
});
```

- AnyEvent::Util::punycode_encode $string

  Punycode-encodes the given `$string` and returns its punycode form. Note that uppercase letters are *not* casefolded - you have to do that yourself.

  Croaks when it cannot encode the string.

- AnyEvent::Util::punycode_decode $string

  Tries to punycode-decode the given `$string` and return its unicode form. Again, uppercase letters are not casefoled, you have to do that yourself.

  Croaks when it cannot decode the string.

- AnyEvent::Util::idn_nameprep $idn[, $display]

  Implements the IDNA nameprep normalisation algorithm. Or actually the UTS#46 algorithm. Or maybe something similar - reality is complicated between IDNA2003, UTS#46 and IDNA2008. If `$display` is true then the name is prepared for display, otherwise it is prepared for lookup (default).

  If you have no clue what this means, look at `idn_to_ascii` instead.

  This function is designed to avoid using a lot of resources - it uses about 1MB of RAM (most of this due to Unicode::Normalize). Also, names that are already "simple" will only be checked for basic validity, without the overhead of full nameprep processing.

- $domainname = AnyEvent::Util::idn_to_ascii $idn

  Converts the given unicode string (`$idn`, international domain name, e.g. 日本語。ＪＰ) to a pure-ASCII domain name (this is usually called the "IDN ToAscii" transform). This transformation is idempotent, which means you can call it just in case and it will do the right thing.

  Unlike some other "ToAscii" implementations, this one works on full domain names and should never fail - if it cannot convert the name, then it will return

it unchanged.

This function is an amalgam of IDNA2003, UTS#46 and IDNA2008 - it tries to be reasonably compatible to other implementations, reasonably secure, as much as IDNs can be secure, and reasonably efficient when confronted with IDNs that are already valid DNS names.

- $idn = AnyEvent::Util::idn_to_unicode $idn

  Converts the given unicode string ($idn, international domain name, e.g. 日本語。ＪＰ, www.deliantra.net, www.xn–l-0ga.de) to unicode form (this is usually called the "IDN ToUnicode" transform). This transformation is idempotent, which means you can call it just in case and it will do the right thing.

  Unlike some other "ToUnicode" implementations, this one works on full domain names and should never fail - if it cannot convert the name, then it will return it unchanged.

  This function is an amalgam of IDNA2003, UTS#46 and IDNA2008 - it tries to be reasonably compatible to other implementations, reasonably secure, as much as IDNs can be secure, and reasonably efficient when confronted with IDNs that are already valid DNS names.

  At the moment, this function simply calls `idn_nameprep $idn, 1`, returning its argument when that function fails.

AUTHOR

```
Marc Lehmann <schmorp@schmorp.de>
http://anyevent.schmorp.de
```

5.2.4 App

Namespace containing the following modules:

- `App::Cpan` - easily interact with CPAN from the command line
- `App::Prove` - Implements the `prove` command.

App::Cpan

easily interact with CPAN from the command line

SYNOPSIS

```
# with arguments and no switches, installs specified modules
cpan module_name [ module_name ... ]

# with switches, installs modules with extra behavior
cpan [-cfFimtTw] module_name [ module_name ... ]

# use local::lib
cpan -I module_name [ module_name ... ]
```

(continues on next page)

(continued from previous page)

```
# one time mirror override for faster mirrors
cpan -p ...

# with just the dot, install from the distribution in the
# current directory
cpan .

# without arguments, starts CPAN.pm shell
cpan

# without arguments, but some switches
cpan [-ahpruvACDLOPX]
```

DESCRIPTION

This script provides a command interface (not a shell) to CPAN. At the moment it uses CPAN.pm to do the work, but it is not a one-shot command runner for CPAN.pm.

Options

- -a

  Creates a CPAN.pm autobundle with CPAN::Shell->autobundle.

- -A module [module ...]

  Shows the primary maintainers for the specified modules.

- -c module

  Runs a `make clean` in the specified module's directories.

- -C module [module ...]

  Show the `Changes` files for the specified modules

- -D module [module ...]

  Show the module details. This prints one line for each out-of-date module (meaning, modules locally installed but have newer versions on CPAN). Each line has three columns: module name, local version, and CPAN version.

- -f

  Force the specified action, when it normally would have failed. Use this to install a module even if its tests fail. When you use this option, -i is not optional for installing a module when you need to force it:

  ```
  % cpan -f -i Module::Foo
  ```

- -F

  Turn off CPAN.pm's attempts to lock anything. You should be careful with this since you might end up with multiple scripts trying to muck in the same directory. This

isn't so much of a concern if you're loading a special config with `-j`, and that config sets up its own work directories.

- -g module [module ...]

  Downloads to the current directory the latest distribution of the module.

- -G module [module ...]

  UNIMPLEMENTED

  Download to the current directory the latest distribution of the modules, unpack each distribution, and create a git repository for each distribution.

  If you want this feature, check out Yanick Champoux's `Git::CPAN::Patch` distribution.

- -h

  Print a help message and exit. When you specify `-h`, it ignores all of the other options and arguments.

- -i module [module ...]

  Install the specified modules. With no other switches, this switch is implied.

- -I

  Load `local::lib` (think like `-I` for loading lib paths). Too bad `-l` was already taken.

- -j Config.pm

  Load the file that has the CPAN configuration data. This should have the same format as the standard `CPAN/Config.pm` file, which defines `$CPAN::Config` as an anonymous hash.

  If the file does not exist, `cpan` dies.

- -J

  Dump the configuration in the same format that CPAN.pm uses. This is useful for checking the configuration as well as using the dump as a starting point for a new, custom configuration.

- -l

  List all installed modules with their versions

- -L author [author ...]

  List the modules by the specified authors.

- -m

  Make the specified modules.

- -M mirror1,mirror2,...

  A comma-separated list of mirrors to use for just this run. The `-P` option can find them for you automatically.

- -n

  Do a dry run, but don't actually install anything. (unimplemented)

- -O

  Show the out-of-date modules.

- -p

  Ping the configured mirrors and print a report

- -P

  Find the best mirrors you could be using and use them for the current session.

- -r

  Recompiles dynamically loaded modules with CPAN::Shell->recompile.

- -s

  Drop in the CPAN.pm shell. This command does this automatically if you don't specify any arguments.

- -t module [module ...]

  Run a `make test` on the specified modules.

- -T

  Do not test modules. Simply install them.

- -u

  Upgrade all installed modules. Blindly doing this can really break things, so keep a backup.

- -v

  Print the script version and CPAN.pm version then exit.

- -V

  Print detailed information about the cpan client.

- -w

  UNIMPLEMENTED

  Turn on cpan warnings. This checks various things, like directory permissions, and tells you about problems you might have.

- -x module [module ...]

  Find close matches to the named modules that you think you might have mistyped. This requires the optional installation of Text::Levenshtein or Text::Levenshtein::Damerau.

- -X

  Dump all the namespaces to standard output.

Examples

```
# print a help message
cpan -h

# print the version numbers
cpan -v

# create an autobundle
cpan -a

# recompile modules
cpan -r

# upgrade all installed modules
cpan -u

# install modules ( sole -i is optional )
cpan -i Netscape::Booksmarks Business::ISBN

# force install modules ( must use -i )
cpan -fi CGI::Minimal URI

# install modules but without testing them
cpan -Ti CGI::Minimal URI
```

Environment variables

There are several components in CPAN.pm that use environment variables. The build tools, *ExtUtils::MakeMaker* and Module::Build use some, while others matter to the levels above them. Some of these are specified by the Perl Toolchain Gang:

Lancaster Consensus: https://github.com/Perl-Toolchain-Gang/toolchain-site/blob/master/lancaster-consensus.md

Oslo Consensus: https://github.com/Perl-Toolchain-Gang/toolchain-site/blob/master/oslo-consensus.md

- NONINTERACTIVE_TESTING

  Assume no one is paying attention and skips prompts for distributions that do that correctly. `cpan(1)` sets this to 1 unless it already has a value (even if that value is false).

- PERL_MM_USE_DEFAULT

  Use the default answer for a prompted questions. `cpan(1)` sets this to 1 unless it already has a value (even if that value is false).

- CPAN_OPTS

  As with PERL5OPT, a string of additional `cpan(1)` options to add to those you specify on the command line.

- CPANSCRIPT_LOGLEVEL

The log level to use, with either the embedded, minimal logger or Log::Log4perl if it is installed. Possible values are the same as the `Log::Log4perl` levels: `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR`, and `FATAL`. The default is `INFO`.

- GIT_COMMAND

  The path to the `git` binary to use for the Git features. The default is `/usr/local/bin/git`.

Methods

- run(ARGS)

  Just do it.

  The `run` method returns 0 on success and a positive number on failure. See the section on EXIT CODES for details on the values.

  CPAN.pm sends all the good stuff either to STDOUT, or to a temp file if $CPAN::Be_Silent is set. I have to intercept that output so I can find out what happened.

  Stolen from File::Path::Expand

EXIT VALUES

The script exits with zero if it thinks that everything worked, or a positive number if it thinks that something failed. Note, however, that in some cases it has to divine a failure by the output of things it does not control. For now, the exit codes are vague:

```
1       An unknown error

2       The was an external problem

4       There was an internal problem with the script

8       A module failed to install
```

TO DO

* There is initial support for Log4perl if it is available, but I haven't gone through everything to make the NullLogger work out correctly if Log4perl is not installed.

* When I capture CPAN.pm output, I need to check for errors and report them to the user.

* Warnings switch

* Check then exit

BUGS

* none noted

SEE ALSO

CPAN, App::cpanminus

SOURCE AVAILABILITY

This code is in Github in the CPAN.pm repository:

```
https://github.com/andk/cpanpm
```

The source used to be tracked separately in another GitHub repo, but the canonical source is now in the above repo.

CREDITS

Japheth Cleaver added the bits to allow a forced install (`-f`).

Jim Brandt suggested and provided the initial implementation for the up-to-date and Changes features.

Adam Kennedy pointed out that `exit()` causes problems on Windows where this script ends up with a .bat extension

David Golden helps integrate this into the `CPAN.pm` repos.

Jim Keenan fixed up various issues with _download

AUTHOR

brian d foy, `<bdfoy@cpan.org>`

COPYRIGHT

Copyright (c) 2001-2021, brian d foy, All Rights Reserved.

You may redistribute this under the same terms as Perl itself.

App::Prove

Implements the `prove` command.

VERSION

Version 3.52

DESCRIPTION

Test::Harness provides a command, `prove`, which runs a TAP based test suite and prints a report. The `prove` command is a minimal wrapper around an instance of this module.

SYNOPSIS

```
use App::Prove;

my $app = App::Prove->new;
$app->process_args(@ARGV);
$app->run;
```

METHODS

Class Methods

new

Create a new `App::Prove`. Optionally a hash ref of attribute initializers may be passed.

state_class

Getter/setter for the name of the class used for maintaining state. This class should either subclass from `App::Prove::State` or provide an identical interface.

state_manager

Getter/setter for the instance of the `state_class`.

add_rc_file

```
$prove->add_rc_file('myproj/.proverc');
```

Called before `process_args` to prepend the contents of an rc file to the options.

process_args

```
$prove->process_args(@args);
```

Processes the command-line arguments. Attributes will be set appropriately. Any filenames may be found in the `argv` attribute.

Dies on invalid arguments.

run

Perform whatever actions the command line args specified. The `prove` command line tool consists of the following code:

```
use App::Prove;

my $app = App::Prove->new;
$app->process_args(@ARGV);
exit( $app->run ? 0 : 1 );  # if you need the exit code
```

require_harness

Load a harness replacement class.

```
$prove->require_harness($for => $class_name);
```

print_version

Display the version numbers of the loaded *TAP::Harness* and the current Perl.

Attributes

After command line parsing the following attributes reflect the values of the corresponding command line switches. They may be altered before calling `run`.

- `archive`
- `argv`
- `backwards`
- `blib`
- `color`
- `directives`
- `dry`
- `exec`
- `extensions`
- `failures`
- `comments`
- `formatter`
- `harness`
- `ignore_exit`
- `includes`
- `jobs`
- `lib`

- merge
- modules
- parse
- plugins
- quiet
- really_quiet
- recurse
- rules
- show_count
- show_help
- show_man
- show_version
- shuffle
- state
- state_class
- taint_fail
- taint_warn
- test_args
- timer
- verbose
- warnings_fail
- warnings_warn
- tapversion
- trap

PLUGINS

App::Prove provides support for 3rd-party plugins. These are currently loaded at run-time, *after* arguments have been parsed (so you can not change the way arguments are processed, sorry), typically with the -P_plugin_ switch, eg:

```
prove -PMyPlugin
```

This will search for a module named App::Prove::Plugin::MyPlugin, or failing that, MyPlugin. If the plugin can't be found, prove will complain & exit.

You can pass an argument to your plugin by appending an = after the plugin name, eg -PMyPlugin=foo. You can pass multiple arguments using commas:

```
prove -PMyPlugin=foo,bar,baz
```

These are passed in to your plugin's `load()` class method (if it has one), along with a reference to the `App::Prove` object that is invoking your plugin:

```
sub load {
    my ($class, $p) = @_;

    my @args = @{ $p->{args} };
    # @args will contain ( 'foo', 'bar', 'baz' )
    $p->{app_prove}->do_something;
    ...
}
```

Sample Plugin

Here's a sample plugin, for your reference:

```
package App::Prove::Plugin::Foo;

# Sample plugin, try running with:
# prove -PFoo=bar -r -j3
# prove -PFoo -Q
# prove -PFoo=bar,My::Formatter

use strict;
use warnings;

sub load {
    my ($class, $p) = @_;
    my @args = @{ $p->{args} };
    my $app  = $p->{app_prove};

    print "loading plugin: $class, args: ", join(', ', @args ), "\n
 ↪";

    # turn on verbosity
    $app->verbose( 1 );

    # set the formatter?
    $app->formatter( $args[1] ) if @args > 1;

    # print some of App::Prove's state:
    for my $attr (qw( jobs quiet really_quiet recurse verbose )) {
        my $val = $app->$attr;
        $val    = 'undef' unless defined( $val );
        print "$attr: $val\n";
    }

    return 1;
```

(continues on next page)

(continued from previous page)

```
}

1;
```

SEE ALSO

prove, *TAP::Harness*

Modules

- `App::Prove::State` - State storage for the `prove` command.

App::Prove::State

State storage for the `prove` command.

VERSION

Version 3.52

DESCRIPTION

The `prove` command supports a `--state` option that instructs it to store persistent state across runs. This module implements that state and the operations that may be performed on it.

SYNOPSIS

```
# Re-run failed tests
$ prove --state=failed,save -rbv
```

METHODS

Class Methods

new

Accepts a hashref with the following key/value pairs:

- `store`

  The filename of the data store holding the data that App::Prove::State reads.

- `extensions` (optional)

  The test name extensions. Defaults to `.t`.

- `result_class` (optional)

  The name of the `result_class`. Defaults to `App::Prove::State::Result`.

result_class

Getter/setter for the name of the class used for tracking test results. This class should either subclass from `App::Prove::State::Result` or provide an identical interface.

extensions

Get or set the list of extensions that files must have in order to be considered tests. Defaults to ['.t'].

results

Get the results of the last test run. Returns a `result_class()` instance.

commit

Save the test results. Should be called after all tests have run.

Instance Methods

apply_switch

```
$self->apply_switch('failed,save');
```

Apply a list of switch options to the state, updating the internal object state as a result. Nothing is returned.

Diagnostics: - "Illegal state option: %s"

- `last`

  Run in the same order as last time

- `failed`

  Run only the failed tests from last time

- `passed`

  Run only the passed tests from last time

- `all`

  Run all tests in normal order

- `hot`

  Run the tests that most recently failed first

- `todo`

  Run the tests ordered by number of todos.

- `slow`

  Run the tests in slowest to fastest order.

- `fast`

  Run test tests in fastest to slowest order.

- `new`

  Run the tests in newest to oldest order.

- `old`

  Run the tests in oldest to newest order.

- `save`

  Save the state on exit.

`get_tests`

Given a list of args get the names of tests that should run

`observe_test`

Store the results of a test.

`save`

Write the state to a file.

`load`

Load the state from a file

Modules

- `App::Prove::State::Result` - Individual test suite results.

App::Prove::State::Result

Individual test suite results.

VERSION

Version 3.52

DESCRIPTION

The `prove` command supports a `--state` option that instructs it to store persistent state across runs. This module encapsulates the results for a single test suite run.

SYNOPSIS

```
# Re-run failed tests
$ prove --state=failed,save -rbv
```

METHODS

Class Methods

new

```
my $result = App::Prove::State::Result->new({
    generation => $generation,
    tests      => \%tests,
});
```

Returns a new `App::Prove::State::Result` instance.

state_version

Returns the current version of state storage.

test_class

Returns the name of the class used for tracking individual tests. This class should either subclass from `App::Prove::State::Result::Test` or provide an identical interface.

generation

Getter/setter for the "generation" of the test suite run. The first generation is 1 (one) and subsequent generations are 2, 3, etc.

last_run_time

Getter/setter for the time of the test suite run.

tests

Returns the tests for a given generation. This is a hashref or a hash, depending on context called. The keys to the hash are the individual test names and the value is a hashref with various interesting values. Each k/v pair might resemble something like this:

```
't/foo.t' => {
   elapsed        => '0.0428488254547119',
   gen            => '7',
   last_pass_time => '1219328376.07815',
   last_result    => '0',
   last_run_time  => '1219328376.07815',
   last_todo      => '0',
```

(continues on next page)

(continued from previous page)

```
    mtime           => '1191708862',
    seq             => '192',
    total_passes    => '6',
 }
```

test

```
my $test = $result->test('t/customer/create.t');
```

Returns an individual `App::Prove::State::Result::Test` instance for the given test name (usually the filename). Will return a new `App::Prove::State::Result::Test` instance if the name is not found.

test_names

Returns an list of test names, sorted by run order.

remove

```
$result->remove($test_name);            # remove the test
my $test = $result->test($test_name);   # fatal error
```

Removes a given test from results. This is a no-op if the test name is not found.

num_tests

Returns the number of tests for a given test suite result.

raw

Returns a hashref of raw results, suitable for serialization by YAML.

Modules

- `App::Prove::State::Result::Test` - Individual test results.

App::Prove::State::Result::Test

Individual test results.

VERSION

Version 3.52

DESCRIPTION

The `prove` command supports a `--state` option that instructs it to store persistent state across runs. This module encapsulates the results for a single test.

SYNOPSIS

```
# Re-run failed tests
$ prove --state=failed,save -rbv
```

METHODS

Class Methods

new

Instance Methods

name

The name of the test. Usually a filename.

elapsed

The total elapsed times the test took to run, in seconds from the epoch..

generation

The number for the “generation” of the test run. The first generation is 1 (one) and subsequent generations are 2, 3, etc.

last_pass_time

The last time the test program passed, in seconds from the epoch.

Returns `undef` if the program has never passed.

last_fail_time

The last time the test suite failed, in seconds from the epoch.

Returns `undef` if the program has never failed.

mtime

Returns the mtime of the test, in seconds from the epoch.

raw

Returns a hashref of raw test data, suitable for serialization by YAML.

result

Currently, whether or not the test suite passed with no 'problems' (such as TODO passed).

run_time

The total time it took for the test to run, in seconds. If `Time::HiRes` is available, it will have finer granularity.

num_todo

The number of tests with TODO directives.

sequence

The order in which this test was run for the given test suite result.

total_passes

The number of times the test has passed.

total_failures

The number of times the test has failed.

parser

The underlying parser object. This is useful if you need the full information for the test program.

5.2.5 Archive

Namespace containing the following modules:

- `Archive::Tar` - module for manipulations of tar archives

Archive::Tar

module for manipulations of tar archives

SYNOPSIS

```
use Archive::Tar;
my $tar = Archive::Tar->new;
```

(continues on next page)

(continued from previous page)

```
$tar->read('origin.tgz');
$tar->extract();

$tar->add_files('file/foo.pl', 'docs/README');
$tar->add_data('file/baz.txt', 'This is the contents now');

$tar->rename('oldname', 'new/file/name');
$tar->chown('/', 'root');
$tar->chown('/', 'root:root');
$tar->chmod('/tmp', '1777');

$tar->write('files.tar');                   # plain tar
$tar->write('files.tgz', COMPRESS_GZIP);    # gzip compressed
$tar->write('files.tbz', COMPRESS_BZIP);    # bzip2 compressed
$tar->write('files.txz', COMPRESS_XZ);      # xz compressed
```

DESCRIPTION

Archive::Tar provides an object oriented mechanism for handling tar files. It provides class methods for quick and easy files handling while also allowing for the creation of tar file objects for custom manipulation. If you have the IO::Zlib module installed, Archive::Tar will also support compressed or gzipped tar files.

An object of class Archive::Tar represents a .tar(.gz) archive full of files and things.

Object Methods

Archive::Tar->new([$file, $compressed])

Returns a new Tar object. If given any arguments, `new()` calls the `read()` method automatically, passing on the arguments provided to the `read()` method.

If `new()` is invoked with arguments and the `read()` method fails for any reason, `new()` returns undef.

$tar->read ($filename|$handle, [$compressed, {opt => 'val'}])

Read the given tar file into memory. The first argument can either be the name of a file or a reference to an already open filehandle (or an IO::Zlib object if it's compressed)

The `read` will *replace* any previous content in `$tar`!

The second argument may be considered optional, but remains for backwards compatibility. Archive::Tar now looks at the file magic to determine what class should be used to open the file and will transparently Do The Right Thing.

Archive::Tar will warn if you try to pass a bzip2 / xz compressed file and the IO::Uncompress::Bunzip2 / IO::Uncompress::UnXz are not available and simply return.

Note that you can currently **not** pass a `gzip` compressed filehandle, which is not opened with `IO::Zlib`, a `bzip2` compressed filehandle, which is not opened with `IO::Uncompress::Bunzip2`, a `xz` compressed filehandle, which is not opened with `IO::Uncom-`

`press::UnXz`, nor a string containing the full archive information (either compressed or uncompressed). These are worth while features, but not currently implemented. See the `TODO` section.

The third argument can be a hash reference with options. Note that all options are case-sensitive.

- limit

  Do not read more than `limit` files. This is useful if you have very big archives, and are only interested in the first few files.

- filter

  Can be set to a regular expression. Only files with names that match the expression will be read.

- md5

  Set to 1 and the md5sum of files will be returned (instead of file data) my $iter = Archive::Tar->iter($file, 1, {md5 => 1}); while(my $f = $iter->()) { print $f->data . "\t" . $f->full_path . $/; }

- extract

  If set to true, immediately extract entries when reading them. This gives you the same memory break as the `extract_archive` function. Note however that entries will not be read into memory, but written straight to disk. This means no `Archive::Tar::File` objects are created for you to inspect.

All files are stored internally as `Archive::Tar::File` objects. Please consult the *Archive::Tar::File* documentation for details.

Returns the number of files read in scalar context, and a list of `Archive::Tar::File` objects in list context.

$tar->contains_file($filename)

Check if the archive contains a certain file. It will return true if the file is in the archive, false otherwise.

Note however, that this function does an exact match using `eq` on the full path. So it cannot compensate for case-insensitive file- systems or compare 2 paths to see if they would point to the same underlying file.

$tar->extract([@filenames])

Write files whose names are equivalent to any of the names in `@filenames` to disk, creating subdirectories as necessary. This might not work too well under VMS. Under MacPerl, the file's modification time will be converted to the MacOS zero of time, and appropriate conversions will be done to the path. However, the length of each element of the path is not inspected to see whether it's longer than MacOS currently allows (32 characters).

If `extract` is called without a list of file names, the entire contents of the archive are extracted.

Returns a list of filenames extracted.

$tar->extract_file($file, [$extract_path])

Write an entry, whose name is equivalent to the file name provided to disk. Optionally takes a second parameter, which is the full native path (including filename) the entry will be written to.

For example:

```
$tar->extract_file( 'name/in/archive', 'name/i/want/to/give/it' );

$tar->extract_file( $at_file_object,   'name/i/want/to/give/it' );
```

Returns true on success, false on failure.

$tar->list_files([\@properties])

Returns a list of the names of all the files in the archive.

If `list_files()` is passed an array reference as its first argument it returns a list of hash references containing the requested properties of each file. The following list of properties is supported: name, size, mtime (last modified date), mode, uid, gid, linkname, uname, gname, devmajor, devminor, prefix.

Passing an array reference containing only one element, 'name', is special cased to return a list of names rather than a list of hash references, making it equivalent to calling `list_files` without arguments.

$tar->get_files([@filenames])

Returns the `Archive::Tar::File` objects matching the filenames provided. If no filename list was passed, all `Archive::Tar::File` objects in the current Tar object are returned.

Please refer to the `Archive::Tar::File` documentation on how to handle these objects.

$tar->get_content($file)

Return the content of the named file.

$tar->replace_content($file, $content)

Make the string $content be the content for the file named $file.

$tar->rename($file, $new_name)

Rename the file of the in-memory archive to $new_name.

Note that you must specify a Unix path for $new_name, since per tar standard, all files in the archive must be Unix paths.

Returns true on success and false on failure.

$tar->chmod($file, $mode)

Change mode of $file to $mode.

Returns true on success and false on failure.

$tar->chown($file, $uname [, $gname])

Change owner $file to $uname and $gname.

Returns true on success and false on failure.

$tar->remove (@filenamelist)

Removes any entries with names matching any of the given filenames from the in-memory archive. Returns a list of `Archive::Tar::File` objects that remain.

$tar->clear

`clear` clears the current in-memory archive. This effectively gives you a 'blank' object, ready to be filled again. Note that `clear` only has effect on the object, not the underlying tarfile.

$tar->write ([$file, $compressed, $prefix])

Write the in-memory archive to disk. The first argument can either be the name of a file or a reference to an already open filehandle (a GLOB reference).

The second argument is used to indicate compression. You can compress using `gzip`, `bzip2` or `xz`. If you pass a digit, it's assumed to be the `gzip` compression level (between 1 and 9), but the use of constants is preferred:

```
# write a gzip compressed file
$tar->write( 'out.tgz', COMPRESS_GZIP );

# write a bzip compressed file
$tar->write( 'out.tbz', COMPRESS_BZIP );

# write a xz compressed file
$tar->write( 'out.txz', COMPRESS_XZ );
```

Note that when you pass in a filehandle, the compression argument is ignored, as all files are printed verbatim to your filehandle. If you wish to enable compression with filehandles, use an `IO::Zlib`, `IO::Compress::Bzip2` or `IO::Compress::Xz` filehandle instead.

The third argument is an optional prefix. All files will be tucked away in the directory you specify as prefix. So if you have files 'a' and 'b' in your archive, and you specify 'foo' as prefix, they will be written to the archive as 'foo/a' and 'foo/b'.

If no arguments are given, `write` returns the entire formatted archive as a string, which could be useful if you'd like to stuff the archive into a socket or a pipe to gzip or something.

$tar->add_files(@filenamelist)

Takes a list of filenames and adds them to the in-memory archive.

The path to the file is automatically converted to a Unix like equivalent for use in the archive, and, if on MacOS, the file's modification time is converted from the MacOS epoch to the Unix epoch. So tar archives created on MacOS with **Archive::Tar** can be read both with *tar* on Unix and applications like *suntar* or *Stuffit Expander* on MacOS.

Be aware that the file's type/creator and resource fork will be lost, which is usually what you want in cross-platform archives.

Instead of a filename, you can also pass it an existing `Archive::Tar::File` object from, for example, another archive. The object will be clone, and effectively be a copy of the original, not an alias.

Returns a list of `Archive::Tar::File` objects that were just added.

$tar->add_data ($filename, $data, [$opthashref])

Takes a filename, a scalar full of data and optionally a reference to a hash with specific options.

Will add a file to the in-memory archive, with name `$filename` and content `$data`. Specific properties can be set using `$opthashref`. The following list of properties is supported: name, size, mtime (last modified date), mode, uid, gid, linkname, uname, gname, devmajor, devminor, prefix, type. (On MacOS, the file's path and modification times are converted to Unix equivalents.)

Valid values for the file type are the following constants defined by Archive::Tar::Constant:

- FILE

  Regular file.

- HARDLINK

- SYMLINK

  Hard and symbolic ("soft") links; linkname should specify target.

- CHARDEV

- BLOCKDEV

  Character and block devices. devmajor and devminor should specify the major and minor device numbers.

- DIR

  Directory.

- FIFO

  FIFO (named pipe).

- SOCKET

  Socket.

Returns the `Archive::Tar::File` object that was just added, or `undef` on failure.

$tar->error([$BOOL])

Returns the current error string (usually, the last error reported). If a true value was specified, it will give the `Carp::longmess` equivalent of the error, in effect giving you a stacktrace.

For backwards compatibility, this error is also available as `$Archive::Tar::error` although it is much recommended you use the method call instead.

$tar->setcwd($cwd);

`Archive::Tar` needs to know the current directory, and it will run `Cwd::cwd()` *every* time it extracts a *relative* entry from the tarfile and saves it in the file system. (As of version 1.30, however, `Archive::Tar` will use the speed optimization described below automatically, so it's only relevant if you're using `extract_file()`).

Since `Archive::Tar` doesn't change the current directory internally while it is extracting the items in a tarball, all calls to `Cwd::cwd()` can be avoided if we can guarantee that the current directory doesn't get changed externally.

To use this performance boost, set the current directory via

```
use Cwd;
$tar->setcwd( cwd() );
```

once before calling a function like `extract_file` and `Archive::Tar` will use the current directory setting from then on and won't call `Cwd::cwd()` internally.

To switch back to the default behaviour, use

```
$tar->setcwd( undef );
```

and `Archive::Tar` will call `Cwd::cwd()` internally again.

If you're using `Archive::Tar`'s `extract()` method, `setcwd()` will be called for you.

Class Methods

Archive::Tar->create_archive($file, $compressed, @filelist)

Creates a tar file from the list of files provided. The first argument can either be the name of the tar file to create or a reference to an open file handle (e.g. a GLOB reference).

The second argument is used to indicate compression. You can compress using `gzip`, `bzip2` or `xz`. If you pass a digit, it's assumed to be the `gzip` compression level (between 1 and 9), but the use of constants is preferred:

```
# write a gzip compressed file
Archive::Tar->create_archive( 'out.tgz', COMPRESS_GZIP, @filelist );

# write a bzip compressed file
Archive::Tar->create_archive( 'out.tbz', COMPRESS_BZIP, @filelist );

# write a xz compressed file
Archive::Tar->create_archive( 'out.txz', COMPRESS_XZ, @filelist );
```

Note that when you pass in a filehandle, the compression argument is ignored, as all files are printed verbatim to your filehandle. If you wish to enable compression with filehandles, use an `IO::Zlib`, `IO::Compress::Bzip2` or `IO::Compress::Xz` filehandle instead.

The remaining arguments list the files to be included in the tar file. These files must all exist. Any files which don't exist or can't be read are silently ignored.

If the archive creation fails for any reason, `create_archive` will return false. Please use the `error` method to find the cause of the failure.

Note that this method does not write on `the fly` as it were; it still reads all the files into memory before writing out the archive. Consult the FAQ below if this is a problem.

Archive::Tar->iter($filename, [$compressed, {opt => $val}])

Returns an iterator function that reads the tar file without loading it all in memory. Each time the function is called it will return the next file in the tarball. The files are returned as `Archive::Tar::File` objects. The iterator function returns the empty list once it has exhausted the files contained.

The second argument can be a hash reference with options, which are identical to the arguments passed to `read()`.

Example usage:

```
my $next = Archive::Tar->iter( "example.tar.gz", 1, {filter => qr/\.
 ↪pm$/} );

while( my $f = $next->() ) {
    print $f->name, "\n";

    $f->extract or warn "Extraction failed";

    # ....
}
```

Archive::Tar->list_archive($file, $compressed, [\@properties])

Returns a list of the names of all the files in the archive. The first argument can either be the name of the tar file to list or a reference to an open file handle (e.g. a GLOB reference).

If `list_archive()` is passed an array reference as its third argument it returns a list of hash references containing the requested properties of each file. The following list of properties is supported: full_path, name, size, mtime (last modified date), mode, uid, gid, linkname, uname, gname, devmajor, devminor, prefix, type.

See `Archive::Tar::File` for details about supported properties.

Passing an array reference containing only one element, 'name', is special cased to return a list of names rather than a list of hash references.

Archive::Tar->extract_archive($file, $compressed)

Extracts the contents of the tar file. The first argument can either be the name of the tar file to create or a reference to an open file handle (e.g. a GLOB reference). All relative paths in the tar file will be created underneath the current working directory.

`extract_archive` will return a list of files it extracted. If the archive extraction fails for any reason, `extract_archive` will return false. Please use the `error` method to find the cause of the failure.

$bool = Archive::Tar->has_io_string

Returns true if we currently have `IO::String` support loaded.

Either `IO::String` or `perlio` support is needed to support writing stringified archives. Currently, `perlio` is the preferred method, if available.

See the `GLOBAL VARIABLES` section to see how to change this preference.

$bool = Archive::Tar->has_perlio

Returns true if we currently have `perlio` support loaded.

This requires `perl-5.8` or higher, compiled with `perlio`

Either `IO::String` or `perlio` support is needed to support writing stringified archives. Currently, `perlio` is the preferred method, if available.

See the `GLOBAL VARIABLES` section to see how to change this preference.

$bool = Archive::Tar->has_zlib_support

Returns true if `Archive::Tar` can extract `zlib` compressed archives

$bool = Archive::Tar->has_bzip2_support

Returns true if `Archive::Tar` can extract `bzip2` compressed archives

$bool = Archive::Tar->has_xz_support

Returns true if `Archive::Tar` can extract `xz` compressed archives

Archive::Tar->can_handle_compressed_files

A simple checking routine, which will return true if `Archive::Tar` is able to uncompress compressed archives on the fly with `IO::Zlib`, `IO::Compress::Bzip2` and `IO::Compress::Xz` or false if not both are installed.

You can use this as a shortcut to determine whether `Archive::Tar` will do what you think before passing compressed archives to its `read` method.

GLOBAL VARIABLES

$Archive::Tar::FOLLOW_SYMLINK

Set this variable to `1` to make `Archive::Tar` effectively make a copy of the file when extracting. Default is `0`, which means the symlink stays intact. Of course, you will have to pack the file linked to as well.

This option is checked when you write out the tarfile using `write` or `create_archive`.

This works just like `/bin/tar`'s `-h` option.

$Archive::Tar::CHOWN

By default, `Archive::Tar` will try to `chown` your files if it is able to. In some cases, this may not be desired. In that case, set this variable to `0` to disable `chown`-ing, even if it were possible.

The default is 1.

$Archive::Tar::CHMOD

By default, `Archive::Tar` will try to `chmod` your files to whatever mode was specified for the particular file in the archive. In some cases, this may not be desired. In that case, set this variable to `0` to disable `chmod`-ing.

The default is 1.

$Archive::Tar::SAME_PERMISSIONS

When, `$Archive::Tar::CHMOD` is enabled, this setting controls whether the permissions on files from the archive are used without modification of if they are filtered by removing any setid bits and applying the current umask.

The default is 1 for the root user and `0` for normal users.

$Archive::Tar::DO_NOT_USE_PREFIX

By default, `Archive::Tar` will try to put paths that are over 100 characters in the `prefix` field of your tar header, as defined per POSIX-standard. However, some (older) tar programs do not implement this spec. To retain compatibility with these older or non-POSIX compliant versions, you can set the `$DO_NOT_USE_PREFIX` variable to a true value, and `Archive::Tar` will use an alternate way of dealing with paths over 100 characters by using the `GNU Extended Header` feature.

Note that clients who do not support the `GNU Extended Header` feature will not be able to read these archives. Such clients include tars on `Solaris`, `Irix` and `AIX`.

The default is `0`.

$Archive::Tar::DEBUG

Set this variable to 1 to always get the `Carp::longmess` output of the warnings, instead of the regular `carp`. This is the same message you would get by doing:

```
$tar->error(1);
```

Defaults to `0`.

$Archive::Tar::WARN

Set this variable to `0` if you do not want any warnings printed. Personally I recommend against doing this, but people asked for the option. Also, be advised that this is of course not threadsafe.

Defaults to 1.

$Archive::Tar::error

Holds the last reported error. Kept for historical reasons, but its use is very much discouraged. Use the `error()` method instead:

```
warn $tar->error unless $tar->extract;
```

Note that in older versions of this module, the `error()` method would return an effectively global value even when called an instance method as above. This has since been fixed, and multiple instances of `Archive::Tar` now have separate error strings.

$Archive::Tar::INSECURE_EXTRACT_MODE

This variable indicates whether `Archive::Tar` should allow files to be extracted outside their current working directory.

Allowing this could have security implications, as a malicious tar archive could alter or replace any file the extracting user has permissions to. Therefor, the default is to not allow insecure extractions.

If you trust the archive, or have other reasons to allow the archive to write files outside your current working directory, set this variable to `true`.

Note that this is a backwards incompatible change from version `1.36` and before.

$Archive::Tar::EXTRACT_HARDLINK

Prior to version `3.06` Archive::Tar would honour hardlinks in archives. Version `3.06` and onwards will skip hardlinks unless this flag is set. Exercise caution when enabling this.

$Archive::Tar::HAS_PERLIO

This variable holds a boolean indicating if we currently have `perlio` support loaded. This will be enabled for any perl greater than `5.8` compiled with `perlio`.

If you feel strongly about disabling it, set this variable to `false`. Note that you will then need `IO::String` installed to support writing stringified archives.

Don't change this variable unless you **really** know what you're doing.

$Archive::Tar::HAS_IO_STRING

This variable holds a boolean indicating if we currently have `IO::String` support loaded. This will be enabled for any perl that has a loadable `IO::String` module.

If you feel strongly about disabling it, set this variable to `false`. Note that you will then need `perlio` support from your perl to be able to write stringified archives.

Don't change this variable unless you **really** know what you're doing.

$Archive::Tar::ZERO_PAD_NUMBERS

This variable holds a boolean indicating if we will create zero padded numbers for `size`, `mtime` and `checksum`. The default is `0`, indicating that we will create space padded numbers. Added for compatibility with `busybox` implementations.

Tuning the way RESOLVE_SYMLINK will works

You can tune the behaviour by setting the $Archive::Tar::RESOLVE_SYMLINK variable, or $ENV{PERL5_AT_RESOLVE_SYMLINK} before loading the module Archive::Tar.

Values can be one of the following:

- none

  Disable this mechanism and failed as it was in previous version (<1.88)

- speed (default)

  If you prefer speed this will read again the whole archive using read() so all entries will be available

- memory

  If you prefer memory

Limitation: It won't work for terminal, pipe or sockets or every non seekable source.

$Archive::Tar::EXTRACT_BLOCK_SIZE

This variable holds an integer with the block size that should be used when writing files during extraction. It defaults to 1 GiB. Please note that this cannot be arbitrarily large since some operating systems limit the number of bytes that can be written in one call to `write(2)`, so if this is too large, extraction may fail with an error.

$Archive::Tar::MAX_FILE_SIZE

This variable holds an upper bound on the per-entry declared size that `Archive::Tar` will accept when reading an archive. Entries whose header claims a larger size are refused with an error before any read allocation. Defaults to 1 GiB. Set to 0 to disable the cap.

FAQ

- What's the minimum perl version required to run Archive::Tar?

  You will need perl version 5.005_03 or newer.

- Isn't Archive::Tar slow?

  Yes it is. It's pure perl, so it's a lot slower then your `/bin/tar` However, it's very portable. If speed is an issue, consider using `/bin/tar` instead.

- Isn't Archive::Tar heavier on memory than /bin/tar?

  Yes it is, see previous answer. Since `Compress::Zlib` and therefore `IO::Zlib` doesn't support `seek` on their filehandles, there is little choice but to read the archive into memory. This is ok if you want to do in-memory manipulation of the archive.

  If you just want to extract, use the `extract_archive` class method instead. It will optimize and write to disk immediately.

  Another option is to use the `iter` class method to iterate over the files in the tarball without reading them all in memory at once.

- Can you lazy-load data instead?

  In some cases, yes. You can use the `iter` class method to iterate over the files in the tarball without reading them all in memory at once.

- How much memory will an X kb tar file need?

  Probably more than X kb, since it will all be read into memory. If this is a problem, and you don't need to do in memory manipulation of the archive, consider using the `iter` class method, or `/bin/tar` instead.

- What do you do with unsupported filetypes in an archive?

  `Unix` has a few filetypes that aren't supported on other platforms, like `Win32`. If we encounter a `hardlink` or `symlink` we'll just try to make a copy of the original file, rather than throwing an error.

  This does require you to read the entire archive in to memory first, since otherwise we wouldn't know what data to fill the copy with. (This means that you cannot use the class methods, including `iter` on archives that have incompatible filetypes and still expect things to work).

  For other filetypes, like `chardevs` and `blockdevs` we'll warn that the extraction of this particular item didn't work.

- I'm using WinZip, or some other non-POSIX client, and files are not being extracted properly!

  By default, `Archive::Tar` is in a completely POSIX-compatible mode, which uses the POSIX-specification of `tar` to store files. For paths greater than 100 characters, this is done using the `POSIX header prefix`. Non-POSIX-compatible clients may not support this part of the specification, and may only support the `GNU Extended Header` functionality. To facilitate those clients, you can set the `$Archive::Tar::DO_NOT_USE_PREFIX` variable to `true`. See the `GLOBAL VARIABLES` section for details on this variable.

Note that GNU tar earlier than version 1.14 does not cope well with the POSIX header prefix. If you use such a version, consider setting the `$Archive::Tar::DO_NOT_USE_PREFIX` variable to `true`.

- How do I extract only files that have property X from an archive?

  Sometimes, you might not wish to extract a complete archive, just the files that are relevant to you, based on some criteria.

  You can do this by filtering a list of `Archive::Tar::File` objects based on your criteria. For example, to extract only files that have the string `foo` in their title, you would use:

```
$tar->extract(
    grep { $_->full_path =~ /foo/ } $tar->get_files
);
```

  This way, you can filter on any attribute of the files in the archive. Consult the `Archive::Tar::File` documentation on how to use these objects.

- How do I access .tar.Z files?

  The `Archive::Tar` module can optionally use `Compress::Zlib` (via the `IO::Zlib` module) to access tar files that have been compressed with `gzip`. Unfortunately tar files compressed with the Unix `compress` utility cannot be read by `Compress::Zlib` and so cannot be directly accesses by `Archive::Tar`.

  If the `uncompress` or `gunzip` programs are available, you can use one of these workarounds to read `.tar.Z` files from `Archive::Tar`

  Firstly with `uncompress`

```
use Archive::Tar;

open F, "uncompress -c $filename |";
my $tar = Archive::Tar->new(*F);
...
```

  and this with `gunzip`

```
use Archive::Tar;

open F, "gunzip -c $filename |";
my $tar = Archive::Tar->new(*F);
...
```

  Similarly, if the `compress` program is available, you can use this to write a `.tar.Z` file

```
use Archive::Tar;
use IO::File;

my $fh = IO::File->new( "| compress -c >$filename" );
my $tar = Archive::Tar->new();
...
```

(continues on next page)

(continued from previous page)

```
$tar->write($fh);
$fh->close ;
```

- How do I handle Unicode strings?

  `Archive::Tar` uses byte semantics for any files it reads from or writes to disk. This is not a problem if you only deal with files and never look at their content or work solely with byte strings. But if you use Unicode strings with character semantics, some additional steps need to be taken.

  For example, if you add a Unicode string like

```
# Problem
$tar->add_data('file.txt', "Euro: \x{20AC}");
```

  then there will be a problem later when the tarfile gets written out to disk via `$tar->write()`:

```
Wide character in print at .../Archive/Tar.pm line 1014.
```

  The data was added as a Unicode string and when writing it out to disk, the `:utf8` line discipline wasn't set by `Archive::Tar`, so Perl tried to convert the string to ISO-8859 and failed. The written file now contains garbage.

  For this reason, Unicode strings need to be converted to UTF-8-encoded bytestrings before they are handed off to `add_data()`:

```
use Encode;
my $data = "Accented character: \x{20AC}";
$data = encode('utf8', $data);

$tar->add_data('file.txt', $data);
```

  A opposite problem occurs if you extract a UTF8-encoded file from a tarball. Using `get_content()` on the `Archive::Tar::File` object will return its content as a bytestring, not as a Unicode string.

  If you want it to be a Unicode string (because you want character semantics with operations like regular expression matching), you need to decode the UTF8-encoded content and have Perl convert it into a Unicode string:

```
use Encode;
my $data = $tar->get_content();

# Make it a Unicode string
$data = decode('utf8', $data);
```

  There is no easy way to provide this functionality in `Archive::Tar`, because a tarball can contain many files, and each of which could be encoded in a different way.

CAVEATS

The AIX tar does not fill all unused space in the tar archive with 0x00. This sometimes leads to warning messages from `Archive::Tar`.

```
Invalid header block at offset nnn
```

A fix for that problem is scheduled to be released in the following levels of AIX, all of which should be coming out in the 4th quarter of 2009:

```
AIX 5.3 TL7 SP10
AIX 5.3 TL8 SP8
AIX 5.3 TL9 SP5
AIX 5.3 TL10 SP2

AIX 6.1 TL0 SP11
AIX 6.1 TL1 SP7
AIX 6.1 TL2 SP6
AIX 6.1 TL3 SP3
```

The IBM APAR number for this problem is IZ50240 (Reported component ID: 5765G0300 / AIX 5.3). It is possible to get an ifix for that problem. If you need an ifix please contact your local IBM AIX support.

TODO

- Check if passed in handles are open for read/write

  Currently I don't know of any portable pure perl way to do this. Suggestions welcome.

- Allow archives to be passed in as string

  Currently, we only allow opened filehandles or filenames, but not strings. The internals would need some reworking to facilitate stringified archives.

- Facilitate processing an opened filehandle of a compressed archive

  Currently, we only support this if the filehandle is an IO::Zlib object. Environments, like apache, will present you with an opened filehandle to an uploaded file, which might be a compressed archive.

SEE ALSO

- The GNU tar specification

  https://www.gnu.org/software/tar/manual/tar.html

- The PAX format specification

  The specification which tar derives from; https://pubs.opengroup.org/onlinepubs/007904975/utilities/pax.html

AUTHOR

This module by Jos Boumans kane@cpan.org.

Please reports bugs to bug-archive-tar@rt.cpan.org.

ACKNOWLEDGEMENTS

Thanks to Sean Burke, Chris Nandor, Chip Salzenberg, Tim Heaney, Gisle Aas, Rainer Tammer and especially Andrew Savige for their help and suggestions.

COPYRIGHT

This module is copyright (c) 2002 - 2009 Jos Boumans kane@cpan.org. All rights reserved.

This library is free software; you may redistribute and/or modify it under the same terms as Perl itself.

Modules

- `Archive::Tar::File` - a subclass for in-memory extracted file from Archive::Tar

Archive::Tar::File

a subclass for in-memory extracted file from Archive::Tar

SYNOPSIS

```
my @items = $tar->get_files;

print $_->name, ' ', $_->size, "\n" for @items;

print $object->get_content;
$object->replace_content('new content');

$object->rename( 'new/full/path/to/file.c' );
```

DESCRIPTION

Archive::Tar::File provides a neat little object layer for in-memory extracted files. It's mostly used internally in Archive::Tar to tidy up the code, but there's no reason users shouldn't use this API as well.

Accessors

A lot of the methods in this package are accessors to the various fields in the tar header:

- name

  The file's name

- mode

  The file's mode

- uid

  The user id owning the file

- gid

  The group id owning the file

- size

  File size in bytes

- mtime

  Modification time. Adjusted to mac-time on MacOS if required

- chksum

  Checksum field for the tar header

- type

  File type – numeric, but comparable to exported constants – see Archive::Tar's documentation

- linkname

  If the file is a symlink, the file it's pointing to

- magic

  Tar magic string – not useful for most users

- version

  Tar version string – not useful for most users

- uname

  The user name that owns the file

- gname

  The group name that owns the file

- devmajor

  Device major number in case of a special file

- devminor

  Device minor number in case of a special file

- prefix

  Any directory to prefix to the extraction path, if any

- raw

  Raw tar header – not useful for most users

Methods

Archive::Tar::File->new(file => $path)

Returns a new Archive::Tar::File object from an existing file.

Returns undef on failure.

Archive::Tar::File->new(data => $path, $data, $opt)

Returns a new Archive::Tar::File object from data.

`$path` defines the file name (which need not exist), `$data` the file contents, and `$opt` is a reference to a hash of attributes which may be used to override the default attributes (fields in the tar header), which are described above in the Accessors section.

Returns undef on failure.

Archive::Tar::File->new(chunk => $chunk)

Returns a new Archive::Tar::File object from a raw 512-byte tar archive chunk.

Returns undef on failure.

$bool = $file->extract([$alternative_name])

Extract this object, optionally to an alternative name.

See `Archive::Tar->extract_file` for details.

Returns true on success and false on failure.

$path = $file->full_path

Returns the full path from the tar header; this is basically a concatenation of the `prefix` and `name` fields.

$bool = $file->validate

Done by Archive::Tar internally when reading the tar file: validate the header against the checksum to ensure integer tar file.

Returns true on success, false on failure

$bool = $file->has_content

Returns a boolean to indicate whether the current object has content. Some special files like directories and so on never will have any content. This method is mainly to make sure you don't get warnings for using uninitialized values when looking at an object's content.

$content = $file->get_content

Returns the current content for the in-memory file

$cref = $file->get_content_by_ref

Returns the current content for the in-memory file as a scalar reference. Normal users won't need this, but it will save memory if you are dealing with very large data files in your tar archive, since it will pass the contents by reference, rather than make a copy of it first.

$bool = $file->replace_content($content)

Replace the current content of the file with the new content. This only affects the in-memory archive, not the on-disk version until you write it.

Returns true on success, false on failure.

$bool = $file->rename($new_name)

Rename the current file to $new_name.

Note that you must specify a Unix path for $new_name, since per tar standard, all files in the archive must be Unix paths.

Returns true on success and false on failure.

$bool = $file->chmod($mode)

Change mode of $file to $mode. The mode can be a string or a number which is interpreted as octal whether or not a leading 0 is given.

Returns true on success and false on failure.

$bool = $file->chown($user [, $group])

Change owner of $file to $user. If a $group is given that is changed as well. You can also pass a single parameter with a colon separating the use and group as in 'root:wheel'.

Returns true on success and false on failure.

Convenience methods

To quickly check the type of a `Archive::Tar::File` object, you can use the following methods:

- $file->is_file

  Returns true if the file is of type `file`

- $file->is_dir

  Returns true if the file is of type `dir`

- $file->is_hardlink

  Returns true if the file is of type `hardlink`

- $file->is_symlink

  Returns true if the file is of type `symlink`

- $file->is_chardev

  Returns true if the file is of type `chardev`

- $file->is_blockdev

  Returns true if the file is of type `blockdev`

- $file->is_fifo

  Returns true if the file is of type `fifo`

- $file->is_socket

  Returns true if the file is of type `socket`

- $file->is_longlink

  Returns true if the file is of type `LongLink`. Should not happen after a successful `read`.

- $file->is_label

  Returns true if the file is of type `Label`. Should not happen after a successful `read`.

- $file->is_unknown

  Returns true if the file type is unknown

5.2.6 Attribute

Namespace containing the following modules:

- `Attribute::Handlers` - Simpler definition of attribute handlers

Attribute::Handlers

Simpler definition of attribute handlers

VERSION

This document describes version 1.03 of Attribute::Handlers.

SYNOPSIS

```
package MyClass;
require 5.006;
use Attribute::Handlers;
no warnings 'redefine';
```

(continues on next page)

(continued from previous page)

```
sub Good : ATTR(SCALAR) {
    my ($package, $symbol, $referent, $attr, $data) = @_;

    # Invoked for any scalar variable with a :Good attribute,
    # provided the variable was declared in MyClass (or
    # a derived class) or typed to MyClass.

    # Do whatever to $referent here (executed in CHECK phase).
    ...
}

sub Bad : ATTR(SCALAR) {
    # Invoked for any scalar variable with a :Bad attribute,
    # provided the variable was declared in MyClass (or
    # a derived class) or typed to MyClass.
    ...
}

sub Good : ATTR(ARRAY) {
    # Invoked for any array variable with a :Good attribute,
    # provided the variable was declared in MyClass (or
    # a derived class) or typed to MyClass.
    ...
}

sub Good : ATTR(HASH) {
    # Invoked for any hash variable with a :Good attribute,
    # provided the variable was declared in MyClass (or
    # a derived class) or typed to MyClass.
    ...
}

sub Ugly : ATTR(CODE) {
    # Invoked for any subroutine declared in MyClass (or a
    # derived class) with an :Ugly attribute.
    ...
}

sub Omni : ATTR {
    # Invoked for any scalar, array, hash, or subroutine
    # with an :Omni attribute, provided the variable or
    # subroutine was declared in MyClass (or a derived class)
    # or the variable was typed to MyClass.
    # Use ref($_[2]) to determine what kind of referent it was.
    ...
}


use Attribute::Handlers autotie => { Cycle => Tie::Cycle };
```

(continues on next page)

(continued from previous page)

```
my $next : Cycle(['A'..'Z']);
```

DESCRIPTION

This module, when inherited by a package, allows that package's class to define attribute handler subroutines for specific attributes. Variables and subroutines subsequently defined in that package, or in packages derived from that package may be given attributes with the same names as the attribute handler subroutines, which will then be called in one of the compilation phases (i.e. in a `BEGIN`, `CHECK`, `INIT`, or `END` block). (`UNITCHECK` blocks don't correspond to a global compilation phase, so they can't be specified here.)

To create a handler, define it as a subroutine with the same name as the desired attribute, and declare the subroutine itself with the
attribute `:ATTR`. For example:

```
package LoudDecl;
use Attribute::Handlers;

sub Loud :ATTR {
    my ($package, $symbol, $referent, $attr, $data, $phase,
        $filename, $linenum) = @_;
    print STDERR
        ref($referent), " ",
        *{$symbol}{NAME}, " ",
        "($referent) ", "was just declared ",
        "and ascribed the ${attr} attribute ",
        "with data ($data)\n",
        "in phase $phase\n",
        "in file $filename at line $linenum\n";
}
```

This creates a handler for the attribute `:Loud` in the class LoudDecl. Thereafter, any subroutine declared with a `:Loud` attribute in the class LoudDecl:

```
package LoudDecl;

sub foo: Loud {...}
```

causes the above handler to be invoked, and passed:

- [0]

  the name of the package into which it was declared;

- [1]

  a reference to the symbol table entry (typeglob) containing the subroutine;

- [2]

  a reference to the subroutine;

- [3]

the name of the attribute;

- [4]

  any data associated with that attribute;

- [5]

  the name of the phase in which the handler is being invoked;

- [6]

  the filename in which the handler is being invoked;

- [7]

  the line number in this file.

Likewise, declaring any variables with the `:Loud` attribute within the package:

```
package LoudDecl;

my $foo :Loud;
my @foo :Loud;
my %foo :Loud;
```

will cause the handler to be called with a similar argument list (except, of course, that `$_[2]` will be a reference to the variable).

The package name argument will typically be the name of the class into which the subroutine was declared, but it may also be the name of a derived class (since handlers are inherited).

If a lexical variable is given an attribute, there is no symbol table to which it belongs, so the symbol table argument (`$_[1]`) is set to the string `'LEXICAL'` in that case. Likewise, ascribing an attribute to an anonymous subroutine results in a symbol table argument of `'ANON'`.

The data argument passes in the value (if any) associated with the attribute. For example, if `&foo` had been declared:

```
sub foo :Loud("turn it up to 11, man!") {...}
```

then a reference to an array containing the string `"turn it up to 11, man!"` would be passed as the last argument.

Attribute::Handlers makes strenuous efforts to convert the data argument (`$_[4]`) to a usable form before passing it to the handler (but see "Non-interpretive attribute handlers"). If those efforts succeed, the interpreted data is passed in an array reference; if they fail, the raw data is passed as a string. For example, all of these:

```
sub foo :Loud(till=>ears=>are=>bleeding) {...}
sub foo :Loud(qw/till ears are bleeding/) {...}
sub foo :Loud(qw/till, ears, are, bleeding/) {...}
sub foo :Loud(till,ears,are,bleeding) {...}
```

causes it to pass `['till','ears','are','bleeding']` as the handler's data argument. While:

```
sub foo :Loud(['till','ears','are','bleeding']) {...}
```

causes it to pass `[ ['till','ears','are','bleeding'] ]`; the array reference specified in the data being passed inside the standard array reference indicating successful interpretation.

However, if the data can't be parsed as valid Perl, then it is passed as an uninterpreted string. For example:

```
sub foo :Loud(my,ears,are,bleeding) {...}
sub foo :Loud(qw/my ears are bleeding) {...}
```

cause the strings `'my,ears,are,bleeding'` and `'qw/my ears are bleeding'` respectively to be passed as the data argument.

If no value is associated with the attribute, `undef` is passed.

Typed lexicals

Regardless of the package in which it is declared, if a lexical variable is ascribed an attribute, the handler that is invoked is the one belonging to the package to which it is typed. For example, the following declarations:

```
package OtherClass;

my LoudDecl $loudobj : Loud;
my LoudDecl @loudobjs : Loud;
my LoudDecl %loudobjex : Loud;
```

causes the LoudDecl::Loud handler to be invoked (even if OtherClass also defines a handler for `:Loud` attributes).

Type-specific attribute handlers

If an attribute handler is declared and the `:ATTR` specifier is given the name of a built-in type (`SCALAR`, `ARRAY`, `HASH`, or `CODE`), the handler is only applied to declarations of that type. For example, the following definition:

```
package LoudDecl;

sub RealLoud :ATTR(SCALAR) { print "Yeeeeow!" }
```

creates an attribute handler that applies only to scalars:

```
package Painful;
use base LoudDecl;

my $metal : RealLoud;           # invokes &LoudDecl::RealLoud
my @metal : RealLoud;           # error: unknown attribute
my %metal : RealLoud;           # error: unknown attribute
sub metal : RealLoud {...}      # error: unknown attribute
```

You can, of course, declare separate handlers for these types as well (but you'll need to specify `no warnings 'redefine'` to do it quietly):

```
package LoudDecl;
use Attribute::Handlers;
no warnings 'redefine';

sub RealLoud :ATTR(SCALAR) { print "Yeeeeow!" }
sub RealLoud :ATTR(ARRAY) { print "Urrrrrrrrrr!" }
sub RealLoud :ATTR(HASH) { print "Arrrrrgggghhhhhh!" }
sub RealLoud :ATTR(CODE) { croak "Real loud sub torpedoed" }
```

You can also explicitly indicate that a single handler is meant to be used for all types of referents like so:

```
package LoudDecl;
use Attribute::Handlers;

sub SeriousLoud :ATTR(ANY) { warn "Hearing loss imminent" }
```

(I.e. `ATTR(ANY)` is a synonym for `:ATTR`).

Non-interpretive attribute handlers

Occasionally the strenuous efforts Attribute::Handlers makes to convert the data argument (`$_[4]`) to a usable form before passing it to the handler get in the way.

You can turn off that eagerness-to-help by declaring an attribute handler with the keyword `RAWDATA`. For example:

```
sub Raw          : ATTR(RAWDATA) {...}
sub Nekkid       : ATTR(SCALAR,RAWDATA) {...}
sub Au::Naturale : ATTR(RAWDATA,ANY) {...}
```

Then the handler makes absolutely no attempt to interpret the data it receives and simply passes it as a string:

```
my $power : Raw(1..100);        # handlers receives "1..100"
```

Phase-specific attribute handlers

By default, attribute handlers are called at the end of the compilation phase (in a `CHECK` block). This seems to be optimal in most cases because most things that can be defined are defined by that point but nothing has been executed.

However, it is possible to set up attribute handlers that are called at other points in the program's compilation or execution, by explicitly stating the phase (or phases) in which you wish the attribute handler to be called. For example:

```
sub Early    :ATTR(SCALAR,BEGIN) {...}
sub Normal   :ATTR(SCALAR,CHECK) {...}
sub Late     :ATTR(SCALAR,INIT) {...}
```

(continues on next page)

(continued from previous page)

```
sub Final    :ATTR(SCALAR,END) {...}
sub Bookends :ATTR(SCALAR,BEGIN,END) {...}
```

As the last example indicates, a handler may be set up to be (re)called in two or more phases. The phase name is passed as the handler's final argument.

Note that attribute handlers that are scheduled for the `BEGIN` phase are handled as soon as the attribute is detected (i.e. before any subsequently defined `BEGIN` blocks are executed).

Attributes as `tie` interfaces

Attributes make an excellent and intuitive interface through which to tie variables. For example:

```
use Attribute::Handlers;
use Tie::Cycle;

sub UNIVERSAL::Cycle : ATTR(SCALAR) {
    my ($package, $symbol, $referent, $attr, $data, $phase) = @_;
    $data = [ $data ] unless ref $data eq 'ARRAY';
    tie $$referent, 'Tie::Cycle', $data;
}

# and thereafter...

package main;

my $next : Cycle('A'..'Z');     # $next is now a tied variable

while (<>) {
    print $next;
}
```

Note that, because the `Cycle` attribute receives its arguments in the `$data` variable, if the attribute is given a list of arguments, `$data` will consist of a single array reference; otherwise, it will consist of the single argument directly. Since Tie::Cycle requires its cycling values to be passed as an array reference, this means that we need to wrap non-array-reference arguments in an array constructor:

```
$data = [ $data ] unless ref $data eq 'ARRAY';
```

Typically, however, things are the other way around: the tieable class expects its arguments as a flattened list, so the attribute looks like:

```
sub UNIVERSAL::Cycle : ATTR(SCALAR) {
    my ($package, $symbol, $referent, $attr, $data, $phase) = @_;
    my @data = ref $data eq 'ARRAY' ? @$data : $data;
    tie $$referent, 'Tie::Whatever', @data;
}
```

This software pattern is so widely applicable that Attribute::Handlers provides a way to automate it: specifying `'autotie'` in the `use Attribute::Handlers` statement. So, the cycling example, could also be written:

```
use Attribute::Handlers autotie => { Cycle => 'Tie::Cycle' };

# and thereafter...

package main;

my $next : Cycle(['A'..'Z']);     # $next is now a tied variable

while (<>) {
    print $next;
}
```

Note that we now have to pass the cycling values as an array reference, since the `autotie` mechanism passes `tie` a list of arguments as a list (as in the Tie::Whatever example), *not* as an array reference (as in the original Tie::Cycle example at the start of this section).

The argument after `'autotie'` is a reference to a hash in which each key is the name of an attribute to be created, and each value is the class to which variables ascribed that attribute should be tied.

Note that there is no longer any need to import the Tie::Cycle module – Attribute::Handlers takes care of that automagically. You can even pass arguments to the module's `import` subroutine, by appending them to the class name. For example:

```
use Attribute::Handlers
     autotie => { Dir => 'Tie::Dir qw(DIR_UNLINK)' };
```

If the attribute name is unqualified, the attribute is installed in the current package. Otherwise it is installed in the qualifier's package:

```
package Here;

use Attribute::Handlers autotie => {
     Other::Good => Tie::SecureHash, # tie attr installed in Other::
             Bad => Tie::Taxes,      # tie attr installed in Here::
 UNIVERSAL::Ugly => Software::Patent # tie attr installed everywhere
};
```

Autoties are most commonly used in the module to which they actually tie, and need to export their attributes to any module that calls them. To facilitate this, Attribute::Handlers recognizes a special "pseudo-class" – `__CALLER__`, which may be specified as the qualifier of an attribute:

```
package Tie::Me::Kangaroo::Down::Sport;

use Attribute::Handlers autotie =>
     { '__CALLER__::Roo' => __PACKAGE__ };
```

This causes Attribute::Handlers to define the Roo attribute in the package that imports the Tie::Me::Kangaroo::Down::Sport module.

Note that it is important to quote the __CALLER__::Roo identifier because a bug in perl 5.8 will refuse to parse it and cause an unknown error.

Passing the tied object to `tie`

Occasionally it is important to pass a reference to the object being tied to the TIESCALAR, TIEHASH, etc. that ties it.

The `autotie` mechanism supports this too. The following code:

```
use Attribute::Handlers autotieref => { Selfish => Tie::Selfish };
my $var : Selfish(@args);
```

has the same effect as:

```
tie my $var, 'Tie::Selfish', @args;
```

But when `"autotieref"` is used instead of `"autotie"`:

```
use Attribute::Handlers autotieref => { Selfish => Tie::Selfish };
my $var : Selfish(@args);
```

the effect is to pass the `tie` call an extra reference to the variable being tied:

```
tie my $var, 'Tie::Selfish', \$var, @args;
```

EXAMPLES

If the class shown in “SYNOPSIS” were placed in the MyClass.pm module, then the following code:

```
package main;
use MyClass;

my MyClass $slr :Good :Bad(1**1-1) :Omni(-vorous);

package SomeOtherClass;
use base MyClass;

sub tent { 'acle' }

sub fn :Ugly(sister) :Omni('po',tent()) {...}
my @arr :Good :Omni(s/cie/nt/);
my %hsh :Good(q/bye/) :Omni(q/bus/);
```

would cause the following handlers to be invoked:

```
# my MyClass $slr :Good :Bad(1**1-1) :Omni(-vorous);
```

(continues on next page)

(continued from previous page)

```
MyClass::Good:ATTR(SCALAR)( 'MyClass',          # class
                            'LEXICAL',          # no typeglob
                            \$slr,              # referent
                            'Good',             # attr name
                            undef               # no attr data
                            'CHECK',            # compiler phase
                          );

MyClass::Bad:ATTR(SCALAR)( 'MyClass',           # class
                           'LEXICAL',           # no typeglob
                           \$slr,               # referent
                           'Bad',               # attr name
                           0                    # eval'd attr data
                           'CHECK',             # compiler phase
                         );

MyClass::Omni:ATTR(SCALAR)( 'MyClass',          # class
                            'LEXICAL',          # no typeglob
                            \$slr,              # referent
                            'Omni',             # attr name
                            '-vorous'           # eval'd attr data
                            'CHECK',            # compiler phase
                          );


# sub fn :Ugly(sister) :Omni('po',tent()) {...}

MyClass::UGLY:ATTR(CODE)( 'SomeOtherClass',     # class
                          \*SomeOtherClass::fn, # typeglob
                          \&SomeOtherClass::fn, # referent
                          'Ugly',               # attr name
                          'sister'              # eval'd attr data
                          'CHECK',              # compiler phase
                        );

MyClass::Omni:ATTR(CODE)( 'SomeOtherClass',     # class
                          \*SomeOtherClass::fn, # typeglob
                          \&SomeOtherClass::fn, # referent
                          'Omni',               # attr name
                          ['po','acle']         # eval'd attr data
                          'CHECK',              # compiler phase
                        );


# my @arr :Good :Omni(s/cie/nt/);

MyClass::Good:ATTR(ARRAY)( 'SomeOtherClass',    # class
                           'LEXICAL',           # no typeglob
                           \@arr,               # referent
```

(continues on next page)

(continued from previous page)

```
                              'Good',              # attr name
                              undef                # no attr data
                              'CHECK',             # compiler phase
                            );

MyClass::Omni:ATTR(ARRAY)( 'SomeOtherClass',     # class
                              'LEXICAL',           # no typeglob
                              \@arr,               # referent
                              'Omni',              # attr name
                              ""                   # eval'd attr data
                              'CHECK',             # compiler phase
                            );


# my %hsh :Good(q/bye) :Omni(q/bus/);

MyClass::Good:ATTR(HASH)( 'SomeOtherClass',      # class
                              'LEXICAL',           # no typeglob
                              \%hsh,               # referent
                              'Good',              # attr name
                              'q/bye'              # raw attr data
                              'CHECK',             # compiler phase
                            );

MyClass::Omni:ATTR(HASH)( 'SomeOtherClass',      # class
                              'LEXICAL',           # no typeglob
                              \%hsh,               # referent
                              'Omni',              # attr name
                              'bus'                # eval'd attr data
                              'CHECK',             # compiler phase
                            );
```

Installing handlers into UNIVERSAL, makes them...err..universal. For example:

```
package Descriptions;
use Attribute::Handlers;

my %name;
sub name { return $name{$_[2]}||*{$_[1]}{NAME} }

sub UNIVERSAL::Name :ATTR {
    $name{$_[2]} = $_[4];
}

sub UNIVERSAL::Purpose :ATTR {
    print STDERR "Purpose of ", &name, " is $_[4]\n";
}

sub UNIVERSAL::Unit :ATTR {
```

(continues on next page)

(continued from previous page)

```
    print STDERR &name, " measured in $_[4]\n";
}
```

Let's you write:

```
use Descriptions;

my $capacity : Name(capacity)
             : Purpose(to store max storage capacity for files)
             : Unit(Gb);


package Other;

sub foo : Purpose(to foo all data before barring it) { }

# etc.
```

UTILITY FUNCTIONS

This module offers a single utility function, `findsym()`.

- findsym

  ```
  my $symbol = Attribute::Handlers::findsym($package, $referent);
  ```

  The function looks in the symbol table of `$package` for the typeglob for `$referent`, which is a reference to a variable or subroutine (SCALAR, ARRAY, HASH, or CODE). If it finds the typeglob, it returns it. Otherwise, it returns undef. Note that `findsym` memoizes the typeglobs it has previously successfully found, so subsequent calls with the same arguments should be much faster.

DIAGNOSTICS

- `Bad attribute type: ATTR(%s)`

  An attribute handler was specified with an `:ATTR(_ref_type_)`, but the type of referent it was defined to handle wasn't one of the five permitted: `SCALAR`, `ARRAY`, `HASH`, `CODE`, or `ANY`.

- `Attribute handler %s doesn't handle %s attributes`

  A handler for attributes of the specified name *was* defined, but not for the specified type of declaration. Typically encountered when trying to apply a `VAR` attribute handler to a subroutine, or a `SCALAR` attribute handler to some other type of variable.

- `Declaration of %s attribute in package %s may clash with future reserved word`

  A handler for an attributes with an all-lowercase name was declared. An attribute with an all-lowercase name might have a meaning to Perl itself some day, even though most don't yet. Use a mixed-case attribute name, instead.

- `Can't have two ATTR specifiers on one subroutine`

  You just can't, okay? Instead, put all the specifications together with commas between them in a single `ATTR(_specification_)`.

- `Can't autotie a %s`

  You can only declare autoties for types `"SCALAR"`, `"ARRAY"`, and `"HASH"`. They're the only things (apart from typeglobs – which are not declarable) that Perl can tie.

- `Internal error: %s symbol went missing`

  Something is rotten in the state of the program. An attributed subroutine ceased to exist between the point it was declared and the point at which its attribute handler(s) would have been called.

- `Won't be able to apply END handler`

  You have defined an END handler for an attribute that is being applied to a lexical variable. Since the variable may not be available during END this won't happen.

AUTHOR

Damian Conway (damian@conway.org). The maintainer of this module is now Rafael Garcia-Suarez (rgarciasuarez@gmail.com).

Maintainer of the CPAN release is Steffen Mueller (smueller@cpan.org). Contact him with technical difficulties with respect to the packaging of the CPAN module.

BUGS

There are undoubtedly serious bugs lurking somewhere in code this funky :-) Bug reports and other feedback are most welcome.

COPYRIGHT AND LICENSE

```
  Copyright (c) 2001-2014, Damian Conway. All Rights Reserved.
This module is free software. It may be used, redistributed
    and/or modified under the same terms as Perl itself.
```

5.2.7 AutoLoader

load subroutines only on demand

SYNOPSIS

```
package Foo;
use AutoLoader 'AUTOLOAD';   # import the default AUTOLOAD␣
↪subroutine

package Bar;
use AutoLoader;              # don't import AUTOLOAD, define our own
sub AUTOLOAD {
```

(continues on next page)

(continued from previous page)

```
    ...
    $AutoLoader::AUTOLOAD = "...";
    goto &AutoLoader::AUTOLOAD;
}
```

DESCRIPTION

The **AutoLoader** module works with the **AutoSplit** module and the `__END__` token to defer the loading of some subroutines until they are used rather than loading them all at once.

To use **AutoLoader**, the author of a module has to place the definitions of subroutines to be autoloaded after an `__END__` token. (See *perldata*.) The **AutoSplit** module can then be run manually to extract the definitions into individual files `auto/funcname.al`.

AutoLoader implements an AUTOLOAD subroutine. When an undefined subroutine in is called in a client module of **AutoLoader**, **AutoLoader**'s AUTOLOAD subroutine attempts to locate the subroutine in a file with a name related to the location of the file from which the client module was read. As an example, if `POSIX.pm` is located in `/usr/local/lib/perl5/POSIX.pm`, **AutoLoader** will look for perl subroutines **POSIX** in `/usr/local/lib/perl5/auto/POSIX/*.al`, where the `.al` file has the same name as the subroutine, sans package. If such a file exists, AUTOLOAD will read and evaluate it, thus (presumably) defining the needed subroutine. AUTOLOAD will then `goto` the newly defined subroutine.

Once this process completes for a given function, it is defined, so future calls to the subroutine will bypass the AUTOLOAD mechanism.

Subroutine Stubs

In order for object method lookup and/or prototype checking to operate correctly even when methods have not yet been defined it is necessary to "forward declare" each subroutine (as in `sub NAME;`). See *"SYNOPSIS" in perlsub*. Such forward declaration creates "subroutine stubs", which are place holders with no code.

The AutoSplit and **AutoLoader** modules automate the creation of forward declarations. The AutoSplit module creates an 'index' file containing forward declarations of all the AutoSplit subroutines. When the AutoLoader module is 'use'd it loads these declarations into its callers package.

Because of this mechanism it is important that **AutoLoader** is always `used` and not `required`.

Using AutoLoader's AUTOLOAD Subroutine

In order to use **AutoLoader**'s AUTOLOAD subroutine you *must* explicitly import it:

```
use AutoLoader 'AUTOLOAD';
```

Overriding AutoLoader's AUTOLOAD Subroutine

Some modules, mainly extensions, provide their own AUTOLOAD subroutines. They typically need to check for some special cases (such as constants) and then fallback to **AutoLoader**'s AUTOLOAD for the rest.

Such modules should *not* import **AutoLoader**'s AUTOLOAD subroutine. Instead, they should define their own AUTOLOAD subroutines along these lines:

```
use AutoLoader;
use Carp;

sub AUTOLOAD {
    my $sub = $AUTOLOAD;
    (my $constname = $sub) =~ s/.*::///;
    my $val = constant($constname, @_ ? $_[0] : 0);
    if ($! != 0) {
        if ($! =~ /Invalid/ || $!{EINVAL}) {
            $AutoLoader::AUTOLOAD = $sub;
            goto &AutoLoader::AUTOLOAD;
        }
        else {
            croak "Your vendor has not defined constant $constname";
        }
    }
    *$sub = sub { $val }; # same as: eval "sub $sub { $val }";
    goto &$sub;
}
```

If any module's own AUTOLOAD subroutine has no need to fallback to the AutoLoader's AUTOLOAD subroutine (because it doesn't have any AutoSplit subroutines), then that module should not use **AutoLoader** at all.

Package Lexicals

Package lexicals declared with `my` in the main block of a package using **AutoLoader** will not be visible to auto-loaded subroutines, due to the fact that the given scope ends at the `__END__` marker. A module using such variables as package globals will not work properly under the **AutoLoader**.

The `vars` pragma (see "vars" in perlmod) may be used in such situations as an alternative to explicitly qualifying all globals with the package namespace. Variables pre-declared with this pragma will be visible to any autoloaded routines (but will not be invisible outside the package, unfortunately).

Not Using AutoLoader

You can stop using AutoLoader by simply

```
no AutoLoader;
```

AutoLoader vs. SelfLoader

The **AutoLoader** is similar in purpose to **SelfLoader**: both delay the loading of subroutines.

SelfLoader uses the `__DATA__` marker rather than `__END__`. While this avoids the use of a hierarchy of disk files and the associated open/close for each routine loaded, **SelfLoader** suffers a startup speed disadvantage in the one-time parsing of the lines after `__DATA__`, after which routines are cached. **SelfLoader** can also handle multiple packages in a file.

AutoLoader only reads code as it is requested, and in many cases should be faster, but requires a mechanism like **AutoSplit** be used to create the individual files. *ExtUtils::MakeMaker* will invoke **AutoSplit** automatically if **AutoLoader** is used in a module source file.

Forcing AutoLoader to Load a Function

Sometimes, it can be necessary or useful to make sure that a certain function is fully loaded by AutoLoader. This is the case, for example, when you need to wrap a function to inject debugging code. It is also helpful to force early loading of code before forking to make use of copy-on-write as much as possible.

Starting with AutoLoader 5.73, you can call the `AutoLoader::autoload_sub` function with the fully-qualified name of the function to load from its `.al` file. The behaviour is exactly the same as if you called the function, triggering the regular `AUTOLOAD` mechanism, but it does not actually execute the autoloaded function.

CAVEATS

AutoLoaders prior to Perl 5.002 had a slightly different interface. Any old modules which use **AutoLoader** should be changed to the new calling style. Typically this just means changing a require to a use, adding the explicit `'AUTOLOAD'` import if needed, and removing **AutoLoader** from `@ISA`.

On systems with restrictions on file name length, the file corresponding to a subroutine may have a shorter name that the routine itself. This can lead to conflicting file names. The *AutoSplit* package warns of these potential conflicts when used to split a module.

AutoLoader may fail to find the autosplit files (or even find the wrong ones) in cases where `@INC` contains relative paths, **and** the program does `chdir`.

SEE ALSO

SelfLoader - an autoloader that doesn't use external files.

AUTHOR

`AutoLoader` is maintained by the perl5-porters. Please direct any questions to the canonical mailing list. Anything that is applicable to the CPAN release can be sent to its maintainer, though.

Author and Maintainer: The Perl5-Porters perl5-porters@perl.org

Maintainer of the CPAN release: Steffen Mueller smueller@cpan.org

COPYRIGHT AND LICENSE

This package has been part of the perl core since the first release of perl5. It has been released separately to CPAN so older installations can benefit from bug fixes.

This package has the same copyright and license as the perl core:

```
        Copyright (C) 1993, 1994, 1995, 1996, 1997, 1998, 1999,
    2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009,
    2011, 2012, 2013
    by Larry Wall and others

                        All rights reserved.

This program is free software; you can redistribute it and/or modify
it under the terms of either:

    a) the GNU General Public License as published by the Free
    Software Foundation; either version 1, or (at your option) any
    later version, or

    b) the "Artistic License" which comes with this Kit.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See either
the GNU General Public License or the Artistic License for more␣
↪details.

You should have received a copy of the Artistic License with this
Kit, in the file named "Artistic".  If not, I'll be glad to provide␣
↪one.

You should also have received a copy of the GNU General Public␣
↪License
along with this program in the file named "Copying". If not, write␣
↪to the
Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston,
MA 02110-1301, USA or visit their web page on the internet at
http://www.gnu.org/copyleft/gpl.html.

For those of you that choose to use the GNU General Public License,
my interpretation of the GNU General Public License is that no Perl
script falls under the terms of the GPL unless you explicitly put
said script under the terms of the GPL yourself.  Furthermore, any
object code linked with perl does not automatically fall under the
terms of the GPL, provided such object code only adds definitions
of subroutines and variables, and does not otherwise impair the
resulting interpreter from executing any standard Perl script.  I
consider linking in C subroutines in this manner to be the moral
equivalent of defining subroutines in the Perl language itself.  You
```

(continues on next page)

(continued from previous page)

```
may sell such an object file as proprietary provided that you␣
 ↪provide
or offer to provide the Perl source, as specified by the GNU General
Public License.  (This is merely an alternate way of specifying␣
 ↪input
to the program.)  You may also sell a binary produced by the dumping␣
 ↪of
a running Perl script that belongs to you, provided that you provide␣
 ↪or
offer to provide the Perl source as specified by the GPL.  (The
fact that a Perl interpreter and your code are in the same binary␣
 ↪file
is, in this case, a form of mere aggregation.)  This is my␣
 ↪interpretation
of the GPL.  If you still have concerns or difficulties␣
 ↪understanding
my intent, feel free to contact me.  Of course, the Artistic License
spells all this out for your protection, so you may prefer to use␣
 ↪that.
```

5.2.8 AutoSplit

split a package for autoloading

SYNOPSIS

```
autosplit($file, $dir, $keep, $check, $modtime);

autosplit_lib_modules(@modules);
```

DESCRIPTION

This function will split up your program into files that the AutoLoader module can handle. It is used by both the standard perl libraries and by the MakeMaker utility, to automatically configure libraries for autoloading.

The `autosplit` interface splits the specified file into a hierarchy rooted at the directory `$dir`. It creates directories as needed to reflect class hierarchy, and creates the file `autosplit.ix`. This file acts as both forward declaration of all package routines, and as timestamp for the last update of the hierarchy.

The remaining three arguments to `autosplit` govern other options to the autosplitter.

- $keep

  If the third argument, *$keep*, is false, then any pre-existing `*.al` files in the autoload directory are removed if they are no longer part of the module (obsoleted functions). $keep defaults to 0.

- $check

The fourth argument, *$check*, instructs `autosplit` to check the module currently being split to ensure that it includes a `use` specification for the AutoLoader module, and skips the module if AutoLoader is not detected. $check defaults to 1.

- $modtime

  Lastly, the *$modtime* argument specifies that `autosplit` is to check the modification time of the module against that of the `autosplit.ix` file, and only split the module if it is newer. $modtime defaults to 1.

Typical use of AutoSplit in the perl MakeMaker utility is via the command-line with:

```
perl -e 'use AutoSplit; autosplit($ARGV[0], $ARGV[1], 0, 1, 1)'
```

Defined as a Make macro, it is invoked with file and directory arguments; `autosplit` will split the specified file into the specified directory and delete obsolete `.al` files, after checking first that the module does use the AutoLoader, and ensuring that the module is not already currently split in its current form (the modtime test).

The `autosplit_lib_modules` form is used in the building of perl. It takes as input a list of files (modules) that are assumed to reside in a directory **lib** relative to the current directory. Each file is sent to the autosplitter one at a time, to be split into the directory **lib/auto**.

In both usages of the autosplitter, only subroutines defined following the perl *__END__* token are split out into separate files. Some routines may be placed prior to this marker to force their immediate loading and parsing.

Multiple packages

As of version 1.01 of the AutoSplit module it is possible to have multiple packages within a single file. Both of the following cases are supported:

```
package NAME;
__END__
sub AAA { ... }
package NAME::option1;
sub BBB { ... }
package NAME::option2;
sub BBB { ... }

package NAME;
__END__
sub AAA { ... }
sub NAME::option1::BBB { ... }
sub NAME::option2::BBB { ... }
```

DIAGNOSTICS

`AutoSplit` will inform the user if it is necessary to create the top-level directory specified in the invocation. It is preferred that the script or installation process that invokes `AutoSplit` have created the full directory path ahead of time. This warning may indicate that the module is being split into an incorrect path.

`AutoSplit` will warn the user of all subroutines whose name causes potential file naming conflicts on machines with drastically limited (8 characters or less) file name length. Since the subroutine name is used as the file name, these warnings can aid in portability to such systems.

Warnings are issued and the file skipped if `AutoSplit` cannot locate either the *__END__* marker or a "package Name;"-style specification.

`AutoSplit` will also emit general diagnostics for inability to create directories or files.

AUTHOR

`AutoSplit` is maintained by the perl5-porters. Please direct any questions to the canonical mailing list. Anything that is applicable to the CPAN release can be sent to its maintainer, though.

Author and Maintainer: The Perl5-Porters perl5-porters@perl.org

Maintainer of the CPAN release: Steffen Mueller smueller@cpan.org

COPYRIGHT AND LICENSE

This package has been part of the perl core since the first release of perl5. It has been released separately to CPAN so older installations can benefit from bug fixes.

This package has the same copyright and license as the perl core:

```
        Copyright (C) 1993, 1994, 1995, 1996, 1997, 1998, 1999,
    2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008
    by Larry Wall and others

                        All rights reserved.

This program is free software; you can redistribute it and/or modify
it under the terms of either:

    a) the GNU General Public License as published by the Free
    Software Foundation; either version 1, or (at your option) any
    later version, or

    b) the "Artistic License" which comes with this Kit.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See either
the GNU General Public License or the Artistic License for more␣
↪details.

You should have received a copy of the Artistic License with this
Kit, in the file named "Artistic".  If not, I'll be glad to provide␣
↪one.

You should also have received a copy of the GNU General Public␣
```

(continues on next page)

(continued from previous page)

```
 ↪License
along with this program in the file named "Copying". If not, write␣
 ↪to the
Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston,␣
 ↪MA
02111-1307, USA or visit their web page on the internet at
http://www.gnu.org/copyleft/gpl.html.

For those of you that choose to use the GNU General Public License,
my interpretation of the GNU General Public License is that no Perl
script falls under the terms of the GPL unless you explicitly put
said script under the terms of the GPL yourself.  Furthermore, any
object code linked with perl does not automatically fall under the
terms of the GPL, provided such object code only adds definitions
of subroutines and variables, and does not otherwise impair the
resulting interpreter from executing any standard Perl script.  I
consider linking in C subroutines in this manner to be the moral
equivalent of defining subroutines in the Perl language itself.  You
may sell such an object file as proprietary provided that you␣
 ↪provide
or offer to provide the Perl source, as specified by the GNU General
Public License.  (This is merely an alternate way of specifying␣
 ↪input
to the program.)  You may also sell a binary produced by the dumping␣
 ↪of
a running Perl script that belongs to you, provided that you provide␣
 ↪or
offer to provide the Perl source as specified by the GPL.  (The
fact that a Perl interpreter and your code are in the same binary␣
 ↪file
is, in this case, a form of mere aggregation.)  This is my␣
 ↪interpretation
of the GPL.  If you still have concerns or difficulties␣
 ↪understanding
my intent, feel free to contact me.  Of course, the Artistic License
spells all this out for your protection, so you may prefer to use␣
 ↪that.
```

5.2.9 B

Inspect a running Perl program's own optree, symbol table, and compiled subs.

`B` is the compiler back-end interface. It hands a script introspective access to the data structures the interpreter already holds: the root of the main optree, every compiled sub's CV, the symbol table (stash) graph, and the SV/AV/HV/GV objects those structures point at. Subclasses such as `B::OP`, `B::CV`, `B::GV`, `B::SV` (and its type-specific children) wrap those pointers as blessed Perl objects with accessor methods.

Common callers are back-end modules like `B::Deparse`, `B::Concise`, and `B::Xref`; debugging tooling that walks the optree; and scripts that introspect the symbol table to

generate code or diagnostics.

```
use B qw(svref_2object class);

sub add { my ($x, $y) = @_; return $x + $y }

my $cv = svref_2object(\&add);                  # a B::CV
printf "%s defined in %s at line %d\n", $cv->GV->NAME, $cv->FILE =~␣
 ↪s{.*/}{}r,
    $cv->START->line;                           # add defined in add.
 ↪pl at line 3

# Walk the sub's ops in execution order.
my @ops;
for (my $op = $cv->START; $$op; $op = $op->next) {
    push @ops, $op->name;
}
print "@ops\n";
# nextstate padrange aassign nextstate padsv padsv add leavesub

my $sv = svref_2object(\"text");
printf "%s, PV %s, length %d\n", class($sv), $sv->PV, $sv->CUR;   #␣
 ↪PV, PV text, length 4
print "main program: ", class(B::main_cv()), ", root ", B::main_
 ↪root()->name, "\n";
# main program: CV, root leave
```

The ops are pperl's own, the ones perl 5.44 builds for the same source: the names, flags, order and tree shape that `B::Concise` and `B::Deparse` read are perl5's.

This is a native implementation of the XS half of B 1.92; its `B.pm` and the `B::*` modules built on it are used unchanged. B exports nothing by default; on request, its functions (`svref_2object`, `class`, `main_root`, `main_start`, `main_cv`, `walkoptree`, `walksymtable`, ppname, opnumber, `begin_av` ...) and its constants - the `SVf_*`, `OPf_*`, `OPp*`, `CVf_*`, `GVf_*`, `PMf_*` flags and the rest.

Differences from upstream

One, where perl 5.44 with B.xs crashes: a `B::OP` method called on a `B::NULL` - the object for a missing op, such as the ROOT of an XSUB - as `B::OP::name($null)` answers as for the null op (name is "null", an op or SV field is a `B::NULL`, a number is 0). B.xs follows the NULL pointer and crashes (t/81-xs-native/B/470-or-better.t).

Modules

- `B::AV`
- `B::BINOP`
- `B::BM`
- `B::COP`

- B::COW - Ask whether a string shares its buffer copy-on-write, and how many scalars share it.
- B::CV
- B::Concise - Walk Perl syntax tree, printing concise info about ops
- B::Deparse - Perl compiler backend to produce perl code
- B::FM
- B::GV
- B::HE
- B::HV
- B::IO
- B::IV
- B::LISTOP
- B::LOGOP
- B::LOOP
- B::MAGIC
- B::METHOP
- B::NV
- B::OP
- B::Op_private - OP op_private flag definitions
- B::PADLIST
- B::PADNAME
- B::PADNAMELIST
- B::PADOP
- B::PMOP
- B::PV
- B::PVLV
- B::PVMG
- B::RHE
- B::SV
- B::SVOP
- B::Showlex - Show lexical variables used in functions or files
- B::Terse - Walk Perl syntax tree, printing terse info about ops
- B::UNOP
- B::UNOP_AUX
- B::Xref - Generates cross reference reports for Perl programs

Functions

Entry points

svref_2object

Wrap a reference's referent as a `B::*` object.

opnumber

Return the numeric op-type for an op name, or `-1` if unknown.

ppname

Return the pp_-prefixed name for a numeric op-type.

Synopsis

```
my $name = B::ppname($op->type);   # e.g. "pp_print"
```

Inverse of `B::opnumber`. Returns `undef` if the number is out of range.

sv_undef

Return a `B::SPECIAL` wrapping perl's shared `PL_sv_undef`.

sv_yes

Return a `B::SPECIAL` wrapping perl's shared `PL_sv_yes`.

sv_no

Return a `B::SPECIAL` wrapping perl's shared `PL_sv_no`.

main_root

Return a `B::OP` for the root of the program's main optree.

main_start

Return a `B::OP` for the first op executed in the main program.

Synopsis

```
my $start = B::main_start;
```

Where `B::main_root` gives the syntactic root, `B::main_start` gives the head of the linked list perl follows at run time via `op_next`.

cast_I32

Truncate `$i` to a signed 32-bit integer.

Synopsis

```
B::cast_I32(0x1_0000_0001);   # 1 (low 32 bits, sign-extended)
```

Exists so Perl back-ends can reproduce perl's own I32 truncation when they need to emit exact values for an op field typed I32 in C.

amagic_generation

Return the overload-magic generation counter (always `0` in modern perl).

comppadlist

Return the padlist of the currently-compiling unit as a `B::PADLIST`.

formfeed

Return `$^L`, the form-feed string, as a `B::SV`.

address

Return the memory address of an SV as an integer.

Synopsis

```
my $addr = B::address(\$x);
```

Mostly useful as an identity key: two `B::*` objects wrapping the same underlying SV will report the same address.

threadsv_names

Return the list of thread-local special-variable names (empty).

minus_c

Request that compilation stop after the compile phase (`-c`).

save_BEGINs

Request that `BEGIN` blocks be saved rather than freed (`save_BEGINs`).

Walking the optree

walkoptree

Recursively walk an optree, calling `$method` on each op.

walkoptree_debug

Get or set the walkoptree debug flag.

String and hash helpers

cstring

Return $sv's contents as a double-quoted C-style string literal.

perlstring

Return $sv's contents as a double-quoted Perl string literal.

Synopsis

```
B::perlstring('say $x@y');    # "say \$x\@y"
```

Like B::cstring, but also escapes $ and @ so the result can be pasted back into Perl source without interpolation surprises.

cchar

Return the first character of $sv as a single-quoted C character literal.

hash

Return perl's hash of a string, formatted as 0xNNNNNNNN.

safename

Return a string with a leading control character rendered as ^X.

B::GV methods

B::GV::SAFENAME

Return the GV's name with a leading control character rendered as ^X.

Interpreter globals

sub_generation

B::sub_generation(): perl's global method-cache generation (PL_sub_generation). It goes up when every method cache is invalidated at once - a change to UNIVERSAL or one of its parents, among a few others; ordinary sub definitions and @ISA changes only invalidate their own classes and leave it alone. B::dowarn() is perl's warning switch word (PL_dowarn): 0 normally, 1 under -w, 3 under -W, 4 under -X.

Also available as B::dowarn.

main_cv

Return a `B::CV` for the implicit CV of the main program.

inc_gv

Return a `B::GV` for `*INC`, the module-loading search list.

defstash

Return a `B::HV` for `%main::`, the default stash.

curstash

Return a `B::HV` for the currently active compile-time stash.

warnhook

Return a `B::SV` for the current `$SIG{__WARN__}` handler.

diehook

Return a `B::SV` for the current `$SIG{__DIE__}` handler.

init_av

Return a `B::AV` of the `INIT` blocks queued for execution.

check_av

Return a `B::AV` of `CHECK` blocks queued for execution.

unitcheck_av

Return a `B::AV` of `UNITCHECK` blocks queued for execution.

begin_av

Return a `B::AV` of `BEGIN` blocks already executed.

end_av

Return a `B::AV` of `END` blocks queued to run at interpreter exit.

B::AV

Functions

Other Functions

FILL

Return the logical last index of the array (AvFILL, i.e. $#a).

MAX

Return the allocated capacity of the array (AvMAX).

ARRAY

Return the array's elements as a list of B:: objects (AvARRAY).

ARRAYelt

Return the Nth element as a B:: object (AvARRAYelt).

B::BINOP

Functions

B::OP methods

last

Return the last child of a BINOP/LISTOP (op_last).

B::BM

Functions

Other Functions

TABLE

Return the Boyer-Moore skip table of a studied string.

PREVIOUS

Always 0.

RARE

Always 0.

USEFUL

Return how many times the Boyer-Moore study has paid for itself.

B::COP

Functions

B::OP methods

cop_seq

Return a COP's compile sequence number (`cop_seq`).

line

Return a COP's source line number (`cop_line`).

hints

Return a COP's hint bits (`cop_hints`).

file

Return a COP's source filename (`CopFILE`).

stash

Return a COP's package stash as a B::HV (`cop_stash`).

stashpv

Return a COP's package name (`CopSTASHPV`).

label

Return a COP's statement label, or undef (`CopLABEL`).

stashoff

Croaks: a COP's stash pad offset only exists on a threaded perl.

B::COW

Ask whether a string shares its buffer copy-on-write, and how many scalars share it.

perl copies a string by sharing its buffer and copying only when one of the copies is modified. These helpers expose that state, mostly to test code that wants to know a copy was avoided.

Synopsis

```
use B::COW qw(can_cow is_cow cowrefcnt cowrefcnt_max);

my $a = "x" x 100;
my $b = $a;                          # shares $a's buffer
print is_cow($b) ? "shared\n" : "own\n";
print cowrefcnt($a), "\n";           # how many share it, as perl
 ↪counts
print cowrefcnt_max(), "\n";         # 255
```

Functions

`can_cow()` is true: this perl copies on write. `is_cow($sv)` is true for a string whose buffer is shared copy-on-write. `cowrefcnt($sv)` is the share count perl keeps in the buffer (0 for a shared hash key string, which has no such count), or `undef` for a value that is not copy-on-write. `cowrefcnt_max()` is the largest count, after which perl copies instead of sharing. Nothing is exported by default.

This is a native implementation of B::COW 0.007's `COW.xs`; its `B/COW.pm` is used unchanged.

Differences from upstream

None.

Functions

Copy on write

`can_cow`

Whether this perl copies strings on write.

Synopsis

```
use B::COW qw(can_cow);
print can_cow() ? "copy on write\n" : "plain copies\n";
```

What you get back

True: pperl, like perl 5.44, shares string buffers copy-on-write.

`is_cow`

Whether a string shares its buffer copy-on-write.

`cowrefcnt`

How many scalars share a copy-on-write buffer, as perl counts.

cowrefcnt_max

The largest copy-on-write share count.

is_cow

Whether a string shares its buffer copy-on-write.

Synopsis

```
use B::COW qw(is_cow);
my $copy = $string;
print is_cow($copy) ? "shared\n" : "own buffer\n";
```

What you get back

True for a string value whose buffer is shared copy-on-write with other scalars or with a hash key, false otherwise - including for a value that is not a string.

cowrefcnt

How many scalars share a copy-on-write buffer, as perl counts.

Synopsis

```
use B::COW qw(cowrefcnt);
my $n = cowrefcnt($string);
```

What you get back

The share count perl keeps in the last byte of the buffer, 0 for a string that shares a hash key's buffer (those have no count), or `undef` when the value is not copy-on-write.

cowrefcnt_max

The largest copy-on-write share count.

Synopsis

```
use B::COW qw(cowrefcnt_max);
print cowrefcnt_max(), "\n";   # 255
```

What you get back

255: the count is one byte, and a string already shared that often is copied instead of shared again.

B::CV

Functions

B::CV methods

GV

Return the `B::GV` that a named CV belongs to.

STASH

Return the `B::HV` for the package the CV was compiled in.

FILE

Return the source filename the CV was compiled from.

Synopsis

```
print $cv->FILE;    # e.g. "MyModule.pm"
```

Returns an empty string for XSUBs that do not carry a file record.

ROOT

Return the root op of the CV's optree, or `B::NULL` for an XSUB.

Synopsis

```
my $root = $cv->ROOT;
```

XSUBs carry compiled C code and have no optree, so `ROOT` returns a `B::NULL`. Test with `$cv->CvFLAGS & B::CVf_ISXSUB` first if it matters.

XSUB

Return the CV's C function pointer as an integer, or `0` for a Perl sub.

Synopsis

```
my $addr = $cv->XSUB;   # non-zero iff the sub is an XSUB
```

XSUBANY

Return `CvXSUBANY` - per-XSUB private data.

Synopsis

```
my $any = $cv->XSUBANY;          # any_iv; 0 for Perl subs
my $sv  = $const_cv->XSUBANY;    # B object of the constant for␣
 ↪CvCONST subs
```

START

Return the first op executed in the CV's body, or `B::NULL` for an XSUB.

DEPTH

Return the current recursion depth of the sub.

CvFLAGS

Return the raw `CvFLAGS` word of the CV.

NAME_HEK

Return the CV's name when it was stored as a HEK rather than via a GV.

OUTSIDE

Return the lexically enclosing CV (`CvOUTSIDE`).

PADLIST

Return the CV's padlist as a `B::PADLIST`, or `B::NULL` for an XSUB.

OUTSIDE_SEQ

Return the COP sequence number at which the CV was closed over.

CONST

Return `CVf_CONST` if the CV is a constant (inlinable) sub, else 0.

HSCXT

Return the XS handshake context pointer as an integer, or `0` for a Perl sub.

const_sv

Return the SV a constant sub yields, as a B:: object (`const_sv`).

GV

Return the `B::GV` that a named CV belongs to.

Synopsis

```
my $cv = B::svref_2object(\&main::foo);
my $gv = $cv->GV;       # B::GV for *main::foo
print $gv->NAME;        # "foo"
```

Anonymous and lexical subs may return a `B::SPECIAL` or a sentinel rather than a real `B::GV`; check with `isa` before calling `NAME`.

NAME_HEK

Return the CV's name when it was stored as a HEK rather than via a GV.

Synopsis

```
my $name = $cv->NAME_HEK;    # undef unless CVf_NAMED is set
```

Modern perl stores some sub names directly in a hash-key slot on the CV (cheaper than a full GV). If the CV does not use that slot, returns `undef`; callers generally fall back to `$cv->GV->NAME`.

B::Concise

Walk Perl syntax tree, printing concise info about ops

SYNOPSIS

```
perl -MO=Concise[,OPTIONS] foo.pl

use B::Concise qw(set_style add_callback);
```

DESCRIPTION

This compiler backend prints the internal OPs of a Perl program's syntax tree in one of several space-efficient text formats suitable for debugging the inner workings of perl or other compiler backends. It can print OPs in the order they appear in the OP tree, in the order they will execute, or in a text approximation to their tree structure, and the format of the information displayed is customizable. Its function is similar to that of perl's **-Dx** debugging flag or the **B::Terse** module, but it is more sophisticated and flexible.

EXAMPLE

Here's two outputs (or 'renderings'), using the -exec and -basic (i.e. default) formatting conventions on the same code snippet.

```
    % perl -MO=Concise,-exec -e '$a = $b + 42'
    1  <0> enter
    2  <;> nextstate(main 1 -e:1) v
    3  <#> gvsv[*b] s
    4  <$> const[IV 42] s
*   5  <2> add[t3] sK/2
    6  <#> gvsv[*a] s
    7  <2> sassign vKS/2
    8  <@> leave[1 ref] vKP/REFC
```

In this -exec rendering, each opcode is executed in the order shown. The add opcode, marked with '*', is discussed in more detail.

The 1st column is the op's sequence number, starting at 1, and is displayed in base 36 by default. Here they're purely linear; the sequences are very helpful when looking at code with loops and branches.

The symbol between angle brackets indicates the op's type, for example; <2> is a BINOP, <@> a LISTOP, and <#> is a PADOP, which is used in threaded perls. (see "OP class abbreviations").

The opname, as in **'add[t1]'**, may be followed by op-specific information in parentheses or brackets (ex **'[t1]'**).

The op-flags (ex **'sK/2'**) are described in ("OP flags abbreviations").

```
   % perl -MO=Concise -e '$a = $b + 42'
   8  <@> leave[1 ref] vKP/REFC ->(end)
   1     <0> enter ->2
   2     <;> nextstate(main 1 -e:1) v ->3
   7     <2> sassign vKS/2 ->8
*  5        <2> add[t1] sK/2 ->6
   -           <1> ex-rv2sv sK/1 ->4
   3              <$> gvsv(*b) s ->4
   4           <$> const(IV 42) s ->5
   -        <1> ex-rv2sv sKRM*/1 ->7
   6           <$> gvsv(*a) s ->7
```

The default rendering is top-down, so they're not in execution order. This form reflects the way the stack is used to parse and evaluate expressions; the add operates on the two terms below it in the tree.

Nullops appear as `ex-opname`, where *opname* is an op that has been optimized away by perl. They're displayed with a sequence-number of '-', because they are not executed (they don't appear in previous example), they're printed here because they reflect the parse.

The arrow points to the sequence number of the next op; they're not displayed in -exec mode, for obvious reasons.

Note that because this rendering was done on a non-threaded perl, the PADOPs in the previous examples are now SVOPs, and some (but not all) of the square brackets have been replaced by round ones. This is a subtle feature to provide some visual distinction between renderings on threaded and un-threaded perls.

OPTIONS

Arguments that don't start with a hyphen are taken to be the names of subroutines or formats to render; if no such functions are specified, the main body of the program (outside any subroutines, and not including use'd or require'd files) is rendered. Passing `BEGIN`, `UNITCHECK`, `CHECK`, `INIT`, or `END` will cause all of the corresponding special blocks to be printed. Arguments must follow options.

Options affect how things are rendered (ie printed). They're presented here by their visual effect, 1st being strongest. They're grouped according to how they interrelate;

within each group the options are mutually exclusive (unless otherwise stated).

Options for Opcode Ordering

These options control the 'vertical display' of opcodes. The display 'order' is also called 'mode' elsewhere in this document.

- **-basic**

  Print OPs in the order they appear in the OP tree (a preorder traversal, starting at the root). The indentation of each OP shows its level in the tree, and the '->' at the end of the line indicates the next opcode in execution order. This mode is the default, so the flag is included simply for completeness.

- **-exec**

  Print OPs in the order they would normally execute (for the majority of constructs this is a postorder traversal of the tree, ending at the root). In most cases the OP that usually follows a given OP will appear directly below it; alternate paths are shown by indentation. In cases like loops when control jumps out of a linear path, a 'goto' line is generated.

- **-tree**

  Print OPs in a text approximation of a tree, with the root of the tree at the left and 'left-to-right' order of children transformed into 'top-to-bottom'. Because this mode grows both to the right and down, it isn't suitable for large programs (unless you have a very wide terminal).

Options for Line-Style

These options select the line-style (or just style) used to render each opcode, and dictates what info is actually printed into each line.

- **-concise**

  Use the author's favorite set of formatting conventions. This is the default, of course.

- **-terse**

  Use formatting conventions that emulate the output of **B::Terse**. The basic mode is almost indistinguishable from the real **B::Terse**, and the exec mode looks very similar, but is in a more logical order and lacks curly brackets. **B::Terse** doesn't have a tree mode, so the tree mode is only vaguely reminiscent of **B::Terse**.

- **-linenoise**

  Use formatting conventions in which the name of each OP, rather than being written out in full, is represented by a one- or two-character abbreviation. This is mainly a joke.

- **-debug**

  Use formatting conventions reminiscent of CPAN module **B::Debug**; these aren't very concise at all.

- **-env**

  Use formatting conventions read from the environment variables `B_CONCISE_FORMAT`, `B_CONCISE_GOTO_FORMAT`, and `B_CONCISE_TREE_FORMAT`.

Options for tree-specific formatting

- **-compact**

  Use a tree format in which the minimum amount of space is used for the lines connecting nodes (one character in most cases). This squeezes out a few precious columns of screen real estate.

- **-loose**

  Use a tree format that uses longer edges to separate OP nodes. This format tends to look better than the compact one, especially in ASCII, and is the default.

- **-vt**

  Use tree connecting characters drawn from the VT100 line-drawing set. This looks better if your terminal supports it.

- **-ascii**

  Draw the tree with standard ASCII characters like + and `|`. These don't look as clean as the VT100 characters, but they'll work with almost any terminal (or the horizontal scrolling mode of less(1)) and are suitable for text documentation or email. This is the default.

These are pairwise exclusive, i.e. compact or loose, vt or ascii.

Options controlling sequence numbering

- **-base***n*

  Print OP sequence numbers in base *n*. If *n* is greater than 10, the digit for 11 will be 'a', and so on. If *n* is greater than 36, the digit for 37 will be 'A', and so on until 62. Values greater than 62 are not currently supported. The default is 36.

- **-bigendian**

  Print sequence numbers with the most significant digit first. This is the usual convention for Arabic numerals, and the default.

- **-littleendian**

  Print sequence numbers with the least significant digit first. This is obviously mutually exclusive with bigendian.

Other options

- **-src**

  With this option, the rendering of each statement (starting with the nextstate OP) will be preceded by the 1st line of source code that generates it. For example:

```
1  <0> enter
# 1: my $i;
2  <;> nextstate(main 1 junk.pl:1) v:{
3  <0> padsv[$i:1,10] vM/LVINTRO
# 3: for $i (0..9) {
4  <;> nextstate(main 3 junk.pl:3) v:{
5  <0> pushmark s
6  <$> const[IV 0] s
7  <$> const[IV 9] s
8  <{> enteriter(next->j last->m redo->9)[$i:1,10] lKS
k  <0> iter s
l  <|> and(other->9) vK/1
# 4:     print "line ";
9      <;> nextstate(main 2 junk.pl:4) v
a      <0> pushmark s
b      <$> const[PV "line "] s
c      <@> print vK
# 5:     print "$i\n";
...
```

- **-stash=”somepackage”**

  With this, “somepackage” will be required, then the stash is inspected, and each function is rendered.

The following options are pairwise exclusive.

- **-main**

  Include the main program in the output, even if subroutines were also specified. This rendering is normally suppressed when a subroutine name or reference is given.

- **-nomain**

  This restores the default behavior after you’ve changed it with ‘-main’ (it’s not normally needed). If no subroutine name/ref is given, main is rendered, regardless of this flag.

- **-nobanner**

  Renderings usually include a banner line identifying the function name or stringified subref. This suppresses the printing of the banner.

  TBC: Remove the stringified coderef; while it provides a ‘cookie’ for each function rendered, the cookies used should be 1,2,3.. not a random hex-address. It also complicates string comparison of two different trees.

- **-banner**

  restores default banner behavior.

- **-banneris** => subref

  TBC: a hookpoint (and an option to set it) for a user-supplied function to produce a banner appropriate for users needs. It’s not ideal, because the rendering-state

variables, which are a natural candidate for use in concise.t, are unavailable to the user.

Option Stickiness

If you invoke Concise more than once in a program, you should know that the options are 'sticky'. This means that the options you provide in the first call will be remembered for the 2nd call, unless you re-specify or change them.

ABBREVIATIONS

The concise style uses symbols to convey maximum info with minimal clutter (like hex addresses). With just a little practice, you can start to see the flowers, not just the branches, in the trees.

OP class abbreviations

These symbols appear before the op-name, and indicate the B:: namespace that represents the ops in your Perl code.

```
0      OP (aka BASEOP)  An OP with no children
1      UNOP             An OP with one child
+      UNOP_AUX         A UNOP with auxillary fields
2      BINOP            An OP with two children
|      LOGOP            A control branch OP
@      LISTOP           An OP that could have lots of children
/      PMOP             An OP with a regular expression
$      SVOP             An OP with an SV
"      PVOP             An OP with a string
{      LOOP             An OP that holds pointers for a loop
;      COP              An OP that marks the start of a statement
#      PADOP            An OP with a GV on the pad
.      METHOP           An OP with method call info
```

OP flags abbreviations

OP flags are either public or private. The public flags alter the behavior of each opcode in consistent ways, and are represented by 0 or more single characters.

```
v      OPf_WANT_VOID    Want nothing (void context)
s      OPf_WANT_SCALAR  Want single value (scalar context)
l      OPf_WANT_LIST    Want list of any length (list context)
                        Want is unknown
K      OPf_KIDS         There is a firstborn child.
P      OPf_PARENS       This operator was parenthesized.
                         (Or block needs explicit scope entry.)
R      OPf_REF          Certified reference.
                         (Return container, not containee).
M      OPf_MOD          Will modify (lvalue).
```

(continues on next page)

(continued from previous page)

```
S       OPf_STACKED     Some arg is arriving on the stack.
*       OPf_SPECIAL     Do something weird for this op (see op.h)
```

Private flags, if any are set for an opcode, are displayed after a '/'

```
8  <@> leave[1 ref] vKP/REFC ->(end)
7     <2> sassign vKS/2 ->8
```

They're opcode specific, and occur less often than the public ones, so they're represented by short mnemonics instead of single-chars; see B::Op_private and `regen/op_private` for more details.

Note that a number after a '/' often indicates the number of arguments. In the *sassign* example above, the OP takes 2 arguments. These values are sometimes used at runtime: in particular, the MAXARG macro makes use of them.

FORMATTING SPECIFICATIONS

For each line-style ('concise', 'terse', 'linenoise', etc.) there are 3 format-specs which control how OPs are rendered.

The first is the 'default' format, which is used in both basic and exec modes to print all opcodes. The 2nd, goto-format, is used in exec mode when branches are encountered. They're not real opcodes, and are inserted to look like a closing curly brace. The tree-format is tree specific.

When a line is rendered, the correct format-spec is copied and scanned for the following items; data is substituted in, and other manipulations like basic indenting are done, for each opcode rendered.

There are 3 kinds of items that may be populated; special patterns, #vars, and literal text, which is copied verbatim. (Yes, it's a set of s///g steps.)

Special Patterns

These items are the primitives used to perform indenting, and to select text from amongst alternatives.

- **(x(***exec_text***;***basic_text***)x)**

  Generates *exec_text* in exec mode, or *basic_text* in basic mode.

- **(*(***text***)*)**

  Generates one copy of *text* for each indentation level.

- **(*(***text1***;***text2***)*)**

  Generates one fewer copies of *text1* than the indentation level, followed by one copy of *text2* if the indentation level is more than 0.

- **(?(***text1#var__Text2***)?)**

  If the value of *var* is true (not empty or zero), generates the value of *var* surrounded by *text1* and *Text2*, otherwise nothing.

- ~

  Any number of tildes and surrounding whitespace will be collapsed to a single space.

Variables

These #vars represent opcode properties that you may want as part of your rendering. The '#' is intended as a private sigil; a #var's value is interpolated into the style-line, much like "read $this".

These vars take 3 forms:

- *#var*

  A property named 'var' is assumed to exist for the opcodes, and is interpolated into the rendering.

- *#var__N*

  Generates the value of *var*, left justified to fill *N* spaces. Note that this means while you can have properties 'foo' and 'foo2', you cannot render 'foo2', but you could with 'foo2a'. You would be wise not to rely on this behavior going forward ;-)

- *#Var*

  This ucfirst form of #var generates a tag-value form of itself for display; it converts '#Var' into a 'Var => #var' style, which is then handled as described above. (Imp-note: #Vars cannot be used for conditional-fills, because the => #var transform is done after the check for #Var's value).

The following variables are 'defined' by B::Concise; when they are used in a style, their respective values are plugged into the rendering of each opcode.

Only some of these are used by the standard styles, the others are provided for you to delve into optree mechanics, should you wish to add a new style (see "add_style" below) that uses them. You can also add new ones using "add_callback".

- **#addr**

  The address of the OP, in hexadecimal.

- **#arg**

  The OP-specific information of the OP (such as the SV for an SVOP, the non-local exit pointers for a LOOP, etc.) enclosed in parentheses.

- **#class**

  The B-determined class of the OP, in all caps.

- **#classsym**

  A single symbol abbreviating the class of the OP.

- **#coplabel**

  The label of the statement or block the OP is the start of, if any.

- **#exname**

  The name of the OP, or 'ex-foo' if the OP is a null that used to be a foo.

- **#extarg**

  The target of the OP, or nothing for a nulled OP.

- **#firstaddr**

  The address of the OP's first child, in hexadecimal.

- **#flags**

  The OP's flags, abbreviated as a series of symbols.

- **#flagval**

  The numeric value of the OP's flags.

- **#hints**

  The COP's hint flags, rendered with abbreviated names if possible. An empty string if this is not a COP. Here are the symbols used:

```
 $ strict refs
 & strict subs
 * strict vars
x$ explicit use/no strict refs
x& explicit use/no strict subs
x* explicit use/no strict vars
 i integers
 l locale
 b bytes
 { block scope
 % localise %^H
 < open in
 > open out
 I overload int
 F overload float
 B overload binary
 S overload string
 R overload re
 T taint
 E eval
 X filetest access
 U utf-8

 us      use feature 'unicode_strings'
 fea=NNN feature bundle number
```

- **#hintsval**

  The numeric value of the COP's hint flags, or an empty string if this is not a COP.

- **#hyphseq**

  The sequence number of the OP, or a hyphen if it doesn't have one.

- **#label**

'NEXT', 'LAST', or 'REDO' if the OP is a target of one of those in exec mode, or empty otherwise.

- **#lastaddr**

  The address of the OP's last child, in hexadecimal.

- **#name**

  The OP's name.

- **#NAME**

  The OP's name, in all caps.

- **#next**

  The sequence number of the OP's next OP.

- **#nextaddr**

  The address of the OP's next OP, in hexadecimal.

- **#noise**

  A one- or two-character abbreviation for the OP's name.

- **#private**

  The OP's private flags, rendered with abbreviated names if possible.

- **#privval**

  The numeric value of the OP's private flags.

- **#seq**

  The sequence number of the OP. Note that this is a sequence number generated by B::Concise.

- **#opt**

  Whether or not the op has been optimized by the peephole optimizer.

- **#sibaddr**

  The address of the OP's next youngest sibling, in hexadecimal.

- **#svaddr**

  The address of the OP's SV, if it has an SV, in hexadecimal.

- **#svclass**

  The class of the OP's SV, if it has one, in all caps (e.g., 'IV').

- **#svval**

  The value of the OP's SV, if it has one, in a short human-readable format.

- **#targ**

  The numeric value of the OP's targ.

- **#targarg**

  The name of the variable the OP's targ refers to, if any, otherwise the letter t followed by the OP's targ in decimal.

- **#targarglife**

  Same as **#targarg**, but followed by the COP sequence numbers that delimit the variable's lifetime (or 'end' for a variable in an open scope) for a variable.

- **#typenum**

  The numeric value of the OP's type, in decimal.

One-Liner Command tips

- perl -MO=Concise,bar foo.pl

  Renders only bar() from foo.pl. To see main, drop the ',bar'. To see both, add ',-main'

- perl -MDigest::MD5=md5 -MO=Concise,md5 -e1

  The export is needed so that BC can find it in main.

- perl -MPOSIX -MO=Concise,_POSIX_ARG_MAX -e1

  Identifies _POSIX_ARG_MAX as a constant sub, optimized to an IV. Although POSIX isn't entirely consistent across platforms, this is likely to be present in virtually all of them.

- perl -MPOSIX -MO=Concise -e 'print _POSIX_SAVED_IDS'

  This renders a print statement, which includes a call to the function. It's identical to rendering a file with a use call and that single statement, except for the filename which appears in the nextstate ops.

- perl -MPOSIX -MO=Concise,a -e 'sub a{print _POSIX_SAVED_IDS}'

  This is **very** similar to previous, only the first two ops differ. This subroutine rendering is more representative, insofar as a single main program will have many subs.

- perl -MB::Concise -e 'B::Concise::compile("-exec","-src", \%B::Concise::)->()'

  This renders all functions in the B::Concise package with the source lines. It eschews the O framework so that the stashref can be passed directly to B::Concise::compile(). See -stash option for a more convenient way to render a package.

Using B::Concise outside of the O framework

The common (and original) usage of B::Concise was for command-line renderings of simple code, as given in EXAMPLE. But you can also use **B::Concise** from your code, and call compile() directly, and repeatedly. By doing so, you can avoid the compile-time only operation of O.pm, and even use the debugger to step through B::Concise::compile() itself.

Once you're doing this, you may alter Concise output by adding new rendering styles, and by optionally adding callback routines which populate new variables, if such were referenced from those (just added) styles.

Example: Altering Concise Renderings

```
use B::Concise qw(set_style add_callback);
add_style($yourStyleName => $defaultfmt, $gotofmt, $treefmt);
add_callback
  ( sub {
        my ($h, $op, $format, $level, $stylename) = @_;
        $h->{variable} = some_func($op);
    });
$walker = B::Concise::compile(@options,@subnames,@subrefs);
$walker->();
```

set_style()

set_style accepts 3 arguments, and updates the three format-specs comprising a line-style (basic-exec, goto, tree). It has one minor drawback though; it doesn't register the style under a new name. This can become an issue if you render more than once and switch styles. Thus you may prefer to use add_style() and/or set_style_standard() instead.

set_style_standard($name)

This restores one of the standard line-styles: `terse`, `concise`, `linenoise`, debug, env, into effect. It also accepts style names previously defined with add_style().

add_style ()

This subroutine accepts a new style name and three style arguments as above, and creates, registers, and selects the newly named style. It is an error to re-add a style; call set_style_standard() to switch between several styles.

add_callback ()

If your newly minted styles refer to any new #variables, you'll need to define a callback subroutine that will populate (or modify) those variables. They are then available for use in the style you've chosen.

The callbacks are called for each opcode visited by Concise, in the same order as they are added. Each subroutine is passed five parameters.

```
1. A hashref, containing the variable names and values which are
   populated into the report-line for the op
2. the op, as a B<B::OP> object
3. a reference to the format string
4. the formatting (indent) level
5. the selected stylename
```

To define your own variables, simply add them to the hash, or change existing values if you need to. The level and format are passed in as references to scalars, but it is unlikely that they will need to be changed or even used.

Running B::Concise::compile()

compile accepts options as described above in "OPTIONS", and arguments, which are either coderefs, or subroutine names.

It constructs and returns a $treewalker coderef, which when invoked, traverses, or walks, and renders the optrees of the given arguments to STDOUT. You can reuse this, and can change the rendering style used each time; thereafter the coderef renders in the new style.

walk_output lets you change the print destination from STDOUT to another open filehandle, or into a string passed as a ref (unless you've built perl with -Uuseperlio).

```
my $walker = B::Concise::compile('-terse','aFuncName', \&aSubRef); #
↪1
walk_output(\my $buf);
$walker->();                          # 1 renders -terse
set_style_standard('concise');        # 2
$walker->();                          # 2 renders -concise
$walker->(@new);                      # 3 renders whatever
print "3 different renderings: terse, concise, and @new: $buf\n";
```

When $walker is called, it traverses the subroutines supplied when it was created, and renders them using the current style. You can change the style afterwards in several different ways:

```
1. call C<compile>, altering style or mode/order
2. call C<set_style_standard>
3. call $walker, passing @new options
```

Passing new options to the $walker is the easiest way to change amongst any pre-defined styles (the ones you add are automatically recognized as options), and is the only way to alter rendering order without calling compile again. Note however that rendering state is still shared amongst multiple $walker objects, so they must still be used in a coordinated manner.

B::Concise::reset_sequence()

This function (not exported) lets you reset the sequence numbers (note that they're numbered arbitrarily, their goal being to be human readable). Its purpose is mostly to support testing, i.e. to compare the concise output from two identical anonymous subroutines (but different instances). Without the reset, B::Concise, seeing that they're separate optrees, generates different sequence numbers in the output.

Errors

Errors in rendering (non-existent function-name, non-existent coderef) are written to the STDOUT, or wherever you've set it via walk_output().

Errors using the various *style* calls, and bad args to walk_output(), result in die(). Use an eval if you wish to catch these errors and continue processing.

AUTHOR

Stephen McCamant, <smcc@CSUA.Berkeley.EDU>.

B::Deparse

Perl compiler backend to produce perl code

SYNOPSIS

perl -MO=Deparse[,-d][,-f*FILE*][,-p][,-q][,-l] [,-s*LETTERS*][,-x*LEVEL*] *prog.pl*

DESCRIPTION

`B::Deparse` is a backend module for the Perl compiler that generates perl source code, based on the internal compiled structure that perl itself creates after parsing a program. The output of `B::Deparse` won't be exactly the same as the original source, since perl doesn't keep track of comments or whitespace, and there isn't a one-to-one correspondence between perl's syntactical constructions and their compiled form, but it will often be close. When you use the **-p** option, the output also includes parentheses even when they are not required by precedence, which can make it easy to see if perl is parsing your expressions the way you intended.

While `B::Deparse` goes to some lengths to try to figure out what your original program was doing, some parts of the language can still trip it up; it still fails even on some parts of Perl's own test suite. If you encounter a failure other than the most common ones described in the BUGS section below, you can help contribute to `B::Deparse`'s ongoing development by submitting a bug report with a small example.

OPTIONS

As with all compiler backend options, these must follow directly after the `-MO=Deparse`, separated by a comma but not any white space.

- **-d**

  Output data values (when they appear as constants) using *Data::Dumper*. Without this option, `B::Deparse` will use some simple routines of its own for the same purpose. Currently, *Data::Dumper* is better for some kinds of data (such as complex structures with sharing and self-reference) while the built-in routines are better for others (such as odd floating-point values).

- **-f***FILE*

  Normally, `B::Deparse` deparses the main code of a program, and all the subs defined in the same file. To include subs defined in other files, pass the **-f** option with the filename. You can pass the **-f** option several times, to include more than one secondary file. (Most of the time you don't want to use it at all.) You can also use this option to include subs which are defined in the scope of a **#line** directive with two parameters.

- **-l**

Add `#line` declarations to the output based on the line and file locations of the original code.

- **-p**

  Print extra parentheses. Without this option, `B::Deparse` includes parentheses in its output only when they are needed, based on the structure of your program. With **-p**, it uses parentheses (almost) whenever they would be legal. This can be useful if you are used to LISP, or if you want to see how perl parses your input. If you say

```
if ($var & 0x7f == 65) {print "Gimme an A!"}
print ($which ? $a : $b), "\n";
$name = $ENV{USER} or "Bob";
```

  `B::Deparse,-p` will print

```
if (($var & 0)) {
    print('Gimme an A!')
};
(print(($which ? $a : $b)), '???');
(($name = $ENV{'USER'}) or '???')
```

  which probably isn't what you intended (the `'???'` is a sign that perl optimized away a constant value).

- **-P**

  Disable prototype checking. With this option, all function calls are deparsed as if no prototype was defined for them. In other words,

```
perl -MO=Deparse,-P -e 'sub foo (\@) { 1 } foo @x'
```

  will print

```
sub foo (\@) {
    1;
}
&foo(\@x);
```

  making clear how the parameters are actually passed to `foo`.

- **-q**

  Expand double-quoted strings into the corresponding combinations of concatenation, `uc`, `ucfirst`, `lc`, `lcfirst`, `quotemeta`, and `join`. For instance, print

```
print "Hello, $world, @ladies, \u$gentlemen\E, \u\L$me!";
```

  as

```
print 'Hello, ' . $world . ', ' . join($", @ladies) . ', '
      . ucfirst($gentlemen) . ', ' . ucfirst(lc $me . '!');
```

  Note that the expanded form represents the way perl handles such constructions internally – this option actually turns off the reverse translation that `B::Deparse`

usually does. On the other hand, note that `$x = "$y"` is not the same as `$x = $y`: the former makes the value of `$y` into a string before doing the assignment.

- **-s***LETTERS*

  Tweak the style of `B::Deparse`'s output. The letters should follow directly after the 's', with no space or punctuation. The following options are available:

  - **C**

    Cuddle `elsif`, `else`, and `continue` blocks. For example, print

    ```
    if (...) {
        ...
    } else {
        ...
    }
    ```

    instead of

    ```
    if (...) {
        ...
    }
    else {
        ...
    }
    ```

    The default is not to cuddle.

  - **i***NUMBER*

    Indent lines by multiples of *NUMBER* columns. The default is 4 columns.

  - **T**

    Use tabs for each 8 columns of indent. The default is to use only spaces. For instance, if the style options are **-si4T**, a line that's indented 3 times will be preceded by one tab and four spaces; if the options were **-si8T**, the same line would be preceded by three tabs.

  - **v***STRING***.**

    Print *STRING* for the value of a constant that can't be determined because it was optimized away (mnemonic: this happens when a constant is used in **v**oid context). The end of the string is marked by a period. The string should be a valid perl expression, generally a constant. Note that unless it's a number, it probably needs to be quoted, and on a command line quotes need to be protected from the shell. Some conventional values include 0, 1, 42, '', 'foo', and 'Useless use of constant omitted' (which may need to be **-sv"'Useless use of constant omitted'."** or something similar depending on your shell). The default is '???'. If you're using `B::Deparse` on a module or other file that's `require`'d, you shouldn't use a value that evaluates to false, since the customary true constant at the end of a module will be in void context when the file is compiled as a main program.

- **-x***LEVEL*

Expand conventional syntax constructions into equivalent ones that expose their internal operation. *LEVEL* should be a digit, with higher values meaning more expansion. As with **-q**, this actually involves turning off special cases in `B::Deparse`'s normal operations.

If *LEVEL* is at least 3, `for` loops will be translated into equivalent while loops with continue blocks; for instance

```
for ($i = 0; $i < 10; ++$i) {
    print $i;
}
```

turns into

```
$i = 0;
while ($i < 10) {
    print $i;
} continue {
    ++$i
}
```

Note that in a few cases this translation can't be perfectly carried back into the source code – if the loop's initializer declares a my variable, for instance, it won't have the correct scope outside of the loop.

If *LEVEL* is at least 5, `use` declarations will be translated into `BEGIN` blocks containing calls to `require` and `import`; for instance,

```
use strict 'refs';
```

turns into

```
sub BEGIN {
    require strict;
    do {
        'strict'->import('refs')
    };
}
```

If *LEVEL* is at least 7, `if` statements will be translated into equivalent expressions using &&, `?:` and do `{}`; for instance

```
print 'hi' if $nice;
if ($nice) {
    print 'hi';
}
if ($nice) {
    print 'hi';
} else {
    print 'bye';
}
```

turns into

```
$nice and print 'hi';
$nice and do { print 'hi' };
$nice ? do { print 'hi' } : do { print 'bye' };
```

Long sequences of elsifs will turn into nested ternary operators, which `B::Deparse` doesn't know how to indent nicely.

USING B::Deparse AS A MODULE

Synopsis

```
use B::Deparse;
$deparse = B::Deparse->new("-p", "-sC");
$body = $deparse->coderef2text(\&func);
eval "sub func $body"; # the inverse operation
```

Description

`B::Deparse` can also be used on a sub-by-sub basis from other perl programs.

new

```
$deparse = B::Deparse->new(OPTIONS)
```

Create an object to store the state of a deparsing operation and any options. The options are the same as those that can be given on the command line (see "OPTIONS"); options that are separated by commas after **-MO=Deparse** should be given as separate strings.

ambient_pragmas

```
$deparse->ambient_pragmas(strict => 'all', '$' => $[);
```

The compilation of a subroutine can be affected by a few compiler directives, **pragmas**. These are:

- use strict;
- use warnings;
- Assigning to the special variable $[
- use integer;
- use bytes;
- use utf8;
- use re;

Ordinarily, if you use `B::Deparse` on a subroutine which has been compiled in the presence of one or more of these pragmas, the output will include statements to turn on the appropriate directives. So if you then compile the code returned by ["coderef2text", it will behave the same way as the subroutine which you deparsed.

However, you may know that you intend to use the results in a particular context, where some pragmas are already in scope. In this case, you use the "ambient_pragmas" method to describe the assumptions you wish to make.

Not all of the options currently have any useful effect. See "BUGS" for more details.

The parameters it accepts are:

- strict

  Takes a string, possibly containing several values separated by whitespace. The special values "all" and "none" mean what you'd expect.

```
$deparse->ambient_pragmas(strict => 'subs refs');
```

- $[

  Takes a number, the value of the array base `$[`. Obsolete: cannot be non-zero.

- bytes
- utf8
- integer

  If the value is true, then the appropriate pragma is assumed to be in the ambient scope, otherwise not.

- re

  Takes a string, possibly containing a whitespace-separated list of values. The values "all" and "none" are special. It's also permissible to pass an array reference here.

```
$deparser->ambient_pragmas(re => 'eval');
```

- warnings

  Takes a string, possibly containing a whitespace-separated list of values. The values "all" and "none" are special, again. It's also permissible to pass an array reference here.

```
$deparser->ambient_pragmas(warnings => [qw[void io]]);
```

  If one of the values is the string "FATAL", then all the warnings in that list will be considered fatal, just as with the **warnings** pragma itself. Should you need to specify that some warnings are fatal, and others are merely enabled, you can pass the **warnings** parameter twice:

```
$deparser->ambient_pragmas(
    warnings => 'all',
    warnings => [FATAL => qw/void io/],
);
```

  See *warnings* for more information about lexical warnings.

- hint_bits
- warning_bits

These two parameters are used to specify the ambient pragmas in the format used by the special variables `$^H` and `${^WARNING_BITS}`.

They exist principally so that you can write code like:

```
{ my ($hint_bits, $warning_bits);
BEGIN {($hint_bits, $warning_bits) = ($^H, ${^WARNING_BITS})}
$deparser->ambient_pragmas (
    hint_bits    => $hint_bits,
    warning_bits => $warning_bits,
    '$'          => 0 + $[
); }
```

which specifies that the ambient pragmas are exactly those which are in scope at the point of calling. However, see also [“ambient_pragmas_from_caller”.

- This parameter is used to specify the ambient pragmas which are stored in the special hash %^H.

ambient_pragmas_from_caller

```
$deparse->ambient_pragmas_from_caller()
```

A convenient shortcut for setting the hints and warnings ambient pragmas to those of the immediately calling code. This uses the *caller* function to determine the hints and warnings bits in effect at the callsite to this method, and sets those as the ambient settings for the deparser.

coderef2text

```
$body = $deparse->coderef2text(\&func)
$body = $deparse->coderef2text(sub ($$) { ... })
```

Return source code for the body of a subroutine (a block, optionally preceded by a prototype in parens), given a reference to the sub. Because a subroutine can have no names, or more than one name, this method doesn’t return a complete subroutine definition – if you want to eval the result, you should prepend “sub subname “, or “sub “ for an anonymous function constructor. Unless the sub was defined in the main:: package, the code will include a package declaration.

Normally, `B::Deparse` will emit code that includes the *feature* pragma if required to enable features that are used in the fragment that follows. However, as “coderef2text” emits only the body of a subroutine and expects the caller to prepend the `sub` and optional name onto the beginning of it, it will not have the opportunity to emit a `use feature 'signatures'` if the subroutine uses a signature, and the signatures feature is not enabled in the ambient pragmas.

In the particular situation of a subroutine that uses the `signatures` feature to parse its arguments being passed to “coderef2text” when the feature is **not** enabled in “ambient_pragmas”, `B::Deparse` will attempt to emit pure-perl code that emulates the behaviour of the signature as closely as possible. This is performed on a **best-effort** basis. It is not guaranteed to perfectly capture the semantics of the signature’s behaviour, only

to offer a human-readable suggestion as to what it might do. Furthermore, it is not guaranteed to be able to reproduce every possible behaviour of signatures in future versions of Perl. It may be that a future version introduces a behaviour that does not have a tidy way to express it in this pure-perl emulation code without using the `signatures` feature.

If this is of importance to you, make sure to use the "ambient_pragmas" or "ambient_pragmas_from_caller" method to enable the `signatures` feature, ensuring that `B::Deparse` will use it to deparse subroutines that use signatures.

BUGS

- The only pragmas to be completely supported are: `use warnings`, `use strict`, `use bytes`, `use integer` and `use feature`.

  Excepting those listed above, we're currently unable to guarantee that `B::Deparse` will produce a pragma at the correct point in the program. (Specifically, pragmas at the beginning of a block often appear right before the start of the block instead.) Since the effects of pragmas are often lexically scoped, this can mean that the pragma holds sway over a different portion of the program than in the input file.

- In fact, the above is a specific instance of a more general problem: we can't guarantee to produce `BEGIN` blocks or `use` declarations in exactly the right place. So if you use a module which affects compilation (such as by over-riding keywords, overloading constants or whatever) then the output code might not work as intended.

- Some constants don't print correctly either with or without **-d**. For instance, neither `B::Deparse` nor *Data::Dumper* know how to print dual-valued scalars correctly, as in:

```
use constant E2BIG => ($!=7); $y = E2BIG; print $y, 0+$y;

use constant H => { "#" => 1 }; H->{"#"};
```

- An input file that uses source filtering probably won't be deparsed into runnable code, because it will still include the **use** declaration for the source filtering module, even though the code that is produced is already ordinary Perl which shouldn't be filtered again.

- Optimized-away statements are rendered as '???'. This includes statements that have a compile-time side-effect, such as the obscure

```
my $x if 0;
```

  which is not, consequently, deparsed correctly.

```
  foreach my $i (@_) { 0 }
=>
  foreach my $i (@_) { '???' }
```

- Lexical (my) variables declared in scopes external to a subroutine appear in "coderef2text" output text as package variables. This is a tricky problem, as perl

has no native facility for referring to a lexical variable defined within a different scope, although *PadWalker* is a good start.

See also Data::Dump::Streamer, which combines B::Deparse and *PadWalker* to serialize closures properly.

- There are probably many more bugs on non-ASCII platforms (EBCDIC).

AUTHOR

Stephen McCamant <smcc@CSUA.Berkeley.EDU>, based on an earlier version by Malcolm Beattie mbeattie@sable.ox.ac.uk, with contributions from Gisle Aas, James Duncan, Albert Dvornik, Robin Houston, Dave Mitchell, Hugo van der Sanden, Gurusamy Sarathy, Nick Ing-Simmons, and Rafael Garcia-Suarez.

B::FM

Functions

Other Functions

LINES

Always `0`.

B::GV

Functions

B::GV methods

NAME

Return the GV's unqualified name.

FILE

Return the source file that first introduced the GV.

Synopsis

```
print $gv->FILE;    # e.g. "MyModule.pm"
```

The filename perl recorded when the glob was created or first accessed as an lvalue. An empty string for globs with no recorded origin.

STASH

Return the `B::HV` for the stash that owns the GV.

SV

Return the scalar slot of the GV as a `B::SV`.

FORM

Return the format slot of the GV as a `B::CV` (stash walking via `O=Deparse` visits every GP slot).

IO

Return the IO slot of the GV as a `B::IO`.

CVGEN

Return the GV's method-cache generation number.

AV

Return the array slot of the GV as a `B::AV`.

HV

Return the hash slot of the GV as a `B::HV`.

CV

Return the code slot of the GV as a `B::CV`.

EGV

Return the "effective" GV (`GvEGV`) - for aliased globs, the canonical one.

GvREFCNT

Return the refcount of the GV's GP (slot-bundle) record.

LINE

Return the source line where the GV was first introduced.

isGV_with_GP

Return true if the SV is a GV that has a GP slot-bundle attached.

is_empty

Return true if the GV has no slot-bundle (GP) at all.

GvFLAGS

Return the GV-specific flag bits.

GP

Return the address of the glob's GP as an integer.

GPFLAGS

Return the GP-level flag bits (GVf_INTRO, GVf_ASSUMECV, ...).

FILEGV

Return the *main::_<FILE glob for the file this GV was declared in.

NAME

Return the GV's unqualified name.

Synopsis

```
my $gv = B::svref_2object(\*main::foo);
print $gv->NAME;     # "foo"
```

The name alone, without the stash prefix. For *main::foo this is "foo"; combine with B::GV::STASH->NAME to recover the full name.

SV

Return the scalar slot of the GV as a B::SV.

Synopsis

```
our $x = 42;
my $gv = B::svref_2object(\*main::x);
print $gv->SV->IV;       # 42
```

Equivalent to *foo{SCALAR} in Perl, but wrapped as a B::SV.

B::HE

Functions

Other Functions

VAL

Return the hash entry's value as a B:: object.

SVKEY_force

Return the hash entry's key as a B:: object, whatever its storage.

HASH

Return the entry's cached hash value.

B::HV

Functions

B::HV methods

NAME

Return the package name of a stash HV.

KEYS

Return the number of keys in the hash (`HvKEYS`).

ARRAY

Return the hash's contents as a key/B-object list (stash walking: B::Deparse's stash_subs via `O=Deparse` is the main consumer).

MAX

Return the hash's bucket-array size (`HvMAX`).

FILL

Return the number of used buckets (`HvFILL`).

RITER

Return the hash's iterator position (`HvRITER`).

NAME

Return the package name of a stash HV.

Synopsis

```
my $stash = B::svref_2object(\%main::);
print $stash->NAME;     # "main"
```

Meaningful only on stash hashes (those accessible as %Pkg::). A plain user hash returns an empty string.

B::IO

Functions

Other Functions

LINES

$. - the input line number for this handle.

PAGE

$% - the current format page number.

PAGE_LEN

$= - lines per format page.

LINES_LEFT

$- - lines left on the current format page.

TOP_NAME

$^ - the name of the top-of-page format, or undef.

TOP_GV

The B::GV holding the top-of-page format.

FMT_NAME

$~ - the name of the body format, or undef.

FMT_GV

The B::GV holding the body format.

BOTTOM_NAME

The name of the bottom-of-page format, or undef.

BOTTOM_GV

The `B::GV` holding the bottom-of-page format.

IoTYPE

The handle's kind as a one-character string: `<` read, `>` write, `+` read-write, `|` pipe, `-` `-` (STDIN via ARGV), `s` socket.

IoFLAGS

The handle's flag bits (`IOf_ARGV`, `IOf_START`, `IOf_FLUSH`, `IOf_DIDTOP`, `IOf_UNTAINT`, `IOf_NOLINE`, `IOf_FAKE_DIRP`).

IsSTD

True if this IO is one of the three standard handles.

IsSTD

True if this IO is one of the three standard handles.

Synopsis

```
$io->IsSTD('stdout') and print "that's STDOUT\n";
```

Compares the actual `PerlIO *`, so it stays right after a reopen that leaves the glob's name unchanged. Croaks on any name other than `stdin`, `stdout` or `stderr`.

B::IV

Functions

B::IV methods

IVX

Return the IV slot of the wrapped SV without coercion.

UVX

Return the UV slot of the wrapped SV without coercion.

packiv

Return the IV as 8 network-order bytes.

needs64bits

True when the IV does not survive a round trip through 32 bits.

IV

Return the SV's IV slot without coercion (`B::IV::IV`).

packiv

Return the IV as 8 network-order bytes.

Synopsis

```
my $packed = B::svref_2object(\$n)->packiv;
```

A fixed-width, endian-independent encoding, so a compiler backend can emit an integer constant once and have it read back the same on any machine. Ask `needs64bits` first if you only want to spend 4 bytes.

B::LISTOP

Functions

B::OP methods

children

Return the number of children in a LISTOP.

B::LOGOP

Functions

B::OP methods

other

Return a LOGOP's alternate branch target (`op_other`).

B::LOOP

Functions

B::OP methods

redoop

Return a LOOP's redo target (op_redoop).

nextop

Return a LOOP's next target (op_nextop).

lastop

Return a LOOP's last target (op_lastop).

B::MAGIC

Functions

Other Functions

MOREMAGIC

The next magic in the chain, or undef (B::MAGIC::MOREMAGIC).

PRIVATE

mg_private (B::MAGIC::PRIVATE).

TYPE

The magic type as a one-character string (B::MAGIC::TYPE).

FLAGS

mg_flags (B::MAGIC::FLAGS).

LENGTH

mg_len (B::MAGIC::LENGTH).

OBJ

The magic's object slot as a B::* object (B::MAGIC::OBJ).

PTR

The magic's pointer slot (B::MAGIC::PTR).

REGEX

The compiled regexp behind r-magic, as an IV (`B::MAGIC::REGEX`).

precomp

The original pattern text of r-magic (`B::MAGIC::precomp`).

B::METHOP

Functions

B::OP methods

first

Return a METHOP's first child for dynamic method calls.

meth_sv

Return a METHOP's method-name SV (`meth_sv`).

rclass

Return a METHOP's redirect class SV (`rclass`).

B::NV

Functions

B::NV methods

NV

Return the floating-point value of the wrapped SV, coercing if needed.

NVX

Return the NV slot of the wrapped SV without coercion.

B::OP

Functions

B::OP methods

next

Return the next op in execution order (`op_next`).

`sibling`

Return the next syntactic sibling.

`targ`

Return the op's pad target index (`op_targ`).

`flags`

Return the op's shared flag byte (`op_flags`).

`private`

Return the op's private flag byte (`op_private`).

`name`

Return the short op name (e.g. "print").

`desc`

Return the human-readable op description.

`ppaddr`

Return a C-source reference to the op's pp function.

`type`

Return the numeric op-type code (`op_type`).

`opt`

Return the `op_opt` bit (peephole-optimised flag).

`moresib`

Return the `op_moresib` bit.

`parent`

Return the syntactic parent of the op.

`oplist`

Return every op reachable from this one in execution order.

oplist

Return every op reachable from this one in execution order.

Synopsis

```
my @ops = B::main_root->oplist;
```

Follows op_next and also descends the side chains it does not thread through - a LOGOP's other, a LOOP's redoop/nextop/lastop, a substitution's replacement, and a sort block's comparator.

This clears op_opt on every op it visits, which is how it avoids walking a shared branch twice. Run it after the peephole optimiser, and expect B::OP::opt to read 0 afterwards.

B::Op_private

OP op_private flag definitions

SYNOPSIS

```
use B::Op_private;

# flag details for bit 7 of OP_AELEM's op_private:
my $name  = $B::Op_private::bits{aelem}{7}; # OPpLVAL_INTRO
my $value = $B::Op_private::defines{$name}; # 128
my $label = $B::Op_private::labels{$name};  # LVINTRO

# the bit field at bits 5..6 of OP_AELEM's op_private:
my $bf  = $B::Op_private::bits{aelem}{6};
my $mask = $bf->{bitmask}; # etc
```

DESCRIPTION

This module provides four global hashes:

```
%B::Op_private::bits
%B::Op_private::defines
%B::Op_private::labels
%B::Op_private::ops_using
```

which contain information about the per-op meanings of the bits in the op_private field.

%bits

This is indexed by op name and then bit number (0..7). For single bit flags, it returns the name of the define (if any) for that bit:

```
$B::Op_private::bits{aelem}{7} eq 'OPpLVAL_INTRO';
```

For bit fields, it returns a hash ref containing details about the field. The same reference will be returned for all bit positions that make up the bit field; so for example these both return the same hash ref:

```
$bitfield = $B::Op_private::bits{aelem}{5};
$bitfield = $B::Op_private::bits{aelem}{6};
```

The general format of this hash ref is

```
{
    # The bit range and mask; these are always present.
    bitmin        => 5,
    bitmax        => 6,
    bitmask       => 0x60,

    # (The remaining keys are optional)

    # The names of any defines that were requested:
    mask_def      => 'OPpFOO_MASK',
    baseshift_def => 'OPpFOO_SHIFT',
    bitcount_def  => 'OPpFOO_BITS',

    # If present, Concise etc will display the value with a 'FOO='
    # prefix. If it equals '-', then Concise will treat the bit
    # field as raw bits and not try to interpret it.
    label         => 'FOO',

    # If present, specifies the names of some defines and the
    # display labels that are used to assign meaning to particu-
    # lar integer values within the bit field; e.g. 3 is dis-
    # played as 'C'.
    enum          => [ qw(
                          1   OPpFOO_A  A
                          2   OPpFOO_B  B
                          3   OPpFOO_C  C
                      )],

};
```

%defines

This gives the value of every OPp define, e.g.

```
$B::Op_private::defines{OPpLVAL_INTRO} == 128;
```

%labels

This gives the short display label for each define, as used by `B::Concise` and `perl -Dx`, e.g.

```
$B::Op_private::labels{OPpLVAL_INTRO} eq 'LVINTRO';
```

If the label equals '-', then Concise will treat the bit as a raw bit and not try to display it symbolically.

%ops_using

For each define, this gives a reference to an array of op names that use the flag.

```
@ops_using_lvintro = @{ $B::Op_private::ops_using{OPp_LVAL_INTRO} };
```

B::PADLIST

Functions

Other Functions

MAX

Return the highest index in the padlist (`PadlistMAX`).

NAMES

Return the padlist's PADNAMELIST (`PadlistNAMES`).

ARRAY

Return (NAMES, pad, pad, ...) as a list (`PadlistARRAY`).

ARRAYelt

Return the Nth padlist element (`PadlistARRAYelt`).

B::PADNAME

Functions

Other Functions

PV

Return the pad name PV with its sigil (e.g. "$x", "@arr") (`PadnamePV`).

LEN

Return the pad name length in bytes (`PadnameLEN`).

FLAGS

Return the pad name flags (`PadnameFLAGS`), with SVf_FAKE OR'd for OUTER.

TYPE

Return the padname's type stash as a B:: object (`PadnameTYPE`).

OURSTASH

Return the 'our' stash for the padname (`PadnameOURSTASH`).

B::PADNAMELIST

Functions

Other Functions

MAX

Return the highest index in the padname list (`PadnamelistMAX`).

ARRAY

Return the padnames as a list of B::PADNAME objects (`PadnamelistARRAY`).

ARRAYelt

Return the Nth padname as a B::PADNAME (`PadnamelistARRAYelt`).

B::PADOP

Functions

B::OP methods

padix

Return a PADOP's pad index (`op_padix`).

B::PMOP

Functions

B::OP methods

pmflags

Return a PMOP's compiled-pattern flags (`op_pmflags`).

pmreplroot

Return a PMOP's replacement-optree root (`pmreplroot`).

precomp

Return a PMOP's precompiled pattern source (`RX_PRECOMP`).

pmregexp

Return a PMOP's compiled regexp as a B::REGEXP.

pmoffset

Croaks: a PMOP's pattern index only exists on a threaded perl.

pmstashpv

Croaks: use `pmstash` instead - `pmstashpv` is the threaded spelling.

B::PV

Functions

B::PV methods

PV

Return the string contents of a PV SV, preserving `SVf_UTF8`.

PVX

Return the PV buffer contents as a C-style NUL-terminated string.

CUR

Return the current length in bytes of the PV buffer (`SvCUR`).

LEN

Return the allocated buffer size in bytes (`SvLEN`).

PVBM

Return the whole `SvPVX` buffer, table and all, as raw bytes.

RV

Return the referent of a reference SV as a B:: object (`B::PV::RV`).

PV

Return the string contents of a PV SV, preserving `SVf_UTF8`.

Synopsis

```
my $pv = B::svref_2object(\"hello");
print $pv->PV;
```

Returns the full `SvCUR` bytes, so embedded `\0` are kept. An SV with no PV slot returns an empty string.

B::PVLV

Functions

Other Functions

TARGOFF

Return where this lvalue's window starts inside its target.

TARGLEN

Return the length of this lvalue's window into its target.

TARG

Return the SV this lvalue is a view onto.

TYPE

Return the one-character lvalue kind: `x` substr, `.` vec, `y` defelem, `/` the `$/`-style tied slot, `t` a `local`ised element.

B::PVMG

Functions

B::SV methods

SvSTASH

B.xs `B::PVMG::SvSTASH` (an ALIAS of the PVMG accessor group, PVMG_stash_ix). B::HV and friends inherit it - sv_stash.t reaches it as `B::HV->SvSTASH`, RT#126410.

Other Functions

MAGIC

Every magic in the SV's chain, as `B::MAGIC` objects (`B::PVMG::MAGIC`).

B::RHE

Functions

Other Functions

HASH

Return the hints hash wrapped by a `B::RHE` object as a plain hash reference (`B::RHE::HASH`).

B::SV

Functions

B::SV methods

REFCNT

Return the reference count of the wrapped SV.

FLAGS

Return the raw `SvFLAGS` word of the wrapped SV.

SvTYPE

Return the SV type as a numeric `SVt_*` code.

object_2svref

Return a Perl reference back to the SV wrapped by a `B::*` object.

IsBOOL

B.xs `B::SV::SvIsBOOL` / `SvTRUE` / `SvTRUE_nomg`, PREFIX = Sv. Three separate XSUBs upstream, each `bool f(B::SV sv)`.

REFCNT

Return the reference count of the wrapped SV.

Synopsis

```
my $sv = B::svref_2object(\$x);
print $sv->REFCNT;
```

Calling `REFCNT` does not itself alter the count the `B::*` wrapper reports (the wrapper holds a weak handle, not a strong reference).

SvTYPE

Return the SV type as a numeric `SVt_*` code.

Synopsis

```
if ($sv->SvTYPE == B::SVt_PVCV) { ... }
```

Compare against `B::SVt_NULL`, `B::SVt_IV`, `B::SVt_PVCV`, etc. The blessed class of the wrapper already encodes the type at a coarser granularity; `SvTYPE` is the precise value.

object_2svref

Return a Perl reference back to the SV wrapped by a `B::*` object.

Synopsis

```
my $b   = B::svref_2object(\$x);
my $ref = $b->object_2svref;   # same as \$x
```

Inverse of `B::svref_2object`. Useful when a back-end has descended into a sub-structure via `B::*` methods and needs a plain Perl reference to the SV it's now looking at.

B::SVOP

Functions

B::OP methods

sv

Return an SVOP's SV payload (`op_sv`).

gv

Return an SVOP's GV payload (`op_sv`, GV-typed).

B::Showlex

Show lexical variables used in functions or files

SYNOPSIS

```
perl -MO=Showlex[,-OPTIONS][,SUBROUTINE] foo.pl
```

DESCRIPTION

When a comma-separated list of subroutine names is given as options, Showlex prints the lexical variables used in those subroutines. Otherwise, it prints the file-scope lexicals in the file.

EXAMPLES

Traditional form:

```
$ perl -MO=Showlex -e 'my ($i,$j,$k)=(1,"foo")'
Pad of lexical names for comppadlist has 4 entries
0: (0x8caea4) undef
1: (0x9db0fb0) $i
2: (0x9db0f38) $j
3: (0x9db0f50) $k
Pad of lexical values for comppadlist has 5 entries
0: SPECIAL #1 &PL_sv_undef
1: NULL (0x9da4234)
2: NULL (0x9db0f2c)
3: NULL (0x9db0f44)
4: NULL (0x9da4264)
-e syntax OK
```

New-style form:

```
$ perl -MO=Showlex,-newlex -e 'my ($i,$j,$k)=(1,"foo")'
main Pad has 4 entries
0: (0x8caea4) undef
1: (0xa0c4fb8) "$i" = NULL (0xa0b8234)
2: (0xa0c4f40) "$j" = NULL (0xa0c4f34)
3: (0xa0c4f58) "$k" = NULL (0xa0c4f4c)
-e syntax OK
```

New form, no specials, outside O framework:

```
$ perl -MB::Showlex -e \
   'my ($i,$j,$k)=(1,"foo"); B::Showlex::compile(-newlex,-nosp)->()'
main Pad has 4 entries
1: (0x998ffb0) "$i" = IV (0x9983234) 1
2: (0x998ff68) "$j" = PV (0x998ff5c) "foo"
3: (0x998ff80) "$k" = NULL (0x998ff74)
```

Note that this example shows the values of the lexicals, whereas the other examples did not (as they're compile-time only).

OPTIONS

The `-newlex` option produces a more readable `name => value` format, and is shown in the second example above.

The `-nosp` option eliminates reporting of SPECIALs, such as `0: SPECIAL \#1 &PL_sv_undef` above. Reporting of SPECIALs can sometimes overwhelm your declared lexicals.

SEE ALSO

B::Showlex can also be used outside of the O framework, as in the third example. See *B::Concise* for a fuller explanation of reasons.

TODO

Some of the reported info, such as hex addresses, is not particularly valuable. Other information would be more useful for the typical programmer, such as line-numbers, pad-slot reuses, etc.. Given this, -newlex is not a particularly good flag-name.

AUTHOR

Malcolm Beattie, `mbeattie@sable.ox.ac.uk`

B::Terse

Walk Perl syntax tree, printing terse info about ops

SYNOPSIS

```
perl -MO=Terse[,OPTIONS] foo.pl
```

DESCRIPTION

This module prints the contents of the parse tree, but without as much information as CPAN module B::Debug. For comparison, `print "Hello, world."` produced 96 lines of output from B::Debug, but only 6 from B::Terse.

This module is useful for people who are writing their own back end, or who are learning about the Perl internals. It's not useful to the average programmer.

This version of B::Terse is really just a wrapper that calls *B::Concise* with the **-terse** option. It is provided for compatibility with old scripts (and habits) but using B::Concise directly is now recommended instead.

For compatibility with the old B::Terse, this module also adds a method named `terse` to B::OP and B::SV objects. The B::SV method is largely compatible with the old one, though authors of new software might be advised to choose a more user-friendly output format.

The B::OP `terse` method, however, doesn't work well. Since B::Terse was first written, much more information in OPs has migrated to the scratchpad datastructure, but the `terse` interface doesn't have any way of getting to the correct pad. As a kludge, the new version will always use the pad for the main program, but for OPs in subroutines this will give the wrong answer or crash.

AUTHOR

The original version of B::Terse was written by Malcolm Beattie, mbeattie@sable.ox.ac.uk. This wrapper was written by Stephen McCamant, <smcc@MIT.EDU>.

B::UNOP

Functions

B::OP methods

`first`

Return the first child of a UNOP (`op_first`).

B::UNOP_AUX

Functions

Other Functions

`string`

Return a string representation of `op_aux` where possible.

`aux_list`

Return the contents of the `op_aux` array as a list of IV/GV/etc.

B::Xref

Generates cross reference reports for Perl programs

SYNOPSIS

perl -MO=Xref[,OPTIONS] foo.pl

DESCRIPTION

The B::Xref module is used to generate a cross reference listing of all definitions and uses of variables, subroutines and formats in a Perl program. It is implemented as a backend for the Perl compiler.

The report generated is in the following format:

```
File filename1
  Subroutine subname1
    Package package1
      object1        line numbers
      object2        line numbers
      ...
    Package package2
    ...
```

Each **File** section reports on a single file. Each **Subroutine** section reports on a single subroutine apart from the special cases "(definitions)" and "(main)". These report, respectively, on subroutine definitions found by the initial symbol table walk and on the main part of the program or module external to all subroutines.

The report is then grouped by the **Package** of each variable, subroutine or format with the special case "(lexicals)" meaning lexical variables. Each **object** name (implicitly qualified by its containing **Package**) includes its type character(s) at the beginning where possible. Lexical variables are easier to track and even included dereferencing information where possible.

The `line numbers` are a comma separated list of line numbers (some preceded by code letters) where that object is used in some way. Simple uses aren't preceded by a code letter. Introductions (such as where a lexical is first defined with `my`) are indicated with the letter "i". Subroutine and method calls are indicated by the character "&". Subroutine definitions are indicated by "s" and format definitions by "f".

For instance, here's part of the report from the *pod2man* program that comes with Perl:

```
Subroutine clear_noremap
  Package (lexical)
    $ready_to_print   i1069, 1079
  Package main
    $&                1086
    $.                1086
    $0                1086
    $1                1087
    $2                1085, 1085
    $3                1085, 1085
    $ARGV             1086
    %HTML_Escapes     1085, 1085
```

This shows the variables used in the subroutine `clear_noremap`. The variable `$ready_to_print` is a my() (lexical) variable, **i**ntroduced (first declared with my()) on line 1069, and used on line 1079. The variable `$&` from the main package is used on 1086, and so on.

A line number may be prefixed by a single letter:

- i

  Lexical variable introduced (declared with my()) for the first time.

- &

  Subroutine or method call.

- s

  Subroutine defined.

- r

  Format defined.

The most useful option the cross referencer has is to save the report to a separate file. For instance, to save the report on *myperlprogram* to the file *report*:

```
$ perl -MO=Xref,-oreport myperlprogram
```

OPTIONS

Option words are separated by commas (not whitespace) and follow the usual conventions of compiler backend options.

- `-oFILENAME`

  Directs output to `FILENAME` instead of standard output.

- `-r`

  Raw output. Instead of producing a human-readable report, outputs a line in machine-readable form for each definition/use of a variable/sub/format.

- `-d`

  Don't output the "(definitions)" sections.

- `-D[tO]`

  (Internal) debug options, probably only useful if `-r` included. The `t` option prints the object on the top of the stack as it's being tracked. The `O` option prints each operator as it's being processed in the execution order of the program.

BUGS

Non-lexical variables are quite difficult to track through a program. Sometimes the type of a non-lexical variable's use is impossible to determine. Introductions of non-lexical non-scalars don't seem to be reported properly.

AUTHOR

Malcolm Beattie, mbeattie@sable.ox.ac.uk.

svref_2object

Wrap a reference's referent as a `B::*` object.

Synopsis

```
my $obj = B::svref_2object(\&some_sub);
my $cv  = B::svref_2object(\&main::foo);   # returns B::CV
my $av  = B::svref_2object(\@ARGV);        # returns B::AV
```

Given a Perl reference, return a blessed `B::*` object that wraps the underlying SV, AV, HV, CV, GV, etc. Croaks with `argument is not a reference` if the input is not a reference. This is the usual entry point into `B`: most introspection starts from a reference the caller already has.

opnumber

Return the numeric op-type for an op name, or `-1` if unknown.

Synopsis

```
my $n = B::opnumber("print");      # numeric type
my $n = B::opnumber("pp_print");   # leading "pp_" is tolerated
```

Accepts either the bare name or the pp_-prefixed form. The returned integer is an index into perl's global op-name / op-desc tables and is the value `B::OP::type` produces.

main_root

Return a `B::OP` for the root of the program's main optree.

Synopsis

```
my $root = B::main_root;
$root->name;     # typically "leave"
```

The root op is the top of the tree that `B::Deparse` and friends walk. Paired with `B::main_start`, which gives the entry point in execution order.

walkoptree

Recursively walk an optree, calling `$method` on each op.

Synopsis

```
B::walkoptree(B::main_root, "visit");
```

Each op is blessed into its `B::*OP` class and then `$method` is dispatched on it. Children are visited after the parent. Back-end classes such as `B::Concise` define a visit method on their own package and call `walkoptree` to drive tree-traversal without writing the walk themselves.

walkoptree_debug

Get or set the `walkoptree` debug flag.

Synopsis

```
my $prev = B::walkoptree_debug(1);    # enable
B::walkoptree_debug($prev);           # restore
```

When enabled, `B::walkoptree` also calls a `walkoptree_debug` method on each node before the regular visit method, which lets a back-end print diagnostics as it walks. Returns the previous flag value.

cstring

Return `$sv`'s contents as a double-quoted C-style string literal.

Synopsis

```
B::cstring("hi\tthere");   # "hi\tthere"
B::cstring(undef);         # 0
```

Unprintable bytes become `\n`, `\t`, `\r`, `\a`, `\b`, `\f`, `\v`, or octal escapes. An undef SV renders as the literal `0` (no quotes), which is the convention `B::Deparse` depends on. Used by back-ends that emit C-level source.

cchar

Return the first character of `$sv` as a single-quoted C character literal.

Synopsis

```
B::cchar("A");    # 'A'
B::cchar("\n");   # '\n'
```

Non-printable bytes are escaped the same way as in `B::cstring`. Only the first byte is inspected.

hash

Return perl's hash of a string, formatted as `0xNNNNNNNN`.

Synopsis

```
my $h = B::hash("foo");    # e.g. "0x2ad53e4d"
```

Uses the same hash function the interpreter uses for hash keys, so the value is useful for reasoning about bucket placement and hash collisions in a live hash, not as a general-purpose checksum.

comppadlist

Return the padlist of the currently-compiling unit as a `B::PADLIST`.

Synopsis

```
my $pad = B::comppadlist;
my ($names, $vals) = $pad->ARRAY;
```

At run time this is `PL_main_cv`'s padlist - the file-scope lexicals of the main program. Inside a `CHECK`/`INIT` block during compilation it is `PL_compcv`'s. `B::Xref` starts every cross-reference here.

B::GV::SAFENAME

Return the GV's name with a leading control character rendered as `^X`.

Synopsis

```
B::svref_2object(\*{"main::\cA"})->SAFENAME;   # "^A"
```

The single-character control names (`*^A`, `*^D`, `*^?`) appear in the stash with raw control bytes in their name; `SAFENAME` renders them in the caret form normally used in Perl source. Only the first character is converted, so `*^WARNING_BITS` gives `^WARNING_BITS`; the conversion itself is `B::safename`.

safename

Return a string with a leading control character rendered as `^X`.

Synopsis

```
use B qw(safename);
safename("\cGx");     # "^Gx"
safename("\x7f");     # "^?"
safename("a\cG");     # "a\cG" - only the first character is
 ↪converted
```

A first character in `\cA` to `\cZ`, `\c[`, `\c\`, `\c]`, `\c^` or `\c_` becomes `^` followed by the matching printable character, and `\c?` (DEL) becomes `^?`; everything else is returned unchanged. This is the conversion `B::GV::SAFENAME` applies to a glob's name, and the form Perl source uses for variables such as `${^WARNING_BITS}`.

Exported on request.

5.2.10 Benchmark

Benchmark running times of Perl code

SYNOPSIS

```
use Benchmark qw(:all);

timethis($count, "code");

# Use Perl code in strings...
timethese($count, {
    'Name1' => '...code1...',
    'Name2' => '...code2...',
});

# ... or use subroutine references.
timethese($count, {
    'Name1' => sub { ...code1... },
    'Name2' => sub { ...code2... },
});

# cmpthese can be used both ways as well
cmpthese($count, {
    'Name1' => '...code1...',
    'Name2' => '...code2...',
});

cmpthese($count, {
    'Name1' => sub { ...code1... },
    'Name2' => sub { ...code2... },
});

# ...or in two stages
$results = timethese($count,
    {
        'Name1' => sub { ...code1... },
        'Name2' => sub { ...code2... },
    },
    'none'
);
cmpthese( $results );

my $t1 = timeit($count, '...other code...');
print "$count loops of other code took:", timestr($t1),"\n";

my $t2 = countit($time, '...other code...');
$count = $t2->iters;
print "$count loops of other code took:", timestr($t2),"\n";

# Enable hires wallclock timing if possible
use Benchmark ':hireswallclock';
```

DESCRIPTION

The `Benchmark` module encapsulates a number of routines to help you figure out how long it takes to execute some code.

`timethis` - Run a chunk of code several times

`timethese` - Run several chunks of code several times

`cmpthese` - Print results of `timethese` as a comparison chart

`timeit` - Run a chunk of code and see how long it goes

`countit` - See how many times a chunk of code runs in a given time

Methods

- new

  Returns the current time. Example:

```
use Benchmark;
my $t0 = Benchmark->new;
# ... your code here ...
my $t1 = Benchmark->new;
my $td = timediff($t1, $t0);
print "the code took:",timestr($td),"\n";
```

- debug

  Enables or disable debugging by setting the `$Benchmark::Debug` flag:

```
Benchmark->debug(1);
my $t = timeit(10, ' 5 ** $Global ');
Benchmark->debug(0);
```

- iters

  Returns the number of iterations.

Standard Exports

The following routines will be exported into your namespace if you use the Benchmark module:

- timeit(COUNT, CODE)

  Arguments: COUNT is the number of times to run the loop, and CODE is the code to run. CODE may be either a code reference or a string to be eval'd; either way it will be run in the caller's package.

  Returns: a Benchmark object.

- timethis (COUNT, CODE, [TITLE, [STYLE]])

  Time COUNT iterations of CODE. CODE may be a string to eval or a code reference; either way the CODE will run in the caller's package. Results will be printed to STDOUT as TITLE followed by the times. TITLE defaults to "timethis COUNT" if

none is provided. STYLE determines the format of the output, as described for timestr() below.

The COUNT can be zero or negative: this means the *minimum number of CPU seconds* to run. A zero signifies the default of 3 seconds. For example to run at least for 10 seconds:

```
timethis(-10, $code);
```

or to run two pieces of code tests for at least 3 seconds:

```
timethese(0, { test1 => '...', test2 => '...'});
```

CPU seconds is, in UNIX terms, the user time plus the system time of the process itself, as opposed to the real (wallclock) time and the time spent by the child processes. Less than 0.1 seconds is not accepted (-0.01 as the count, for example, will cause a fatal runtime exception).

Note that the CPU seconds is the **minimum** time: CPU scheduling and other operating system factors may complicate the attempt so that a little bit more time is spent. The benchmark output will, however, also tell the number of `$code` runs/second, which should be a more interesting number than the actually spent seconds.

Returns a Benchmark object.

- timethese (COUNT, CODEHASHREF, [STYLE])

  The CODEHASHREF is a reference to a hash containing names as keys and either a string to eval or a code reference for each value. For each (KEY, VALUE) pair in the CODEHASHREF, this routine will call

  ```
  timethis(COUNT, VALUE, KEY, STYLE);
  ```

  The routines are called in string comparison order of KEY.

  The COUNT can be zero or negative, see timethis().

  Returns a hash reference of Benchmark objects, keyed by name.

- timediff (T1, T2)

  Returns the difference between two Benchmark times as a Benchmark object suitable for passing to timestr().

- timestr (TIMEDIFF, [STYLE, [FORMAT]])

  Returns a string that formats the times in the TIMEDIFF object in the requested STYLE. TIMEDIFF is expected to be a Benchmark object similar to that returned by timediff().

  STYLE can be any of 'all', 'none', 'noc', 'nop' or 'auto'. 'all' shows each of the 5 times available ('wallclock' time, user time, system time, user time of children, and system time of children). 'noc' shows all except the two children times. 'nop' shows only wallclock and the two children times. 'auto' (the default) will act as 'all' unless the children times are both zero, in which case it acts as 'noc'. 'none' prevents output.

  FORMAT is the printf(3)-style format specifier (without the leading '%') to use to print the times. It defaults to '5.2f'.

Optional Exports

The following routines will be exported into your namespace if you specifically ask that they be imported:

- clearcache (COUNT)

  Clear the cached time for COUNT rounds of the null loop.

- clearallcache ( )

  Clear all cached times.

- cmpthese (COUNT, CODEHASHREF, [STYLE])

- cmpthese (RESULTSHASHREF, [STYLE])

  Optionally calls timethese(), then outputs comparison chart. This:

```
cmpthese( -1, { a => "++\$i", b => "\$i *= 2" } );
```

  outputs a chart like:

```
       Rate    b    a
b 2831802/s   -- -61%
a 7208959/s 155%   --
```

  This chart is sorted from slowest to fastest, and shows the percent speed difference between each pair of tests.

  `cmpthese` can also be passed the data structure that timethese() returns:

```
my $results = timethese( -1,
    { a => "++\$i", b => "\$i *= 2" } );
cmpthese( $results );
```

  in case you want to see both sets of results. If the first argument is an unblessed hash reference, that is RESULTSHASHREF; otherwise that is COUNT.

  Returns a reference to an ARRAY of rows, each row is an ARRAY of cells from the above chart, including labels. This:

```
my $rows = cmpthese( -1,
    { a => '++$i', b => '$i *= 2' }, "none" );
```

  returns a data structure like:

```
[
    [ '',       'Rate',   'b',    'a' ],
    [ 'b', '2885232/s',  '--', '-59%' ],
    [ 'a', '7099126/s', '146%',  '--' ],
]
```

  NOTE: This result value differs from previous versions, which returned the `timethese()` result structure. If you want that, just use the two statement `timethese...cmpthese` idiom shown above.

Incidentally, note the variance in the result values between the two examples; this is typical of benchmarking. If this were a real benchmark, you would probably want to run a lot more iterations.

- countit(TIME, CODE)

  Arguments: TIME is the minimum length of time to run CODE for, and CODE is the code to run. CODE may be either a code reference or a string to be eval'd; either way it will be run in the caller's package.

  TIME is *not* negative. countit() will run the loop many times to calculate the speed of CODE before running it for TIME. The actual time run for will usually be greater than TIME due to system clock resolution, so it's best to look at the number of iterations divided by the times that you are concerned with, not just the iterations.

  Returns: a Benchmark object.

- disablecache ( )

  Disable caching of timings for the null loop. This will force Benchmark to recalculate these timings for each new piece of code timed.

- enablecache ( )

  Enable caching of timings for the null loop. The time taken for COUNT rounds of the null loop will be calculated only once for each different COUNT used.

- timesum (T1, T2)

  Returns the sum of two Benchmark times as a Benchmark object suitable for passing to timestr().

:hireswallclock

If the Time::HiRes module has been installed, you can specify the special tag `:hireswallclock` for Benchmark (if Time::HiRes is not available, the tag will be silently ignored). This tag will cause the wallclock time to be measured in microseconds, instead of integer seconds. Note though that the speed computations are still conducted in CPU time, not wallclock time.

Benchmark Object

Many of the functions in this module return a Benchmark object, or in the case of `timethese()`, a reference to a hash, the values of which are Benchmark objects. This is useful if you want to store or further process results from Benchmark functions.

Internally the Benchmark object holds timing values, described in "NOTES" below. The following methods can be used to access them:

- cpu_p

  Total CPU (User + System) of the main (parent) process.

- cpu_c

  Total CPU (User + System) of any children processes.

- cpu_a

  Total CPU of parent and any children processes.

- real

  Real elapsed time “wallclock seconds”.

- iters

  Number of iterations run.

The following illustrates use of the Benchmark object:

```
my $result = timethis(100000, sub { ... });
print "total CPU = ", $result->cpu_a, "\n";
```

NOTES

The data is stored as a list of values from the time and times functions:

```
($real, $user, $system, $children_user, $children_system, $iters)
```

in seconds for the whole loop (not divided by the number of rounds).

The timing is done using time(3) and times(3).

Code is executed in the caller’s package.

The time of the null loop (a loop with the same number of rounds but empty loop body) is subtracted from the time of the real loop.

The null loop times can be cached, the key being the number of rounds. The caching can be controlled using calls like these:

```
clearcache($key);
clearallcache();

disablecache();
enablecache();
```

Caching is off by default, as it can (usually slightly) decrease accuracy and does not usually noticeably affect runtimes.

EXAMPLES

For example,

```
use Benchmark qw( cmpthese ) ;
my $x = 3;
cmpthese( -5, {
    a => sub{$x*$x},
    b => sub{$x**2},
} );
```

outputs something like this:

```
Benchmark: running a, b, each for at least 5 CPU seconds...
       Rate    b    a
```

(continues on next page)

(continued from previous page)

```
b 1559428/s   -- -62%
a 4152037/s 166%   --
```

while

```
use Benchmark qw( timethese cmpthese ) ;
my $x = 3;
my $r = timethese( -5, {
    a => sub{$x*$x},
    b => sub{$x**2},
} );
cmpthese $r;
```

outputs something like this:

```
Benchmark: running a, b, each for at least 5 CPU seconds...
         a: 10 wallclock secs ( 5.14 usr +  0.13 sys =  5.27 CPU) @␣
↪3835055.60/s (n=20210743)
         b:  5 wallclock secs ( 5.41 usr +  0.00 sys =  5.41 CPU) @␣
↪1574944.92/s (n=8520452)
       Rate    b    a
b 1574945/s   -- -59%
a 3835056/s 144%   --
```

INHERITANCE

Benchmark inherits from no other class, except of course from Exporter.

CAVEATS

Comparing eval'd strings with code references will give you inaccurate results: a code reference will show a slightly slower execution time than the equivalent eval'd string.

The real time timing is done using time(2) and the granularity is therefore only one second.

Short tests may produce negative figures because perl can appear to take longer to execute the empty loop than a short test; try:

```
timethis(100,'1');
```

The system time of the null loop might be slightly more than the system time of the loop with the actual code and therefore the difference might end up being < 0.

SEE ALSO

Devel::NYTProf - a Perl code profiler

AUTHORS

Jarkko Hietaniemi <jhi@iki.fi>, Tim Bunce <Tim.Bunce@ig.co.uk>

MODIFICATION HISTORY

September 8th, 1994; by Tim Bunce.

March 28th, 1997; by Hugo van der Sanden: added support for code references and the already documented 'debug' method; revamped documentation.

April 04-07th, 1997: by Jarkko Hietaniemi, added the run-for-some-time functionality.

September, 1999; by Barrie Slaymaker: math fixes and accuracy and efficiency tweaks. Added cmpthese(). A result is now returned from timethese(). Exposed countit() (was runfor()).

December, 2001; by Nicholas Clark: make timestr() recognise the style 'none' and return an empty string. If cmpthese is calling timethese, make it pass the style in. (so that 'none' will suppress output). Make sub new dump its debugging output to STDERR, to be consistent with everything else. All bugs found while writing a regression test.

September, 2002; by Jarkko Hietaniemi: add ':hireswallclock' special tag.

February, 2004; by Chia-liang Kao: make cmpthese and timestr use time statistics for children instead of parent when the style is 'nop'.

November, 2007; by Christophe Grosjean: make cmpthese and timestr compute time consistently with style argument, default is 'all' not 'noc' any more.

5.2.11 CPAN

query, download and build perl modules from CPAN sites

SYNOPSIS

Interactive mode:

```
perl -MCPAN -e shell
```

–or–

```
cpan
```

Basic commands:

```
# Modules:

cpan> install Acme::Meta                        # in the shell

CPAN::Shell->install("Acme::Meta");              # in perl

# Distributions:

cpan> install NWCLARK/Acme-Meta-0.02.tar.gz     # in the shell
```

(continues on next page)

(continued from previous page)

```
CPAN::Shell->
  install("NWCLARK/Acme-Meta-0.02.tar.gz");    # in perl

# module objects:

$mo = CPAN::Shell->expandany($mod);
$mo = CPAN::Shell->expand("Module",$mod);      # same thing

# distribution objects:

$do = CPAN::Shell->expand("Module",$mod)->distribution;
$do = CPAN::Shell->expandany($distro);         # same thing
$do = CPAN::Shell->expand("Distribution",
                          $distro);            # same thing
```

DESCRIPTION

The CPAN module automates or at least simplifies the make and install of perl modules and extensions. It includes some primitive searching capabilities and knows how to use LWP, HTTP::Tiny, Net::FTP and certain external download clients to fetch distributions from the net.

These are fetched from one or more mirrored CPAN (Comprehensive Perl Archive Network) sites and unpacked in a dedicated directory.

The CPAN module also supports named and versioned *bundles* of modules. Bundles simplify handling of sets of related modules. See Bundles below.

The package contains a session manager and a cache manager. The session manager keeps track of what has been fetched, built, and installed in the current session. The cache manager keeps track of the disk space occupied by the make processes and deletes excess space using a simple FIFO mechanism.

All methods provided are accessible in a programmer style and in an interactive shell style.

CPAN::shell([$prompt, $command]) Starting Interactive Mode

Enter interactive mode by running

```
perl -MCPAN -e shell
```

or

```
cpan
```

which puts you into a readline interface. If `Term::ReadKey` and either of `Term::ReadLine::Perl` or `Term::ReadLine::Gnu` are installed, history and command completion are supported.

Once at the command line, type h for one-page help screen; the rest should be self-explanatory.

The function call `shell` takes two optional arguments: one the prompt, the second the default initial command line (the latter only works if a real ReadLine interface module is installed).

The most common uses of the interactive modes are

- Searching for authors, bundles, distribution files and modules

  There are corresponding one-letter commands `a`, `b`, `d`, and `m` for each of the four categories and another, `i` for any of the mentioned four. Each of the four entities is implemented as a class with slightly differing methods for displaying an object.

  Arguments to these commands are either strings exactly matching the identification string of an object, or regular expressions matched case-insensitively against various attributes of the objects. The parser only recognizes a regular expression when you enclose it with slashes.

  The principle is that the number of objects found influences how an item is displayed. If the search finds one item, the result is displayed with the rather verbose method `as_string`, but if more than one is found, each object is displayed with the terse method `as_glimpse`.

  Examples:

```
cpan> m Acme::MetaSyntactic
Module id = Acme::MetaSyntactic
    CPAN_USERID  BOOK (Philippe Bruhat (BooK) <[...]>)
    CPAN_VERSION 0.99
    CPAN_FILE    B/BO/BOOK/Acme-MetaSyntactic-0.99.tar.gz
    UPLOAD_DATE  2006-11-06
    MANPAGE      Acme::MetaSyntactic - Themed metasyntactic␣
 ↪variables names
    INST_FILE    /usr/local/lib/perl/5.10.0/Acme/MetaSyntactic.
 ↪pm
    INST_VERSION 0.99
cpan> a BOOK
Author id = BOOK
    EMAIL        [...]
    FULLNAME     Philippe Bruhat (BooK)
cpan> d BOOK/Acme-MetaSyntactic-0.99.tar.gz
Distribution id = B/BO/BOOK/Acme-MetaSyntactic-0.99.tar.gz
    CPAN_USERID  BOOK (Philippe Bruhat (BooK) <[...]>)
    CONTAINSMODS Acme::MetaSyntactic Acme::MetaSyntactic::Alias␣
 ↪[...]
    UPLOAD_DATE  2006-11-06
cpan> m /lorem/
Module  = Acme::MetaSyntactic::loremipsum (BOOK/Acme-
 ↪MetaSyntactic-0.99.tar.gz)
Module    Text::Lorem            (ADEOLA/Text-Lorem-0.3.tar.gz)
Module    Text::Lorem::More      (RKRIMEN/Text-Lorem-More-0.12.
 ↪tar.gz)
```

(continues on next page)

(continued from previous page)

```
Module    Text::Lorem::More::Source (RKRIMEN/Text-Lorem-More-0.
↪12.tar.gz)
cpan> i /berlin/
Distribution    BEATNIK/Filter-NumberLines-0.02.tar.gz
Module  = DateTime::TimeZone::Europe::Berlin (DROLSKY/DateTime-
↪TimeZone-0.7904.tar.gz)
Module    Filter::NumberLines    (BEATNIK/Filter-NumberLines-0.
↪02.tar.gz)
Author          [...]
```

The examples illustrate several aspects: the first three queries target modules, authors, or distros directly and yield exactly one result. The last two use regular expressions and yield several results. The last one targets all of bundles, modules, authors, and distros simultaneously. When more than one result is available, they are printed in one-line format.

- `get`, `make`, `test`, `install`, `clean` modules or distributions

  These commands take any number of arguments and investigate what is necessary to perform the action. Argument processing is as follows:

```
known module name in format Foo/Bar.pm   module
other embedded slash                     distribution
  - with trailing slash dot              directory
enclosing slashes                        regexp
known module name in format Foo::Bar     module
```

  If the argument is a distribution file name (recognized by embedded slashes), it is processed. If it is a module, CPAN determines the distribution file in which this module is included and processes that, following any dependencies named in the module's META.yml or Makefile.PL (this behavior is controlled by the configuration parameter `prerequisites_policy`). If an argument is enclosed in slashes it is treated as a regular expression: it is expanded and if the result is a single object (distribution, bundle or module), this object is processed.

  Example:

```
install Dummy::Perl                    # installs the module
install AUXXX/Dummy-Perl-3.14.tar.gz   # installs that␣
↪distribution
install /Dummy-Perl-3.14/              # same if the regexp is␣
↪unambiguous
```

  `get` downloads a distribution file and untars or unzips it, `make` builds it, `test` runs the test suite, and `install` installs it.

  Any `make` or `test` is run unconditionally. An

```
install <distribution_file>
```

  is also run unconditionally. But for

```
install <module>
```

CPAN checks whether an install is needed and prints *module up to date* if the distribution file containing the module doesn't need updating.

CPAN also keeps track of what it has done within the current session and doesn't try to build a package a second time regardless of whether it succeeded or not. It does not repeat a test run if the test has been run successfully before. Same for install runs.

The `force` pragma may precede another command (currently: `get`, `make`, `test`, or `install`) to execute the command from scratch and attempt to continue past certain errors. See the section below on the `force` and the `fforce` pragma.

The `notest` pragma skips the test part in the build process.

Example:

```
cpan> notest install Tk
```

A `clean` command results in a

```
make clean
```

being executed within the distribution file's working directory.

- `readme`, `perldoc`, `look` module or distribution

  `readme` displays the README file of the associated distribution. `Look` gets and untars (if not yet done) the distribution file, changes to the appropriate directory and opens a subshell process in that directory. `perldoc` displays the module's pod documentation in html or plain text format.

- `ls` author
- `ls` globbing_expression

  The first form lists all distribution files in and below an author's CPAN directory as stored in the CHECKSUMS files distributed on CPAN. The listing recurses into subdirectories.

  The second form limits or expands the output with shell globbing as in the following examples:

```
ls JV/make*
ls GSAR/*make*
ls */*make*
```

  The last example is very slow and outputs extra progress indicators that break the alignment of the result.

  Note that globbing only lists directories explicitly asked for, for example FOO/* will not list FOO/bar/Acme-Sthg-n.nn.tar.gz. This may be regarded as a bug that may be changed in some future version.

- `failed`

The `failed` command reports all distributions that failed on one of `make`, `test` or `install` for some reason in the currently running shell session.

- Persistence between sessions

  If the `YAML` or the `YAML::Syck` module is installed a record of the internal state of all modules is written to disk after each step. The files contain a signature of the currently running perl version for later perusal.

  If the configurations variable `build_dir_reuse` is set to a true value, then CPAN.pm reads the collected YAML files. If the stored signature matches the currently running perl, the stored state is loaded into memory such that persistence between sessions is effectively established.

- The `force` and the `fforce` pragma

  To speed things up in complex installation scenarios, CPAN.pm keeps track of what it has already done and refuses to do some things a second time. A `get`, a `make`, and an `install` are not repeated. A `test` is repeated only if the previous test was unsuccessful. The diagnostic message when CPAN.pm refuses to do something a second time is one of _Has already been `_unwrapped|made|tested successfully` or something similar. Another situation where CPAN refuses to act is an `install` if the corresponding `test` was not successful.

  In all these cases, the user can override this stubborn behaviour by prepending the command with the word force, for example:

```
cpan> force get Foo
cpan> force make AUTHOR/Bar-3.14.tar.gz
cpan> force test Baz
cpan> force install Acme::Meta
```

  Each *forced* command is executed with the corresponding part of its memory erased.

  The `fforce` pragma is a variant that emulates a `force get` which erases the entire memory followed by the action specified, effectively restarting the whole get/make/test/install procedure from scratch.

- Lockfile

  Interactive sessions maintain a lockfile, by default `~/.cpan/.lock`. Batch jobs can run without a lockfile and not disturb each other.

  The shell offers to run in *downgraded mode* when another process is holding the lockfile. This is an experimental feature that is not yet tested very well. This second shell then does not write the history file, does not use the metadata file, and has a different prompt.

- Signals

  CPAN.pm installs signal handlers for SIGINT and SIGTERM. While you are in the cpan-shell, it is intended that you can press `^C` anytime and return to the cpan-shell prompt. A SIGTERM will cause the cpan-shell to clean up and leave the shell loop. You can emulate the effect of a SIGTERM by sending two consecutive SIGINTs, which usually means by pressing `^C` twice.

CPAN.pm ignores SIGPIPE. If the user sets `inactivity_timeout`, a SIGALRM is used during the run of the `perl Makefile.PL` or `perl Build.PL` subprocess. A SIGALRM is also used during module version parsing, and is controlled by `version_timeout`.

CPAN::Shell

The commands available in the shell interface are methods in the package CPAN::Shell. If you enter the shell command, your input is split by the Text::ParseWords::shellwords() routine, which acts like most shells do. The first word is interpreted as the method to be invoked, and the rest of the words are treated as the method's arguments. Continuation lines are supported by ending a line with a literal backslash.

autobundle

`autobundle` writes a bundle file into the `$CPAN::Config->{cpan_home}/Bundle` directory. The file contains a list of all modules that are both available from CPAN and currently installed within @INC. Duplicates of each distribution are suppressed. The name of the bundle file is based on the current date and a counter, e.g. `Bundle/Snapshot_2012_05_21_00.pm`. This is installed again by running `cpan Bundle::Snapshot_2012_05_21_00`, or installing `Bundle::Snapshot_2012_05_21_00` from the CPAN shell.

Return value: path to the written file.

hosts

Note: this feature is still in alpha state and may change in future versions of CPAN.pm

This commands provides a statistical overview over recent download activities. The data for this is collected in the YAML file `FTPstats.yml` in your `cpan_home` directory. If no YAML module is configured or YAML not installed, or if `ftpstats_size` is set to a value `<=0`, no stats are provided.

install_tested

Install all distributions that have been tested successfully but have not yet been installed. See also `is_tested`.

is_tested

List all build directories of distributions that have been tested successfully but have not yet been installed. See also `install_tested`.

mkmyconfig

mkmyconfig() writes your own CPAN::MyConfig file into your `~/.cpan/` directory so that you can save your own preferences instead of the system-wide ones.

r [Module|/Regexp/]...

scans current perl installation for modules that have a newer version available on CPAN and provides a list of them. If called without argument, all potential upgrades are listed; if called with arguments the list is filtered to the modules and regexps given as arguments.

The listing looks something like this:

```
Package namespace         installed    latest  in CPAN file
CPAN                        1.94_64    1.9600  ANDK/CPAN-1.9600.tar.
 ↪gz
CPAN::Reporter               1.1801    1.1902  DAGOLDEN/CPAN-
 ↪Reporter-1.1902.tar.gz
YAML                           0.70      0.73  INGY/YAML-0.73.tar.gz
YAML::Syck                     1.14      1.17  AVAR/YAML-Syck-1.17.
 ↪tar.gz
YAML::Tiny                     1.44      1.50  ADAMK/YAML-Tiny-1.50.
 ↪tar.gz
CGI                            3.43      3.55  MARKSTOS/CGI.pm-3.55.
 ↪tar.gz
Module::Build::YAML            1.40      1.41  DAGOLDEN/Module-
 ↪Build-0.3800.tar.gz
TAP::Parser::Result::YAML      3.22      3.23  ANDYA/Test-Harness-3.
 ↪23.tar.gz
YAML::XS                       0.34      0.35  INGY/YAML-LibYAML-0.
 ↪35.tar.gz
```

It suppresses duplicates in the column `in CPAN file` such that distributions with many upgradeable modules are listed only once.

Note that the list is not sorted.

recent ***EXPERIMENTAL COMMAND***

The `recent` command downloads a list of recent uploads to CPAN and displays them *slowly*. While the command is running, a $SIG{INT} exits the loop after displaying the current item.

Note: This command requires XML::LibXML installed.

Note: This whole command currently is just a hack and will probably change in future versions of CPAN.pm, but the general approach will likely remain.

Note: See also smoke

recompile

The primary purpose of this command is to finish a network installation. Imagine you have a common source tree for two different architectures. You decide to do a completely independent fresh installation. You start on one architecture with the help of a Bundle file produced earlier. CPAN installs the whole Bundle for you, but when you try to repeat the job on the second architecture, CPAN responds with a `"Foo up to date"`

message for all modules. So you invoke CPAN's recompile on the second architecture and you're done.

Another popular use for `recompile` is to act as a rescue in case your perl breaks binary compatibility. If one of the modules that CPAN uses is in turn depending on binary compatibility (so you cannot run CPAN commands), then you should try the CPAN::Nox module for recovery.

report Bundle|Distribution|Module

The `report` command temporarily turns on the `test_report` config variable, then runs the `force test` command with the given arguments. The `force` pragma reruns the tests and repeats every step that might have failed before.

smoke ***EXPERIMENTAL COMMAND***

***** WARNING: this command downloads and executes software from CPAN to your computer of completely unknown status. You should never do this with your normal account and better have a dedicated well separated and secured machine to do this. *****

The `smoke` command takes the list of recent uploads to CPAN as provided by the `recent` command and tests them all. While the command is running $SIG{INT} is defined to mean that the current item shall be skipped.

Note: This whole command currently is just a hack and will probably change in future versions of CPAN.pm, but the general approach will likely remain.

Note: See also recent

upgrade [Module|/Regexp/]...

The `upgrade` command first runs an `r` command with the given arguments and then installs the newest versions of all modules that were listed by that.

The four CPAN::* Classes: Author, Bundle, Module, Distribution

Although it may be considered internal, the class hierarchy does matter for both users and programmer. CPAN.pm deals with the four classes mentioned above, and those classes all share a set of methods. Classical single polymorphism is in effect. A metaclass object registers all objects of all kinds and indexes them with a string. The strings referencing objects have a separated namespace (well, not completely separated):

```
      Namespace                         Class

words containing a "/" (slash)      Distribution
 words starting with Bundle::          Bundle
       everything else             Module or Author
```

Modules know their associated Distribution objects. They always refer to the most recent official release. Developers may mark their releases as unstable development versions (by inserting an underscore into the module version number which will also be reflected in the distribution name when you run 'make dist'), so the really hottest and

newest distribution is not always the default. If a module Foo circulates on CPAN in both version 1.23 and 1.23_90, CPAN.pm offers a convenient way to install version 1.23 by saying

```
install Foo
```

This would install the complete distribution file (say BAR/Foo-1.23.tar.gz) with all accompanying material. But if you would like to install version 1.23_90, you need to know where the distribution file resides on CPAN relative to the authors/id/ directory. If the author is BAR, this might be BAR/Foo-1.23_90.tar.gz; so you would have to say

```
install BAR/Foo-1.23_90.tar.gz
```

The first example will be driven by an object of the class CPAN::Module, the second by an object of class CPAN::Distribution.

Integrating local directories

Note: this feature is still in alpha state and may change in future versions of CPAN.pm

Distribution objects are normally distributions from the CPAN, but there is a slightly degenerate case for Distribution objects, too, of projects held on the local disk. These distribution objects have the same name as the local directory and end with a dot. A dot by itself is also allowed for the current directory at the time CPAN.pm was used. All actions such as `make`, `test`, and `install` are applied directly to that directory. This gives the command `cpan .` an interesting touch: while the normal mantra of installing a CPAN module without CPAN.pm is one of

```
perl Makefile.PL                  perl Build.PL
       ( go and get prerequisites )
make                              ./Build
make test                         ./Build test
make install                      ./Build install
```

the command `cpan .` does all of this at once. It figures out which of the two mantras is appropriate, fetches and installs all prerequisites, takes care of them recursively, and finally finishes the installation of the module in the current directory, be it a CPAN module or not.

The typical usage case is for private modules or working copies of projects from remote repositories on the local disk.

Redirection

The usual shell redirection symbols | and > are recognized by the cpan shell **only when surrounded by whitespace**. So piping to pager or redirecting output into a file works somewhat as in a normal shell, with the stipulation that you must type extra spaces.

Plugin support ***EXPERIMENTAL***

Plugins are objects that implement any of currently eight methods:

```
pre_get
post_get
pre_make
post_make
pre_test
post_test
pre_install
post_install
```

The `plugin_list` configuration parameter holds a list of strings of the form

```
Modulename=arg0,arg1,arg2,arg3,...
```

eg:

```
CPAN::Plugin::Flurb=dir,/opt/pkgs/flurb/raw,verbose,1
```

At run time, each listed plugin is instantiated as a singleton object by running the equivalent of this pseudo code:

```
my $plugin = <string representation from config>;
<generate Modulename and arguments from $plugin>;
my $p = $instance{$plugin} ||= Modulename->new($arg0,$arg1,...);
```

The generated singletons are kept around from instantiation until the end of the shell session. <plugin_list> can be reconfigured at any time at run time. While the cpan shell is running, it checks all activated plugins at each of the 8 reference points listed above and runs the respective method if it is implemented for that object. The method is called with the active CPAN::Distribution object passed in as an argument.

CONFIGURATION

When the CPAN module is used for the first time, a configuration dialogue tries to determine a couple of site specific options. The result of the dialog is stored in a hash reference `$CPAN::Config` in a file CPAN/Config.pm.

Default values defined in the CPAN/Config.pm file can be overridden in a user specific file: CPAN/MyConfig.pm. Such a file is best placed in `$HOME/.cpan/CPAN/MyConfig.pm`, because `$HOME/.cpan` is added to the search path of the CPAN module before the use() or require() statements. The mkmyconfig command writes this file for you.

If you want to keep your own CPAN/MyConfig.pm somewhere else, you should load it before loading CPAN.pm, e.g.:

```
 perl -I/tmp/somewhere -MCPAN::MyConfig -MCPAN -eshell

 --or--

perl -I/tmp/somewhere -MCPAN::MyConfig -S cpan
```

Once you are in the shell you can change your configuration as follows.

The `o conf` command has various bells and whistles:

- completion support

  If you have a ReadLine module installed, you can hit TAB at any point of the commandline and `o conf` will offer you completion for the built-in subcommands and/or config variable names.

- displaying some help: o conf help

  Displays a short help

- displaying current values: o conf [KEY]

  Displays the current value(s) for this config variable. Without KEY, displays all subcommands and config variables.

  Example:

```
o conf shell
```

  If KEY starts and ends with a slash, the string in between is treated as a regular expression and only keys matching this regexp are displayed

  Example:

```
o conf /color/
```

- changing of scalar values: o conf KEY VALUE

  Sets the config variable KEY to VALUE. The empty string can be specified as usual in shells, with `''` or `""`

  Example:

```
o conf wget /usr/bin/wget
```

- changing of list values: o conf KEY SHIFT|UNSHIFT|PUSH|POP|SPLICE|LIST

  If a config variable name ends with `list`, it is a list. `o conf KEY shift` removes the first element of the list, `o conf KEY pop` removes the last element of the list. `o conf KEYS unshift LIST` prepends a list of values to the list, `o conf KEYS push LIST` appends a list of valued to the list.

  Likewise, `o conf KEY splice LIST` passes the LIST to the corresponding splice command.

  Finally, any other list of arguments is taken as a new list value for the KEY variable discarding the previous value.

  Examples:

```
o conf urllist unshift http://cpan.dev.local/CPAN
o conf urllist splice 3 1
o conf urllist http://cpan1.local http://cpan2.local ftp://ftp.
 ↪perl.org
```

- reverting to saved: o conf defaults

  Reverts all config variables to the state in the saved config file.

- saving the config: o conf commit

  Saves all config variables to the current config file (CPAN/Config.pm or CPAN/MyConfig.pm that was loaded at start).

The configuration dialog can be started any time later again by issuing the command o `conf init` in the CPAN shell. A subset of the configuration dialog can be run by issuing `o conf init WORD` where WORD is any valid config variable or a regular expression.

Config Variables

The following keys in the hash reference $CPAN::Config are currently defined:

```
allow_installing_module_downgrades
                    allow or disallow installing module downgrades
allow_installing_outdated_dists
                    allow or disallow installing modules that are
                    indexed in the cpan index pointing to a distro
                    with a higher distro-version number
applypatch          path to external prg
auto_commit         commit all changes to config variables to disk
build_cache         size of cache for directories to build modules
build_dir           locally accessible directory to build modules
build_dir_reuse     boolean if distros in build_dir are persistent
build_requires_install_policy
                    to install or not to install when a module is
                    only needed for building. yes|no|ask/yes|ask/no
bzip2               path to external prg
cache_metadata      use serializer to cache metadata
check_sigs          if signatures should be verified
cleanup_after_install
                    remove build directory immediately after a
                    successful install and remember that for the
                    duration of the session
colorize_debug      Term::ANSIColor attributes for debugging output
colorize_output     boolean if Term::ANSIColor should colorize output
colorize_print      Term::ANSIColor attributes for normal output
colorize_warn       Term::ANSIColor attributes for warnings
commandnumber_in_prompt
                    boolean if you want to see current command number
commands_quote      preferred character to use for quoting external
                    commands when running them. Defaults to double
                    quote on Windows, single tick everywhere else;
                    can be set to space to disable quoting
connect_to_internet_ok
                    whether to ask if opening a connection is ok␣
↪before
                    urllist is specified
```

(continues on next page)

(continued from previous page)

```
cpan_home           local directory reserved for this package
curl                path to external prg
dontload_hash       DEPRECATED
dontload_list       arrayref: modules in the list will not be
                    loaded by the CPAN::has_inst() routine
ftp                 path to external prg
ftp_passive         if set, the environment variable FTP_PASSIVE is␣
 ↪set
                    for downloads
ftp_proxy           proxy host for ftp requests
ftpstats_period     max number of days to keep download statistics
ftpstats_size       max number of items to keep in the download␣
 ↪statistics
getcwd              see below
gpg                 path to external prg
gzip                location of external program gzip
halt_on_failure     stop processing after the first failure of queued
                    items or dependencies
histfile            file to maintain history between sessions
histsize            maximum number of lines to keep in histfile
http_proxy          proxy host for http requests
inactivity_timeout breaks interactive Makefile.PLs or Build.PLs
                    after this many seconds inactivity. Set to 0 to
                    disable timeouts.
index_expire        refetch index files after this many days
inhibit_startup_message
                    if true, suppress the startup message
keep_source_where   directory in which to keep the source (if we do)
load_module_verbosity
                    report loading of optional modules used by CPAN.
 ↪pm
lynx                path to external prg
make                location of external make program
make_arg            arguments that should always be passed to 'make'
make_install_make_command
                    the make command for running 'make install', for
                    example 'sudo make'
make_install_arg    same as make_arg for 'make install'
makepl_arg          arguments passed to 'perl Makefile.PL'
mbuild_arg          arguments passed to './Build'
mbuild_install_arg arguments passed to './Build install'
mbuild_install_build_command
                    command to use instead of './Build' when we are
                    in the install stage, for example 'sudo ./Build'
mbuildpl_arg        arguments passed to 'perl Build.PL'
ncftp               path to external prg
ncftpget            path to external prg
no_proxy            don't proxy to these hosts/domains (comma␣
 ↪separated list)
```

(continues on next page)

(continued from previous page)

```
pager              location of external program more (or any pager)
password           your password if you CPAN server wants one
patch              path to external prg
patches_dir        local directory containing patch files
perl5lib_verbosity verbosity level for PERL5LIB additions
plugin_list        list of active hooks (see Plugin support above
                   and the CPAN::Plugin module)
prefer_external_tar
                   per default all untar operations are done with
                   Archive::Tar; by setting this variable to true
                   the external tar command is used if available
prefer_installer   legal values are MB and EUMM: if a module comes
                   with both a Makefile.PL and a Build.PL, use the
                   former (EUMM) or the latter (MB); if the module
                   comes with only one of the two, that one will be
                   used no matter the setting
prerequisites_policy
                   what to do if you are missing module
 ↪prerequisites
                   ('follow' automatically, 'ask' me, or 'ignore')
                   For 'follow', also sets PERL_AUTOINSTALL and
                   PERL_EXTUTILS_AUTOINSTALL for "--defaultdeps" if
                   not already set
prefs_dir          local directory to store per-distro build options
proxy_user         username for accessing an authenticating proxy
proxy_pass         password for accessing an authenticating proxy
pushy_https        use https to cpan.org when possible, otherwise
 ↪use http
                   to cpan.org and issue a warning
randomize_urllist  add some randomness to the sequence of the
 ↪urllist
recommends_policy  whether recommended prerequisites should be
 ↪included
scan_cache         controls scanning of cache ('atstart', 'atexit'
 ↪or 'never')
shell              your favorite shell
show_unparsable_versions
                   boolean if r command tells which modules are
 ↪versionless
show_upload_date   boolean if commands should try to determine
 ↪upload date
show_zero_versions boolean if r command tells for which modules
 ↪$version==0
suggests_policy    whether suggested prerequisites should be
 ↪included
tar                location of external program tar
tar_verbosity      verbosity level for the tar command
term_is_latin      deprecated: if true Unicode is translated to ISO-
 ↪8859-1
```

(continues on next page)

(continued from previous page)

```
                    (and nonsense for characters outside latin range)
term_ornaments      boolean to turn ReadLine ornamenting on/off
test_report         email test reports (if CPAN::Reporter is␣
↪installed)
trust_test_report_history
                    skip testing when previously tested ok (according␣
↪to
                    CPAN::Reporter history)
unzip               location of external program unzip
urllist             arrayref to nearby CPAN sites (or equivalent␣
↪locations)
urllist_ping_external
                    use external ping command when autoselecting␣
↪mirrors
urllist_ping_verbose
                    increase verbosity when autoselecting mirrors
use_prompt_default set PERL_MM_USE_DEFAULT for configure/make/test/
↪install
use_sqlite          use CPAN::SQLite for metadata storage (fast and␣
↪lean)
username            your username if you CPAN server wants one
version_timeout     stops version parsing after this many seconds.
                    Default is 15 secs. Set to 0 to disable.
wait_list           arrayref to a wait server to try (See CPAN::WAIT)
wget                path to external prg
yaml_load_code      enable YAML code deserialisation via␣
↪CPAN::DeferredCode
yaml_module         which module to use to read/write YAML files
```

You can set and query each of these options interactively in the cpan shell with the `o conf` or the `o conf init` command as specified below.

- `o conf <scalar option>`

  prints the current value of the *scalar option*

- `o conf <scalar option> <value>`

  Sets the value of the *scalar option* to *value*

- `o conf <list option>`

  prints the current value of the *list option* in MakeMaker's neatvalue format.

- `o conf <list option> [shift|pop]`

  shifts or pops the array in the *list option* variable

- `o conf <list option> [unshift|push|splice] <list>`

  works like the corresponding perl commands.

- interactive editing: o conf init [MATCH|LIST]

  Runs an interactive configuration dialog for matching variables. Without argument runs the dialog over all supported config variables. To specify a MATCH the

argument must be enclosed by slashes.

Examples:

```
o conf init ftp_passive ftp_proxy
o conf init /color/
```

Note: this method of setting config variables often provides more explanation about the functioning of a variable than the manpage.

CPAN::anycwd($path): Note on config variable getcwd

CPAN.pm changes the current working directory often and needs to determine its own current working directory. By default it uses Cwd::cwd, but if for some reason this doesn't work on your system, configure alternatives according to the following table:

- cwd

  Calls Cwd::cwd

- getcwd

  Calls Cwd::getcwd

- fastcwd

  Calls Cwd::fastcwd

- getdcwd

  Calls Cwd::getdcwd

- backtickcwd

  Calls the external command cwd.

Note on the format of the urllist parameter

urllist parameters are URLs according to RFC 1738. We do a little guessing if your URL is not compliant, but if you have problems with `file` URLs, please try the correct format. Either:

```
file://localhost/whatever/ftp/pub/CPAN/
```

or

```
file:///home/ftp/pub/CPAN/
```

The urllist parameter has CD-ROM support

The `urllist` parameter of the configuration table contains a list of URLs used for downloading. If the list contains any `file` URLs, CPAN always tries there first. This feature is disabled for index files. So the recommendation for the owner of a CD-ROM with CPAN contents is: include your local, possibly outdated CD-ROM as a `file` URL at the end of urllist, e.g.

```
o conf urllist push file://localhost/CDROM/CPAN
```

CPAN.pm will then fetch the index files from one of the CPAN sites that come at the beginning of urllist. It will later check for each module to see whether there is a local copy of the most recent version.

Another peculiarity of urllist is that the site that we could successfully fetch the last file from automatically gets a preference token and is tried as the first site for the next request. So if you add a new site at runtime it may happen that the previously preferred site will be tried another time. This means that if you want to disallow a site for the next transfer, it must be explicitly removed from urllist.

Maintaining the urllist parameter

If you have YAML.pm (or some other YAML module configured in `yaml_module`) installed, CPAN.pm collects a few statistical data about recent downloads. You can view the statistics with the `hosts` command or inspect them directly by looking into the `FTPstats.yml` file in your `cpan_home` directory.

To get some interesting statistics, it is recommended that `randomize_urllist` be set; this introduces some amount of randomness into the URL selection.

The `requires` and `build_requires` dependency declarations

Since CPAN.pm version 1.88_51 modules declared as `build_requires` by a distribution are treated differently depending on the config variable `build_requires_install_policy`. By setting `build_requires_install_policy` to `no`, such a module is not installed. It is only built and tested, and then kept in the list of tested but uninstalled modules. As such, it is available during the build of the dependent module by integrating the path to the `blib/arch` and `blib/lib` directories in the environment variable PERL5LIB. If `build_requires_install_policy` is set to `yes`, then both modules declared as `requires` and those declared as `build_requires` are treated alike. By setting to `ask/yes` or `ask/no`, CPAN.pm asks the user and sets the default accordingly.

Configuration of the allow_installing_* parameters

The `allow_installing_*` parameters are evaluated during the `make` phase. If set to `yes`, they allow the testing and the installation of the current distro and otherwise have no effect. If set to `no`, they may abort the build (preventing testing and installing), depending on the contents of the `blib/` directory. The `blib/` directory is the directory that holds all the files that would usually be installed in the `install` phase.

`allow_installing_outdated_dists` compares the `blib/` directory with the CPAN index. If it finds something there that belongs, according to the index, to a different dist, it aborts the current build.

`allow_installing_module_downgrades` compares the `blib/` directory with already installed modules, actually their version numbers, as determined by ExtUtils::MakeMaker or equivalent. If a to-be-installed module would downgrade an already installed module, the current build is aborted.

An interesting twist occurs when a distroprefs document demands the installation of an outdated dist via goto while `allow_installing_outdated_dists` forbids it. With-

out additional provisions, this would let the `allow_installing_outdated_dists` win and the distroprefs lose. So the proper arrangement in such a case is to write a second distroprefs document for the distro that `goto` points to and overrule the `cpanconfig` there. E.g.:

```
---
match:
  distribution: "^MAUKE/Keyword-Simple-0.04.tar.gz"
goto: "MAUKE/Keyword-Simple-0.03.tar.gz"
---
match:
  distribution: "^MAUKE/Keyword-Simple-0.03.tar.gz"
cpanconfig:
  allow_installing_outdated_dists: yes
```

Configuration for individual distributions (*Distroprefs*)

(**Note:** This feature has been introduced in CPAN.pm 1.8854)

Distributions on CPAN usually behave according to what we call the CPAN mantra. Or since the advent of Module::Build we should talk about two mantras:

```
perl Makefile.PL     perl Build.PL
make                 ./Build
make test            ./Build test
make install         ./Build install
```

But some modules cannot be built with this mantra. They try to get some extra data from the user via the environment, extra arguments, or interactively–thus disturbing the installation of large bundles like Phalanx100 or modules with many dependencies like Plagger.

The distroprefs system of `CPAN.pm` addresses this problem by allowing the user to specify extra informations and recipes in YAML files to either

- pass additional arguments to one of the four commands,
- set environment variables
- instantiate an Expect object that reads from the console, waits for some regular expressions and enters some answers
- temporarily override assorted `CPAN.pm` configuration variables
- specify dependencies the original maintainer forgot
- disable the installation of an object altogether

See the YAML and Data::Dumper files that come with the `CPAN.pm` distribution in the `distroprefs/` directory for examples.

Filenames

The YAML files themselves must have the `.yml` extension; all other files are ignored (for two exceptions see *Fallback Data::Dumper and Storable* below). The containing directory can be specified in `CPAN.pm` in the `prefs_dir` config variable. Try `o conf init prefs_dir` in the CPAN shell to set and activate the distroprefs system.

Every YAML file may contain arbitrary documents according to the YAML specification, and every document is treated as an entity that can specify the treatment of a single distribution.

Filenames can be picked arbitrarily; `CPAN.pm` always reads all files (in alphabetical order) and takes the key `match` (see below in *Language Specs*) as a hashref containing match criteria that determine if the current distribution matches the YAML document or not.

Fallback Data::Dumper and Storable

If neither your configured `yaml_module` nor YAML.pm is installed, CPAN.pm falls back to using Data::Dumper and Storable and looks for files with the extensions `.dd` or `.st` in the `prefs_dir` directory. These files are expected to contain one or more hashrefs. For Data::Dumper generated files, this is expected to be done with by defining `$VAR1`, `$VAR2`, etc. The YAML shell would produce these with the command

```
ysh < somefile.yml > somefile.dd
```

For Storable files the rule is that they must be constructed such that `Storable::retrieve(file)` returns an array reference and the array elements represent one distropref object each. The conversion from YAML would look like so:

```
perl -MYAML=LoadFile -MStorable=nstore -e '
    @y=LoadFile(shift);
    nstore(\@y, shift)' somefile.yml somefile.st
```

In bootstrapping situations it is usually sufficient to translate only a few YAML files to Data::Dumper for crucial modules like `YAML::Syck`, `YAML.pm` and `Expect.pm`. If you prefer Storable over Data::Dumper, remember to pull out a Storable version that writes an older format than all the other Storable versions that will need to read them.

Blueprint

The following example contains all supported keywords and structures with the exception of `eexpect` which can be used instead of `expect`.

```
---
comment: "Demo"
match:
  module: "Dancing::Queen"
  distribution: "^CHACHACHA/Dancing-"
  not_distribution: "\.zip$"
  perl: "/usr/local/cariba-perl/bin/perl"
  perlconfig:
```

(continues on next page)

(continued from previous page)

```
    archname: "freebsd"
    not_cc: "gcc"
  env:
    DANCING_FLOOR: "Shubiduh"
disabled: 1
cpanconfig:
  make: gmake
pl:
  args:
    - "--somearg=specialcase"

  env: {}

  expect:
    - "Which is your favorite fruit"
    - "apple\n"

make:
  args:
    - all
    - extra-all

  env: {}

  expect: []

  commandline: "echo SKIPPING make"

test:
  args: []

  env: {}

  expect: []

install:
  args: []

  env:
    WANT_TO_INSTALL: YES

  expect:
    - "Do you really want to install"
    - "y\n"

patches:
  - "ABCDE/Fedcba-3.14-ABCDE-01.patch"

depends:
```

(continues on next page)

(continued from previous page)

```
  configure_requires:
    LWP: 5.8
  build_requires:
    Test::Exception: 0.25
  requires:
    Spiffy: 0.30
```

Language Specs

Every YAML document represents a single hash reference. The valid keys in this hash are as follows:

- comment [scalar]

  A comment

- cpanconfig [hash]

  Temporarily override assorted `CPAN.pm` configuration variables.

  Supported are: `build_requires_install_policy`, `check_sigs`, `make`, `make_install_make_command`, `prefer_installer`, `test_report`. Please report as a bug when you need another one supported.

- depends [hash] *** EXPERIMENTAL FEATURE ***

  All three types, namely `configure_requires`, `build_requires`, and `requires` are supported in the way specified in the META.yml specification. The current implementation *merges* the specified dependencies with those declared by the package maintainer. In a future implementation this may be changed to override the original declaration.

- disabled [boolean]

  Specifies that this distribution shall not be processed at all.

- features [array] *** EXPERIMENTAL FEATURE ***

  Experimental implementation to deal with optional_features from META.yml. Still needs coordination with installer software and currently works only for META.yml declaring `dynamic_config=0`. Use with caution.

- goto [string]

  The canonical name of a delegate distribution to install instead. Useful when a new version, although it tests OK itself, breaks something else or a developer release or a fork is already uploaded that is better than the last released version.

- install [hash]

  Processing instructions for the `make install` or `./Build install` phase of the CPAN mantra. See below under *Processing Instructions*.

- make [hash]

  Processing instructions for the `make` or `./Build` phase of the CPAN mantra. See below under *Processing Instructions*.

- match [hash]

  A hashref with one or more of the keys `distribution`, `module`, `perl`, `perlconfig`, and `env` that specify whether a document is targeted at a specific CPAN distribution or installation. Keys prefixed with `not_` negates the corresponding match.

  The corresponding values are interpreted as regular expressions. The `distribution` related one will be matched against the canonical distribution name, e.g. "AUTHOR/Foo-Bar-3.14.tar.gz".

  The `module` related one will be matched against *all* modules contained in the distribution until one module matches.

  The `perl` related one will be matched against `$^X` (but with the absolute path).

  The value associated with `perlconfig` is itself a hashref that is matched against corresponding values in the `%Config::Config` hash living in the `Config.pm` module. Keys prefixed with `not_` negates the corresponding match.

  The value associated with `env` is itself a hashref that is matched against corresponding values in the `%ENV` hash. Keys prefixed with `not_` negates the corresponding match.

  If more than one restriction of `module`, `distribution`, etc. is specified, the results of the separately computed match values must all match. If so, the hashref represented by the YAML document is returned as the preference structure for the current distribution.

- patches [array]

  An array of patches on CPAN or on the local disk to be applied in order via an external patch program. If the value for the `-p` parameter is `0` or `1` is determined by reading the patch beforehand. The path to each patch is either an absolute path on the local filesystem or relative to a patch directory specified in the `patches_dir` configuration variable or in the format of a canonical distro name. For examples please consult the distroprefs/ directory in the CPAN.pm distribution (these examples are not installed by default).

  Note: if the `applypatch` program is installed and `CPAN::Config` knows about it **and** a patch is written by the `makepatch` program, then `CPAN.pm` lets `applypatch` apply the patch. Both `makepatch` and `applypatch` are available from CPAN in the `JV/makepatch-*` distribution.

- pl [hash]

  Processing instructions for the `perl Makefile.PL` or `perl Build.PL` phase of the CPAN mantra. See below under *Processing Instructions*.

- test [hash]

  Processing instructions for the `make test` or `./Build test` phase of the CPAN mantra. See below under *Processing Instructions*.

Processing Instructions

- args [array]

  Arguments to be added to the command line

- commandline

  A full commandline to run via `system()`. During execution, the environment variable PERL is set to $^X (but with an absolute path). If `commandline` is specified, `args` is not used.

- eexpect [hash]

  Extended `expect`. This is a hash reference with four allowed keys, `mode`, `timeout`, `reuse`, and `talk`.

  You must install the `Expect` module to use `eexpect`. CPAN.pm does not install it for you.

  `mode` may have the values `deterministic` for the case where all questions come in the order written down and `anyorder` for the case where the questions may come in any order. The default mode is `deterministic`.

  `timeout` denotes a timeout in seconds. Floating-point timeouts are OK. With `mode=deterministic`, the timeout denotes the timeout per question; with `mode=anyorder` it denotes the timeout per byte received from the stream or questions.

  `talk` is a reference to an array that contains alternating questions and answers. Questions are regular expressions and answers are literal strings. The Expect module watches the stream from the execution of the external program (`perl Makefile.PL`, `perl Build.PL`, `make`, etc.).

  For `mode=deterministic`, the CPAN.pm injects the corresponding answer as soon as the stream matches the regular expression.

  For `mode=anyorder` CPAN.pm answers a question as soon as the timeout is reached for the next byte in the input stream. In this mode you can use the `reuse` parameter to decide what will happen with a question-answer pair after it has been used. In the default case (reuse=0) it is removed from the array, avoiding being used again accidentally. If you want to answer the question `Do you really want to do that` several times, then it must be included in the array at least as often as you want this answer to be given. Setting the parameter `reuse` to 1 makes this repetition unnecessary.

- env [hash]

  Environment variables to be set during the command

- expect [array]

  You must install the `Expect` module to use `expect`. CPAN.pm does not install it for you.

  `expect: <array>` is a short notation for this `eexpect`:

```
eexpect:
        mode: deterministic
```

(continues on next page)

(continued from previous page)

```
        timeout: 15
        talk: <array>
```

Schema verification with Kwalify

If you have the Kwalify module installed (which is part of the Bundle::CPANxxl), then all your distroprefs files are checked for syntactic correctness.

Example Distroprefs Files

CPAN.pm comes with a collection of example YAML files. Note that these are really just examples and should not be used without care because they cannot fit everybody's purpose. After all, the authors of the packages that ask questions had a need to ask, so you should watch their questions and adjust the examples to your environment and your needs. You have been warned:-)

PROGRAMMER'S INTERFACE

If you do not enter the shell, shell commands are available both as methods (`CPAN::Shell->install(...)`) and as functions in the calling package (`install(...)`). Before calling low-level commands, it makes sense to initialize components of CPAN you need, e.g.:

```
CPAN::HandleConfig->load;
CPAN::Shell::setup_output;
CPAN::Index->reload;
```

High-level commands do such initializations automatically.

There's currently only one class that has a stable interface - CPAN::Shell. All commands that are available in the CPAN shell are methods of the class CPAN::Shell. The arguments on the commandline are passed as arguments to the method.

So if you take for example the shell command

```
notest install A B C
```

the actually executed command is

```
CPAN::Shell->notest("install","A","B","C");
```

Each of the commands that produce listings of modules (`r`, `autobundle`, `u`) also return a list of the IDs of all modules within the list.

- expand($type,@things)

  The IDs of all objects available within a program are strings that can be expanded to the corresponding real objects with the `CPAN::Shell->expand("Module", @things)` method. Expand returns a list of CPAN::Module objects according to the `@things` arguments given. In scalar context, it returns only the first element of the list.

- expandany(@things)

  Like expand, but returns objects of the appropriate type, i.e. CPAN::Bundle objects for bundles, CPAN::Module objects for modules, and CPAN::Distribution objects for distributions. Note: it does not expand to CPAN::Author objects.

- Programming Examples

  This enables the programmer to do operations that combine functionalities that are available in the shell.

```
# install everything that is outdated on my disk:
perl -MCPAN -e 'CPAN::Shell->install(CPAN::Shell->r)'

# install my favorite programs if necessary:
for $mod (qw(Net::FTP Digest::SHA Data::Dumper)) {
    CPAN::Shell->install($mod);
}

# list all modules on my disk that have no VERSION number
for $mod (CPAN::Shell->expand("Module","/./")) {
    next unless $mod->inst_file;
    # MakeMaker convention for undefined $VERSION:
    next unless $mod->inst_version eq "undef";
    print "No VERSION in ", $mod->id, "\n";
}

# find out which distribution on CPAN contains a module:
print CPAN::Shell->expand("Module","Apache::Constants")->cpan_
 ↪file
```

  Or if you want to schedule a *cron* job to watch CPAN, you could list all modules that need updating. First a quick and dirty way:

```
perl -e 'use CPAN; CPAN::Shell->r;'
```

  If you don't want any output should all modules be up to date, parse the output of above command for the regular expression `/modules are up to date/` and decide to mail the output only if it doesn't match.

  If you prefer to do it more in a programmerish style in one single process, something like this may better suit you:

```
# list all modules on my disk that have newer versions on CPAN
for $mod (CPAN::Shell->expand("Module","/./")) {
  next unless $mod->inst_file;
  next if $mod->uptodate;
  printf "Module %s is installed as %s, could be updated to %s␣
 ↪from CPAN\n",
      $mod->id, $mod->inst_version, $mod->cpan_version;
}
```

  If that gives too much output every day, you may want to watch only for three modules. You can write

```
for $mod (CPAN::Shell->expand("Module","/Apache|LWP|CGI/")) {
```

as the first line instead. Or you can combine some of the above tricks:

```
# watch only for a new mod_perl module
$mod = CPAN::Shell->expand("Module","mod_perl");
exit if $mod->uptodate;
# new mod_perl arrived, let me know all update recommendations
CPAN::Shell->r;
```

Methods in the other Classes

- CPAN::Author::as_glimpse()

  Returns a one-line description of the author

- CPAN::Author::as_string()

  Returns a multi-line description of the author

- CPAN::Author::email()

  Returns the author's email address

- CPAN::Author::fullname()

  Returns the author's name

- CPAN::Author::name()

  An alias for fullname

- CPAN::Bundle::as_glimpse()

  Returns a one-line description of the bundle

- CPAN::Bundle::as_string()

  Returns a multi-line description of the bundle

- CPAN::Bundle::clean()

  Recursively runs the `clean` method on all items contained in the bundle.

- CPAN::Bundle::contains()

  Returns a list of objects' IDs contained in a bundle. The associated objects may be bundles, modules or distributions.

- CPAN::Bundle::force($method,@args)

  Forces CPAN to perform a task that it normally would have refused to do. Force takes as arguments a method name to be called and any number of additional arguments that should be passed to the called method. The internals of the object get the needed changes so that CPAN.pm does not refuse to take the action. The `force` is passed recursively to all contained objects. See also the section above on the `force` and the `fforce` pragma.

- CPAN::Bundle::get()

  Recursively runs the `get` method on all items contained in the bundle

- CPAN::Bundle::inst_file()

  Returns the highest installed version of the bundle in either @INC or `$CPAN::Config->{cpan_home}`. Note that this is different from CPAN::Module::inst_file.

- CPAN::Bundle::inst_version()

  Like CPAN::Bundle::inst_file, but returns the $VERSION

- CPAN::Bundle::uptodate()

  Returns 1 if the bundle itself and all its members are up-to-date.

- CPAN::Bundle::install()

  Recursively runs the `install` method on all items contained in the bundle

- CPAN::Bundle::make()

  Recursively runs the `make` method on all items contained in the bundle

- CPAN::Bundle::readme()

  Recursively runs the `readme` method on all items contained in the bundle

- CPAN::Bundle::test()

  Recursively runs the `test` method on all items contained in the bundle

- CPAN::Distribution::as_glimpse()

  Returns a one-line description of the distribution

- CPAN::Distribution::as_string()

  Returns a multi-line description of the distribution

- CPAN::Distribution::author

  Returns the CPAN::Author object of the maintainer who uploaded this distribution

- CPAN::Distribution::pretty_id()

  Returns a string of the form “AUTHORID/TARBALL”, where AUTHORID is the author’s PAUSE ID and TARBALL is the distribution filename.

- CPAN::Distribution::base_id()

  Returns the distribution filename without any archive suffix. E.g “Foo-Bar-0.01”

- CPAN::Distribution::clean()

  Changes to the directory where the distribution has been unpacked and runs `make clean` there.

- CPAN::Distribution::containsmods()

  Returns a list of IDs of modules contained in a distribution file. Works only for distributions listed in the 02packages.details.txt.gz file. This typically means that just most recent version of a distribution is covered.

- CPAN::Distribution::cvs_import()

  Changes to the directory where the distribution has been unpacked and runs something like

```
cvs -d $cvs_root import -m $cvs_log $cvs_dir $userid v$version
```

there.

- CPAN::Distribution::dir()

  Returns the directory into which this distribution has been unpacked.

- CPAN::Distribution::force($method,@args)

  Forces CPAN to perform a task that it normally would have refused to do. Force takes as arguments a method name to be called and any number of additional arguments that should be passed to the called method. The internals of the object get the needed changes so that CPAN.pm does not refuse to take the action. See also the section above on the `force` and the `fforce` pragma.

- CPAN::Distribution::get()

  Downloads the distribution from CPAN and unpacks it. Does nothing if the distribution has already been downloaded and unpacked within the current session.

- CPAN::Distribution::install()

  Changes to the directory where the distribution has been unpacked and runs the external command `make install` there. If `make` has not yet been run, it will be run first. A `make test` is issued in any case and if this fails, the install is cancelled. The cancellation can be avoided by letting `force` run the `install` for you.

  This install method only has the power to install the distribution if there are no dependencies in the way. To install an object along with all its dependencies, use CPAN::Shell->install.

  Note that install() gives no meaningful return value. See uptodate().

- CPAN::Distribution::isa_perl()

  Returns 1 if this distribution file seems to be a perl distribution. Normally this is derived from the file name only, but the index from CPAN can contain a hint to achieve a return value of true for other filenames too.

- CPAN::Distribution::look()

  Changes to the directory where the distribution has been unpacked and opens a subshell there. Exiting the subshell returns.

- CPAN::Distribution::make()

  First runs the `get` method to make sure the distribution is downloaded and unpacked. Changes to the directory where the distribution has been unpacked and runs the external commands `perl Makefile.PL` or `perl Build.PL` and `make` there.

- CPAN::Distribution::perldoc()

  Downloads the pod documentation of the file associated with a distribution (in HTML format) and runs it through the external command *lynx* specified in `$CPAN::Config->{lynx}`. If *lynx* isn't available, it converts it to plain text with the external command *html2text* and runs it through the pager specified in `$CPAN::Config->{pager}`.

- CPAN::Distribution::prefs()

  Returns the hash reference from the first matching YAML file that the user has deposited in the `prefs_dir/` directory. The first succeeding match wins. The files in the `prefs_dir/` are processed alphabetically, and the canonical distro name (e.g. AUTHOR/Foo-Bar-3.14.tar.gz) is matched against the regular expressions stored in the $root->{match}{distribution} attribute value. Additionally all module names contained in a distribution are matched against the regular expressions in the $root->{match}{module} attribute value. The two match values are ANDed together. Each of the two attributes are optional.

- CPAN::Distribution::prereq_pm()

  Returns the hash reference that has been announced by a distribution as the `requires` and `build_requires` elements. These can be declared either by the `META.yml` (if authoritative) or can be deposited after the run of `Build.PL` in the file `./_build/prereqs` or after the run of `Makefile.PL` written as the `PREREQ_PM` hash in a comment in the produced `Makefile`. *Note*: this method only works after an attempt has been made to `make` the distribution. Returns undef otherwise.

- CPAN::Distribution::readme()

  Downloads the README file associated with a distribution and runs it through the pager specified in `$CPAN::Config->{pager}`.

- CPAN::Distribution::reports()

  Downloads report data for this distribution from www.cpantesters.org and displays a subset of them.

- CPAN::Distribution::read_yaml()

  Returns the content of the META.yml of this distro as a hashref. Note: works only after an attempt has been made to `make` the distribution. Returns undef otherwise. Also returns undef if the content of META.yml is not authoritative. (The rules about what exactly makes the content authoritative are still in flux.)

- CPAN::Distribution::test()

  Changes to the directory where the distribution has been unpacked and runs `make test` there.

- CPAN::Distribution::uptodate()

  Returns 1 if all the modules contained in the distribution are up-to-date. Relies on containsmods.

- CPAN::Index::force_reload()

  Forces a reload of all indices.

- CPAN::Index::reload()

  Reloads all indices if they have not been read for more than `$CPAN::Config->{index_expire}` days.

- CPAN::InfoObj::dump()

  CPAN::Author, CPAN::Bundle, CPAN::Module, and CPAN::Distribution inherit this method. It prints the data structure associated with an object. Useful for debug-

ging. Note: the data structure is considered internal and thus subject to change without notice.

- CPAN::Module::as_glimpse()

  Returns a one-line description of the module in four columns: The first column contains the word `Module`, the second column consists of one character: an equals sign if this module is already installed and up-to-date, a less-than sign if this module is installed but can be upgraded, and a space if the module is not installed. The third column is the name of the module and the fourth column gives maintainer or distribution information.

- CPAN::Module::as_string()

  Returns a multi-line description of the module

- CPAN::Module::clean()

  Runs a clean on the distribution associated with this module.

- CPAN::Module::cpan_file()

  Returns the filename on CPAN that is associated with the module.

- CPAN::Module::cpan_version()

  Returns the latest version of this module available on CPAN.

- CPAN::Module::cvs_import()

  Runs a cvs_import on the distribution associated with this module.

- CPAN::Module::description()

  Returns a 44 character description of this module. Only available for modules listed in The Module List (CPAN/modules/00modlist.long.html or 00modlist.long.txt.gz)

- CPAN::Module::distribution()

  Returns the CPAN::Distribution object that contains the current version of this module.

- CPAN::Module::dslip_status()

  Returns a hash reference. The keys of the hash are the letters `D`, `S`, `L`, `I`, and , for development status, support level, language, interface and public licence respectively. The data for the DSLIP status are collected by pause.perl.org when authors register their namespaces. The values of the 5 hash elements are one-character words whose meaning is described in the table below. There are also 5 hash elements `DV`, `SV`, `LV`, `IV`, and that carry a more verbose value of the 5 status variables.

  Where the 'DSLIP' characters have the following meanings:

```
D - Development Stage  (Note: *NO IMPLIED TIMESCALES*):
  i   - Idea, listed to gain consensus or as a placeholder
  c   - under construction but pre-alpha (not yet released)
  a/b - Alpha/Beta testing
  R   - Released
  M   - Mature (no rigorous definition)
  S   - Standard, supplied with Perl 5
```

(continues on next page)

(continued from previous page)

```
S - Support Level:
  m    - Mailing-list
  d    - Developer
  u    - Usenet newsgroup comp.lang.perl.modules
  n    - None known, try comp.lang.perl.modules
  a    - abandoned; volunteers welcome to take over maintenance

L - Language Used:
  p    - Perl-only, no compiler needed, should be platform␣
↪independent
  c    - C and perl, a C compiler will be needed
  h    - Hybrid, written in perl with optional C code, no␣
↪compiler needed
  +    - C++ and perl, a C++ compiler will be needed
  o    - perl and another language other than C or C++

I - Interface Style
  f    - plain Functions, no references used
  h    - hybrid, object and function interfaces available
  n    - no interface at all (huh?)
  r    - some use of unblessed References or ties
  O    - Object oriented using blessed references and/or␣
↪inheritance

P - Public License
  p    - Standard-Perl: user may choose between GPL and Artistic
  g    - GPL: GNU General Public License
  l    - LGPL: "GNU Lesser General Public License" (previously␣
↪known as
         "GNU Library General Public License")
  b    - BSD: The BSD License
  a    - Artistic license alone
  2    - Artistic license 2.0 or later
  o    - open source: approved by www.opensource.org
  d    - allows distribution without restrictions
  r    - restricted distribution
  n    - no license at all
```

- CPAN::Module::force($method,@args)

  Forces CPAN to perform a task it would normally refuse to do. Force takes as arguments a method name to be invoked and any number of additional arguments to pass that method. The internals of the object get the needed changes so that CPAN.pm does not refuse to take the action. See also the section above on the `force` and the `fforce` pragma.

- CPAN::Module::get()

  Runs a get on the distribution associated with this module.

- CPAN::Module::inst_file()

Returns the filename of the module found in @INC. The first file found is reported, just as perl itself stops searching @INC once it finds a module.

- CPAN::Module::available_file()

  Returns the filename of the module found in PERL5LIB or @INC. The first file found is reported. The advantage of this method over `inst_file` is that modules that have been tested but not yet installed are included because PERL5LIB keeps track of tested modules.

- CPAN::Module::inst_version()

  Returns the version number of the installed module in readable format.

- CPAN::Module::available_version()

  Returns the version number of the available module in readable format.

- CPAN::Module::install()

  Runs an `install` on the distribution associated with this module.

- CPAN::Module::look()

  Changes to the directory where the distribution associated with this module has been unpacked and opens a subshell there. Exiting the subshell returns.

- CPAN::Module::make()

  Runs a `make` on the distribution associated with this module.

- CPAN::Module::manpage_headline()

  If module is installed, peeks into the module's manpage, reads the headline, and returns it. Moreover, if the module has been downloaded within this session, does the equivalent on the downloaded module even if it hasn't been installed yet.

- CPAN::Module::perldoc()

  Runs a `perldoc` on this module.

- CPAN::Module::readme()

  Runs a `readme` on the distribution associated with this module.

- CPAN::Module::reports()

  Calls the reports() method on the associated distribution object.

- CPAN::Module::test()

  Runs a `test` on the distribution associated with this module.

- CPAN::Module::uptodate()

  Returns 1 if the module is installed and up-to-date.

- CPAN::Module::userid()

  Returns the author's ID of the module.

Cache Manager

Currently the cache manager only keeps track of the build directory ($CPAN::Config->{build_dir}). It is a simple FIFO mechanism that deletes complete directories below `build_dir` as soon as the size of all directories there gets bigger than $CPAN::Config->{build_cache} (in MB). The contents of this cache may be used for later re-installations that you intend to do manually, but will never be trusted by CPAN itself. This is due to the fact that the user might use these directories for building modules on different architectures.

There is another directory ($CPAN::Config->{keep_source_where}) where the original distribution files are kept. This directory is not covered by the cache manager and must be controlled by the user. If you choose to have the same directory as build_dir and as keep_source_where directory, then your sources will be deleted with the same fifo mechanism.

Bundles

A bundle is just a perl module in the namespace Bundle:: that does not define any functions or methods. It usually only contains documentation.

It starts like a perl module with a package declaration and a $VERSION variable. After that the pod section looks like any other pod with the only difference being that *one special pod section* exists starting with (verbatim):

```
=head1 CONTENTS
```

In this pod section each line obeys the format

```
Module_Name [Version_String] [- optional text]
```

The only required part is the first field, the name of a module (e.g. Foo::Bar, i.e. *not* the name of the distribution file). The rest of the line is optional. The comment part is delimited by a dash just as in the man page header.

The distribution of a bundle should follow the same convention as other distributions.

Bundles are treated specially in the CPAN package. If you say 'install Bundle::Tkkit' (assuming such a bundle exists), CPAN will install all the modules in the CONTENTS section of the pod. You can install your own Bundles locally by placing a conformant Bundle file somewhere into your @INC path. The autobundle() command which is available in the shell interface does that for you by including all currently installed modules in a snapshot bundle file.

PREREQUISITES

The CPAN program is trying to depend on as little as possible so the user can use it in hostile environment. It works better the more goodies the environment provides. For example if you try in the CPAN shell

```
install Bundle::CPAN
```

or

```
install Bundle::CPANxxl
```

you will find the shell more convenient than the bare shell before.

If you have a local mirror of CPAN and can access all files with "file:" URLs, then you only need a perl later than perl5.003 to run this module. Otherwise Net::FTP is strongly recommended. LWP may be required for non-UNIX systems, or if your nearest CPAN site is associated with a URL that is not `ftp:`.

If you have neither Net::FTP nor LWP, there is a fallback mechanism implemented for an external ftp command or for an external lynx command.

UTILITIES

Finding packages and VERSION

This module presumes that all packages on CPAN

- declare their $VERSION variable in an easy to parse manner. This prerequisite can hardly be relaxed because it consumes far too much memory to load all packages into the running program just to determine the $VERSION variable. Currently all programs that are dealing with version use something like this

  ```
  perl -MExtUtils::MakeMaker -le \
      'print MM->parse_version(shift)' filename
  ```

  If you are author of a package and wonder if your $VERSION can be parsed, please try the above method.

- come as compressed or gzipped tarfiles or as zip files and contain a `Makefile.PL` or `Build.PL` (well, we try to handle a bit more, but with little enthusiasm).

Debugging

Debugging this module is more than a bit complex due to interference from the software producing the indices on CPAN, the mirroring process on CPAN, packaging, configuration, synchronicity, and even (gasp!) due to bugs within the CPAN.pm module itself.

For debugging the code of CPAN.pm itself in interactive mode, some debugging aid can be turned on for most packages within CPAN.pm with one of

- o debug package...

  sets debug mode for packages.

- o debug -package...

  unsets debug mode for packages.

- o debug all

  turns debugging on for all packages.

- o debug number

which sets the debugging packages directly. Note that `o debug 0` turns debugging off.

What seems a successful strategy is the combination of `reload cpan` and the debugging switches. Add a new debug statement while running in the shell and then issue a `reload cpan` and see the new debugging messages immediately without losing the current context.

`o debug` without an argument lists the valid package names and the current set of packages in debugging mode. `o debug` has built-in completion support.

For debugging of CPAN data there is the `dump` command which takes the same arguments as make/test/install and outputs each object's Data::Dumper dump. If an argument looks like a perl variable and contains one of `$`, `@` or `%`, it is eval()ed and fed to Data::Dumper directly.

Floppy, Zip, Offline Mode

CPAN.pm works nicely without network access, too. If you maintain machines that are not networked at all, you should consider working with `file:` URLs. You'll have to collect your modules somewhere first. So you might use CPAN.pm to put together all you need on a networked machine. Then copy the $CPAN::Config->{keep_source_where} (but not $CPAN::Config->{build_dir}) directory on a floppy. This floppy is kind of a personal CPAN. CPAN.pm on the non-networked machines works nicely with this floppy. See also below the paragraph about CD-ROM support.

Basic Utilities for Programmers

- has_inst($module)

  Returns true if the module is installed. Used to load all modules into the running CPAN.pm that are considered optional. The config variable `dontload_list` intercepts the `has_inst()` call such that an optional module is not loaded despite being available. For example, the following command will prevent `YAML.pm` from being loaded:

  ```
  cpan> o conf dontload_list push YAML
  ```

  See the source for details.

- use_inst($module)

  Similary to has_inst() tries to load optional library but also dies if library is not available

- has_usable($module)

  Returns true if the module is installed and in a usable state. Only useful for a handful of modules that are used internally. See the source for details.

- instance($module)

  The constructor for all the singletons used to represent modules, distributions, authors, and bundles. If the object already exists, this method returns the object; otherwise, it calls the constructor.

- frontend()

- frontend($new_frontend)

Getter/setter for frontend object. Method just allows to subclass CPAN.pm.

SECURITY

There's no strong security layer in CPAN.pm. CPAN.pm helps you to install foreign, unmasked, unsigned code on your machine. We compare to a checksum that comes from the net just as the distribution file itself. But we try to make it easy to add security on demand:

Cryptographically signed modules

Since release 1.77, CPAN.pm has been able to verify cryptographically signed module distributions using Module::Signature. The CPAN modules can be signed by their authors, thus giving more security. The simple unsigned MD5 checksums that were used before by CPAN protect mainly against accidental file corruption.

You will need to have Module::Signature installed, which in turn requires that you have at least one of Crypt::OpenPGP module or the command-line gpg tool installed.

You will also need to be able to connect over the Internet to the public key servers, like pgp.mit.edu, and their port 11731 (the HKP protocol).

The configuration parameter check_sigs is there to turn signature checking on or off.

EXPORT

Most functions in package CPAN are exported by default. The reason for this is that the primary use is intended for the cpan shell or for one-liners.

ENVIRONMENT

When the CPAN shell enters a subshell via the look command, it sets the environment CPAN_SHELL_LEVEL to 1, or increments that variable if it is already set.

When CPAN runs, it sets the environment variable PERL5_CPAN_IS_RUNNING to the ID of the running process. It also sets PERL5_CPANPLUS_IS_RUNNING to prevent runaway processes which could happen with older versions of Module::Install.

When running `perl Makefile.PL`, the environment variable `PERL5_CPAN_IS_EXECUTING` is set to the full path of the `Makefile.PL` that is being executed. This prevents runaway processes with newer versions of Module::Install.

When the config variable ftp_passive is set, all downloads will be run with the environment variable FTP_PASSIVE set to this value. This is in general a good idea as it influences both Net::FTP and LWP based connections. The same effect can be achieved by starting the cpan shell with this environment variable set. For Net::FTP alone, one can also always set passive mode by running libnetcfg.

POPULATE AN INSTALLATION WITH LOTS OF MODULES

Populating a freshly installed perl with one's favorite modules is pretty easy if you maintain a private bundle definition file. To get a useful blueprint of a bundle definition file, the command autobundle can be used on the CPAN shell command line. This command writes a bundle definition file for all modules installed for the current perl interpreter.

It's recommended to run this command once only, and from then on maintain the file manually under a private name, say Bundle/my_bundle.pm. With a clever bundle file you can then simply say

```
cpan> install Bundle::my_bundle
```

then answer a few questions and go out for coffee (possibly even in a different city).

Maintaining a bundle definition file means keeping track of two things: dependencies and interactivity. CPAN.pm sometimes fails on calculating dependencies because not all modules define all MakeMaker attributes correctly, so a bundle definition file should specify prerequisites as early as possible. On the other hand, it's annoying that so many distributions need some interactive configuring. So what you can try to accomplish in your private bundle file is to have the packages that need to be configured early in the file and the gentle ones later, so you can go out for coffee after a few minutes and leave CPAN.pm to churn away unattended.

WORKING WITH CPAN.pm BEHIND FIREWALLS

Thanks to Graham Barr for contributing the following paragraphs about the interaction between perl, and various firewall configurations. For further information on firewalls, it is recommended to consult the documentation that comes with the *ncftp* program. If you are unable to go through the firewall with a simple Perl setup, it is likely that you can configure *ncftp* so that it works through your firewall.

Three basic types of firewalls

Firewalls can be categorized into three basic types.

- http firewall

  This is when the firewall machine runs a web server, and to access the outside world, you must do so via that web server. If you set environment variables like http_proxy or ftp_proxy to values beginning with http://, or in your web browser you've proxy information set, then you know you are running behind an http firewall.

  To access servers outside these types of firewalls with perl (even for ftp), you need LWP or HTTP::Tiny.

- ftp firewall

  This where the firewall machine runs an ftp server. This kind of firewall will only let you access ftp servers outside the firewall. This is usually done by connecting to the firewall with ftp, then entering a username like "user@outside.host.com".

  To access servers outside these type of firewalls with perl, you need Net::FTP.

- One-way visibility

  One-way visibility means these firewalls try to make themselves invisible to users inside the firewall. An FTP data connection is normally created by sending your IP address to the remote server and then listening for the return connection. But the remote server will not be able to connect to you because of the firewall. For these types of firewall, FTP connections need to be done in a passive mode.

There are two that I can think off.

– SOCKS

If you are using a SOCKS firewall, you will need to compile perl and link it with the SOCKS library. This is what is normally called a 'socksified' perl. With this executable you will be able to connect to servers outside the firewall as if it were not there.

– IP Masquerade

This is when the firewall implemented in the kernel (via NAT, or networking address translation), it allows you to hide a complete network behind one IP address. With this firewall no special compiling is needed as you can access hosts directly.

For accessing ftp servers behind such firewalls you usually need to set the environment variable `FTP_PASSIVE` or the config variable ftp_passive to a true value.

Configuring lynx or ncftp for going through a firewall

If you can go through your firewall with e.g. lynx, presumably with a command such as

```
/usr/local/bin/lynx -pscott:tiger
```

then you would configure CPAN.pm with the command

```
o conf lynx "/usr/local/bin/lynx -pscott:tiger"
```

That's all. Similarly for ncftp or ftp, you would configure something like

```
o conf ncftp "/usr/bin/ncftp -f /home/scott/ncftplogin.cfg"
```

Your mileage may vary...

FAQ

- 1)

  I installed a new version of module X but CPAN keeps saying, I have the old version installed

  Probably you **do** have the old version installed. This can happen if a module installs itself into a different directory in the @INC path than it was previously installed. This is not really a CPAN.pm problem, you would have the same problem when installing the module manually. The easiest way to prevent this behaviour is to add the argument `UNINST=1` to the `make install` call, and that is why many people add this argument permanently by configuring

  ```
  o conf make_install_arg UNINST=1
  ```

- 2)

  So why is UNINST=1 not the default?

Because there are people who have their precise expectations about who may install where in the @INC path and who uses which @INC array. In fine tuned environments `UNINST=1` can cause damage.

- 3)

  I want to clean up my mess, and install a new perl along with all modules I have. How do I go about it?

  Run the autobundle command for your old perl and optionally rename the resulting bundle file (e.g. Bundle/mybundle.pm), install the new perl with the Configure option prefix, e.g.

```
./Configure -Dprefix=/usr/local/perl-5.6.78.9
```

  Install the bundle file you produced in the first step with something like

```
cpan> install Bundle::mybundle
```

  and you're done.

- 4)

  When I install bundles or multiple modules with one command there is too much output to keep track of.

  You may want to configure something like

```
o conf make_arg "| tee -ai /root/.cpan/logs/make.out"
o conf make_install_arg "| tee -ai /root/.cpan/logs/make_
↪install.out"
```

  so that STDOUT is captured in a file for later inspection.

- 5)

  I am not root, how can I install a module in a personal directory?

  As of CPAN 1.9463, if you do not have permission to write the default perl library directories, CPAN's configuration process will ask you whether you want to bootstrap <local::lib>, which makes keeping a personal perl library directory easy.

  Another thing you should bear in mind is that the UNINST parameter can be dangerous when you are installing into a private area because you might accidentally remove modules that other people depend on that are not using the private area.

- 6)

  How to get a package, unwrap it, and make a change before building it?

  Have a look at the `look` (!) command.

- 7)

  I installed a Bundle and had a couple of fails. When I retried, everything resolved nicely. Can this be fixed to work on first try?

  The reason for this is that CPAN does not know the dependencies of all modules when it starts out. To decide about the additional items to install, it just uses data found in the META.yml file or the generated Makefile. An undetected missing piece

breaks the process. But it may well be that your Bundle installs some prerequisite later than some depending item and thus your second try is able to resolve everything. Please note, CPAN.pm does not know the dependency tree in advance and cannot sort the queue of things to install in a topologically correct order. It resolves perfectly well **if** all modules declare the prerequisites correctly with the PREREQ_PM attribute to MakeMaker or the `requires` stanza of Module::Build. For bundles which fail and you need to install often, it is recommended to sort the Bundle definition file manually.

- 8)

  In our intranet, we have many modules for internal use. How can I integrate these modules with CPAN.pm but without uploading the modules to CPAN?

  Have a look at the CPAN::Site module.

- 9)

  When I run CPAN's shell, I get an error message about things in my `/etc/inputrc` (or `~/.inputrc`) file.

  These are readline issues and can only be fixed by studying readline configuration on your architecture and adjusting the referenced file accordingly. Please make a backup of the `/etc/inputrc` or `~/.inputrc` and edit them. Quite often harmless changes like uppercasing or lowercasing some arguments solves the problem.

- 10)

  Some authors have strange characters in their names.

  Internally CPAN.pm uses the UTF-8 charset. If your terminal is expecting ISO-8859-1 charset, a converter can be activated by setting term_is_latin to a true value in your config file. One way of doing so would be

  ```
  cpan> o conf term_is_latin 1
  ```

  If other charset support is needed, please file a bug report against CPAN.pm at rt.cpan.org and describe your needs. Maybe we can extend the support or maybe UTF-8 terminals become widely available.

  Note: this config variable is deprecated and will be removed in a future version of CPAN.pm. It will be replaced with the conventions around the family of $LANG and $LC_* environment variables.

- 11)

  When an install fails for some reason and then I correct the error condition and retry, CPAN.pm refuses to install the module, saying `Already tried without success`.

  You could use the force pragma like so

  ```
  force install Foo::Bar
  ```

  Or, to avoid a force install (which would install even if the tests fail), you can force only the test and then install:

```
force test Foo::Bar
install Foo::Bar
```

Or you can use

```
look Foo::Bar
```

and then `make install` directly in the subshell.

- 12)

  How do I install a "DEVELOPER RELEASE" of a module?

  By default, CPAN will install the latest non-developer release of a module. If you want to install a dev release, you have to specify the partial path starting with the author id to the tarball you wish to install, like so:

```
cpan> install KWILLIAMS/Module-Build-0.27_07.tar.gz
```

  Note that you can use the `ls` command to get this path listed.

- 13)

  How do I install a module and all its dependencies from the commandline, without being prompted for anything, despite my CPAN configuration (or lack thereof)?

  CPAN uses ExtUtils::MakeMaker's prompt() function to ask its questions, so if you set the PERL_MM_USE_DEFAULT environment variable, you shouldn't be asked any questions at all (assuming the modules you are installing are nice about obeying that variable as well):

```
% PERL_MM_USE_DEFAULT=1 perl -MCPAN -e 'install My::Module'
```

- 14)

  How do I create a Module::Build based Build.PL derived from an ExtUtils::MakeMaker focused Makefile.PL?

  http://search.cpan.org/dist/Module-Build-Convert/

- 15)

  I'm frequently irritated with the CPAN shell's inability to help me select a good mirror.

  CPAN can now help you select a "good" mirror, based on which ones have the lowest 'ping' round-trip times. From the shell, use the command 'o conf init urllist' and allow CPAN to automatically select mirrors for you.

  Beyond that help, the urllist config parameter is yours. You can add and remove sites at will. You should find out which sites have the best up-to-dateness, bandwidth, reliability, etc. and are topologically close to you. Some people prefer fast downloads, others up-to-dateness, others reliability. You decide which to try in which order.

  Henk P. Penning maintains a site that collects data about CPAN sites:

```
http://mirrors.cpan.org/
```

Also, feel free to play with experimental features. Run

```
o conf init randomize_urllist ftpstats_period ftpstats_size
```

and choose your favorite parameters. After a few downloads running the `hosts` command will probably assist you in choosing the best mirror sites.

- 16)

  Why do I get asked the same questions every time I start the shell?

  You can make your configuration changes permanent by calling the command `o conf commit`. Alternatively set the `auto_commit` variable to true by running `o conf init auto_commit` and answering the following question with yes.

- 17)

  Older versions of CPAN.pm had the original root directory of all tarballs in the build directory. Now there are always random characters appended to these directory names. Why was this done?

  The random characters are provided by File::Temp and ensure that each module's individual build directory is unique. This makes running CPAN.pm in concurrent processes simultaneously safe.

- 18)

  Speaking of the build directory. Do I have to clean it up myself?

  You have the choice to set the config variable `scan_cache` to `never`. Then you must clean it up yourself. The other possible values, `atstart` and `atexit` clean up the build directory when you start (or more precisely, after the first extraction into the build directory) or exit the CPAN shell, respectively. If you never start up the CPAN shell, you probably also have to clean up the build directory yourself.

- 19)

  How can I switch to sudo instead of local::lib?

  The following 5 environment veriables need to be reset to the previous values: PATH, PERL5LIB, PERL_LOCAL_LIB_ROOT, PERL_MB_OPT, PERL_MM_OPT; and these two CPAN.pm config variables must be reconfigured: make_install_make_command and mbuild_install_build_command. The five env variables have probably been overwritten in your $HOME/.bashrc or some equivalent. You either find them there and delete their traces and logout/login or you override them temporarily, depending on your exact desire. The two cpanpm config variables can be set with:

  ```
  o conf init /install_.*_command/
  ```

  probably followed by

  ```
  o conf commit
  ```

- 20)

How do recommends_policy and suggests_policy work, exactly?

The terms `recommends` and `suggests` have been standardized in https://metacpan.org/pod/CPAN::Meta::Spec

In CPAN.pm, if you set `recommands_policy` to a true value, that means: if you then install a distribution `Foo` that *recommends* a module `Bar`, both `Foo` and `Bar` will be tested and potentially installed.

Similarly, if you set `suggests_policy` to a true value, it means: if you install a distribution `Foo` that *suggests* a module `Bar`, both `Foo` and `Bar` will be tested and potentially installed.

In either case, when `Foo` passes its tests and `Bar` does not pass its tests, `Foo` will be installed nontheless. But if `Foo` does not pass its tests, neither will be installed.

This also works recursively for all recommends and suggests of the module `Bar`.

This has also been illustrated by a cpan tester, who wrote:

I just tested Starlink-AST-3.03 which recommends Tk::Zinc; Tk-Zinc-3.306 fails with http://www.cpantesters.org/cpan/report/a2de7c38-810d-11ee-9ad4-e2167316189a ; nonetheless Starlink-AST-3.03 succeeds with http://www.cpantesters.org/cpan/report/9352e754-810d-11ee-90e9-46117316189a

COMPATIBILITY

OLD PERL VERSIONS

CPAN.pm is regularly tested to run under 5.005 and assorted newer versions. It is getting more and more difficult to get the minimal prerequisites working on older perls. It is close to impossible to get the whole Bundle::CPAN working there. If you're in the position to have only these old versions, be advised that CPAN is designed to work fine without the Bundle::CPAN installed.

To get things going, note that GBARR/Scalar-List-Utils-1.18.tar.gz is compatible with ancient perls and that File::Temp is listed as a prerequisite but CPAN has reasonable workarounds if it is missing.

CPANPLUS

This module and its competitor, the CPANPLUS module, are both much cooler than the other. CPAN.pm is older. CPANPLUS was designed to be more modular, but it was never intended to be compatible with CPAN.pm.

CPANMINUS

In the year 2010 App::cpanminus was launched as a new approach to a cpan shell with a considerably smaller footprint. Very cool stuff.

SECURITY ADVICE

This software enables you to upgrade software on your computer and so is inherently dangerous because the newly installed software may contain bugs and may alter the way your computer works or even make it unusable. Please consider backing up your data before every upgrade.

BUGS

Please report bugs via http://rt.cpan.org/

Before submitting a bug, please make sure that the traditional method of building a Perl module package from a shell by following the installation instructions of that package still works in your environment.

AUTHOR

Andreas Koenig <andk@cpan.org>

LICENSE

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See http://www.perl.com/perl/misc/Artistic.html

TRANSLATIONS

Kawai,Takanori provides a Japanese translation of a very old version of this manpage at http://homepage3.nifty.com/hippo2000/perltips/CPAN.htm

SEE ALSO

Many people enter the CPAN shell by running the cpan utility program which is installed in the same directory as perl itself. So if you have this directory in your PATH variable (or some equivalent in your operating system) then typing `cpan` in a console window will work for you as well. Above that the utility provides several commandline shortcuts.

melezhik (Alexey) sent me a link where he published a chef recipe to work with CPAN.pm: http://community.opscode.com/cookbooks/cpan.

Modules

- `CPAN::Debug` - internal debugging for CPAN.pm
- `CPAN::Distroprefs` - read and match distroprefs
- `CPAN::FirstTime` - Utility for CPAN::Config file Initialization
- `CPAN::HandleConfig` - internal configuration handling for CPAN.pm
- `CPAN::Kwalify` - Interface between CPAN.pm and Kwalify.pm
- `CPAN::Meta` - the distribution metadata for a CPAN dist
- `CPAN::Mirrors` - Get CPAN mirror information and select a fast one

- `CPAN::Nox` - Interactive mode:
- `CPAN::Plugin` - Base class for CPAN shell extensions
- `CPAN::Queue` - internal queue support for CPAN.pm
- `CPAN::Tarzip` - internal handling of tar archives for CPAN.pm
- `CPAN::Version` - utility functions to compare CPAN versions

CPAN::Debug

internal debugging for CPAN.pm

LICENSE

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

CPAN::Distroprefs

read and match distroprefs

SYNOPSIS

```
use CPAN::Distroprefs;

my %info = (... distribution/environment info ...);

my $finder = CPAN::Distroprefs->find($prefs_dir, \%ext_map);

while (my $result = $finder->next) {

    die $result->as_string if $result->is_fatal;

    warn($result->as_string), next if $result->is_warning;

    for my $pref (@{ $result->prefs }) {
        if ($pref->matches(\%info)) {
            return $pref;
        }
    }
}
```

DESCRIPTION

This module encapsulates reading *Distroprefs* and matching them against CPAN distributions.

INTERFACE

```
my $finder = CPAN::Distroprefs->find($dir, \%ext_map);

while (my $result = $finder->next) { ... }
```

Build an iterator which finds distroprefs files in the tree below the given directory. Within the tree directories matching `m/^[._]/` are pruned.

`%ext_map` is a hashref whose keys are file extensions and whose values are modules used to load matching files:

```
{
    'yml' => 'YAML::Syck',
    'dd'  => 'Data::Dumper',
    ...
}
```

Each time `$finder->next` is called, the iterator returns one of two possible values:

- a CPAN::Distroprefs::Result object
- `undef`, indicating that no prefs files remain to be found

RESULTS

`find()` returns CPAN::Distroprefs::Result objects to indicate success or failure when reading a prefs file.

Common

All results share some common attributes:

type

`success`, `warning`, or `fatal`

file

the file from which these prefs were read, or to which this error refers (relative filename)

ext

the file's extension, which determines how to load it

dir

the directory the file was read from

abs

the absolute path to the file

Errors

Error results (warning and fatal) contain:

msg

the error message (usually either `$!` or a YAML error)

Successes

Success results contain:

prefs

an arrayref of CPAN::Distroprefs::Pref objects

PREFS

CPAN::Distroprefs::Pref objects represent individual distroprefs documents. They are constructed automatically as part of `success` results from `find()`.

data

the pref information as a hashref, suitable for e.g. passing to Kwalify

match_attributes

returns a list of the valid match attributes (see the Distroprefs section in *CPAN*)

currently: `env perl perlconfig distribution module`

has_any_match

true if this pref has a 'match' attribute at all

has_valid_subkeys

true if this pref has a 'match' attribute and at least one valid match attribute

matches

```
if ($pref->matches(\%arg)) { ... }
```

true if this pref matches the passed-in hashref, which must have a value for each of the `match_attributes` (above)

LICENSE

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

CPAN::FirstTime

Utility for CPAN::Config file Initialization

SYNOPSIS

CPAN::FirstTime::init()

DESCRIPTION

The init routine asks a few questions and writes a CPAN/Config.pm or CPAN/MyConfig.pm file (depending on what it is currently using).

In the following all questions and explanations regarding config variables are collected.

- allow_installing_module_downgrades

  The CPAN shell can watch the `blib/` directories that are built up before running `make test` to determine whether the current distribution will end up with modules being overwritten with decreasing module version numbers. It can then let the build of this distro fail when it discovers a downgrade.

  Do you want to allow installing distros with decreasing module versions compared to what you have installed (yes, no, ask/yes, ask/no)?

- allow_installing_outdated_dists

  The CPAN shell can watch the `blib/` directories that are built up before running `make test` to determine whether the current distribution contains modules that are indexed with a distro with a higher distro-version number than the current one. It can then let the build of this distro fail when it would not represent the most up-to-date version of the distro.

  Note: choosing anything but 'yes' for this option will need CPAN::DistnameInfo being installed for taking effect.

  Do you want to allow installing distros that are not indexed as the highest distro-version for all contained modules (yes, no, ask/yes, ask/no)?

- auto_commit

  Normally CPAN.pm keeps config variables in memory and changes need to be saved in a separate 'o conf commit' command to make them permanent between sessions. If you set the 'auto_commit' option to true, changes to a config variable are always automatically committed to disk.

  Always commit changes to config variables to disk?

- build_cache

  CPAN.pm can limit the size of the disk area for keeping the build directories with all the intermediate files.

Cache size for build directory (in MB)?

- build_dir

  Directory where the build process takes place?

- build_dir_reuse

  Until version 1.88 CPAN.pm never trusted the contents of the build_dir directory between sessions. Since 1.88_58 CPAN.pm has a YAML-based mechanism that makes it possible to share the contents of the build_dir/ directory between different sessions with the same version of perl. People who prefer to test things several days before installing will like this feature because it saves a lot of time.

  If you say yes to the following question, CPAN will try to store enough information about the build process so that it can pick up in future sessions at the same state of affairs as it left a previous session.

  Store and re-use state information about distributions between CPAN.pm sessions?

- build_requires_install_policy

  When a module declares another one as a 'build_requires' prerequisite this means that the other module is only needed for building or testing the module but need not be installed permanently. In this case you may wish to install that other module nonetheless or just keep it in the 'build_dir' directory to have it available only temporarily. Installing saves time on future installations but makes the perl installation bigger.

  You can choose if you want to always install (yes), never install (no) or be always asked. In the latter case you can set the default answer for the question to yes (ask/yes) or no (ask/no).

  Policy on installing 'build_requires' modules (yes, no, ask/yes, ask/no)?

- cache_metadata

  To considerably speed up the initial CPAN shell startup, it is possible to use Storable to create a cache of metadata. If Storable is not available, the normal index mechanism will be used.

  Note: this mechanism is not used when use_sqlite is on and SQLite is running.

  Cache metadata (yes/no)?

- check_sigs

  CPAN packages can be digitally signed by authors and thus verified with the security provided by strong cryptography. The exact mechanism is defined in the Module::Signature module. While this is generally considered a good thing, it is not always convenient to the end user to install modules that are signed incorrectly or where the key of the author is not available or where some prerequisite for Module::Signature has a bug and so on.

  With the check_sigs parameter you can turn signature checking on and off. The default is off for now because the whole tool chain for the functionality is not yet considered mature by some. The author of CPAN.pm would recommend setting it to true most of the time and turning it off only if it turns out to be annoying.

Note that if you do not have Module::Signature installed, no signature checks will be performed at all.

Always try to check and verify signatures if a SIGNATURE file is in the package and Module::Signature is installed (yes/no)?

- cleanup_after_install

  Users who install modules and do not intend to look back, can free occupied disk space quickly by letting CPAN.pm cleanup each build directory immediately after a successful install.

  Remove build directory after a successful install? (yes/no)?

- colorize_output

  When you have Term::ANSIColor installed, you can turn on colorized output to have some visual differences between normal CPAN.pm output, warnings, debugging output, and the output of the modules being installed. Set your favorite colors after some experimenting with the Term::ANSIColor module.

  Please note that on Windows platforms colorized output also requires the Win32::Console::ANSI module.

  Do you want to turn on colored output?

- colorize_print

  Color for normal output?

- colorize_warn

  Color for warnings?

- colorize_debug

  Color for debugging messages?

- commandnumber_in_prompt

  The prompt of the cpan shell can contain the current command number for easier tracking of the session or be a plain string.

  Do you want the command number in the prompt (yes/no)?

- connect_to_internet_ok

  If you have never defined your own `urllist` in your configuration then `CPAN.pm` will be hesitant to use the built in default sites for downloading. It will ask you once per session if a connection to the internet is OK and only if you say yes, it will try to connect. But to avoid this question, you can choose your favorite download sites once and get away with it. Or, if you have no favorite download sites answer yes to the following question.

  If no urllist has been chosen yet, would you prefer CPAN.pm to connect to the built-in default sites without asking? (yes/no)?

- ftp_passive

  Shall we always set the FTP_PASSIVE environment variable when dealing with ftp download (yes/no)?

- ftpstats_period

  Statistics about downloads are truncated by size and period simultaneously.

  How many days shall we keep statistics about downloads?

- ftpstats_size

  Statistics about downloads are truncated by size and period simultaneously. Setting this to zero or negative disables download statistics.

  How many items shall we keep in the statistics about downloads?

- getcwd

  CPAN.pm changes the current working directory often and needs to determine its own current working directory. Per default it uses Cwd::cwd but if this doesn't work on your system for some reason, alternatives can be configured according to the following table:

```
cwd          Cwd::cwd
getcwd       Cwd::getcwd
fastcwd      Cwd::fastcwd
getdcwd      Cwd::getdcwd
backtickcwd external command cwd
```

  Preferred method for determining the current working directory?

- halt_on_failure

  Normally, CPAN.pm continues processing the full list of targets and dependencies, even if one of them fails. However, you can specify that CPAN should halt after the first failure. (Note that optional recommended or suggested modules that fail will not cause a halt.)

  Do you want to halt on failure (yes/no)?

- histfile

  If you have one of the readline packages (Term::ReadLine::Perl, Term::ReadLine::Gnu, possibly others) installed, the interactive CPAN shell will have history support. The next two questions deal with the filename of the history file and with its size. If you do not want to set this variable, please hit SPACE ENTER to the following question.

  File to save your history?

- histsize

  Number of lines to save?

- inactivity_timeout

  Sometimes you may wish to leave the processes run by CPAN alone without caring about them. Because the Makefile.PL or the Build.PL sometimes contains question you're expected to answer, you can set a timer that will kill a 'perl Makefile.PL' process after the specified time in seconds.

  If you set this value to 0, these processes will wait forever. This is the default and recommended setting.

Timeout for inactivity during {Makefile,Build}.PL?

- index_expire

  The CPAN indexes are usually rebuilt once or twice per hour, but the typical CPAN mirror mirrors only once or twice per day. Depending on the quality of your mirror and your desire to be on the bleeding edge, you may want to set the following value to more or less than one day (which is the default). It determines after how many days CPAN.pm downloads new indexes.

  Let the index expire after how many days?

- inhibit_startup_message

  When the CPAN shell is started it normally displays a greeting message that contains the running version and the status of readline support.

  Do you want to turn this message off?

- keep_source_where

  Unless you are accessing the CPAN on your filesystem via a file: URL, CPAN.pm needs to keep the source files it downloads somewhere. Please supply a directory where the downloaded files are to be kept.

  Download target directory?

- load_module_verbosity

  When CPAN.pm loads a module it needs for some optional feature, it usually reports about module name and version. Choose 'v' to get this message, 'none' to suppress it.

  Verbosity level for loading modules (none or v)?

- makepl_arg

  Every Makefile.PL is run by perl in a separate process. Likewise we run 'make' and 'make install' in separate processes. If you have any parameters (e.g. PREFIX, UNINST or the like) you want to pass to the calls, please specify them here.

  If you don't understand this question, just press ENTER.

  Typical frequently used settings:

```
PREFIX=~/perl    # non-root users (please see manual for more␣
↪hints)
```

  Parameters for the 'perl Makefile.PL' command?

- make_arg

  Parameters for the 'make' command? Typical frequently used setting:

```
-j3              # dual processor system (on GNU make)
```

  Your choice:

- make_install_arg

  Parameters for the 'make install' command? Typical frequently used setting:

```
UNINST=1          # to always uninstall potentially conflicting␣
↪files
                  # (but do NOT use with local::lib or INSTALL_
↪BASE)
```

Your choice:

- make_install_make_command

  Do you want to use a different make command for 'make install'? Cautious people will probably prefer:

```
   su root -c make
or
   sudo make
or
   /path1/to/sudo -u admin_account /path2/to/make
```

  or some such. Your choice:

- mbuildpl_arg

  A Build.PL is run by perl in a separate process. Likewise we run './Build' and './Build install' in separate processes. If you have any parameters you want to pass to the calls, please specify them here.

  Typical frequently used settings:

```
--install_base /home/xxx             # different installation␣
↪directory
```

  Parameters for the 'perl Build.PL' command?

- mbuild_arg

  Parameters for the './Build' command? Setting might be:

```
--extra_linker_flags -L/usr/foo/lib  # non-standard library␣
↪location
```

  Your choice:

- mbuild_install_arg

  Parameters for the './Build install' command? Typical frequently used setting:

```
--uninst 1        # uninstall conflicting files
                  # (but do NOT use with local::lib or INSTALL_
↪BASE)
```

  Your choice:

- mbuild_install_build_command

  Do you want to use a different command for './Build install'? Sudo users will probably prefer:

```
   su root -c ./Build
or
   sudo ./Build
or
   /path1/to/sudo -u admin_account ./Build
```

or some such. Your choice:

- pager

  What is your favorite pager program?

- prefer_installer

  When you have Module::Build installed and a module comes with both a Makefile.PL and a Build.PL, which shall have precedence?

  The main two standard installer modules are the old and well established ExtUtils::MakeMaker (for short: EUMM) which uses the Makefile.PL. And the next generation installer Module::Build (MB) which works with the Build.PL (and often comes with a Makefile.PL too). If a module comes only with one of the two we will use that one but if both are supplied then a decision must be made between EUMM and MB. See also http://rt.cpan.org/Ticket/Display.html?id=29235 for a discussion about the right default.

  Or, as a third option you can choose RAND which will make a random decision (something regular CPAN testers will enjoy).

  In case you can choose between running a Makefile.PL or a Build.PL, which installer would you prefer (EUMM or MB or RAND)?

- prefs_dir

  CPAN.pm can store customized build environments based on regular expressions for distribution names. These are YAML files where the default options for CPAN.pm and the environment can be overridden and dialog sequences can be stored that can later be executed by an Expect.pm object. The CPAN.pm distribution comes with some prefab YAML files that cover sample distributions that can be used as blueprints to store your own prefs. Please check out the distroprefs/ directory of the CPAN.pm distribution to get a quick start into the prefs system.

  Directory where to store default options/environment/dialogs for building modules that need some customization?

- prerequisites_policy

  The CPAN module can detect when a module which you are trying to build depends on prerequisites. If this happens, it can build the prerequisites for you automatically ('follow'), ask you for confirmation ('ask'), or just ignore them ('ignore'). Choosing 'follow' also sets PERL_AUTOINSTALL and PERL_EXTUTILS_AUTOINSTALL for "–defaultdeps" if not already set.

  Please set your policy to one of the three values.

  Policy on building prerequisites (follow, ask or ignore)?

- pushy_https

Boolean. Defaults to true. If this option is true, the cpan shell will use https://cpan.org/ to download stuff from the CPAN. It will fall back to http://cpan.org/ if it can't handle https for some reason (missing modules, missing programs). Whenever it falls back to the http protocol, it will issue a warning.

If this option is true, the option `urllist` will be ignored. Consequently, if you want to work with local mirrors via your own configured list of URLs, you will have to choose no below.

Do you want to turn the pushy_https behaviour on?

- randomize_urllist

  CPAN.pm can introduce some randomness when using hosts for download that are configured in the urllist parameter. Enter a numeric value between 0 and 1 to indicate how often you want to let CPAN.pm try a random host from the urllist. A value of one specifies to always use a random host as the first try. A value of zero means no randomness at all. Anything in between specifies how often, on average, a random host should be tried first.

  Randomize parameter

- recommends_policy

  (Experimental feature!) Some CPAN modules recommend additional, optional dependencies. These should generally be installed except in resource constrained environments. When this policy is true, recommended modules will be included with required modules.

  Include recommended modules?

- scan_cache

  By default, each time the CPAN module is started, cache scanning is performed to keep the cache size in sync ('atstart'). Alternatively, scanning and cleanup can happen when CPAN exits ('atexit'). To prevent any cache cleanup, answer 'never'.

  Perform cache scanning ('atstart', 'atexit' or 'never')?

- shell

  What is your favorite shell?

- show_unparsable_versions

  During the 'r' command CPAN.pm finds modules without version number. When the command finishes, it prints a report about this. If you want this report to be very verbose, say yes to the following variable.

  Show all individual modules that have no $VERSION?

- show_upload_date

  The 'd' and the 'm' command normally only show you information they have in their in-memory database and thus will never connect to the internet. If you set the 'show_upload_date' variable to true, 'm' and 'd' will additionally show you the upload date of the module or distribution. Per default this feature is off because it may require a net connection to get at the upload date.

  Always try to show upload date with 'd' and 'm' command (yes/no)?

- show_zero_versions

  During the 'r' command CPAN.pm finds modules with a version number of zero. When the command finishes, it prints a report about this. If you want this report to be very verbose, say yes to the following variable.

  Show all individual modules that have a $VERSION of zero?

- suggests_policy

  (Experimental feature!) Some CPAN modules suggest additional, optional dependencies. These 'suggest' dependencies provide enhanced operation. When this policy is true, suggested modules will be included with required modules.

  Include suggested modules?

- tar_verbosity

  When CPAN.pm uses the tar command, which switch for the verbosity shall be used? Choose 'none' for quiet operation, 'v' for file name listing, 'vv' for full listing.

  Tar command verbosity level (none or v or vv)?

- term_is_latin

  The next option deals with the charset (a.k.a. character set) your terminal supports. In general, CPAN is English speaking territory, so the charset does not matter much but some CPAN have names that are outside the ASCII range. If your terminal supports UTF-8, you should say no to the next question. If it expects ISO-8859-1 (also known as LATIN1) then you should say yes. If it supports neither, your answer does not matter because you will not be able to read the names of some authors anyway. If you answer no, names will be output in UTF-8.

  Your terminal expects ISO-8859-1 (yes/no)?

- term_ornaments

  When using Term::ReadLine, you can turn ornaments on so that your input stands out against the output from CPAN.pm.

  Do you want to turn ornaments on?

- test_report

  The goal of the CPAN Testers project (http://testers.cpan.org/) is to test as many CPAN packages as possible on as many platforms as possible. This provides valuable feedback to module authors and potential users to identify bugs or platform compatibility issues and improves the overall quality and value of CPAN.

  One way you can contribute is to send test results for each module that you install. If you install the CPAN::Reporter module, you have the option to automatically generate and deliver test reports to CPAN Testers whenever you run tests on a CPAN package.

  See the CPAN::Reporter documentation for additional details and configuration settings. If your firewall blocks outgoing traffic, you may need to configure CPAN::Reporter before sending reports.

  Generate test reports if CPAN::Reporter is installed (yes/no)?

- perl5lib_verbosity

  When CPAN.pm extends @INC via PERL5LIB, it prints a list of directories added (or a summary of how many directories are added). Choose 'v' to get this message, 'none' to suppress it.

  Verbosity level for PERL5LIB changes (none or v)?

- prefer_external_tar

  Per default all untar operations are done with the perl module Archive::Tar; by setting this variable to true the external tar command is used if available; on Unix this is usually preferred because they have a reliable and fast gnutar implementation.

  Use the external tar program instead of Archive::Tar?

- trust_test_report_history

  When a distribution has already been tested by CPAN::Reporter on this machine, CPAN can skip the test phase and just rely on the test report history instead.

  Note that this will not apply to distributions that failed tests because of missing dependencies. Also, tests can be run regardless of the history using "force".

  Do you want to rely on the test report history (yes/no)?

- urllist_ping_external

  When automatic selection of the nearest cpan mirrors is performed, turn on the use of the external ping via Net::Ping::External. This is recommended in the case the local network has a transparent proxy.

  Do you want to use the external ping command when autoselecting mirrors?

- urllist_ping_verbose

  When automatic selection of the nearest cpan mirrors is performed, this option can be used to turn on verbosity during the selection process.

  Do you want to see verbosity turned on when autoselecting mirrors?

- use_prompt_default

  When this is true, CPAN will set PERL_MM_USE_DEFAULT to a true value. This causes ExtUtils::MakeMaker (and compatible) prompts to use default values instead of stopping to prompt you to answer questions. It also sets NONINTERACTIVE_TESTING to a true value to signal more generally that distributions should not try to interact with you.

  Do you want to use prompt defaults (yes/no)?

- use_sqlite

  CPAN::SQLite is a layer between the index files that are downloaded from the CPAN and CPAN.pm that speeds up metadata queries and reduces memory consumption of CPAN.pm considerably.

  Use CPAN::SQLite if available? (yes/no)?

- version_timeout

  This timeout prevents CPAN from hanging when trying to parse a pathologically coded $VERSION from a module.

The default is 15 seconds. If you set this value to 0, no timeout will occur, but this is not recommended.

Timeout for parsing module versions?

- yaml_load_code

  Both YAML.pm and YAML::Syck are capable of deserialising code. As this requires a string eval, which might be a security risk, you can use this option to enable or disable the deserialisation of code via CPAN::DeferredCode. (Note: This does not work under perl 5.6)

  Do you want to enable code deserialisation (yes/no)?

- yaml_module

  At the time of this writing (2009-03) there are three YAML implementations working: YAML, YAML::Syck, and YAML::XS. The latter two are faster but need a C compiler installed on your system. There may be more alternative YAML conforming modules. When I tried two other players, YAML::Tiny and YAML::Perl, they seemed not powerful enough to work with CPAN.pm. This may have changed in the meantime.

  Which YAML implementation would you prefer?

LICENSE

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

CPAN::HandleConfig

internal configuration handling for CPAN.pm

CLASS->safe_quote ITEM

Quotes an item to become safe against spaces in shell interpolation. An item is enclosed in double quotes if:

```
- the item contains spaces in the middle
- the item does not start with a quote
```

This happens to avoid shell interpolation problems when whitespace is present in directory names.

This method uses `commands_quote` to determine the correct quote. If `commands_quote` is a space, no quoting will take place.

if it starts and ends with the same quote character: leave it as it is

if it contains no whitespace: leave it as it is

if it contains whitespace, then

if it contains quotes: better leave it as it is

else: quote it with the correct quote type for the box we're on

LICENSE

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

CPAN::Kwalify

Interface between CPAN.pm and Kwalify.pm

SYNOPSIS

```
use CPAN::Kwalify;
validate($schema_name, $data, $file, $doc);
```

DESCRIPTION

- _validate($schema_name, $data, $file, $doc)

  $schema_name is the name of a supported schema. Currently only `distroprefs` is supported. $data is the data to be validated. $file is the absolute path to the file the data are coming from. $doc is the index of the document within $doc that is to be validated. The last two arguments are only there for better error reporting.

  Relies on being called from within CPAN.pm.

  Dies if something fails. Does not return anything useful.

- yaml($schema_name)

  Returns the YAML text of that schema. Dies if something fails.

AUTHOR

Andreas Koenig <andk@cpan.org>

LICENSE

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See http://www.perl.com/perl/misc/Artistic.html

CPAN::Meta

the distribution metadata for a CPAN dist

VERSION

version 2.150013

SYNOPSIS

```
use v5.10;
use strict;
use warnings;
use CPAN::Meta;
use Module::Load;

my $meta = CPAN::Meta->load_file('META.json');

printf "testing requirements for %s version %s\n",
$meta->name,
$meta->version;

my $prereqs = $meta->effective_prereqs;

for my $phase ( qw/configure runtime build test/ ) {
    say "Requirements for $phase:";
    my $reqs = $prereqs->requirements_for($phase, "requires");
    for my $module ( sort $reqs->required_modules ) {
        my $status;
        if ( eval { load $module unless $module eq 'perl'; 1 } ) {
            my $version = $module eq 'perl' ? $] : $module->VERSION;
            $status = $reqs->accepts_module($module, $version)
                    ? "$version ok" : "$version not ok";
        } else {
            $status = "missing"
        };
        say "  $module ($status)";
    }
}
```

DESCRIPTION

Software distributions released to the CPAN include a `META.json` or, for older distributions, `META.yml`, which describes the distribution, its contents, and the requirements for building and installing the distribution. The data structure stored in the `META.json` file is described in *CPAN::Meta::Spec*.

CPAN::Meta provides a simple class to represent this distribution metadata (or *distmeta*), along with some helpful methods for interrogating that data.

The documentation below is only for the methods of the CPAN::Meta object. For information on the meaning of individual fields, consult the spec.

METHODS

new

```
my $meta = CPAN::Meta->new($distmeta_struct, \%options);
```

Returns a valid CPAN::Meta object or dies if the supplied metadata hash reference fails to validate. Older-format metadata will be up-converted to version 2 if they validate against the original stated specification.

It takes an optional hashref of options. Valid options include:

- lazy_validation – if true, new will attempt to convert the given metadata to version 2 before attempting to validate it. This means than any fixable errors will be handled by CPAN::Meta::Converter before validation. (Note that this might result in invalid optional data being silently dropped.) The default is false.

create

```
my $meta = CPAN::Meta->create($distmeta_struct, \%options);
```

This is same as `new()`, except that `generated_by` and `meta-spec` fields will be generated if not provided. This means the metadata structure is assumed to otherwise follow the latest *CPAN::Meta::Spec*.

load_file

```
my $meta = CPAN::Meta->load_file($distmeta_file, \%options);
```

Given a pathname to a file containing metadata, this deserializes the file according to its file suffix and constructs a new `CPAN::Meta` object, just like `new()`. It will die if the deserialized version fails to validate against its stated specification version.

It takes the same options as `new()` but `lazy_validation` defaults to true.

load_yaml_string

```
my $meta = CPAN::Meta->load_yaml_string($yaml, \%options);
```

This method returns a new CPAN::Meta object using the first document in the given YAML string. In other respects it is identical to `load_file()`.

load_json_string

```
my $meta = CPAN::Meta->load_json_string($json, \%options);
```

This method returns a new CPAN::Meta object using the structure represented by the given JSON string. In other respects it is identical to `load_file()`.

load_string

```
my $meta = CPAN::Meta->load_string($string, \%options);
```

If you don't know if a string contains YAML or JSON, this method will use *Parse::CPAN::Meta* to guess. In other respects it is identical to `load_file()`.

save

```
$meta->save($distmeta_file, \%options);
```

Serializes the object as JSON and writes it to the given file. The only valid option is `version`, which defaults to '2'. On Perl 5.8.1 or later, the file is saved with UTF-8 encoding.

For `version` 2 (or higher), the filename should end in '.json'. *JSON::PP* is the default JSON backend. Using another JSON backend requires *JSON* 2.5 or later and you must set the `$ENV{PERL_JSON_BACKEND}` to a supported alternate backend like *JSON::XS*.

For `version` less than 2, the filename should end in '.yml'. *CPAN::Meta::Converter* is used to generate an older metadata structure, which is serialized to YAML. CPAN::Meta::YAML is the default YAML backend. You may set the `$ENV{PERL_YAML_BACKEND}` to a supported alternative backend, though this is not recommended due to subtle incompatibilities between YAML parsers on CPAN.

meta_spec_version

This method returns the version part of the `meta_spec` entry in the distmeta structure. It is equivalent to:

```
$meta->meta_spec->{version};
```

effective_prereqs

```
my $prereqs = $meta->effective_prereqs;

my $prereqs = $meta->effective_prereqs( \@feature_identifiers );
```

This method returns a *CPAN::Meta::Prereqs* object describing all the prereqs for the distribution. If an arrayref of feature identifiers is given, the prereqs for the identified features are merged together with the distribution's core prereqs before the CPAN::Meta::Prereqs object is returned.

should_index_file

```
... if $meta->should_index_file( $filename );
```

This method returns true if the given file should be indexed. It decides this by checking the `file` and `directory` keys in the `no_index` property of the distmeta structure. Note that neither the version format nor `release_status` are considered.

`$filename` should be given in unix format.

should_index_package

```
... if $meta->should_index_package( $package );
```

This method returns true if the given package should be indexed. It decides this by checking the `package` and `namespace` keys in the `no_index` property of the distmeta structure. Note that neither the version format nor `release_status` are considered.

features

```
my @feature_objects = $meta->features;
```

This method returns a list of *CPAN::Meta::Feature* objects, one for each optional feature described by the distribution's metadata.

feature

```
my $feature_object = $meta->feature( $identifier );
```

This method returns a *CPAN::Meta::Feature* object for the optional feature with the given identifier. If no feature with that identifier exists, an exception will be raised.

as_struct

```
my $copy = $meta->as_struct( \%options );
```

This method returns a deep copy of the object's metadata as an unblessed hash reference. It takes an optional hashref of options. If the hashref contains a `version` argument, the copied metadata will be converted to the version of the specification and returned. For example:

```
my $old_spec = $meta->as_struct( {version => "1.4"} );
```

as_string

```
my $string = $meta->as_string( \%options );
```

This method returns a serialized copy of the object's metadata as a character string. (The strings are **not** UTF-8 encoded.) It takes an optional hashref of options. If the hashref contains a `version` argument, the copied metadata will be converted to the version of the specification and returned. For example:

```
my $string = $meta->as_string( {version => "1.4"} );
```

For `version` greater than or equal to 2, the string will be serialized as JSON. For `version` less than 2, the string will be serialized as YAML. In both cases, the same rules are followed as in the `save()` method for choosing a serialization backend.

The serialized structure will include a `x_serialization_backend` entry giving the package and version used to serialize. Any existing key in the given `$meta` object will be clobbered.

STRING DATA

The following methods return a single value, which is the value for the corresponding entry in the distmeta structure. Values should be either undef or strings.

- abstract
- description
- dynamic_config
- generated_by
- name
- release_status
- version

LIST DATA

These methods return lists of string values, which might be represented in the distmeta structure as arrayrefs or scalars:

- authors
- keywords
- licenses

The `authors` and `licenses` methods may also be called as `author` and `license`, respectively, to match the field name in the distmeta structure.

MAP DATA

These readers return hashrefs of arbitrary unblessed data structures, each described more fully in the specification:

- meta_spec
- resources
- provides
- no_index
- prereqs
- optional_features

CUSTOM DATA

A list of custom keys are available from the `custom_keys` method and particular keys may be retrieved with the `custom` method.

```
say $meta->custom($_) for $meta->custom_keys;
```

If a custom key refers to a data structure, a deep clone is returned.

BUGS

Please report any bugs or feature using the CPAN Request Tracker. Bugs can be submitted through the web interface at http://rt.cpan.org/Dist/Display.html?Queue=CPAN-Meta

When submitting a bug or request, please include a test-file or a patch to an existing test-file that illustrates the bug or desired feature.

SEE ALSO

- *CPAN::Meta::Converter*
- *CPAN::Meta::Validator*

SUPPORT

Bugs / Feature Requests

Please report any bugs or feature requests through the issue tracker at https://github.com/Perl-Toolchain-Gang/CPAN-Meta/issues. You will be notified automatically of any progress on your issue.

Source Code

This is open source software. The code repository is available for public review and contribution under the terms of the license.

https://github.com/Perl-Toolchain-Gang/CPAN-Meta

```
git clone https://github.com/Perl-Toolchain-Gang/CPAN-Meta.git
```

AUTHORS

- David Golden dagolden@cpan.org
- Ricardo Signes cpan@semiotic.systems
- Adam Kennedy adamk@cpan.org

CONTRIBUTORS

- Ansgar Burchardt ansgar@cpan.org
- Avar Arnfjord Bjarmason avar@cpan.org
- Benjamin Noggle agwind@users.noreply.github.com
- Christopher J. Madsen cjm@cpan.org
- Chuck Adams cja987@gmail.com
- Cory G Watson gphat@cpan.org
- Damyan Ivanov dam@cpan.org
- Dan Book grinnz@gmail.com

- Eric Wilhelm ewilhelm@cpan.org
- Graham Knop haarg@haarg.org
- Gregor Hermann gregoa@debian.org
- Karen Etheridge ether@cpan.org
- Kenichi Ishigaki ishigaki@cpan.org
- Kent Fredric kentfredric@gmail.com
- Ken Williams kwilliams@cpan.org
- Lars Dieckow daxim@cpan.org
- Leon Timmermans leont@cpan.org
- majensen maj@fortinbras.us
- Mark Fowler markf@cpan.org
- Matt S Trout mst@shadowcat.co.uk
- Michael G. Schwern mschwern@cpan.org
- Mohammad S Anwar mohammad.anwar@yahoo.com
- mohawk2 mohawk2@users.noreply.github.com
- moznion moznion@gmail.com
- Niko Tyni ntyni@debian.org
- Olaf Alders olaf@wundersolutions.com
- Olivier Mengué dolmen@cpan.org
- Philippe Bruhat (BooK) book@cpan.org
- Randy Sims randys@thepierianspring.org
- Ricardo Signes rjbs@semiotic.systems
- Tomohiro Hosaka bokutin@bokut.in

COPYRIGHT AND LICENSE

This software is copyright (c) 2010 by David Golden, Ricardo Signes, Adam Kennedy and Contributors.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

Modules

- `CPAN::Meta::Converter` - Convert CPAN distribution metadata structures
- `CPAN::Meta::Feature` - an optional feature provided by a CPAN distribution
- `CPAN::Meta::History` - history of CPAN Meta Spec changes
- `CPAN::Meta::Merge` - Merging CPAN Meta fragments

- `CPAN::Meta::Prereqs` - a set of distribution prerequisites by phase and type
- `CPAN::Meta::Requirements` - a set of version requirements for a CPAN dist
- `CPAN::Meta::Spec` - specification for CPAN distribution metadata
- `CPAN::Meta::Validator` - validate CPAN distribution metadata structures
- `CPAN::Meta::YAML` - Read and write a subset of YAML for CPAN Meta files

CPAN::Meta::Converter

Convert CPAN distribution metadata structures

VERSION

version 2.150013

SYNOPSIS

```
my $struct = decode_json_file('META.json');

my $cmc = CPAN::Meta::Converter->new( $struct );

my $new_struct = $cmc->convert( version => "2" );
```

DESCRIPTION

This module converts CPAN Meta structures from one form to another. The primary use is to convert older structures to the most modern version of the specification, but other transformations may be implemented in the future as needed. (E.g. stripping all custom fields or stripping all optional fields.)

METHODS

new

```
my $cmc = CPAN::Meta::Converter->new( $struct );
```

The constructor should be passed a valid metadata structure but invalid structures are accepted. If no meta-spec version is provided, version 1.0 will be assumed.

Optionally, you can provide a `default_version` argument after `$struct`:

```
my $cmc = CPAN::Meta::Converter->new( $struct, default_version =>
↪"1.4" );
```

This is only needed when converting a metadata fragment that does not include a `meta-spec` field.

convert

```
my $new_struct = $cmc->convert( version => "2" );
```

Returns a new hash reference with the metadata converted to a different form. `convert` will die if any conversion/standardization still results in an invalid structure.

Valid parameters include:

- `version` – Indicates the desired specification version (e.g. “1.0”, “1.1” ... “1.4”, “2”). Defaults to the latest version of the CPAN Meta Spec.

Conversion proceeds through each version in turn. For example, a version 1.2 structure might be converted to 1.3 then 1.4 then finally to version 2. The conversion process attempts to clean-up simple errors and standardize data. For example, if `author` is given as a scalar, it will converted to an array reference containing the item. (Converting a structure to its own version will also clean-up and standardize.)

When data are cleaned and standardized, missing or invalid fields will be replaced with sensible defaults when possible. This may be lossy or imprecise. For example, some badly structured META.yml files on CPAN have prerequisite modules listed as both keys and values:

```
requires => { 'Foo::Bar' => 'Bam::Baz' }
```

These would be split and each converted to a prerequisite with a minimum version of zero.

When some mandatory fields are missing or invalid, the conversion will attempt to provide a sensible default or will fill them with a value of ‘unknown’. For example a missing or unrecognized `license` field will result in a `license` field of ‘unknown’. Fields that may get an ‘unknown’ include:

- abstract
- author
- license

upgrade_fragment

```
my $new_struct = $cmc->upgrade_fragment;
```

Returns a new hash reference with the metadata converted to the latest version of the CPAN Meta Spec. No validation is done on the result – you must validate after merging fragments into a complete metadata document.

Available since version 2.141170.

BUGS

Please report any bugs or feature using the CPAN Request Tracker. Bugs can be submitted through the web interface at http://rt.cpan.org/Dist/Display.html?Queue=CPAN-Meta

When submitting a bug or request, please include a test-file or a patch to an existing test-file that illustrates the bug or desired feature.

AUTHORS

- David Golden dagolden@cpan.org
- Ricardo Signes cpan@semiotic.systems
- Adam Kennedy adamk@cpan.org

COPYRIGHT AND LICENSE

This software is copyright (c) 2010 by David Golden, Ricardo Signes, Adam Kennedy and Contributors.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

CPAN::Meta::Feature

an optional feature provided by a CPAN distribution

VERSION

version 2.150013

DESCRIPTION

A CPAN::Meta::Feature object describes an optional feature offered by a CPAN distribution and specified in the distribution's `META.json` (or `META.yml`) file.

For the most part, this class will only be used when operating on the result of the `feature` or `features` methods on a *CPAN::Meta* object.

METHODS

new

```
my $feature = CPAN::Meta::Feature->new( $identifier => \%spec );
```

This returns a new Feature object. The `%spec` argument to the constructor should be the same as the value of the `optional_feature` entry in the distmeta. It must contain entries for `description` and `prereqs`.

identifier

This method returns the feature's identifier.

description

This method returns the feature's long description.

prereqs

This method returns the feature's prerequisites as a *CPAN::Meta::Prereqs* object.

BUGS

Please report any bugs or feature using the CPAN Request Tracker. Bugs can be submitted through the web interface at http://rt.cpan.org/Dist/Display.html?Queue=CPAN-Meta

When submitting a bug or request, please include a test-file or a patch to an existing test-file that illustrates the bug or desired feature.

AUTHORS

- David Golden dagolden@cpan.org
- Ricardo Signes cpan@semiotic.systems
- Adam Kennedy adamk@cpan.org

COPYRIGHT AND LICENSE

This software is copyright (c) 2010 by David Golden, Ricardo Signes, Adam Kennedy and Contributors.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

CPAN::Meta::History

history of CPAN Meta Spec changes

VERSION

version 2.150013

DESCRIPTION

The CPAN Meta Spec has gone through several iterations. It was originally written in HTML and later revised into POD (though published in HTML generated from the POD). Fields were added, removed or changed, sometimes by design and sometimes to reflect real-world usage after the fact.

This document reconstructs the history of the CPAN Meta Spec based on change logs, repository commit messages and the published HTML files. In some cases, particularly prior to version 1.2, the exact version when certain fields were introduced or changed is inconsistent between sources. When in doubt, the published HTML files for versions 1.0 to 1.4 as they existed when version 2 was developed are used as the definitive source.

Starting with version 2, the specification document is part of the CPAN-Meta distribution and will be published on CPAN as *CPAN::Meta::Spec*.

Going forward, specification version numbers will be integers and decimal portions will correspond to a release date for the CPAN::Meta library.

HISTORY

Version 2

April 2010

- Revised spec examples as perl data structures rather than YAML
- Switched to JSON serialization from YAML
- Specified allowed version number formats
- Replaced 'requires', 'build_requires', 'configure_requires', 'recommends' and 'conflicts' with new 'prereqs' data structure divided by *phase* (configure, build, test, runtime, etc.) and *relationship* (requires, recommends, suggests, conflicts)
- Added support for 'develop' phase for requirements for maintaining a list of authoring tools
- Changed 'license' to a list and revised the set of valid licenses
- Made 'dynamic_config' mandatory to reduce confusion
- Changed 'resources' subkey 'repository' to a hash that clarifies repository type, url for browsing and url for checkout
- Changed 'resources' subkey 'bugtracker' to a hash for either web or mailto resource
- Changed specification of 'optional_features':
  - Added formal specification and usage guide instead of just example
  - Changed to use new prereqs data structure instead of individual keys
- Clarified intended use of 'author' as generalized contact list
- Added 'release_status' field to indicate stable, testing or unstable status to provide hints to indexers
- Added 'description' field for a longer description of the distribution
- Formalized use of "x_" or "X_" for all custom keys not listed in the official spec

Version 1.4

June 2008

- Noted explicit support for 'perl' in prerequisites
- Added 'configure_requires' prerequisite type
- Changed 'optional_features'
  - Example corrected to show map of maps instead of list of maps (though descriptive text said 'map' even in v1.3)
  - Removed 'requires_packages', 'requires_os' and 'excluded_os' as valid subkeys

Version 1.3

November 2006

- Added 'no_index' subkey 'directory' and removed 'dir' to match actual usage in the wild
- Added a 'repository' subkey to 'resources'

Version 1.2

August 2005

- Re-wrote and restructured spec in POD syntax
- Changed 'name' to be mandatory
- Changed 'generated_by' to be mandatory
- Changed 'license' to be mandatory
- Added version range specifications for prerequisites
- Added required 'abstract' field
- Added required 'author' field
- Added required 'meta-spec' field to define 'version' (and 'url') of the CPAN Meta Spec used for metadata
- Added 'provides' field
- Added 'no_index' field and deprecated 'private' field. 'no_index' subkeys include 'file', 'dir', 'package' and 'namespace'
- Added 'keywords' field
- Added 'resources' field with subkeys 'homepage', 'license', and 'bugtracker'
- Added 'optional_features' field as an alternate under 'recommends'. Includes 'description', 'requires', 'build_requires', 'conflicts', 'requires_packages', 'requires_os' and 'excluded_os' as valid subkeys
- Removed 'license_uri' field

Version 1.1

May 2003

- Changed 'version' to be mandatory
- Added 'private' field
- Added 'license_uri' field

Version 1.0

March 2003

- Original release (in HTML format only)

- Included ‘name’, ‘version’, ‘license’, ‘distribution_type’, ‘requires’, ‘recommends’, ‘build_requires’, ‘conflicts’, ‘dynamic_config’, ‘generated_by’

AUTHORS

- David Golden dagolden@cpan.org
- Ricardo Signes cpan@semiotic.systems
- Adam Kennedy adamk@cpan.org

COPYRIGHT AND LICENSE

This software is copyright (c) 2010 by David Golden, Ricardo Signes, Adam Kennedy and Contributors.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

CPAN::Meta::Merge

Merging CPAN Meta fragments

VERSION

version 2.150013

SYNOPSIS

```
my $merger = CPAN::Meta::Merge->new(default_version => "2");
my $meta = $merger->merge($base, @additional);
```

DESCRIPTION

METHODS

new

This creates a CPAN::Meta::Merge object. It takes one mandatory named argument, `version`, declaring the version of the meta-spec that must be used for the merge. It can optionally take an `extra_mappings` argument that allows one to add additional merging functions for specific elements.

The `extra_mappings` arguments takes a hash ref with the same type of structure as described in *CPAN::Meta::Spec*, except with its values as one of the defined merge strategies or a code ref to a merging function.

```
my $merger = CPAN::Meta::Merge->new(
    default_version => '2',
    extra_mappings => {
        'optional_features' => \&custom_merge_function,
```

(continues on next page)

(continued from previous page)

```
        'x_custom' => 'set_addition',
        'x_meta_meta' => {
            name => 'identical',
            tags => 'set_addition',
        }
    }
);
```

merge(@fragments)

Merge all `@fragments` together. It will accept both CPAN::Meta objects and (possibly incomplete) hashrefs of metadata.

MERGE STRATEGIES

`merge` uses various strategies to combine different elements of the CPAN::Meta objects. The following strategies can be used with the extra_mappings argument of new:

- identical

  The elements must be identical

- set_addition

  The union of two array refs

  ```
  [ a, b ] U [ a, c]  = [ a, b, c ]
  ```

- uniq_map

  Key value pairs from the right hash are merged to the left hash. Key collisions are only allowed if their values are the same. This merge function will recurse into nested hash refs following the same merge rules.

- improvise

  This merge strategy will try to pick the appropriate predefined strategy based on what element type. Array refs will try to use the `set_addition` strategy, Hash refs will try to use the `uniq_map` strategy, and everything else will try the `identical` strategy.

AUTHORS

- David Golden dagolden@cpan.org
- Ricardo Signes cpan@semiotic.systems
- Adam Kennedy adamk@cpan.org

COPYRIGHT AND LICENSE

This software is copyright (c) 2010 by David Golden, Ricardo Signes, Adam Kennedy and Contributors.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

CPAN::Meta::Prereqs

a set of distribution prerequisites by phase and type

VERSION

version 2.150013

DESCRIPTION

A CPAN::Meta::Prereqs object represents the prerequisites for a CPAN distribution or one of its optional features. Each set of prereqs is organized by phase and type, as described in *CPAN::Meta::Prereqs*.

METHODS

new

```
my $prereq = CPAN::Meta::Prereqs->new( \%prereq_spec );
```

This method returns a new set of Prereqs. The input should look like the contents of the `prereqs` field described in *CPAN::Meta::Spec*, meaning something more or less like this:

```
my $prereq = CPAN::Meta::Prereqs->new({
  runtime => {
    requires => {
      'Some::Module' => '1.234',
      ...,
    },
    ...,
  },
  ...,
});
```

You can also construct an empty set of prereqs with:

```
my $prereqs = CPAN::Meta::Prereqs->new;
```

This empty set of prereqs is useful for accumulating new prereqs before finally dumping the whole set into a structure or string.

requirements_for

```
my $requirements = $prereqs->requirements_for( $phase, $type );
```

This method returns a *CPAN::Meta::Requirements* object for the given phase/type combination. If no prerequisites are registered for that combination, a new CPAN::Meta::Requirements object will be returned, and it may be added to as needed.

If `$phase` or `$type` are undefined or otherwise invalid, an exception will be raised.

phases

```
my @phases = $prereqs->phases;
```

This method returns the list of all phases currently populated in the prereqs object, suitable for iterating.

types_in

```
my @runtime_types = $prereqs->types_in('runtime');
```

This method returns the list of all types currently populated in the prereqs object for the provided phase, suitable for iterating.

with_merged_prereqs

```
my $new_prereqs = $prereqs->with_merged_prereqs( $other_prereqs );

my $new_prereqs = $prereqs->with_merged_prereqs( \@other_prereqs );
```

This method returns a new CPAN::Meta::Prereqs objects in which all the other prerequisites given are merged into the current set. This is primarily provided for combining a distribution's core prereqs with the prereqs of one of its optional features.

The new prereqs object has no ties to the originals, and altering it further will not alter them.

merged_requirements

```
my $new_reqs = $prereqs->merged_requirements( \@phases, \@types );
my $new_reqs = $prereqs->merged_requirements( \@phases );
my $new_reqs = $prereqs->merged_requirements();
```

This method joins together all requirements across a number of phases and types into a new *CPAN::Meta::Requirements* object. If arguments are omitted, it defaults to "runtime", "build" and "test" for phases and "requires" and "recommends" for types.

as_string_hash

This method returns a hashref containing structures suitable for dumping into a distmeta data structure. It is made up of hashes and strings, only; there will be no Prereqs, CPAN::Meta::Requirements, or `version` objects inside it.

is_finalized

This method returns true if the set of prereqs has been marked "finalized," and cannot be altered.

finalize

Calling `finalize` on a Prereqs object will close it for further modification. Attempting to make any changes that would actually alter the prereqs will result in an exception being thrown.

clone

```
my $cloned_prereqs = $prereqs->clone;
```

This method returns a Prereqs object that is identical to the original object, but can be altered without affecting the original object. Finalization does not survive cloning, meaning that you may clone a finalized set of prereqs and then modify the clone.

BUGS

Please report any bugs or feature using the CPAN Request Tracker. Bugs can be submitted through the web interface at http://rt.cpan.org/Dist/Display.html?Queue=CPAN-Meta

When submitting a bug or request, please include a test-file or a patch to an existing test-file that illustrates the bug or desired feature.

AUTHORS

- David Golden dagolden@cpan.org
- Ricardo Signes cpan@semiotic.systems
- Adam Kennedy adamk@cpan.org

COPYRIGHT AND LICENSE

This software is copyright (c) 2010 by David Golden, Ricardo Signes, Adam Kennedy and Contributors.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

CPAN::Meta::Requirements

a set of version requirements for a CPAN dist

VERSION

version 2.145

SYNOPSIS

```
use CPAN::Meta::Requirements;

my $build_requires = CPAN::Meta::Requirements->new;

$build_requires->add_minimum('Library::Foo' => 1.208);

$build_requires->add_minimum('Library::Foo' => 2.602);

$build_requires->add_minimum('Module::Bar'  => 'v1.2.3');

$METAyml->{build_requires} = $build_requires->as_string_hash;
```

DESCRIPTION

A CPAN::Meta::Requirements object models a set of version constraints like those specified in the `META.yml` or `META.json` files in CPAN distributions, and as defined by *CPAN::Meta::Spec*. It can be built up by adding more and more constraints, and it will reduce them to the simplest representation.

Logically impossible constraints will be identified immediately by thrown exceptions.

METHODS

new

```
my $req = CPAN::Meta::Requirements->new;
```

This returns a new CPAN::Meta::Requirements object. It takes an optional hash reference argument. Currently, only one key is supported:

- `bad_version_hook` – if provided, when a version cannot be parsed into a version object, this code reference will be called with the invalid version string as first argument, and the module name as second argument. It must return a valid version object.

All other keys are ignored.

add_minimum

```
$req->add_minimum( $module => $version );
```

This adds a new minimum version requirement. If the new requirement is redundant to the existing specification, this has no effect.

Minimum requirements are inclusive. `$version` is required, along with any greater version number.

This method returns the requirements object.

add_maximum

```
$req->add_maximum( $module => $version );
```

This adds a new maximum version requirement. If the new requirement is redundant to the existing specification, this has no effect.

Maximum requirements are inclusive. No version strictly greater than the given version is allowed.

This method returns the requirements object.

add_exclusion

```
$req->add_exclusion( $module => $version );
```

This adds a new excluded version. For example, you might use these three method calls:

```
$req->add_minimum( $module => '1.00' );
$req->add_maximum( $module => '1.82' );

$req->add_exclusion( $module => '1.75' );
```

Any version between 1.00 and 1.82 inclusive would be acceptable, except for 1.75.

This method returns the requirements object.

exact_version

```
$req->exact_version( $module => $version );
```

This sets the version required for the given module to *exactly* the given version. No other version would be considered acceptable.

This method returns the requirements object.

version_range_for_module

```
$req->version_range_for_module( $another_req_object );
```

add_requirements

```
$req->add_requirements( $another_req_object );
```

This method adds all the requirements in the given CPAN::Meta::Requirements object to the requirements object on which it was called. If there are any conflicts, an exception is thrown.

This method returns the requirements object.

accepts_module

```
my $bool = $req->accepts_module($module => $version);
```

Given an module and version, this method returns true if the version specification for the module accepts the provided version. In other words, given:

```
Module => '>= 1.00, < 2.00'
```

We will accept 1.00 and 1.75 but not 0.50 or 2.00.

For modules that do not appear in the requirements, this method will return true.

clear_requirement

```
$req->clear_requirement( $module );
```

This removes the requirement for a given module from the object.

This method returns the requirements object.

requirements_for_module

```
$req->requirements_for_module( $module );
```

This returns a string containing the version requirements for a given module in the format described in *CPAN::Meta::Spec* or undef if the given module has no requirements. This should only be used for informational purposes such as error messages and should not be interpreted or used for comparison (see “accepts_module” instead).

structured_requirements_for_module

```
$req->structured_requirements_for_module( $module );
```

This returns a data structure containing the version requirements for a given module or undef if the given module has no requirements. This should not be used for version checks (see “accepts_module” instead).

Added in version 2.134.

required_modules

This method returns a list of all the modules for which requirements have been specified.

clone

```
$req->clone;
```

This method returns a clone of the invocant. The clone and the original object can then be changed independent of one another.

is_simple

This method returns true if and only if all requirements are inclusive minimums – that is, if their string expression is just the version number.

is_finalized

This method returns true if the requirements have been finalized by having the `finalize` method called on them.

finalize

This method marks the requirements finalized. Subsequent attempts to change the requirements will be fatal, *if* they would result in a change. If they would not alter the requirements, they have no effect.

If a finalized set of requirements is cloned, the cloned requirements are not also finalized.

as_string_hash

This returns a reference to a hash describing the requirements using the strings in the *CPAN::Meta::Spec* specification.

For example after the following program:

```
my $req = CPAN::Meta::Requirements->new;

$req->add_minimum('CPAN::Meta::Requirements' => 0.102);

$req->add_minimum('Library::Foo' => 1.208);

$req->add_maximum('Library::Foo' => 2.602);

$req->add_minimum('Module::Bar'  => 'v1.2.3');

$req->add_exclusion('Module::Bar'  => 'v1.2.8');
```

(continues on next page)

(continued from previous page)

```
$req->exact_version('Xyzzy'  => '6.01');

my $hashref = $req->as_string_hash;
```

$hashref would contain:

```
{
  'CPAN::Meta::Requirements' => '0.102',
  'Library::Foo' => '>= 1.208, <= 2.206',
  'Module::Bar'  => '>= v1.2.3, != v1.2.8',
  'Xyzzy'        => '== 6.01',
}
```

add_string_requirement

```
$req->add_string_requirement('Library::Foo' => '>= 1.208, <= 2.206
↪');
$req->add_string_requirement('Library::Foo' => v1.208);
```

This method parses the passed in string and adds the appropriate requirement for the given module. A version can be a Perl "v-string". It understands version ranges as described in the *"Version Ranges" in CPAN::Meta::Spec*. For example:

- 1.3
- = 1.3
- <= 1.3
- == 1.3
- != 1.3
- 1.3
- < 1.3
- = 1.3, != 1.5, <= 2.0

  A version number without an operator is equivalent to specifying a minimum (>=). Extra whitespace is allowed.

from_string_hash

```
my $req = CPAN::Meta::Requirements->from_string_hash( \%hash );
my $req = CPAN::Meta::Requirements->from_string_hash( \%hash, \%opts␣
↪);
```

This is an alternate constructor for a CPAN::Meta::Requirements object. It takes a hash of module names and version requirement strings and returns a new CPAN::Meta::Requirements object. As with add_string_requirement, a version can be a Perl "v-string". Optionally, you can supply a hash-reference of options, exactly as with the "new" method.

SUPPORT

Bugs / Feature Requests

Please report any bugs or feature requests through the issue tracker at https://github.com/Perl-Toolchain-Gang/CPAN-Meta-Requirements/issues. You will be notified automatically of any progress on your issue.

Source Code

This is open source software. The code repository is available for public review and contribution under the terms of the license.

https://github.com/Perl-Toolchain-Gang/CPAN-Meta-Requirements

```
git clone https://github.com/Perl-Toolchain-Gang/CPAN-Meta-
↪Requirements.git
```

AUTHORS

- David Golden dagolden@cpan.org
- Ricardo Signes cpan@semiotic.systems

CONTRIBUTORS

- Ed J mohawk2@users.noreply.github.com
- Graham Knop haarg@haarg.org
- Karen Etheridge ether@cpan.org
- Leon Timmermans fawaka@gmail.com
- Paul Howarth paul@city-fan.org
- Philippe Bruhat (BooK) book@cpan.org
- Ricardo Signes rjbs@semiotic.systems
- robario webmaster@robario.com
- Tatsuhiko Miyagawa miyagawa@bulknews.net

COPYRIGHT AND LICENSE

This software is copyright (c) 2010 by David Golden and Ricardo Signes.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

Modules

- `CPAN::Meta::Requirements::Range` - a set of version requirements for a CPAN dist

CPAN::Meta::Requirements::Range

a set of version requirements for a CPAN dist

VERSION

version 2.145

SYNOPSIS

```
use CPAN::Meta::Requirements::Range;

my $range = CPAN::Meta::Requirements::Range->with_minimum(1);

$range = $range->with_maximum('v2.2');

my $stringified = $range->as_string;
```

DESCRIPTION

A CPAN::Meta::Requirements::Range object models a set of version constraints like those specified in the `META.yml` or `META.json` files in CPAN distributions, and as defined by *CPAN::Meta::Spec*; It can be built up by adding more and more constraints, and it will reduce them to the simplest representation.

Logically impossible constraints will be identified immediately by thrown exceptions.

METHODS

with_string_requirement

```
$req->with_string_requirement('>= 1.208, <= 2.206');
$req->with_string_requirement(v1.208);
```

This method parses the passed in string and adds the appropriate requirement. A version can be a Perl "v-string". It understands version ranges as described in the *"Version Ranges" in CPAN::Meta::Spec*. For example:

- 1.3
- = 1.3
- <= 1.3
- == 1.3
- != 1.3

- 1.3
- < 1.3
- = 1.3, != 1.5, <= 2.0

  A version number without an operator is equivalent to specifying a minimum (>=). Extra whitespace is allowed.

with_range

```
$range->with_range($other_range)
```

This creates a new range object that is a merge two others.

with_exact_version

```
$range->with_exact_version( $version );
```

This sets the version required to *exactly* the given version. No other version would be considered acceptable.

This method returns the version range object.

with_minimum

```
$range->with_minimum( $version );
```

This adds a new minimum version requirement. If the new requirement is redundant to the existing specification, this has no effect.

Minimum requirements are inclusive. `$version` is required, along with any greater version number.

This method returns the version range object.

with_maximum

```
$range->with_maximum( $version );
```

This adds a new maximum version requirement. If the new requirement is redundant to the existing specification, this has no effect.

Maximum requirements are inclusive. No version strictly greater than the given version is allowed.

This method returns the version range object.

with_exclusion

```
$range->with_exclusion( $version );
```

This adds a new excluded version. For example, you might use these three method calls:

```
$range->with_minimum( '1.00' );
$range->with_maximum( '1.82' );

$range->with_exclusion( '1.75' );
```

Any version between 1.00 and 1.82 inclusive would be acceptable, except for 1.75.

This method returns the requirements object.

as_struct

```
$range->as_struct( $module );
```

This returns a data structure containing the version requirements. This should not be used for version checks (see "accepts_module" instead).

as_string

```
$range->as_string;
```

This returns a string containing the version requirements in the format described in *CPAN::Meta::Spec*. This should only be used for informational purposes such as error messages and should not be interpreted or used for comparison (see "accepts" instead).

accepts

```
my $bool = $range->accepts($version);
```

Given a version, this method returns true if the version specification accepts the provided version. In other words, given:

```
'>= 1.00, < 2.00'
```

We will accept 1.00 and 1.75 but not 0.50 or 2.00.

is_simple

This method returns true if and only if the range is an inclusive minimum – that is, if their string expression is just the version number.

AUTHORS

- David Golden dagolden@cpan.org
- Ricardo Signes cpan@semiotic.systems

COPYRIGHT AND LICENSE

This software is copyright (c) 2010 by David Golden and Ricardo Signes.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

CPAN::Meta::Spec

specification for CPAN distribution metadata

VERSION

version 2.150013

SYNOPSIS

```
my $distmeta = {
  name => 'Module-Build',
  abstract => 'Build and install Perl modules',
  description =>  "Module::Build is a system for "
    . "building, testing, and installing Perl modules. "
    . "It is meant to ... blah blah blah ...",
  version  => '0.36',
  release_status => 'stable',
  author   => [
    'Ken Williams <kwilliams@cpan.org>',
    'Module-Build List <module-build@perl.org>', # additional
↪contact
  ],
  license  => [ 'perl_5' ],
  prereqs => {
    runtime => {
      requires => {
        'perl'   => '5.006',
        'ExtUtils::Install' => '0',
        'File::Basename' => '0',
        'File::Compare'  => '0',
        'IO::File'   => '0',
      },
      recommends => {
        'Archive::Tar' => '1.00',
        'ExtUtils::Install' => '0.3',
        'ExtUtils::ParseXS' => '2.02',
      },
    },
    build => {
      requires => {
        'Test::More' => '0',
      },
```

(continues on next page)

(continued from previous page)

```
    }
  },
  resources => {
    license => ['http://dev.perl.org/licenses/'],
  },
  optional_features => {
    domination => {
      description => 'Take over the world',
      prereqs     => {
        develop => { requires => { 'Genius::Evil'     => '1.234' } }
↪,
        runtime => { requires => { 'Machine::Weather' => '2.0'   } }
↪,
      },
    },
  },
  dynamic_config => 1,
  keywords => [ qw/ toolchain cpan dual-life / ],
  'meta-spec' => {
    version => '2',
    url     => 'https://metacpan.org/pod/CPAN::Meta::Spec',
  },
  generated_by => 'Module::Build version 0.36',
};
```

DESCRIPTION

This document describes version 2 of the CPAN distribution metadata specification, also known as the "CPAN Meta Spec".

Revisions of this specification for typo corrections and prose clarifications may be issued as CPAN::Meta::Spec 2.*x*. These revisions will never change semantics or add or remove specified behavior.

Distribution metadata describe important properties of Perl distributions. Distribution building tools like Module::Build, Module::Install, ExtUtils::MakeMaker or Dist::Zilla should create a metadata file in accordance with this specification and include it with the distribution for use by automated tools that index, examine, package or install Perl distributions.

TERMINOLOGY

- distribution

  This is the primary object described by the metadata. In the context of this document it usually refers to a collection of modules, scripts, and/or documents that are distributed together for other developers to use. Examples of distributions are `Class-Container`, `libwww-perl`, or `DBI`.

- module

This refers to a reusable library of code contained in a single file. Modules usually contain one or more packages and are often referred to by the name of a primary package that can be mapped to the file name. For example, one might refer to `File::Spec` instead of `File/Spec.pm`

- package

  This refers to a namespace declared with the Perl `package` statement. In Perl, packages often have a version number property given by the `$VERSION` variable in the namespace.

- consumer

  This refers to code that reads a metadata file, deserializes it into a data structure in memory, or interprets a data structure of metadata elements.

- producer

  This refers to code that constructs a metadata data structure, serializes into a bytestream and/or writes it to disk.

- must, should, may, etc.

  These terms are interpreted as described in IETF RFC 2119.

DATA TYPES

Fields in the "STRUCTURE" section describe data elements, each of which has an associated data type as described herein. There are four primitive types: Boolean, String, List and Map. Other types are subtypes of primitives and define compound data structures or define constraints on the values of a data element.

Boolean

A *Boolean* is used to provide a true or false value. It **must** be represented as a defined value that is either "1" or "0" or stringifies to those values.

String

A *String* is data element containing a non-zero length sequence of Unicode characters, such as an ordinary Perl scalar that is not a reference.

List

A *List* is an ordered collection of zero or more data elements. Elements of a List may be of mixed types.

Producers **must** represent List elements using a data structure which unambiguously indicates that multiple values are possible, such as a reference to a Perl array (an "arrayref").

Consumers expecting a List **must** consider a String as equivalent to a List of length 1.

Map

A *Map* is an unordered collection of zero or more data elements ("values"), indexed by associated String elements ("keys"). The Map's value elements may be of mixed types.

License String

A *License String* is a subtype of String with a restricted set of values. Valid values are described in detail in the description of the "license" field.

URL

URL is a subtype of String containing a Uniform Resource Locator or Identifier. [This type is called URL and not URI for historical reasons.]

Version

A *Version* is a subtype of String containing a value that describes the version number of packages or distributions. Restrictions on format are described in detail in the "Version Formats" section.

Version Range

The *Version Range* type is a subtype of String. It describes a range of Versions that may be present or installed to fulfill prerequisites. It is specified in detail in the "Version Ranges" section.

STRUCTURE

The metadata structure is a data element of type Map. This section describes valid keys within the Map.

Any keys not described in this specification document (whether top-level or within compound data structures described herein) are considered *custom keys* and **must** begin with an "x" or "X" and be followed by an underscore; i.e. they must match the pattern: `qr{\Ax_}i`. If a custom key refers to a compound data structure, subkeys within it do not need an "x_" or "X_" prefix.

Consumers of metadata may ignore any or all custom keys. All other keys not described herein are invalid and should be ignored by consumers. Producers must not generate or output invalid keys.

For each key, an example is provided followed by a description. The description begins with the version of spec in which the key was added or in which the definition was modified, whether the key is *required* or *optional* and the data type of the corresponding data element. These items are in parentheses, brackets and braces, respectively.

If a data type is a Map or Map subtype, valid subkeys will be described as well.

Some fields are marked *Deprecated*. These are shown for historical context and must not be produced in or consumed from any metadata structure of version 2 or higher.

REQUIRED FIELDS

abstract

Example:

```
abstract => 'Build and install Perl modules'
```

(Spec 1.2) [required] {String}

This is a short description of the purpose of the distribution.

author

Example:

```
author => [ 'Ken Williams <kwilliams@cpan.org>' ]
```

(Spec 1.2) [required] {List of one or more Strings}

This List indicates the person(s) to contact concerning the distribution. The preferred form of the contact string is:

```
contact-name <email-address>
```

This field provides a general contact list independent of other structured fields provided within the "resources" field, such as `bugtracker`. The addressee(s) can be contacted for any purpose including but not limited to (security) problems with the distribution, questions about the distribution or bugs in the distribution.

A distribution's original author is usually the contact listed within this field. Co-maintainers, successor maintainers or mailing lists devoted to the distribution may also be listed in addition to or instead of the original author.

dynamic_config

Example:

```
dynamic_config => 1
```

(Spec 2) [required] {Boolean}

A boolean flag indicating whether a `Build.PL` or `Makefile.PL` (or similar) must be executed to determine prerequisites.

This field should be set to a true value if the distribution performs some dynamic configuration (asking questions, sensing the environment, etc.) as part of its configuration. This field should be set to a false value to indicate that prerequisites included in metadata may be considered final and valid for static analysis.

Note: when this field is true, post-configuration prerequisites are not guaranteed to bear any relation whatsoever to those stated in the metadata, and relying on them doing so is an error. See also "Prerequisites for dynamically configured distributions" in the implementors' notes.

This field explicitly **does not** indicate whether installation may be safely performed without using a Makefile or Build file, as there may be special files to install or custom installation targets (e.g. for dual-life modules that exist on CPAN as well as in the Perl core). This field only defines whether or not prerequisites are exactly as given in the metadata.

generated_by

Example:

```
generated_by => 'Module::Build version 0.36'
```

(Spec 1.0) [required] {String}

This field indicates the tool that was used to create this metadata. There are no defined semantics for this field, but it is traditional to use a string in the form "Generating::Package version 1.23" or the author's name, if the file was generated by hand.

license

Example:

```
license => [ 'perl_5' ]

license => [ 'apache_2_0', 'mozilla_1_0' ]
```

(Spec 2) [required] {List of one or more License Strings}

One or more licenses that apply to some or all of the files in the distribution. If multiple licenses are listed, the distribution documentation should be consulted to clarify the interpretation of multiple licenses.

The following list of license strings are valid:

```
string          description
-------------   ---------------------------------------------
agpl_3          GNU Affero General Public License, Version 3
apache_1_1      Apache Software License, Version 1.1
apache_2_0      Apache License, Version 2.0
artistic_1      Artistic License, (Version 1)
artistic_2      Artistic License, Version 2.0
bsd             BSD License (three-clause)
freebsd         FreeBSD License (two-clause)
gfdl_1_2        GNU Free Documentation License, Version 1.2
gfdl_1_3        GNU Free Documentation License, Version 1.3
gpl_1           GNU General Public License, Version 1
gpl_2           GNU General Public License, Version 2
gpl_3           GNU General Public License, Version 3
lgpl_2_1        GNU Lesser General Public License, Version 2.1
lgpl_3_0        GNU Lesser General Public License, Version 3.0
mit             MIT (aka X11) License
mozilla_1_0     Mozilla Public License, Version 1.0
```

(continues on next page)

(continued from previous page)

```
mozilla_1_1     Mozilla Public License, Version 1.1
openssl         OpenSSL License
perl_5          The Perl 5 License (Artistic 1 & GPL 1 or later)
qpl_1_0         Q Public License, Version 1.0
ssleay          Original SSLeay License
sun             Sun Internet Standards Source License (SISSL)
zlib            zlib License
```

The following license strings are also valid and indicate other licensing not described above:

```
string          description
-------------   ------------------------------------------------
open_source     Other Open Source Initiative (OSI) approved license
restricted      Requires special permission from copyright holder
unrestricted    Not an OSI approved license, but not restricted
unknown         License not provided in metadata
```

All other strings are invalid in the license field.

meta-spec

Example:

```
'meta-spec' => {
  version => '2',
  url     => 'http://search.cpan.org/perldoc?CPAN::Meta::Spec',
}
```

(Spec 1.2) [required] {Map}

This field indicates the version of the CPAN Meta Spec that should be used to interpret the metadata. Consumers must check this key as soon as possible and abort further metadata processing if the meta-spec version is not supported by the consumer.

The following keys are valid, but only `version` is required.

- version

  This subkey gives the integer *Version* of the CPAN Meta Spec against which the document was generated.

- url

  This is a *URL* of the metadata specification document corresponding to the given version. This is strictly for human-consumption and should not impact the interpretation of the document.

  For the version 2 spec, either of these are recommended:

  – `https://metacpan.org/pod/CPAN::Meta::Spec`

  – `http://search.cpan.org/perldoc?CPAN::Meta::Spec`

name

Example:

```
name => 'Module-Build'
```

(Spec 1.0) [required] {String}

This field is the name of the distribution. This is often created by taking the "main package" in the distribution and changing `::` to `-`, but the name may be completely unrelated to the packages within the distribution. For example, LWP::UserAgent is distributed as part of the distribution name "libwww-perl".

release_status

Example:

```
release_status => 'stable'
```

(Spec 2) [required] {String}

This field provides the release status of this distribution. If the `version` field contains an underscore character, then `release_status` **must not** be "stable."

The `release_status` field **must** have one of the following values:

- stable

  This indicates an ordinary, "final" release that should be indexed by PAUSE or other indexers.

- testing

  This indicates a "beta" release that is substantially complete, but has an elevated risk of bugs and requires additional testing. The distribution should not be installed over a stable release without an explicit request or other confirmation from a user. This release status may also be used for "release candidate" versions of a distribution.

- unstable

  This indicates an "alpha" release that is under active development, but has been released for early feedback or testing and may be missing features or may have serious bugs. The distribution should not be installed over a stable release without an explicit request or other confirmation from a user.

Consumers **may** use this field to determine how to index the distribution for CPAN or other repositories in addition to or in replacement of heuristics based on version number or file name.

version

Example:

```
version => '0.36'
```

(Spec 1.0) [required] {Version}

This field gives the version of the distribution to which the metadata structure refers.

OPTIONAL FIELDS

description

Example:

```
description =>  "Module::Build is a system for "
  . "building, testing, and installing Perl modules. "
  . "It is meant to ... blah blah blah ...",
```

(Spec 2) [optional] {String}

A longer, more complete description of the purpose or intended use of the distribution than the one provided by the `abstract` key.

keywords

Example:

```
keywords => [ qw/ toolchain cpan dual-life / ]
```

(Spec 1.1) [optional] {List of zero or more Strings}

A List of keywords that describe this distribution. Keywords **must not** include whitespace.

no_index

Example:

```
no_index => {
  file      => [ 'My/Module.pm' ],
  directory => [ 'My/Private' ],
  package   => [ 'My::Module::Secret' ],
  namespace => [ 'My::Module::Sample' ],
}
```

(Spec 1.2) [optional] {Map}

This Map describes any files, directories, packages, and namespaces that are private to the packaging or implementation of the distribution and should be ignored by indexing or search tools. Note that this is a list of exclusions, and the spec does not define what to *include* - see “Indexing distributions a la PAUSE” in the implementors notes for more information.

Valid subkeys are as follows:

- file

  A *List* of relative paths to files. Paths **must be** specified with unix conventions.

- directory

  A *List* of relative paths to directories. Paths **must be** specified with unix conventions.

  [Note: previous editions of the spec had `dir` instead of `directory`]

- package

  A *List* of package names.

- namespace

  A *List* of package namespaces, where anything below the namespace must be ignored, but *not* the namespace itself.

  In the example above for `no_index`, `My::Module::Sample::Foo` would be ignored, but `My::Module::Sample` would not.

optional_features

Example:

```
optional_features => {
  sqlite => {
    description => 'Provides SQLite support',
    prereqs => {
      runtime => {
        requires => {
          'DBD::SQLite' => '1.25'
        }
      }
    }
  }
}
```

(Spec 2) [optional] {Map}

This Map describes optional features with incremental prerequisites. Each key of the `optional_features` Map is a String used to identify the feature and each value is a Map with additional information about the feature. Valid subkeys include:

- description

  This is a String describing the feature. Every optional feature should provide a description

- prereqs

  This entry is required and has the same structure as that of the `"prereqs"` key. It provides a list of package requirements that must be satisfied for the feature to be supported or enabled.

  There is one crucial restriction: the prereqs of an optional feature **must not** include `configure` phase prereqs.

Consumers **must not** include optional features as prerequisites without explicit instruction from users (whether via interactive prompting, a function parameter or a configuration value, etc.).

If an optional feature is used by a consumer to add additional prerequisites, the consumer should merge the optional feature prerequisites into those given by the `prereqs` key using the same semantics. See “Merging and Resolving Prerequisites” for details on merging prerequisites.

Suggestion for disuse: Because there is currently no way for a distribution to specify a dependency on an optional feature of another dependency, the use of `optional_feature` is discouraged. Instead, create a separate, installable distribution that ensures the desired feature is available. For example, if `Foo::Bar` has a `Baz` feature, release a separate `Foo-Bar-Baz` distribution that satisfies requirements for the feature.

prereqs

Example:

```
prereqs => {
  runtime => {
    requires => {
      'perl'           => '5.006',
      'File::Spec'     => '0.86',
      'JSON'           => '2.16',
    },
    recommends => {
      'JSON::XS'       => '2.26',
    },
    suggests => {
      'Archive::Tar'   => '0',
    },
  },
  build => {
    requires => {
      'Alien::SDL'     => '1.00',
    },
  },
  test => {
    recommends => {
      'Test::Deep'     => '0.10',
    },
  }
}
```

(Spec 2) [optional] {Map}

This is a Map that describes all the prerequisites of the distribution. The keys are phases of activity, such as `configure`, `build`, `test` or `runtime`. Values are Maps in which the keys name the type of prerequisite relationship such as `requires`, `recommends`, or `suggests` and the value provides a set of prerequisite relations. The set of relations **must** be specified as a Map of package names to version ranges.

The full definition for this field is given in the "Prereq Spec" section.

provides

Example:

```
provides => {
  'Foo::Bar' => {
    file    => 'lib/Foo/Bar.pm',
    version => '0.27_02',
  },
  'Foo::Bar::Blah' => {
    file    => 'lib/Foo/Bar/Blah.pm',
  },
  'Foo::Bar::Baz' => {
    file    => 'lib/Foo/Bar/Baz.pm',
    version => '0.3',
  },
}
```

(Spec 1.2) [optional] {Map}

This describes all packages provided by this distribution. This information is used by distribution and automation mechanisms like PAUSE, CPAN, metacpan.org and search.cpan.org to build indexes saying in which distribution various packages can be found.

The keys of `provides` are package names that can be found within the distribution. If a package name key is provided, it must have a Map with the following valid subkeys:

- file

  This field is required. It must contain a Unix-style relative file path from the root of the distribution directory to a file that contains or generates the package. It may be given as `META.yml` or `META.json` to claim a package for indexing without needing a `*.pm`.

- version

  If it exists, this field must contains a *Version* String for the package. If the package does not have a `$VERSION`, this field must be omitted.

resources

Example:

```
resources => {
  license     => [ 'http://dev.perl.org/licenses/' ],
  homepage    => 'http://sourceforge.net/projects/module-build',
  bugtracker  => {
    web     => 'http://rt.cpan.org/Public/Dist/Display.html?
↪Name=CPAN-Meta',
    mailto => 'meta-bugs@example.com',
  },
```

(continues on next page)

(continued from previous page)

```
  repository  => {
    url  => 'git://github.com/dagolden/cpan-meta.git',
    web  => 'http://github.com/dagolden/cpan-meta',
    type => 'git',
  },
  x_twitter   => 'http://twitter.com/cpan_linked/',
}
```

(Spec 2) [optional] {Map}

This field describes resources related to this distribution.

Valid subkeys include:

- homepage

  The official home of this project on the web.

- license

  A List of *URL*'s that relate to this distribution's license. As with the top-level `license` field, distribution documentation should be consulted to clarify the interpretation of multiple licenses provided here.

- bugtracker

  This entry describes the bug tracking system for this distribution. It is a Map with the following valid keys:

  ```
  web    - a URL pointing to a web front-end for the bug tracker
  mailto - an email address to which bugs can be sent
  ```

- repository

  This entry describes the source control repository for this distribution. It is a Map with the following valid keys:

  ```
  url  - a URL pointing to the repository itself
  web  - a URL pointing to a web front-end for the repository
  type - a lowercase string indicating the VCS used
  ```

  Because a url like `http://myrepo.example.com/` is ambiguous as to type, producers should provide a `type` whenever a `url` key is given. The `type` field should be the name of the most common program used to work with the repository, e.g. `git`, `svn`, `cvs`, `darcs`, `bzr` or `hg`.

DEPRECATED FIELDS

build_requires

(Deprecated in Spec 2) [optional] {String}

Replaced by `prereqs`

configure_requires

(Deprecated in Spec 2) [optional] {String}

Replaced by `prereqs`

conflicts

(Deprecated in Spec 2) [optional] {String}

Replaced by `prereqs`

distribution_type

(Deprecated in Spec 2) [optional] {String}

This field indicated 'module' or 'script' but was considered meaningless, since many distributions are hybrids of several kinds of things.

license_uri

(Deprecated in Spec 1.2) [optional] {URL}

Replaced by `license` in `resources`

private

(Deprecated in Spec 1.2) [optional] {Map}

This field has been renamed to "no_index".

recommends

(Deprecated in Spec 2) [optional] {String}

Replaced by `prereqs`

requires

(Deprecated in Spec 2) [optional] {String}

Replaced by `prereqs`

VERSION NUMBERS

Version Formats

This section defines the Version type, used by several fields in the CPAN Meta Spec.

Version numbers must be treated as strings, not numbers. For example, `1.200` **must not** be serialized as `1.2`. Version comparison should be delegated to the Perl *version* module, version 0.80 or newer.

Unless otherwise specified, version numbers **must** appear in one of two formats:

- Decimal versions

  Decimal versions are regular "decimal numbers", with some limitations. They **must** be non-negative and **must** begin and end with a digit. A single underscore **may** be included, but **must** be between two digits. They **must not** use exponential notation ("1.23e-2").

```
version => '1.234'       # OK
version => '1.23_04'     # OK

version => '1.23_04_05'  # Illegal
version => '1.'          # Illegal
version => '.1'          # Illegal
```

- Dotted-integer versions

  Dotted-integer (also known as dotted-decimal) versions consist of positive integers separated by full stop characters (i.e. "dots", "periods" or "decimal points"). This are equivalent in format to Perl "v-strings", with some additional restrictions on form. They must be given in "normal" form, which has a leading "v" character and at least three integer components. To retain a one-to-one mapping with decimal versions, all components after the first **should** be restricted to the range 0 to 999. The final component **may** be separated by an underscore character instead of a period.

```
version => 'v1.2.3'      # OK
version => 'v1.2_3'      # OK
version => 'v1.2.3.4'    # OK
version => 'v1.2.3_4'    # OK
version => 'v2009.10.31' # OK

version => 'v1.2'          # Illegal
version => '1.2.3'         # Illegal
version => 'v1.2_3_4'      # Illegal
version => 'v1.2009.10.31' # Not recommended
```

Version Ranges

Some fields (prereq, optional_features) indicate the particular version(s) of some other module that may be required as a prerequisite. This section details the Version Range type used to provide this information.

The simplest format for a Version Range is just the version number itself, e.g. `2.4`. This means that **at least** version 2.4 must be present. To indicate that **any** version of a prerequisite is okay, even if the prerequisite doesn't define a version at all, use the version `0`.

Alternatively, a version range **may** use the operators < (less than), <= (less than or equal), > (greater than), >= (greater than or equal), == (equal), and != (not equal). For example, the specification `< 2.0` means that any version of the prerequisite less than 2.0 is suitable.

For more complicated situations, version specifications **may** be AND-ed together using commas. The specification `>= 1.2, != 1.5, < 2.0` indicates a version that must be

at least 1.2, **less than** 2.0, and **not equal to** 1.5.

PREREQUISITES

Prereq Spec

The `prereqs` key in the top-level metadata and within `optional_features` define the relationship between a distribution and other packages. The prereq spec structure is a hierarchical data structure which divides prerequisites into *Phases* of activity in the installation process and *Relationships* that indicate how prerequisites should be resolved.

For example, to specify that `Data::Dumper` is `required` during the `test` phase, this entry would appear in the distribution metadata:

```
prereqs => {
  test => {
    requires => {
      'Data::Dumper' => '2.00'
    }
  }
}
```

Phases

Requirements for regular use must be listed in the `runtime` phase. Other requirements should be listed in the earliest stage in which they are required and consumers must accumulate and satisfy requirements across phases before executing the activity. For example, `build` requirements must also be available during the `test` phase.

```
before action       requirements that must be met
----------------    ------------------------------
perl Build.PL       configure
perl Makefile.PL

make                configure, runtime, build
Build

make test           configure, runtime, build, test
Build test
```

Consumers that install the distribution must ensure that *runtime* requirements are also installed and may install dependencies from other phases.

```
after action        requirements that must be met
----------------    ------------------------------
make install        runtime
Build install
```

- configure

  The configure phase occurs before any dynamic configuration has been attempted. Libraries required by the configure phase **must** be available for use before the distribution building tool has been executed.

- build

  The build phase is when the distribution's source code is compiled (if necessary) and otherwise made ready for installation.

- test

  The test phase is when the distribution's automated test suite is run. Any library that is needed only for testing and not for subsequent use should be listed here.

- runtime

  The runtime phase refers not only to when the distribution's contents are installed, but also to its continued use. Any library that is a prerequisite for regular use of this distribution should be indicated here.

- develop

  The develop phase's prereqs are libraries needed to work on the distribution's source code as its author does. These tools might be needed to build a release tarball, to run author-only tests, or to perform other tasks related to developing new versions of the distribution.

Relationships

- requires

  These dependencies **must** be installed for proper completion of the phase.

- recommends

  Recommended dependencies are *strongly* encouraged and should be satisfied except in resource constrained environments.

- suggests

  These dependencies are optional, but are suggested for enhanced operation of the described distribution.

- conflicts

  These libraries cannot be installed when the phase is in operation. This is a very rare situation, and the `conflicts` relationship should be used with great caution, or not at all.

Merging and Resolving Prerequisites

Whenever metadata consumers merge prerequisites, either from different phases or from `optional_features`, they should merged in a way which preserves the intended semantics of the prerequisite structure. Generally, this means concatenating the version specifications using commas, as described in the "Version Ranges" section.

Another subtle error that can occur in resolving prerequisites comes from the way that modules in prerequisites are indexed to distribution files on CPAN. When a module is deleted from a distribution, prerequisites calling for that module could indicate an older distribution should be installed, potentially overwriting files from a newer distribution.

For example, as of Oct 31, 2009, the CPAN index file contained these module-distribution mappings:

```
Class::MOP                      0.94  D/DR/DROLSKY/Class-MOP-0.94.tar.
 ↪gz
Class::MOP::Class               0.94  D/DR/DROLSKY/Class-MOP-0.94.tar.
 ↪gz
Class::MOP::Class::Immutable 0.04  S/ST/STEVAN/Class-MOP-0.36.tar.gz
```

Consider the case where "Class::MOP" 0.94 is installed. If a distribution specified "Class::MOP::Class::Immutable" as a prerequisite, it could result in Class-MOP-0.36.tar.gz being installed, overwriting any files from Class-MOP-0.94.tar.gz.

Consumers of metadata **should** test whether prerequisites would result in installed module files being "downgraded" to an older version and **may** warn users or ignore the prerequisite that would cause such a result.

SERIALIZATION

Distribution metadata should be serialized (as a hashref) as JSON-encoded data and packaged with distributions as the file `META.json`.

In the past, the distribution metadata structure had been packed with distributions as `META.yml`, a file in the YAML Tiny format (for which, see YAML::Tiny). Tools that consume distribution metadata from disk should be capable of loading `META.yml`, but should prefer `META.json` if both are found.

NOTES FOR IMPLEMENTORS

Extracting Version Numbers from Perl Modules

To get the version number from a Perl module, consumers should use the `MM->parse_version($file)` method provided by *ExtUtils::MakeMaker* or *Module::Metadata*. For example, for the module given by `$mod`, the version may be retrieved in one of the following ways:

```
# via ExtUtils::MakeMaker
my $file = MM->_installed_file_for_module($mod);
my $version = MM->parse_version($file)
```

The private `_installed_file_for_module` method may be replaced with other methods for locating a module in `@INC`.

```
# via Module::Metadata
my $info = Module::Metadata->new_from_module($mod);
my $version = $info->version;
```

If only a filename is available, the following approach may be used:

```
# via Module::Metadata
my $info = Module::Metadata->new_from_file($file);
my $version = $info->version;
```

Comparing Version Numbers

The *version* module provides the most reliable way to compare version numbers in all the various ways they might be provided or might exist within modules. Given two strings containing version numbers, `$v1` and `$v2`, they should be converted to `version` objects before using ordinary comparison operators. For example:

```
use version;
if ( version->new($v1) <=> version->new($v2) ) {
  print "Versions are not equal\n";
}
```

If the only comparison needed is whether an installed module is of a sufficiently high version, a direct test may be done using the `VERSION` method. For example, for module `$mod` and version prerequisite `$prereq`:

```
use Module::Load 'load';
if ( $mod =~ m/\A[\w:']+\z/a and eval { load $mod; $mod->VERSION(
↪$prereq); 1 } ) {
  print "Module $mod version is OK.\n";
}
```

The regexp checks that `$mod` only includes characters legal for module names before passing it to *"load" in Module::Load*, which also accepts file paths that may escape `@INC`. Alternatively, if Module::Runtime is installed, the `use_module` function can load the module and perform the version check at the same time, and does not accept file paths:

```
use Module::Runtime 'use_module';
if ( eval { use_module $mod, $prereq; 1 } ) {
  print "Module $mod version is OK.\n";
}
```

Prerequisites for dynamically configured distributions

When `dynamic_config` is true, it is an error to presume that the prerequisites given in distribution metadata will have any relationship whatsoever to the actual prerequisites of the distribution.

In practice, however, one can generally expect such prerequisites to be one of two things:

- The minimum prerequisites for the distribution, to which dynamic configuration will only add items
- Whatever the distribution configured with on the releaser's machine at release time

The second case often turns out to have identical results to the first case, albeit only by accident.

As such, consumers may use this data for informational analysis, but presenting it to the user as canonical or relying on it as such is invariably the height of folly.

Indexing distributions a la PAUSE

While no_index tells you what must be ignored when indexing, this spec holds no opinion on how you should get your initial candidate list of things to possibly index. For "normal" distributions you might consider simply indexing the contents of lib/, but there are many fascinating oddities on CPAN and many dists from the days when it was normal to put the main .pm file in the root of the distribution archive - so PAUSE currently indexes all .pm and .PL files that are not either (a) specifically excluded by no_index (b) in `inc`, `xt`, or `t` directories, or common 'mistake' directories such as `perl5`.

Or: If you're trying to be PAUSE-like, make sure you skip `inc`, `xt` and `t` as well as anything marked as no_index.

Also remember: If the META file contains a provides field, you shouldn't be indexing anything in the first place - just use that.

SEE ALSO

- CPAN, http://www.cpan.org/
- JSON, http://json.org/
- YAML, http://www.yaml.org/
- *CPAN*
- CPANPLUS
- *ExtUtils::MakeMaker*
- Module::Build
- Module::Install
- CPAN::Meta::History::Meta_1_4

HISTORY

Ken Williams wrote the original CPAN Meta Spec (also known as the "META.yml spec") in 2003 and maintained it through several revisions with input from various members of the community. In 2005, Randy Sims redrafted it from HTML to POD for the version 1.2 release. Ken continued to maintain the spec through version 1.4.

In late 2009, David Golden organized the version 2 proposal review process. David and Ricardo Signes drafted the final version 2 spec in April 2010 based on the version 1.4 spec and patches contributed during the proposal process.

AUTHORS

- David Golden dagolden@cpan.org
- Ricardo Signes cpan@semiotic.systems
- Adam Kennedy adamk@cpan.org

COPYRIGHT AND LICENSE

This software is copyright (c) 2010 by David Golden, Ricardo Signes, Adam Kennedy and Contributors.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

CPAN::Meta::Validator

validate CPAN distribution metadata structures

VERSION

version 2.150013

SYNOPSIS

```
my $struct = decode_json_file('META.json');

my $cmv = CPAN::Meta::Validator->new( $struct );

unless ( $cmv->is_valid ) {
  my $msg = "Invalid META structure.  Errors found:\n";
  $msg .= join( "\n", $cmv->errors );
  die $msg;
}
```

DESCRIPTION

This module validates a CPAN Meta structure against the version of the the specification claimed in the `meta-spec` field of the structure.

METHODS

new

```
my $cmv = CPAN::Meta::Validator->new( $struct )
```

The constructor must be passed a metadata structure.

is_valid

```
if ( $cmv->is_valid ) {
  ...
}
```

Returns a boolean value indicating whether the metadata provided is valid.

errors

```
warn( join "\n", $cmv->errors );
```

Returns a list of errors seen during validation.

BUGS

Please report any bugs or feature using the CPAN Request Tracker. Bugs can be submitted through the web interface at http://rt.cpan.org/Dist/Display.html?Queue=CPAN-Meta

When submitting a bug or request, please include a test-file or a patch to an existing test-file that illustrates the bug or desired feature.

AUTHORS

- David Golden dagolden@cpan.org
- Ricardo Signes cpan@semiotic.systems
- Adam Kennedy adamk@cpan.org

COPYRIGHT AND LICENSE

This software is copyright (c) 2010 by David Golden, Ricardo Signes, Adam Kennedy and Contributors.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

CPAN::Meta::YAML

Read and write a subset of YAML for CPAN Meta files

VERSION

version 0.020

SYNOPSIS

```
use CPAN::Meta::YAML;

# reading a META file
open $fh, "<:utf8", "META.yml";
$yaml_text = do { local $/; <$fh> };
$yaml = CPAN::Meta::YAML->read_string($yaml_text)
  or die CPAN::Meta::YAML->errstr;

# finding the metadata
$meta = $yaml->[0];
```

(continues on next page)

(continued from previous page)

```
# writing a META file
$yaml_text = $yaml->write_string
  or die CPAN::Meta::YAML->errstr;
open $fh, ">:utf8", "META.yml";
print $fh $yaml_text;
```

DESCRIPTION

This module implements a subset of the YAML specification for use in reading and writing CPAN metadata files like `META.yml` and `MYMETA.yml`. It should not be used for any other general YAML parsing or generation task.

NOTE: `META.yml` (and `MYMETA.yml`) files should be UTF-8 encoded. Users are responsible for proper encoding and decoding. In particular, the `read` and `write` methods do **not** support UTF-8 and should not be used.

SUPPORT

This module is currently derived from YAML::Tiny by Adam Kennedy. If there are bugs in how it parses a particular META.yml file, please file a bug report in the YAML::Tiny bugtracker: https://github.com/Perl-Toolchain-Gang/YAML-Tiny/issues

SEE ALSO

YAML::Tiny, *YAML*, *YAML::XS*

AUTHORS

- Adam Kennedy adamk@cpan.org
- David Golden dagolden@cpan.org

CONTRIBUTOR

Karen Etheridge ether@cpan.org

COPYRIGHT AND LICENSE

This software is copyright (c) 2010 by Adam Kennedy.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

CPAN::Mirrors

Get CPAN mirror information and select a fast one

SYNOPSIS

```
use CPAN::Mirrors;

my $mirrors = CPAN::Mirrors->new( $mirrored_by_file );

my $seen = {};

my $best_continent = $mirrors->find_best_continents( { seen => $seen
 ↪} );
my @mirrors        = $mirrors->get_mirrors_by_continents( $best_
 ↪continent );

my $callback = sub {
    my( $m ) = @_;
    printf "%s = %s\n", $m->hostname, $m->rtt
    };
$mirrors->get_mirrors_timings( \@mirrors, $seen, $callback, %args );

@mirrors = sort { $a->rtt <=> $b->rtt } @mirrors;

print "Best mirrors are ", map( { $_->rtt } @mirrors[0..3] ), "\n";
```

DESCRIPTION

- new(LOCAL_FILE_NAME)

  Create a new CPAN::Mirrors object from LOCAL_FILE_NAME. This file should look like that in http://www.cpan.org/MIRRORED.BY .

- continents()

  Return a list of continents based on those defined in `MIRRORED.BY`.

- countries([CONTINENTS])

  Return a list of countries based on those defined in `MIRRORED.BY`. It only returns countries for the continents you specify (as defined in `continents`). If you don't specify any continents, it returns all of the countries listed in `MIRRORED.BY`.

- mirrors([COUNTRIES])

  Return a list of mirrors based on those defined in `MIRRORED.BY`. It only returns mirrors for the countries you specify (as defined in `countries`). If you don't specify any countries, it returns all of the mirrors listed in `MIRRORED.BY`.

- get_mirrors_by_countries([COUNTRIES])

  A more sensible synonym for mirrors.

- get_mirrors_by_continents([CONTINENTS])

  Return a list of mirrors for all of continents you specify. If you don't specify any continents, it returns all of the mirrors.

  You can specify a single continent or an array reference of continents.

- get_countries_by_continents([CONTINENTS])

  A more sensible synonym for countries.

- default_mirror

  Returns the default mirror, http://www.cpan.org/ . This mirror uses dynamic DNS to give a close mirror.

- best_mirrors

  `best_mirrors` checks for the best mirrors based on the list of continents you pass, or, without that, all continents, as defined by `CPAN::Mirrored::By`. It pings each mirror, up to the value of `how_many`. In list context, it returns up to `how_many` mirrors. In scalar context, it returns the single best mirror.

  Arguments

```
how_many      - the number of mirrors to return. Default: 1
callback      - a callback for find_best_continents
verbose       - true or false on all the whining and moaning.␣
↪Default: false
continents    - an array ref of the continents to check
external_ping - if true, use external ping via␣
↪Net::Ping::External. Default: false
```

  If you don't specify the continents, `best_mirrors` calls `find_best_continents` to get the list of continents to check.

  If you don't have *Net::Ping* v2.13 or later, needed for timings, this returns the default mirror.

  `external_ping` should be set and then `Net::Ping::External` needs to be installed, if the local network has a transparent proxy.

- get_n_random_mirrors_by_continents(N, [CONTINENTS])

  Returns up to N random mirrors for the specified continents. Specify the continents as an array reference.

- get_mirrors_timings(MIRROR_LIST, SEEN, CALLBACK, %ARGS);

  Pings the listed mirrors and returns a list of mirrors sorted in ascending ping times.

  `MIRROR_LIST` is an anonymous array of `CPAN::Mirrored::By` objects to ping.

  The optional argument `SEEN` is a hash reference used to track the mirrors you've already pinged.

  The optional argument `CALLBACK` is a subroutine reference to call after each ping. It gets the `CPAN::Mirrored::By` object after each ping.

- find_best_continents(HASH_REF);

  `find_best_continents` goes through each continent and pings `N` random mirrors on that continent. It then orders the continents by ascending median ping time. In list context, it returns the ordered list of continent. In scalar context, it returns the same list as an anonymous array.

  Arguments:

```
n        - the number of hosts to ping for each continent. Default: 3
seen     - a hashref of cached hostname ping times
verbose  - true or false for noisy or quiet. Default: false
callback - a subroutine to run after each ping.
ping_cache_limit - how long, in seconds, to reuse previous ping times.
   Default: 1 day
```

The seen hash has hostnames as keys and anonymous arrays as values. The anonymous array is a triplet of a CPAN::Mirrored::By object, a ping time, and the epoch time for the measurement.

The callback subroutine gets the CPAN::Mirrored::By object, the ping time, and measurement time (the same things in the seen hashref) as arguments. find_best_continents doesn't care what the callback does and ignores the return value.

With a low value for N, a single mirror might skew the results enough to choose a worse continent. If you have that problem, try a larger value.

AUTHOR

Andreas Koenig <andk@cpan.org>, David Golden <dagolden@cpan.org>, brian d foy <bdfoy@cpan.org>

LICENSE

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See http://www.perl.com/perl/misc/Artistic.html

CPAN::Nox

SYNOPSIS

Interactive mode:

```
perl -MCPAN::Nox -e shell;
```

DESCRIPTION

This package has the same functionality as CPAN.pm, but tries to prevent the usage of compiled extensions during its own execution. Its primary purpose is a rescue in case you upgraded perl and broke binary compatibility somehow.

LICENSE

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

SEE ALSO

CPAN

CPAN::Plugin

Base class for CPAN shell extensions

SYNOPSIS

```
package CPAN::Plugin::Flurb;
use parent 'CPAN::Plugin';

sub post_test {
  my ($self, $distribution_object) = @_;
  $self = $self->new (distribution_object => $distribution_object);
  ...;
}
```

DESCRIPTION

Alpha Status

The plugin system in the CPAN shell was introduced in version 2.07 and is still considered experimental.

How Plugins work?

See *"Plugin support" in CPAN*.

METHODS

plugin_requires

returns list of packages given plugin requires for functionality. This list is evaluated using `CPAN->use_inst` method.

distribution_object

Get current distribution object.

distribution

distribution_info

build_dir

Simple delegatees for misc parameters derived from distribution

is_xs

AUTHOR

Branislav Zahradnik barney@cpan.org

Modules

- `CPAN::Plugin::Specfile` - Proof of concept implementation of a trivial CPAN::Plugin

CPAN::Plugin::Specfile

Proof of concept implementation of a trivial CPAN::Plugin

SYNOPSIS

```
# once in the cpan shell
o conf plugin_list push CPAN::Plugin::Specfile

# make permanent
o conf commit

# any time in the cpan shell to write a spec file
test Acme::Meta

# disable
# if it is the last in plugin_list:
o conf plugin_list pop
# otherwise, determine the index to splice:
o conf plugin_list
# and then use splice, e.g. to splice position 3:
o conf plugin_list splice 3 1
```

DESCRIPTION

Implemented as a post-test hook, this plugin writes a specfile after every successful test run. The content is also written to the terminal.

As a side effect, the timestamps of the written specfiles reflect the linear order of all dependencies.

WARNING: This code is just a small demo how to use the plugin system of the CPAN shell, not a full fledged spec file writer. Do not expect new features in this plugin.

OPTIONS

The target directory to store the spec files in can be set using `dir` as in

```
o conf plugin_list push CPAN::Plugin::Specfile=dir,/tmp/specfiles-
↪000042
```

The default directory for this is the `plugins/CPAN::Plugin::Specfile` directory in the *cpan_home* directory.

AUTHOR

Andreas Koenig andk@cpan.org, Branislav Zahradnik barney@cpan.org

CPAN::Queue

internal queue support for CPAN.pm

LICENSE

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

CPAN::Tarzip

internal handling of tar archives for CPAN.pm

LICENSE

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

CPAN::Version

utility functions to compare CPAN versions

SYNOPSIS

```
use CPAN::Version;

CPAN::Version->vgt("1.1","1.1.1");    # 1 bc. 1.1 > 1.001001

CPAN::Version->vlt("1.1","1.1");      # 0 bc. 1.1 not < 1.1

CPAN::Version->vcmp("1.1","1.1.1");   # 1 bc. first is larger

CPAN::Version->vcmp("1.1.1","1.1");   # -1 bc. first is smaller

CPAN::Version->readable(v1.2.3);      # "v1.2.3"
```

(continues on next page)

(continued from previous page)

```
CPAN::Version->vstring("v1.2.3");      # v1.2.3

CPAN::Version->float2vv(1.002003);     # "v1.2.3"
```

DESCRIPTION

This module mediates between some version that perl sees in a package and the version that is published by the CPAN indexer.

It's only written as a helper module for both CPAN.pm and CPANPLUS.pm.

As it stands it predates version.pm but has the same goal: make version strings visible and comparable.

LICENSE

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

5.2.12 Cairo

Perl bindings for cairo, the 2D vector graphics library.

This is the `Cairo` module, version 1.109 of the CPAN API: the same packages, methods, arguments and return values that Perl code written against `Cairo` already expects. Drawing goes to the `libcairo.so.2` installed on the machine, which is located when the module is first used; if that library is absent, the first Cairo call dies with a message naming the distribution package to install.

Objects and their lifetime

A cairo object is an ordinary blessed reference. Nothing has to be freed by hand: the underlying cairo resource is released when the last Perl reference to it goes out of scope. An object obtained from a getter - the pattern from `$cr->get_source`, the surface from `$cr->get_target` - stays valid on its own and may outlive the context it came from.

One class does not follow that shape: `Cairo::Path` is a tied array of hashes rather than an object with methods, so the path returned by `$cr->copy_path` is read with ordinary array and hash syntax.

```
use Cairo;

my $surface = Cairo::ImageSurface->create('argb32', 100, 50);
my $cr = Cairo::Context->create($surface);
$cr->set_source_rgb(0.2, 0.4, 0.8);
$cr->rectangle(10, 10, 80, 30);
$cr->fill;
$cr->set_source_rgb(1, 1, 1);
$cr->move_to(20, 32);
```

(continues on next page)

(continued from previous page)

```
$cr->show_text("cairo");
printf "%dx%d %s, status %s\n", $surface->get_width, $surface->get_
↪height,
    $surface->get_format, $cr->status;          # 100x50 argb32,␣
↪status success
$surface->write_to_png('out.png');
```

Enumerations are their nicknames as strings (`'argb32'`, `'success'`), as in the CPAN module. The classes are documented on the pages below: `Cairo::Context`, the surfaces, patterns, fonts, matrices, paths and regions.

Differences from upstream

None.

Modules

- `Cairo::Context` - the drawing context, cairo's central object
- `Cairo::Font` - The `Cairo::Font` family: font faces, scaled fonts and font options.
- `Cairo::FontFace`
- `Cairo::FontOptions`
- `Cairo::Format`
- `Cairo::FtFontFace`
- `Cairo::Gradient`
- `Cairo::ImageSurface`
- `Cairo::LinearGradient`
- `Cairo::Matrix` - the affine transform every drawing operation goes
- `Cairo::Pattern` - `Cairo::Pattern` and its subclasses - the sources a context paints with.
- `Cairo::PdfSurface`
- `Cairo::PsSurface`
- `Cairo::RadialGradient`
- `Cairo::RecordingSurface`
- `Cairo::Region` - a set of integer-aligned rectangles
- `Cairo::ScaledFont`
- `Cairo::SolidPattern`
- `Cairo::Surface` - `Cairo::Surface` and its subclasses - the things cairo draws onto.
- `Cairo::SurfacePattern`
- `Cairo::SvgSurface`

- `Cairo::ToyFontFace`

Functions

Library information

`version`

Returns the version of the cairo library in use, as a single integer encoded `major * 10000 + minor * 100 + micro`.

Also available as `Cairo::lib_version`.

`version_string`

Returns the version of the cairo library in use as a printable string, for example "1.18.4".

Also available as `Cairo::lib_version_string`.

`VERSION_ENCODE`

Encodes a cairo version triple into the single integer that `Cairo::LIB_VERSION` returns: `major * 10000 + minor * 100 + micro`.

Also available as `Cairo::LIB_VERSION_ENCODE`.

`HAS_PNG_FUNCTIONS`

Returns true if the cairo library in use can read and write PNG files, that is, if `Cairo::ImageSurface->create_from_png` and `$surface->write_to_png` will work.

`HAS_PS_SURFACE`

Returns true if the cairo library in use can write PostScript, that is, if `Cairo::PsSurface->create` will work.

`HAS_PDF_SURFACE`

Returns true if the cairo library in use can write PDF, that is, if `Cairo::PdfSurface->create` will work.

`HAS_SVG_SURFACE`

Returns true if the cairo library in use can write SVG, that is, if `Cairo::SvgSurface->create` will work.

`HAS_RECORDING_SURFACE`

Returns true if the cairo library in use has recording surfaces (`Cairo::RecordingSurface->create`).

HAS_FT_FONT

Returns true if the cairo library in use can take its fonts from FreeType (`Cairo::FtFontFace->create`).

Cairo::Context

`Cairo::Context` - the drawing context, cairo's central object.

A context is made for a surface with `Cairo::Context->create($surface)` and holds the whole drawing state: the current source, line width, dash pattern, font, transformation matrix, clip region and current path. The drawing operations - `fill`, `stroke`, `paint`, `show_text` - apply that state to the context's target surface. `save` and `restore` bracket temporary changes to it.

The getters that hand back another object - `get_source`, `get_target`, `get_group_target`, `get_font_face`, `get_scaled_font` - return the context's own object rather than a copy, so changing it changes what the context will draw with. Each keeps itself alive, so it is still usable after the context has gone. `get_matrix`, `get_font_matrix` and `get_font_options` are the exception: those three return a fresh copy that is yours to modify.

Functions

Other Functions

create

Creates a drawing context for `$surface` and returns it as a `Cairo::Context`.

status

Returns the context's error state as a status name such as `'success'` or `'invalid-restore'`.

pop_group

Ends the group started by `push_group` and returns everything drawn inside it as a pattern.

get_source

Returns the context's current source as a `Cairo::Pattern` subclass matching the kind of source it is.

set_dash

Sets the dash pattern used by `stroke`.

get_dash

Returns the current dash pattern as a flat list, the offset first and the dash lengths after it.

get_matrix

Returns the context's current transformation matrix as a `Cairo::Matrix`.

has_current_point

Returns true if the current path has a current point, that is, if a relative operation such as `rel_line_to` has something to start from.

get_current_point

Returns the current point of the path as the two-element list `($x, $y)`, in user-space coordinates.

in_clip

Returns true if the user-space point `($x, $y)` is inside the area that the current clip would let drawing through.

select_font_face

Selects a font by family name for subsequent `show_text` and `text_extents` calls.

get_font_matrix

Returns the context's current font matrix as a `Cairo::Matrix`.

get_font_options

Returns the context's current font options as a fresh `Cairo::FontOptions`.

get_scaled_font

Returns the context's current scaled font - the font face together with the size and transformation in force - as a `Cairo::ScaledFont`.

show_text

Draws `$text` at the current point using the current font and source, then advances the current point past what was drawn.

text_path

Adds the outlines of `$text` to the current path instead of drawing it, so the text can then be filled, stroked or clipped with.

get_font_face

Returns the context's current font face as a `Cairo::FontFace` subclass matching the kind of face it is.

font_extents

Returns the metrics of the current font as a hash reference with the keys `ascent`, `descent`, `height`, `max_x_advance` and `max_y_advance`.

text_extents

Measures `$text` in the current font and returns a hash reference with the keys `x_bearing`, `y_bearing`, `width`, `height`, `x_advance` and `y_advance`.

get_target

Returns the surface the context draws onto, as the concrete `Cairo::Surface` subclass it really is.

get_group_target

Returns the surface drawing currently goes to: the innermost group's surface while a `push_group` is open, and otherwise the same surface `get_target` returns.

tag_begin

Opens a tagged region, used to attach structure such as links and document outlines to PDF output.

tag_end

Closes the tagged region opened by `tag_begin` with the same `$tag_name`.

show_glyphs

Draws a list of already-positioned glyphs with the current font and source.

glyph_path

Adds the outlines of a list of positioned glyphs to the current path instead of drawing them, so they can be filled, stroked or clipped with.

glyph_extents

Measures a list of positioned glyphs and returns a hash reference with the same keys as `text_extents`: `x_bearing`, `y_bearing`, `width`, `height`, `x_advance` and `y_advance`.

show_text_glyphs

Draws positioned glyphs while also recording which characters of `$text` they came from, so that backends such as PDF and SVG can make the result selectable and searchable.

copy_path

Returns a copy of the current path as a `Cairo::Path`.

copy_path_flat

Returns a copy of the current path as a `Cairo::Path`, with every curve replaced by a chain of line segments.

append_path

Appends a path to the context's current path.

copy_clip_rectangle_list

Returns the current clip region as a plain list of hash references, each with the keys `x`, `y`, `width` and `height` in user space.

select_font_face

Selects a font by family name for subsequent `show_text` and `text_extents` calls.

$cr->select_font_face('Sans', 'normal', 'bold');

`$family` is a font family such as `'Sans'` or `'Serif'`, `$slant` is one of `'normal'`, `'italic'` or `'oblique'`, and `$weight` is `'normal'` or `'bold'`. The family is interpreted as text, so accented and non-Latin names work. This is the simple font-selection interface; `set_font_face` takes a font object instead. Wraps `cairo_select_font_face`.

show_text

Draws `$text` at the current point using the current font and source, then advances the current point past what was drawn.

$cr->move_to(10, 40); $cr->show_text('Hello');

The string is taken as text, so non-ASCII characters are drawn correctly. This is the convenience interface meant for simple labels; use `show_glyphs` or a text-shaping library for real typography. Wraps `cairo_show_text`.

tag_begin

Opens a tagged region, used to attach structure such as links and document outlines to PDF output.

$cr->tag_begin('Link', "uri='https://example.org'"); $cr->show_text('example'); $cr->tag_end('Link');

`$attributes` is a single string of `name=value` pairs. Backends other than PDF ignore tags. Needs cairo 1.16 or newer; on an older library the call dies saying so. Wraps `cairo_tag_begin`.

show_glyphs

Draws a list of already-positioned glyphs with the current font and source.

$cr->show_glyphs({ index => 43, x => 10, y => 40 }, ...);

Each glyph is a hash reference with the keys `index` (the glyph number within the font, not a character), `x` and `y` (its position in user space). Missing keys count as 0. Unlike `show_text` this does not advance the current point, since every glyph carries its own position. Returns nothing. Wraps `cairo_show_glyphs`.

show_text_glyphs

Draws positioned glyphs while also recording which characters of `$text` they came from, so that backends such as PDF and SVG can make the result selectable and searchable.

$cr->show_text_glyphs($text, @glyphs, @clusters, ['backward']);

`\@glyphs` holds `{ index, x, y }` hash references as for `show_glyphs`. `\@clusters` holds `{ num_bytes, num_glyphs }` hash references, each mapping that many bytes of `$text` onto that many glyphs; the clusters must together account for the whole string and the whole glyph list. `$cluster_flags` is an array reference of flag names (only `'backward'` is defined), a single flag name, or `undef` for none. Note that unlike the glyph and cluster lists, which are passed by reference, `show_glyphs` takes its glyphs as a flat list. Returns nothing. Needs cairo 1.8 or newer; on an older library the call dies saying so. Wraps `cairo_show_text_glyphs`.

copy_path

Returns a copy of the current path as a `Cairo::Path`.

A `Cairo::Path` is a tied array of hashes, so it is read with ordinary Perl syntax rather than through methods:

```
my $path = $cr->copy_path;
for my $elem (@$path) {
    print "$elem->{type}\n";      # 'move-to', 'line-to', ...
}
```

The path is a snapshot: later drawing on the context does not change it, and it may be handed back to append_path on this or any other context. Wraps `cairo_copy_path`.

append_path

Appends a path to the context's current path.

$cr->append_path($path);

`$path` may be a `Cairo::Path` obtained from `copy_path` or `copy_path_flat`, or a plain array reference of `{ type => ..., points => [...] }` hashes built by hand. Either way the path is copied into the context, so the argument stays usable afterwards. Returns nothing. Wraps `cairo_append_path`.

copy_clip_rectangle_list

Returns the current clip region as a plain list of hash references, each with the keys `x`, `y`, `width` and `height` in user space.

for my $rect ($cr->copy_clip_rectangle_list) { printf "%g %g %g %g\n", @$rect{qw(x y width height)}; }

A clip that cannot be expressed as a list of rectangles makes the call die with `$@` set to a status name such as `'clip-not-representable'`, so wrap it in `eval` when the clip may come from arbitrary paths. Needs cairo 1.4 or newer; on an older library the call dies saying so. Wraps `cairo_copy_clip_rectangle_list`.

Cairo::Font

The `Cairo::Font` family: font faces, scaled fonts and font options.

This is the font half of the Cairo API. `Cairo::ToyFontFace` picks a font by family name, slant and weight; `Cairo::ScaledFont` is that face fixed at a particular size and transform, and is what answers questions about measurements - font extents, text extents, glyph extents, and turning a string into glyphs. `Cairo::FontOptions` carries the rendering settings (antialiasing, subpixel order, hinting, and on cairo 1.18 the color-font controls) that a context or a scaled font uses. `Cairo::FontFace` is the base class the face classes share, and `Cairo::FtFontFace` is the FreeType face - see its `create` for why it cannot be built here.

Font faces come back as subclasses

Anything returning a font face - `ScaledFont::get_font_face`, `Context::get_font_face` - blesses it into the class matching the font backend cairo reports: a toy face into `Cairo::ToyFontFace`, a FreeType face into `Cairo::FtFontFace`, anything else into the base `Cairo::FontFace`. All of them answer `status`, `get_type` and `DESTROY`, which the subclasses inherit.

Object lifetime

Font faces, scaled fonts and font options are reference-counted by cairo. An object you create owns its reference, and a getter that hands back an object its parent also holds - `ScaledFont::get_font_face` - takes a reference of its own, so what you get stays valid for as long as you hold it. Every object releases its reference when Perl destroys it; nothing has to be freed by hand.

Methods report their cairo version at call time

Each method looks up its cairo entry point in the installed libcairo the first time it is used. A method that libcairo does not provide croaks then and there, naming the version it needs (for instance `Cairo: text_extents requires libcairo >= 1.2`), rather than failing when the module is loaded.

Cairo::FontFace

Functions

Other Functions

`status`

Returns the font face's error status as a nickname string, `success` when the face is usable.

`get_type`

Returns the backend behind this font face as a nickname string: `toy`, `ft`, `win32`, `quartz`, `user` or `dwrite`. Needs libcairo 1.2 or newer.

Cairo::FontOptions

Functions

Other Functions

`create`

Creates a font options object with everything at its default (cairo_font_options_create): `Cairo::FontOptions->create`. Takes no arguments.

`status`

Returns the font options' error status as a nickname string, `success` when they are usable.

`merge`

Merges `$other` into `$opts` in place: every option that `$other` sets to something other than its default replaces the one in `$opts`. The argument is required. Returns nothing.

`equal`

Returns true if `$opts` and `$other` hold the same settings. The argument is required.

hash

Returns a hash value for the options, computed from the settings they hold, for use as a key when caching option objects.

set_color_mode

Sets whether color glyphs are used when the font has them: `default`, `no-color` or `color`. Needs libcairo 1.18 or newer, and croaks with `Cairo: set_color_mode requires cairo >= 1.18` on anything older.

get_color_mode

Returns the color-glyph mode as a nickname string. Needs libcairo 1.18 or newer, and croaks with `Cairo: get_color_mode requires cairo >= 1.18` on anything older.

set_color_palette

Chooses which color palette a color font uses, by index; 0 is the font's default palette. Needs libcairo 1.18 or newer, and croaks with `Cairo: set_color_palette requires cairo >= 1.18` on anything older.

get_color_palette

Returns the index of the color palette in use. Needs libcairo 1.18 or newer, and croaks with `Cairo: get_color_palette requires cairo >= 1.18` on anything older.

set_custom_palette_color

Overrides one entry of the color palette: `$opts->set_custom_palette_color($index, $r, $g, $b, $a)`. All five arguments are required. Needs libcairo 1.18 or newer, and croaks with `Cairo: set_custom_palette_color requires cairo >= 1.18` on anything older.

get_custom_palette_color

Reads back one overridden palette entry, returning the list `($status, $r, $g, $b, $a)`. The status nickname comes first and is `success` only when that entry has been overridden - an index with no override yields `invalid-index`, and the four components are then meaningless. Needs libcairo 1.18 or newer, and croaks with `Cairo: get_custom_palette_color requires cairo >= 1.18` on anything older.

Cairo::Format

Functions

Other Functions

stride_for_width

Returns the number of bytes one row of pixels occupies for a given format and width - the `$stride` that `create_for_data` wants.

stride_for_width

Returns the number of bytes one row of pixels occupies for a given format and width - the `$stride` that `create_for_data` wants.

my $stride = Cairo::Format::stride_for_width('argb32', $width);

Called as a plain FUNCTION, not as a method: write `Cairo::Format::stride_for_width(...)`, not `Cairo::Format->stride_for_width(...)`. The result is usually more than width times bytes-per-pixel, because rows are padded for alignment. Returns a negative number for a format or width cairo cannot handle. Wraps `cairo_format_stride_for_width`.

Cairo::FtFontFace

Functions

Other Functions

create

Not available: calling it always croaks.

create

Not available: calling it always croaks.

A FreeType font face can only be built from a `Font::FreeType::Face` object, and pperl does not provide `Font::FreeType`, so nothing can produce the argument this method needs. It will work once `Font::FreeType` is available as a native libfreetype binding. Until then only the class itself exists: `Cairo::FtFontFace` still inherits from `Cairo::FontFace`, so `isa` and `can` answer correctly, and a face cairo reports as a FreeType one is still blessed into this class.

Cairo::Gradient

Functions

Other Functions

add_color_stop_rgb

Adds an opaque color stop to the gradient at `$offset`, a position from 0 to 1 along the gradient's control vector: `$gradient->add_color_stop_rgb($offset, $r, $g, $b)`. Returns nothing.

add_color_stop_rgba

Adds a translucent color stop to the gradient at `$offset`: `$gradient->add_color_stop_rgba($offset, $r, $g, $b, $a)`. Returns nothing.

get_color_stops

Returns every color stop of the gradient, one five-element arrayref `[$offset, $r, $g, $b, $a]` per stop, in the order cairo holds them. On a pattern that is not a gradient it croaks with `pattern-type-mismatch`. Needs libcairo 1.4 or newer.

Cairo::ImageSurface

Functions

Other Functions

create

Creates an in-memory image surface of `$width` by `$height` pixels and returns it as a `Cairo::ImageSurface`.

create_for_data

Creates an image surface that draws directly into a scalar you already hold, rather than into memory of its own.

get_data

Returns the surface's pixels as a string of `height * stride` bytes, or `undef` if the surface has no accessible pixel data.

get_format

Returns the surface's pixel format as a name such as `'argb32'` or `'rgb24'`.

create_from_png

Reads a PNG file and returns its contents as a `Cairo::ImageSurface`.

create_from_png_stream

Reads a PNG by asking a callback of yours for successive chunks of it, and returns the result as a `Cairo::ImageSurface`.

create_for_data

Creates an image surface that draws directly into a scalar you already hold, rather than into memory of its own.

my $surface = Cairo::ImageSurface->create_for_data($data, 'argb32', $width, $height, $stride);

`$stride` is the number of bytes between the start of one row of pixels and the start of the next; `Cairo::Format::stride_for_width` computes the right value. `$data` must already be at least `$height * $stride` bytes long.

The scalar is NOT copied: the surface writes into its bytes in place. That makes the scalar the pixels, but it also means the scalar has to stay alive and unmodified for as long as the surface exists. Letting it go out of scope first, or assigning to it, leaves the surface drawing into memory that is no longer there. Wraps `cairo_image_surface_create_for_data`.

create_from_png

Reads a PNG file and returns its contents as a `Cairo::ImageSurface`.

my $surface = Cairo::ImageSurface->create_from_png('logo.png');

A file that cannot be read or is not a PNG still yields a surface, but one whose `status` reports the error rather than `'success'`, so check it before drawing. Dies if the cairo installed on this machine was built without PNG support; `Cairo::HAS_PNG_FUNCTIONS` tells you in advance. Wraps `cairo_image_surface_create_from_png`.

create_from_png_stream

Reads a PNG by asking a callback of yours for successive chunks of it, and returns the result as a `Cairo::ImageSurface`.

my $surface = Cairo::ImageSurface->create_from_png_stream(sub { my ($data, $length) = @_; read($fh, my $buf, $length); return $buf; });

The callback is called with two arguments - the optional `$data` value you passed after it, and the number of bytes wanted - and must RETURN the bytes it read. To report a failure, die with a cairo status name such as `'read-error'`; the surface then comes back with that status rather than the call dying. The callback is only used during construction and is not retained afterwards. Dies if the cairo installed on this machine was built without PNG support. Wraps `cairo_image_surface_create_from_png_stream`.

Cairo::LinearGradient

Functions

Other Functions

`create`

Creates a linear gradient running from `($x0, $y0)` to `($x1, $y1)`: `Cairo::LinearGradient->create($x0, $y0, $x1, $y1)`. The gradient has no color until you add stops with `add_color_stop_rgb/_rgba`.

get_points

Returns the linear gradient's two endpoints as the four-element list `($x0, $y0, $x1, $y1)`. On a pattern that is not a linear gradient it croaks with `pattern-type-mismatch`. Needs libcairo 1.4 or newer.

Cairo::Matrix

`Cairo::Matrix` - the affine transform every drawing operation goes through.

A matrix holds the six numbers `(xx, yx, xy, yy, x0, y0)` of a 2D affine transform. The `Cairo::Matrix->init*` constructors build one, `translate`/`scale`/`rotate` modify it in place, `multiply` combines two, `invert` inverts it in place, and `transform_point` / `transform_distance` apply it to a coordinate pair.

Unlike the other cairo objects a matrix is a plain value, not a shared reference-counted handle. Every method that hands one back - `Context::get_matrix`, `Pattern::get_matrix`, `ScaledFont::get_font_matrix` - returns a fresh copy, so changing the matrix you got back never changes the object it came from. The copy is released when Perl destroys it; nothing has to be freed by hand.

The matrix API is stable since cairo 1.0, so no method here depends on a particular libcairo version.

Functions

Other Functions

init

Builds a new matrix from the six affine components, in cairo's order: `Cairo::Matrix->init($xx, $yx, $xy, $yy, $x0, $y0)`. All six are required; a shorter argument list croaks with a usage message.

init_identity

Builds a new matrix set to the identity transform: `Cairo::Matrix->init_identity`.

init_translate

Builds a new matrix that translates by `($tx, $ty)`: `Cairo::Matrix->init_translate($tx, $ty)`.

init_scale

Builds a new matrix that scales by `$sx` in x and `$sy` in y: `Cairo::Matrix->init_scale($sx, $sy)`.

init_rotate

Builds a new matrix that rotates by `$radians`: `Cairo::Matrix->init_rotate($radians)`.

translate

Applies a translation of `($tx, $ty)` to `$matrix` in place (cairo_matrix_translate). Returns nothing.

scale

Applies a scale of `($sx, $sy)` to `$matrix` in place (cairo_matrix_scale). Returns nothing.

rotate

Applies a rotation of `$radians` to `$matrix` in place (cairo_matrix_rotate). Returns nothing.

invert

Inverts `$matrix` in place (cairo_matrix_invert) and returns the outcome as a status nickname: `success`, or `invalid-matrix` if the matrix is not invertible.

multiply

Returns a **new** `Cairo::Matrix` holding the product of `$a` and `$b`.

transform_distance

Transforms the distance vector `($dx, $dy)` by `$matrix` and returns the transformed pair as a two-element list. Only the matrix's linear part is applied - the translation is ignored, which is what makes this a distance rather than a point.

transform_point

Transforms the point `($x, $y)` by `$matrix` and returns the transformed pair as a two-element list. The full affine transform is applied, translation included.

Cairo::Pattern

`Cairo::Pattern` and its subclasses - the sources a context paints with.

A pattern is what `Cairo::Context` draws with: a flat color (`Cairo::SolidPattern`), another surface used as an image or a tile (`Cairo::SurfacePattern`), or a color ramp - `Cairo::LinearGradient` and `Cairo::RadialGradient`, which take their color-stop methods from the abstract `Cairo::Gradient`. Settings shared by all of them (the pattern matrix, the extend mode, the filter) live on the base `Cairo::Pattern`.

Constructors return the concrete subclass

A constructor blesses its result into the class matching what cairo actually built, not into the class you called it on: `Cairo::SolidPattern->create_rgb` hands back a `Cairo::SolidPattern`, `Cairo::LinearGradient->create` a `Cairo::LinearGradient`, and anything cairo reports as some other kind comes back as the base `Cairo::Pattern`. `get_surface` behaves the same way, returning the surface blessed into its own class (`Cairo::ImageSurface`, `Cairo::PdfSurface` and so on).

Object lifetime

Patterns are reference-counted by cairo. A pattern you create owns its reference, and the surface `get_surface` returns takes one of its own, so it stays alive for as long as you hold it. Every object drops its reference when Perl destroys it; nothing has to be freed by hand.

Gradient dithering (cairo 1.18)

`set_dither` and `get_dither` control the dithering cairo applies to a gradient to hide banding. They need libcairo 1.18 or newer; on anything older they croak with `requires cairo >= 1.18`.

Functions

Other Functions

`set_matrix`

Sets the pattern's transformation matrix (cairo_pattern_set_matrix). `$matrix` is a `Cairo::Matrix`, and it maps user space to pattern space. Returns nothing.

`get_matrix`

Returns the pattern's transformation matrix as a fresh `Cairo::Matrix`. It is a copy: changing it does not change the pattern, which needs a `set_matrix` call.

`status`

Returns the pattern's error status as a nickname string, `success` when the pattern is usable.

`set_dither`

Sets how a gradient is dithered, to hide banding: `none`, `default`, `fast`, `good` or `best`. Needs libcairo 1.18 or newer, and croaks with `Cairo: set_dither requires cairo >= 1.18` on anything older.

get_dither

Returns the pattern's dither setting as a nickname string. Needs libcairo 1.18 or newer, and croaks with `Cairo: get_dither requires cairo >= 1.18` on anything older.

Cairo::PdfSurface

Functions

Other Functions

restrict_to_version

Restricts the surface to features present in the given PDF version, named as `'1-4'`, `'1-5'`, `'1-6'` or `'1-7'`.

set_metadata

Sets one document metadata field of the PDF.

set_page_label

Sets the label a PDF viewer shows for the current page, instead of its sheet number.

set_thumbnail_size

Asks for a thumbnail image of the current page, at most `$width` by `$height` pixels, to be embedded in the PDF.

add_outline

Adds an entry to the PDF's document outline (its bookmark tree) and returns the entry's id, for use as the parent of further entries.

set_metadata

Sets one document metadata field of the PDF.

$pdf->set_metadata('title', 'Quarterly report');

`$metadata` names the field: `'title'`, `'author'`, `'subject'`, `'keywords'`, `'creator'`, `'create-date'` or `'mod-date'`. The two date fields take an ISO 8601 string. Needs cairo 1.16 or newer; on an older library the call dies saying so. Wraps `cairo_pdf_surface_set_metadata`.

add_outline

Adds an entry to the PDF's document outline (its bookmark tree) and returns the entry's id, for use as the parent of further entries.

my $id = $pdf->add_outline(Cairo::PdfSurface::OUTLINE_ROOT, 'Chapter 1', "page=1", 0); $pdf->add_outline($id, 'Section 1.1', "page=2", 0);

`$parent_id` is `Cairo::PdfSurface::OUTLINE_ROOT` for a top-level entry, `$link_attribs` a string of `name=value` pairs saying where the entry points, and `$flags` a plain INTEGER bitmask - unlike the other enumerated arguments in this module, it is not a name. Needs cairo 1.16 or newer; on an older library the call dies saying so. Wraps `cairo_pdf_surface_add_outline`.

Cairo::PsSurface

Functions

Other Functions

dsc_comment

Emits a raw DSC comment into the PostScript output.

restrict_to_level

Restricts the surface to features present in the given PostScript language level, named as `'2'` or `'3'`.

set_eps

Switches the surface between plain PostScript and Encapsulated PostScript output.

get_eps

Returns 1 if the surface is producing Encapsulated PostScript, and 0 if it is producing plain PostScript.

Cairo::RadialGradient

Functions

Other Functions

create

Creates a radial gradient between two circles, the first centered on `($cx0, $cy0)` with radius `$r0` and the second on `($cx1, $cy1)` with radius `$r1`: `Cairo::RadialGradient->create($cx0, $cy0, $r0, $cx1, $cy1, $r1)`. The gradient has no color until you add stops.

get_circles

Returns the radial gradient's two circles as the six-element list `($x0, $y0, $r0, $x1, $y1, $r1)`. On a pattern that is not a radial gradient it croaks with `pattern-type-mismatch`. Needs libcairo 1.4 or newer.

Cairo::RecordingSurface

Functions

Other Functions

create

Creates a surface that records the drawing operations performed on it instead of rendering them, so they can be replayed later onto any other surface at any resolution.

ink_extents

Returns the area actually covered by ink so far, as the four-element list `($x0, $y0, $width, $height)`.

get_extents

Returns the bounds the surface was created with, as a hash reference with the keys `x`, `y`, `width` and `height`.

create

Creates a surface that records the drawing operations performed on it instead of rendering them, so they can be replayed later onto any other surface at any resolution.

my $rec = Cairo::RecordingSurface->create('color-alpha', { x => 0, y => 0, width => 100, height => 100 }); my $unbounded = Cairo::RecordingSurface->create('color-alpha');

`$content` is `'color'`, `'alpha'` or `'color-alpha'`. The second argument bounds the recording: a plain hash reference with the keys `x`, `y`, `width` and `height`, or `undef`/omitted for an unbounded surface that accepts drawing anywhere. Replay a recording by using it as a source, for instance through `$cr->set_source_surface`. Wraps `cairo_recording_surface_create`.

Cairo::Region

`Cairo::Region` - a set of integer-aligned rectangles.

Regions describe pixel-aligned areas: build one from rectangles, combine it with another using `intersect`, `union`, `subtract` or `xor`, move it with `translate`, and ask whether a point or a rectangle falls inside it. A region owns its cairo object and releases it when Perl destroys the Perl object, so nothing has to be freed by hand.

Requires cairo >= 1.10

The region API arrived in cairo 1.10. If the installed libcairo is older, every method here croaks with `Cairo::Region requires cairo >= 1.10` when it is called.

Rectangles are plain hashrefs

Everywhere a rectangle is passed in or handed back it is an unblessed hashref with exactly the keys `x`, `y`, `width` and `height`, and the values are **integers** - a region is pixel-aligned, so there are no fractional coordinates. In a rectangle you pass in, a missing key or an `undef` value counts as 0; an argument that is not a hash reference croaks with `cairo_rectangle_int_t must be a hash reference`.

Functions

Other Functions

`create`

Creates a region, dispatching on how many rectangles you pass:

`status`

Returns the region's error status as a nickname string, `success` when the region is usable.

`get_extents`

Returns the smallest rectangle containing the whole region, as a hashref `{x, y, width, height}`.

`num_rectangles`

Returns how many rectangles the region is made of.

`get_rectangle`

Returns the `$nth` rectangle of the region (counting from 0) as a hashref `{x, y, width, height}`.

`is_empty`

Returns true if the region covers nothing at all.

`contains_point`

Returns true if the integer point `($x, $y)` lies inside the region.

`contains_rectangle`

Reports how `$rect` sits relative to the region, as a nickname string: `in` when the rectangle is entirely inside, `out` when it is entirely outside, `part` when it is partly in.

equal

Returns true if `$region` and `$other` cover exactly the same area.

translate

Moves the region by the integer offset `($dx, $dy)`, in place. Returns nothing.

create

Creates a region, dispatching on how many rectangles you pass:

```
Cairo::Region->create()          # empty region
Cairo::Region->create($rect)     # one rectangle
Cairo::Region->create(@rects)    # the union of several rectangles
```

Each rectangle is a hashref `{x, y, width, height}` with integer values.

Cairo::ScaledFont

Functions

Other Functions

create

Creates a scaled font - a font face fixed at one size and transform (cairo_scaled_font_create): `Cairo::ScaledFont->create($face, $font_matrix, $ctm, $options)`. `$font_matrix` sizes and shapes the glyphs, `$ctm` is the transform they will be drawn under, both `Cairo::Matrix`, and `$options` is a `Cairo::FontOptions`. All four arguments are required; a shorter list croaks with a usage message.

status

Returns the scaled font's error status as a nickname string, `success` when it is usable.

get_type

Returns the backend behind this scaled font as a nickname string: `toy`, `ft`, `win32`, `quartz`, `user` or `dwrite`. Needs libcairo 1.2 or newer.

extents

Returns the font's own metrics as a hashref with the keys `ascent`, `descent`, `height`, `max_x_advance` and `max_y_advance`. These describe the font at this size, independent of any particular string.

text_extents

Measures a string in this font and returns a hashref with the keys `x_bearing`, `y_bearing`, `width`, `height`, `x_advance` and `y_advance`. The string is taken as UTF-8. It is required; without it the call croaks with a usage message. Needs libcairo 1.2 or newer.

get_font_face

Returns the font face this scaled font was made from, blessed into the class matching its backend (`Cairo::ToyFontFace`, `Cairo::FtFontFace`, or the base `Cairo::FontFace`). The face stays valid for as long as you hold it, even if the scaled font goes away. Needs libcairo 1.2 or newer.

get_font_matrix

Returns the font matrix the scaled font was created with, as a fresh `Cairo::Matrix`. It is a copy: changing it does not change the scaled font. Needs libcairo 1.2 or newer.

get_ctm

Returns the transform matrix the scaled font was created with, as a fresh `Cairo::Matrix`. It is a copy: changing it does not change the scaled font. Needs libcairo 1.2 or newer.

get_scale_matrix

Returns the scale matrix - the font matrix combined with the transform - as a fresh `Cairo::Matrix`. It is a copy: changing it does not change the scaled font. Needs libcairo 1.8 or newer.

get_font_options

Returns the rendering options the scaled font was created with, as a fresh `Cairo::FontOptions`. It is a separate object: changing it does not change the scaled font. Needs libcairo 1.2 or newer.

glyph_extents

Measures a run of glyphs in this font and returns a hashref with the keys `x_bearing`, `y_bearing`, `width`, `height`, `x_advance` and `y_advance`.

text_to_glyphs

Converts a string into the glyphs this font would use to draw it, laid out starting at `($x, $y)`: `$sf->text_to_glyphs($x, $y, $utf8)`.

text_to_glyphs

Converts a string into the glyphs this font would use to draw it, laid out starting at `($x, $y)`: `$sf->text_to_glyphs($x, $y, $utf8)`.

Returns a **list**: `($status, $glyphs, $clusters, $cluster_flags)`. The status nickname is always returned; the other three come back only when it is `success`, so check the status before using them. `$glyphs` is an arrayref of `{index, x, y}` hashrefs, `$clusters` an arrayref of `{num_bytes, num_glyphs}` hashrefs mapping the text to those glyphs, and `$cluster_flags` an arrayref of set flag names (`backward`, or empty).

All three arguments are required; a shorter list croaks with a usage message. Needs libcairo 1.8 or newer.

Cairo::SolidPattern

Functions

Other Functions

create_rgb

Creates an opaque solid-color pattern from the three components (cairo_pattern_create_rgb): `Cairo::SolidPattern->create_rgb($r, $g, $b)`. Each is in the range 0 to 1. All three are required; a shorter argument list croaks with a usage message. The result is blessed `Cairo::SolidPattern`.

create_rgba

Creates a translucent solid-color pattern (cairo_pattern_create_rgba): `Cairo::SolidPattern->create_rgba($r, $g, $b, $a)`. All four components are in the range 0 to 1 and all four are required.

get_rgba

Returns the solid pattern's color as the four-element list `($r, $g, $b, $a)`. On a pattern that is not solid it croaks with `pattern-type-mismatch`. Needs libcairo 1.4 or newer.

Cairo::Surface

`Cairo::Surface` and its subclasses - the things cairo draws onto.

A surface is where drawing ends up, and its class decides in what form: `Cairo::ImageSurface` for pixels in memory, `Cairo::PdfSurface`, `Cairo::PsSurface` and `Cairo::SvgSurface` for the corresponding file formats, and `Cairo::RecordingSurface` for a replayable record of the drawing operations themselves. All of them inherit from `Cairo::Surface`, which carries the methods common to every backend.

Every constructor returns the concrete subclass, never the bare `Cairo::Surface`, so the surface you get back from `create_similar` or from `$cr->get_target` already responds to the methods of whatever kind of surface it turned out to be. Draw on a surface by creating a context for it with `Cairo::Context->create($surface)`.

Methods that name a string constant - `status`, `get_type`, `get_content`, `get_format` - return a short name such as `'success'`, `'image'`, `'color-alpha'` or `'argb32'`, never a number.

When a backend is missing

Which backends exist depends on how the cairo installed on this machine was built. A cairo without PDF, PostScript, SVG or PNG support still loads, but the methods belonging to the missing backend die with a message saying so, for example "libcairo was built without PDF support". `Cairo::HAS_PNG_FUNCTIONS` reports the PNG case up front. The PDF metadata and outline methods additionally need cairo 1.16 or newer.

Constants

The MIME types `set_mime_data` and `supports_mime_type` take are constants of `Cairo::Surface`: `MIME_TYPE_JPEG` (`"image/jpeg"`), `MIME_TYPE_PNG`, `MIME_TYPE_JP2`, `MIME_TYPE_URI`, `MIME_TYPE_UNIQUE_ID`, `MIME_TYPE_JBIG2`, `MIME_TYPE_JBIG2_GLOBAL`, `MIME_TYPE_JBIG2_GLOBAL_ID`, `MIME_TYPE_CCITT_FAX`, `MIME_TYPE_CCITT_FAX_PARAMS`, `MIME_TYPE_EPS` and `MIME_TYPE_EPS_PARAMS`. `Cairo::PdfSurface::OUTLINE_ROOT` (0) is the parent id to give `add_outline` for a top-level bookmark.

Functions

Other Functions

`status`

Returns the surface's error state as a status name such as `'success'`, `'file-not-found'` or `'no-memory'`.

`get_type`

Returns the kind of surface this is, as a name such as `'image'`, `'pdf'`, `'ps'`, `'svg'` or `'recording'`.

`get_content`

Returns what the surface stores, as one of `'color'`, `'alpha'` or `'color-alpha'`.

`write_to_png`

Writes the surface to `$filename` as a PNG and returns a status name.

`create_similar`

Creates a new surface of the same backend as an existing one, sized `$width` by `$height`, and returns it as the concrete subclass.

set_device_offset

Shifts the surface's origin, so that user-space point (0, 0) lands at device point `($x, $y)` instead.

get_device_offset

Returns the surface's device offset as the two-element list `($x_offset, $y_offset)`.

set_fallback_resolution

Sets the resolution, in pixels per inch, at which a vector backend rasterises anything it cannot express directly.

get_fallback_resolution

Returns the surface's fallback resolution as the two-element list `($x_ppi, $y_ppi)`.

get_font_options

Returns the font rendering options this surface prefers, as a fresh `Cairo::FontOptions`.

mark_dirty_rectangle

Tells cairo that the given rectangle of the surface was changed behind its back, so that any cached copy of that area is discarded.

has_show_text_glyphs

Returns 1 if this surface can record the character-to-glyph mapping that `$cr->show_text_glyphs` supplies, and 0 if it cannot.

create_for_rectangle

Creates a surface that is a window onto part of another one.

supports_mime_type

Returns 1 if the surface can embed data of the given MIME type unchanged in its output, and 0 if it cannot.

write_to_png_stream

Writes the surface as a PNG by handing successive chunks of it to a callback of yours, and returns a status name.

set_mime_data

Attaches the original compressed form of an image to the surface, so that a backend able to embed it does so instead of re-encoding the pixels, and returns a status name.

get_mime_data

Returns the data previously attached with `set_mime_data` for the given MIME type, or `undef` if the surface carries none.

create_similar

Creates a new surface of the same backend as an existing one, sized `$width` by `$height`, and returns it as the concrete subclass.

It can be called either way round, as a class method naming the model surface or as a method on the model surface itself:

```
Cairo::Surface->create_similar($other, $content, $width, $height)
$other->create_similar($content, $width, $height)
```

`$content` is `'color'`, `'alpha'` or `'color-alpha'`. The result is a blank surface, not a copy of `$other`; only the backend and its device characteristics are inherited, which is what makes it the efficient choice for an intermediate you will later composite back. Wraps `cairo_surface_create_similar`.

create_for_rectangle

Creates a surface that is a window onto part of another one.

my $part = Cairo::Surface->create_for_rectangle($target, $x, $y, $width, $height);

Drawing on the result goes through to `$target`, clipped to the given rectangle and with its origin moved to the rectangle's corner. The coordinates are in `$target`'s device space and need not be whole pixels. `$target` is kept alive by the returned surface. Wraps `cairo_surface_create_for_rectangle`.

supports_mime_type

Returns 1 if the surface can embed data of the given MIME type unchanged in its output, and 0 if it cannot.

if ($pdf->supports_mime_type(Cairo::Surface::MIME_TYPE_JPEG)) { $image->set_mime_data(Cairo::Surface::MIME_TYPE_JPEG, $jpeg); }

Worth asking before `set_mime_data`, since attaching data a backend cannot use just wastes memory. Wraps `cairo_surface_supports_mime_type`.

write_to_png_stream

Writes the surface as a PNG by handing successive chunks of it to a callback of yours, and returns a status name.

$surface->write_to_png_stream(sub { my ($data, $chunk) = @_; print {$fh} $chunk; });

The callback is called with two arguments: the optional `$data` value you passed after it, and the chunk of PNG bytes to consume. Its return value is ignored; to report a failure, die with a cairo status name such as `'write-error'`, which then becomes this method's result. The callback is only used during the call and is not retained afterwards. Dies if the cairo installed on this machine was built without PNG support. Wraps `cairo_surface_write_to_png_stream`.

set_mime_data

Attaches the original compressed form of an image to the surface, so that a backend able to embed it does so instead of re-encoding the pixels, and returns a status name.

$image->set_mime_data(Cairo::Surface::MIME_TYPE_JPEG, $jpeg_bytes);

`$mime_type` is one of the `Cairo::Surface::MIME_TYPE_*` constants. Ask `supports_mime_type` first, since a backend that cannot use the data simply ignores it.

The scalar itself is retained, NOT copied, so do not modify `$data` after handing it over; the surface holds on to it until the surface goes away or other data replaces it. Needs cairo 1.10 or newer; on an older library the call dies saying so. Wraps `cairo_surface_set_mime_data`.

Cairo::SurfacePattern

Functions

Other Functions

create

Creates a pattern that paints with the contents of `$surface` (cairo_pattern_create_for_surface): `Cairo::SurfacePattern->create($surface)`. The surface argument is required. The result is blessed `Cairo::SurfacePattern`.

get_surface

Returns the surface the pattern paints with, blessed into its own class (`Cairo::ImageSurface`, `Cairo::PdfSurface`, ...). The surface stays valid for as long as you hold the returned object, even if the pattern goes away. On a pattern that is not a surface pattern it croaks with `pattern-type-mismatch`. Needs libcairo 1.4 or newer.

Cairo::SvgSurface

Functions

Other Functions

restrict_to_version

Restricts the surface to features present in the given SVG version, named as `'1-1'` or `'1-2'`.

Cairo::ToyFontFace

Functions

Other Functions

`create`

Creates a font face for a family name selected the simple way (cairo_toy_font_face_create): `Cairo::ToyFontFace->create($family, $slant, $weight)`. `$slant` is one of `normal`, `italic`, `oblique` and `$weight` one of `normal`, `bold`. All three arguments are required; a shorter list croaks with a usage message. Needs libcairo 1.8 or newer.

`get_family`

Returns the family name the face was created with, as a UTF-8 string.

`get_slant`

Returns the face's slant as a nickname string: `normal`, `italic` or `oblique`.

`get_weight`

Returns the face's weight as a nickname string: `normal` or `bold`.

VERSION_ENCODE

Encodes a cairo version triple into the single integer that `Cairo::LIB_VERSION` returns: `major * 10000 + minor * 100 + micro`.

Also available as `Cairo::LIB_VERSION_ENCODE`.

Spelled `Cairo::VERSION_ENCODE` or `Cairo::LIB_VERSION_ENCODE`, and callable both as a plain function, `Cairo::VERSION_ENCODE(1, 2, 0)`, and as a class method, `Cairo->VERSION_ENCODE(1, 2, 0)`. The last three arguments are the ones read, so the class-method form's leading class name is ignored. Called with fewer numbers, the missing minor and micro components count as 0.

5.2.13 Capture

Namespace containing the following modules:

- `Capture::Tiny` - version 0.50

Capture::Tiny

VERSION

version 0.50

SYNOPSIS

```
use Capture::Tiny ':all';

# capture from external command

($stdout, $stderr, $exit) = capture {
  system( $cmd, @args );
};

# capture from arbitrary code (Perl or external)

($stdout, $stderr, @result) = capture {
  # your code here
};

# capture partial or merged output

$stdout = capture_stdout { ... };
$stderr = capture_stderr { ... };
$merged = capture_merged { ... };

# tee output

($stdout, $stderr) = tee {
  # your code here
};

$stdout = tee_stdout { ... };
$stderr = tee_stderr { ... };
$merged = tee_merged { ... };
```

DESCRIPTION

Optionally, output can be teed so that it is captured while being passed through to the original filehandles. Yes, it even works on Windows (usually). Stop guessing which of a dozen capturing modules to use in any particular situation and just use this one.

USAGE

The following functions are available. None are exported by default.

capture

```
($stdout, $stderr, @result) = capture \&code;
$stdout = capture \&code;
```

The `capture` function takes a code reference and returns what is sent to STDOUT and STDERR as well as any return values from the code reference. In scalar context, it returns only STDOUT. If no output was received for a filehandle, it returns an empty string for

that filehandle. Regardless of calling context, all output is captured – nothing is passed to the existing filehandles.

It is prototyped to take a subroutine reference as an argument. Thus, it can be called in block form:

```
($stdout, $stderr) = capture {
  # your code here ...
};
```

Note that the coderef is evaluated in list context. If you wish to force scalar context on the return value, you must use the `scalar` keyword.

```
($stdout, $stderr, $count) = capture {
  my @list = qw/one two three/;
  return scalar @list; # $count will be 3
};
```

Also note that within the coderef, the `@_` variable will be empty. So don't use arguments from a surrounding subroutine without copying them to an array first:

```
sub wont_work {
  my ($stdout, $stderr) = capture { do_stuff( @_ ) };    # WRONG
  ...
}

sub will_work {
  my @args = @_;
  my ($stdout, $stderr) = capture { do_stuff( @args ) }; # RIGHT
  ...
}
```

Captures are normally done to an anonymous temporary filehandle. To capture via a named file (e.g. to externally monitor a long-running capture), provide custom filehandles as a trailing list of option pairs:

```
my $out_fh = IO::File->new("out.txt", "w+");
my $err_fh = IO::File->new("err.txt", "w+");
capture { ... } stdout => $out_fh, stderr => $err_fh;
```

The filehandles must be read/write and seekable. Modifying the files or filehandles during a capture operation will give unpredictable results. Existing IO layers on them may be changed by the capture.

When called in void context, `capture` saves memory and time by not reading back from the capture handles.

capture_stdout

```
($stdout, @result) = capture_stdout \&code;
$stdout = capture_stdout \&code;
```

The `capture_stdout` function works just like `capture` except only STDOUT is captured. STDERR is not captured.

capture_stderr

```
($stderr, @result) = capture_stderr \&code;
$stderr = capture_stderr \&code;
```

The `capture_stderr` function works just like `capture` except only STDERR is captured. STDOUT is not captured.

capture_merged

```
($merged, @result) = capture_merged \&code;
$merged = capture_merged \&code;
```

The `capture_merged` function works just like `capture` except STDOUT and STDERR are merged. (Technically, STDERR is redirected to the same capturing handle as STDOUT before executing the function.)

Caution: STDOUT and STDERR output in the merged result are not guaranteed to be properly ordered due to buffering.

tee

```
($stdout, $stderr, @result) = tee \&code;
$stdout = tee \&code;
```

The `tee` function works just like `capture`, except that output is captured as well as passed on to the original STDOUT and STDERR.

When called in void context, `tee` saves memory and time by not reading back from the capture handles, except when the original STDOUT or STDERR were tied or opened to a scalar handle.

tee_stdout

```
($stdout, @result) = tee_stdout \&code;
$stdout = tee_stdout \&code;
```

The `tee_stdout` function works just like `tee` except only STDOUT is teed. STDERR is not teed (output goes to STDERR as usual).

tee_stderr

```
($stderr, @result) = tee_stderr \&code;
$stderr = tee_stderr \&code;
```

The `tee_stderr` function works just like `tee` except only STDERR is teed. STDOUT is not teed (output goes to STDOUT as usual).

tee_merged

```
($merged, @result) = tee_merged \&code;
$merged = tee_merged \&code;
```

The `tee_merged` function works just like `capture_merged` except that output is captured as well as passed on to STDOUT.

Caution: STDOUT and STDERR output in the merged result are not guaranteed to be properly ordered due to buffering.

LIMITATIONS

Portability

Portability is a goal, not a guarantee. `tee` requires fork, except on Windows where `system(1, @cmd)` is used instead. Not tested on any particularly esoteric platforms yet. See the CPAN Testers Matrix for test result by platform.

PerlIO layers

Capture::Tiny does its best to preserve PerlIO layers such as ':utf8' or ':crlf' when capturing (only for Perl 5.8.1+) . Layers should be applied to STDOUT or STDERR *before* the call to `capture` or `tee`. This may not work for tied filehandles (see below).

Modifying filehandles before capturing

Generally speaking, you should do little or no manipulation of the standard IO filehandles prior to using Capture::Tiny. In particular, closing, reopening, localizing or tying standard filehandles prior to capture may cause a variety of unexpected, undesirable and/or unreliable behaviors, as described below. Capture::Tiny does its best to compensate for these situations, but the results may not be what you desire.

Closed filehandles

Capture::Tiny will work even if STDIN, STDOUT or STDERR have been previously closed. However, since they will be reopened to capture or tee output, any code within the captured block that depends on finding them closed will, of course, not find them to be closed. If they started closed, Capture::Tiny will close them again when the capture block finishes.

Note that this reopening will happen even for STDIN or a filehandle not being captured to ensure that the filehandle used for capture is not opened to file descriptor 0, as this causes problems on various platforms.

Prior to Perl 5.12, closed STDIN combined with PERL_UNICODE=D leaks filehandles and also breaks tee() for undiagnosed reasons. So don't do that.

Localized filehandles

If code localizes any of Perl's standard filehandles before capturing, the capture will affect the localized filehandles and not the original ones. External system calls are not affected by localizing a filehandle in Perl and will continue to send output to the original filehandles (which will thus not be captured).

Scalar filehandles

If STDOUT or STDERR are reopened to scalar filehandles prior to the call to `capture` or `tee`, then Capture::Tiny will override the output filehandle for the duration of the `capture` or `tee` call and then, for `tee`, send captured output to the output filehandle after the capture is complete. (Requires Perl 5.8)

Capture::Tiny attempts to preserve the semantics of STDIN opened to a scalar reference, but note that external processes will not be able to read from such a handle. Capture::Tiny tries to ensure that external processes will read from the null device instead, but this is not guaranteed.

Tied output filehandles

If STDOUT or STDERR are tied prior to the call to `capture` or `tee`, then Capture::Tiny will attempt to override the tie for the duration of the `capture` or `tee` call and then send captured output to the tied filehandle after the capture is complete. (Requires Perl 5.8)

Capture::Tiny may not succeed resending UTF-8 encoded data to a tied STDOUT or STDERR filehandle. Characters may appear as bytes. If the tied filehandle is based on *Tie::StdHandle*, then Capture::Tiny will attempt to determine appropriate layers like `:utf8` from the underlying filehandle and do the right thing.

Tied input filehandle

Capture::Tiny attempts to preserve the semantics of tied STDIN, but this requires Perl 5.8 and is not entirely predictable. External processes will not be able to read from such a handle.

Unless having STDIN tied is crucial, it may be safest to localize STDIN when capturing:

```
my ($out, $err) = do { local *STDIN; capture { ... } };
```

Modifying filehandles during a capture

Attempting to modify STDIN, STDOUT or STDERR *during* `capture` or `tee` is almost certainly going to cause problems. Don't do that.

Forking inside a capture

Forks aren't portable. The behavior of filehandles during a fork is even less so. If Capture::Tiny detects that a fork has occurred within a capture, it will shortcut in the child process and return empty strings for captures. Other problems may occur in the child or parent, as well. Forking in a capture block is not recommended.

Using threads

Filehandles are global. Mixing up I/O and captures in different threads without coordination is going to cause problems. Besides, threads are officially discouraged.

Dropping privileges during a capture

If you drop privileges during a capture, temporary files created to facilitate the capture may not be cleaned up afterwards.

No support for Perl 5.8.0

It's just too buggy when it comes to layers and UTF-8. Perl 5.8.1 or later is recommended.

Limited support for Perl 5.6

Perl 5.6 predates PerlIO. UTF-8 data may not be captured correctly.

ENVIRONMENT

PERL_CAPTURE_TINY_TIMEOUT

Capture::Tiny uses subprocesses internally for `tee`. By default, Capture::Tiny will timeout with an error if such subprocesses are not ready to receive data within 30 seconds (or whatever is the value of `$Capture::Tiny::TIMEOUT`). An alternate timeout may be specified by setting the `PERL_CAPTURE_TINY_TIMEOUT` environment variable. Setting it to zero will disable timeouts. **NOTE**, this does not timeout the code reference being captured – this only prevents Capture::Tiny itself from hanging your process waiting for its child processes to be ready to proceed.

SEE ALSO

This module was inspired by IO::CaptureOutput, which provides similar functionality without the ability to tee output and with more complicated code and API. IO::CaptureOutput does not handle layers or most of the unusual cases described in the "LIMITATIONS" section and I no longer recommend it.

There are many other CPAN modules that provide some sort of output capture, albeit with various limitations that make them appropriate only in particular circumstances. I'm probably missing some. The long list is provided to show why I felt Capture::Tiny was necessary.

- IO::Capture
- IO::Capture::Extended
- IO::CaptureOutput
- IPC::Capture
- *IPC::Cmd*
- *IPC::Open2*
- *IPC::Open3*

- IPC::Open3::Simple
- IPC::Open3::Utils
- IPC::Run
- IPC::Run::SafeHandles
- IPC::Run::Simple
- IPC::Run3
- IPC::System::Simple
- Tee
- IO::Tee
- File::Tee
- Filter::Handle
- Tie::STDERR
- Tie::STDOUT
- Test::Output

SUPPORT

Bugs / Feature Requests

Please report any bugs or feature requests through the issue tracker at https://github.com/dagolden/Capture-Tiny/issues. You will be notified automatically of any progress on your issue.

Source Code

This is open source software. The code repository is available for public review and contribution under the terms of the license.

https://github.com/dagolden/Capture-Tiny

```
git clone https://github.com/dagolden/Capture-Tiny.git
```

AUTHOR

David Golden dagolden@cpan.org

CONTRIBUTORS

- Dagfinn Ilmari Mannsåker ilmari@ilmari.org
- David E. Wheeler david@justatheory.com
- Ed Sabol esabol@users.noreply.github.com
- fecundf <not.com+github@gmail.com>

- Graham Knop haarg@haarg.org
- Karen Etheridge ether@cpan.org
- Mohammad S Anwar mohammad.anwar@yahoo.com
- Peter Rabbitson ribasushi@cpan.org
- Sven Kirmess sven.kirmess@kzone.ch

COPYRIGHT AND LICENSE

This software is Copyright (c) 2009 by David Golden.

This is free software, licensed under:

```
The Apache License, Version 2.0, January 2004
```

5.2.14 Carp

alternative warn and die for modules

SYNOPSIS

```
use Carp;

# warn user (from perspective of caller)
carp "string trimmed to 80 chars";

# die of errors (from perspective of caller)
croak "We're outta here!";

# die of errors with stack backtrace
confess "not implemented";

# cluck, longmess and shortmess not exported by default
use Carp qw(cluck longmess shortmess);
cluck "This is how we got here!"; # warn with stack backtrace
my $long_message   = longmess( "message from cluck() or confess()"
 ↪);
my $short_message  = shortmess( "message from carp() or croak()" );
```

DESCRIPTION

The Carp routines are useful in your own modules because they act like `die()` or `warn()`, but with a message which is more likely to be useful to a user of your module. In the case of `cluck()` and `confess()`, that context is a summary of every call in the call-stack; `longmess()` returns the contents of the error message.

For a shorter message you can use `carp()` or `croak()` which report the error as being from where your module was called. `shortmess()` returns the contents of this error message. There is no guarantee that that is where the error was, but it is a good educated guess.

Carp takes care not to clobber the status variables $! and $^E in the course of assembling its error messages. This means that a $SIG{__DIE__} or $SIG{__WARN__} handler can capture the error information held in those variables, if it is required to augment the error message, and if the code calling Carp left useful values there. Of course, Carp can't guarantee the latter.

You can also alter the way the output and logic of Carp works, by changing some global variables in the Carp namespace. See the section on "GLOBAL VARIABLES" below.

Here is a more complete description of how carp and croak work. What they do is search the call-stack for a function call stack where they have not been told that there shouldn't be an error. If every call is marked safe, they give up and give a full stack backtrace instead. In other words they presume that the first likely looking potential suspect is guilty. Their rules for telling whether a call shouldn't generate errors work as follows:

1. Any call from a package to itself is safe.
2. Packages claim that there won't be errors on calls to or from packages explicitly marked as safe by inclusion in @CARP_NOT, or (if that array is empty) @ISA. The ability to override what @ISA says is new in 5.8.
3. The trust in item 2 is transitive. If A trusts B, and B trusts C, then A trusts C. So if you do not override @ISA with @CARP_NOT, then this trust relationship is identical to, "inherits from".
4. Any call from an internal Perl module is safe. (Nothing keeps user modules from marking themselves as internal to Perl, but this practice is discouraged.)
5. Any call to Perl's warning system (eg Carp itself) is safe. (This rule is what keeps it from reporting the error at the point where you call carp or croak.)
6. $Carp::CarpLevel can be set to skip a fixed number of additional call levels. Using this is not recommended because it is very difficult to get it to behave correctly.

Forcing a Stack Trace

As a debugging aid, you can force Carp to treat a croak as a confess and a carp as a cluck across *all* modules. In other words, force a detailed stack trace to be given. This can be very helpful when trying to understand why, or from where, a warning or error is being generated.

This feature is enabled by 'importing' the non-existent symbol 'verbose'. You would typically enable it by saying

```
perl -MCarp=verbose script.pl
```

or by including the string -MCarp=verbose in the PERL5OPT environment variable.

Alternately, you can set the global variable $Carp::Verbose to true. See the "GLOBAL VARIABLES" section below.

Stack Trace formatting

At each stack level, the subroutine's name is displayed along with its parameters. For simple scalars, this is sufficient. For complex data types, such as objects and other references, this can simply display `'HASH(0x1ab36d8)'`.

Carp gives two ways to control this.

1. For objects, a method, `CARP_TRACE`, will be called, if it exists. If this method doesn't exist, or it recurses into `Carp`, or it otherwise throws an exception, this is skipped, and Carp moves on to the next option, otherwise checking stops and the string returned is used. It is recommended that the object's type is part of the string to make debugging easier.
2. For any type of reference, `$Carp::RefArgFormatter` is checked (see below). This variable is expected to be a code reference, and the current parameter is passed in. If this function doesn't exist (the variable is undef), or it recurses into `Carp`, or it otherwise throws an exception, this is skipped, and Carp moves on to the next option, otherwise checking stops and the string returned is used.
3. Otherwise, if neither `CARP_TRACE` nor `$Carp::RefArgFormatter` is available, stringify the value ignoring any overloading.

GLOBAL VARIABLES

$Carp::MaxEvalLen

This variable determines how many characters of a string-eval are to be shown in the output. Use a value of `0` to show all text.

Defaults to `0`.

$Carp::MaxArgLen

This variable determines how many characters of each argument to a function to print. Use a value of `0` to show the full length of the argument.

Defaults to `64`.

$Carp::MaxArgNums

This variable determines how many arguments to each function to show. Use a false value to show all arguments to a function call. To suppress all arguments, use `-1` or `'0 but true'`.

Defaults to `8`.

$Carp::Verbose

This variable makes `carp()` and `croak()` generate stack backtraces just like `cluck()` and `confess()`. This is how `use Carp 'verbose'` is implemented internally.

Defaults to `0`.

$Carp::RefArgFormatter

This variable sets a general argument formatter to display references. Plain scalars and objects that implement CARP_TRACE will not go through this formatter. Calling Carp from within this function is not supported.

```
local $Carp::RefArgFormatter = sub {
    require Data::Dumper;
    Data::Dumper->Dump($_[0]); # not necessarily safe
};
```

@CARP_NOT

This variable, *in your package*, says which packages are *not* to be considered as the location of an error. The carp() and cluck() functions will skip over callers when reporting where an error occurred.

NB: This variable must be in the package's symbol table, thus:

```
# These work
our @CARP_NOT; # file scope
use vars qw(@CARP_NOT); # package scope
@My::Package::CARP_NOT = ... ; # explicit package variable

# These don't work
sub xyz { ... @CARP_NOT = ... } # w/o declarations above
my @CARP_NOT; # even at top-level
```

Example of use:

```
package My::Carping::Package;
use Carp;
our @CARP_NOT;
sub bar     { .... or _error('Wrong input') }
sub _error  {
    # temporary control of where'ness, __PACKAGE__ is implicit
    local @CARP_NOT = qw(My::Friendly::Caller);
    carp(@_)
}
```

This would make Carp report the error as coming from a caller not in My::Carping::Package, nor from My::Friendly::Caller.

Also read the "DESCRIPTION" section above, about how Carp decides where the error is reported from.

Use @CARP_NOT, instead of $Carp::CarpLevel.

Overrides Carp's use of @ISA.

%Carp::Internal

This says what packages are internal to Perl. `Carp` will never report an error as being from a line in a package that is internal to Perl. For example:

```
$Carp::Internal{ (__PACKAGE__) }++;
# time passes...
sub foo { ... or confess("whatever") };
```

would give a full stack backtrace starting from the first caller outside of __PACKAGE__. (Unless that package was also internal to Perl.)

%Carp::CarpInternal

This says which packages are internal to Perl's warning system. For generating a full stack backtrace this is the same as being internal to Perl, the stack backtrace will not start inside packages that are listed in `%Carp::CarpInternal`. But it is slightly different for the summary message generated by `carp` or `croak`. There errors will not be reported on any lines that are calling packages in `%Carp::CarpInternal`.

For example `Carp` itself is listed in `%Carp::CarpInternal`. Therefore the full stack backtrace from `confess` will not start inside of `Carp`, and the short message from calling `croak` is not placed on the line where `croak` was called.

$Carp::CarpLevel

This variable determines how many additional call frames are to be skipped that would not otherwise be when reporting where an error occurred on a call to one of `Carp`'s functions. It is fairly easy to count these call frames on calls that generate a full stack backtrace. However it is much harder to do this accounting for calls that generate a short message. Usually people skip too many call frames. If they are lucky they skip enough that `Carp` goes all of the way through the call stack, realizes that something is wrong, and then generates a full stack backtrace. If they are unlucky then the error is reported from somewhere misleading very high in the call stack.

Therefore it is best to avoid `$Carp::CarpLevel`. Instead use `@CARP_NOT`, `%Carp::Internal` and `%Carp::CarpInternal`.

Defaults to `0`.

BUGS

The Carp routines don't handle exception objects currently. If called with a first argument that is a reference, they simply call die() or warn(), as appropriate.

SEE ALSO

Carp::Always, Carp::Clan

CONTRIBUTING

Carp is maintained by the perl 5 porters as part of the core perl 5 version control repository. Please see the perlhack perldoc for how to submit patches and contribute to it.

AUTHOR

The Carp module first appeared in Larry Wall's perl 5.000 distribution. Since then it has been modified by several of the perl 5 porters. Andrew Main (Zefram) zefram@fysh.org divested Carp into an independent distribution.

COPYRIGHT

Copyright (C) 1994-2013 Larry Wall

Copyright (C) 2011, 2012, 2013 Andrew Main (Zefram) zefram@fysh.org

LICENSE

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

5.2.15 Class

Namespace containing the following modules:

- `Class::Inspector` - Get information about a class and its structure
- `Class::Load`
- `Class::MOP`
- `Class::Struct` - declare struct-like datatypes as Perl classes
- `Class::XSAccessor` - fast accessors, getters, setters, predicates and

Class::Inspector

Get information about a class and its structure

VERSION

version 1.36

SYNOPSIS

```
use Class::Inspector;

# Is a class installed and/or loaded
Class::Inspector->installed( 'Foo::Class' );
Class::Inspector->loaded( 'Foo::Class' );

# Filename related information
```

(continues on next page)

(continued from previous page)

```
Class::Inspector->filename( 'Foo::Class' );
Class::Inspector->resolved_filename( 'Foo::Class' );

# Get subroutine related information
Class::Inspector->functions( 'Foo::Class' );
Class::Inspector->function_refs( 'Foo::Class' );
Class::Inspector->function_exists( 'Foo::Class', 'bar' );
Class::Inspector->methods( 'Foo::Class', 'full', 'public' );

# Find all loaded subclasses or something
Class::Inspector->subclasses( 'Foo::Class' );
```

DESCRIPTION

Class::Inspector allows you to get information about a loaded class. Most or all of this information can be found in other ways, but they aren't always very friendly, and usually involve a relatively high level of Perl wizardry, or strange and unusual looking code. Class::Inspector attempts to provide an easier, more friendly interface to this information.

METHODS

installed

```
my $bool = Class::Inspector->installed($class);
```

The `installed` static method tries to determine if a class is installed on the machine, or at least available to Perl. It does this by wrapping around `resolved_filename`.

Returns true if installed/available, false if the class is not installed, or `undef` if the class name is invalid.

loaded

```
my $bool = Class::Inspector->loaded($class);
```

The `loaded` static method tries to determine if a class is loaded by looking for symbol table entries.

This method it uses to determine this will work even if the class does not have its own file, but is contained inside a single file with multiple classes in it. Even in the case of some sort of run-time loading class being used, these typically leave some trace in the symbol table, so an Autoload or Class::Autouse-based class should correctly appear loaded.

Returns true if the class is loaded, false if not, or `undef` if the class name is invalid.

filename

```
my $filename = Class::Inspector->filename($class);
```

For a given class, returns the base filename for the class. This will NOT be a fully resolved filename, just the part of the filename BELOW the `@INC` entry.

```
print Class->filename( 'Foo::Bar' );
> Foo/Bar.pm
```

This filename will be returned with the right separator for the local platform, and should work on all platforms.

Returns the filename on success or `undef` if the class name is invalid.

resolved_filename

```
my $filename = Class::Inspector->resolved_filename($class);
my $filename = Class::Inspector->resolved_filename($class, @try_
↪first);
```

For a given class, the `resolved_filename` static method returns the fully resolved filename for a class. That is, the file that the class would be loaded from.

This is not necessarily the file that the class WAS loaded from, as the value returned is determined each time it runs, and the `@INC` include path may change.

To get the actual file for a loaded class, see the `loaded_filename` method.

Returns the filename for the class, or `undef` if the class name is invalid.

loaded_filename

```
my $filename = Class::Inspector->loaded_filename($class);
```

For a given loaded class, the `loaded_filename` static method determines (via the `%INC` hash) the name of the file that it was originally loaded from.

Returns a resolved file path, or false if the class did not have it's own file.

functions

```
my $arrayref = Class::Inspector->functions($class);
```

For a loaded class, the `functions` static method returns a list of the names of all the functions in the classes immediate namespace.

Note that this is not the METHODS of the class, just the functions.

Returns a reference to an array of the function names on success, or `undef` if the class name is invalid or the class is not loaded.

function_refs

```
my $arrayref = Class::Inspector->function_refs($class);
```

For a loaded class, the `function_refs` static method returns references to all the functions in the classes immediate namespace.

Note that this is not the METHODS of the class, just the functions.

Returns a reference to an array of `CODE` refs of the functions on success, or `undef` if the class is not loaded.

function_exists

```
my $bool = Class::Inspector->function_exists($class, $functon);
```

Given a class and function name the `function_exists` static method will check to see if the function exists in the class.

Note that this is as a function, not as a method. To see if a method exists for a class, use the `can` method for any class or object.

Returns true if the function exists, false if not, or `undef` if the class or function name are invalid, or the class is not loaded.

methods

```
my $arrayref = Class::Inspector->methods($class, @options);
```

For a given class name, the `methods` static method will returns ALL the methods available to that class. This includes all methods available from every class up the class' `@ISA` tree.

Returns a reference to an array of the names of all the available methods on success, or `undef` if the class name is invalid or the class is not loaded.

A number of options are available to the `methods` method that will alter the results returned. These should be listed after the class name, in any order.

```
# Only get public methods
my $method = Class::Inspector->methods( 'My::Class', 'public' );
```

- public

  The `public` option will return only 'public' methods, as defined by the Perl convention of prepending an underscore to any 'private' methods. The `public` option will effectively remove any methods that start with an underscore.

- private

  The `private` options will return only 'private' methods, as defined by the Perl convention of prepending an underscore to an private methods. The `private` option will effectively remove an method that do not start with an underscore.

  Note: The `public` and `private` options are mutually exclusive

- full

  `methods` normally returns just the method name. Supplying the `full` option will cause the methods to be returned as the full names. That is, instead of returning `[ 'method1', 'method2', 'method3' ]`, you would instead get `[ 'Class::method1', 'AnotherClass::method2', 'Class::method3' ]`.

- expanded

  The `expanded` option will cause a lot more information about method to be returned. Instead of just the method name, you will instead get an array reference containing the method name as a single combined name, a la `full`, the separate class and method, and a CODE ref to the actual function (if available). Please note that the function reference is not guaranteed to be available. `Class::Inspector` is intended at some later time, to work with modules that have some kind of common run-time loader in place (e.g `Autoloader` or `Class::Autouse` for example.

  The response from `methods( 'Class', 'expanded' )` would look something like the following.

```
[
  [ 'Class::method1',   'Class',   'method1', \&Class::method1 ],
  [ 'Another::method2', 'Another', 'method2', \&Another::method2 ],
  [ 'Foo::bar',         'Foo',     'bar',     \&Foo::bar ],
]
```

subclasses

```
my $arrayref = Class::Inspector->subclasses($class);
```

The `subclasses` static method will search then entire namespace (and thus **all** currently loaded classes) to find all classes that are subclasses of the class provided as a the parameter.

The actual test will be done by calling `isa` on the class as a static method. (i.e. `My::Class->isa($class)`.

Returns a reference to a list of the loaded classes that match the class provided, or false is none match, or `undef` if the class name provided is invalid.

SEE ALSO

http://ali.as/, Class::Handle, *Class::Inspector::Functions*

AUTHOR

Original author: Adam Kennedy adamk@cpan.org

Current maintainer: Graham Ollis plicease@cpan.org

Contributors:

Tom Wyant

Steffen Müller

Kivanc Yazan (KYZN)

COPYRIGHT AND LICENSE

This software is copyright (c) 2002-2019 by Adam Kennedy.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

Modules

- `Class::Inspector::Functions` - Get information about a class and its structure

Class::Inspector::Functions

Get information about a class and its structure

VERSION

version 1.36

SYNOPSIS

```
use Class::Inspector::Functions;
# Class::Inspector provides a non-polluting,
# method based interface!

# Is a class installed and/or loaded
installed( 'Foo::Class' );
loaded( 'Foo::Class' );

# Filename related information
filename( 'Foo::Class' );
resolved_filename( 'Foo::Class' );

# Get subroutine related information
functions( 'Foo::Class' );
function_refs( 'Foo::Class' );
function_exists( 'Foo::Class', 'bar' );
methods( 'Foo::Class', 'full', 'public' );

# Find all loaded subclasses or something
subclasses( 'Foo::Class' );
```

DESCRIPTION

Class::Inspector::Functions is a function based interface of *Class::Inspector*. For a thorough documentation of the available functions, please check the manual for the main module.

Exports

The following functions are exported by default.

```
installed
loaded
filename
functions
methods
subclasses
```

The following functions are exported only by request.

```
resolved_filename
loaded_filename
function_refs
function_exists
```

All the functions may be imported using the `:ALL` tag.

SEE ALSO

http://ali.as/, Class::Handle, *Class::Inspector*

AUTHOR

Original author: Adam Kennedy adamk@cpan.org

Current maintainer: Graham Ollis plicease@cpan.org

Contributors:

Tom Wyant

Steffen Müller

Kivanc Yazan (KYZN)

COPYRIGHT AND LICENSE

This software is copyright (c) 2002-2019 by Adam Kennedy.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

Class::Load

Namespace containing the following modules:

- `Class::Load::XS` - The compiled `is_class_loaded` of Class::Load: whether a package has been loaded, judged from its symbol table.

Class::Load::XS

The compiled `is_class_loaded` of Class::Load: whether a package has been loaded, judged from its symbol table.

A package counts as loaded when it has a `$VERSION` (a version object, a reference to a defined value, or any defined value), a non-empty `@ISA`, or at least one sub, constant or stub. With `{ -version => $v }` it counts only when `$class->VERSION($v)` does not die. Class::Load calls this in place of its pure-Perl version; it is not normally called directly.

This is a native implementation of Class::Load::XS 0.10's `XS.xs`; its `Class/Load/XS.pm` is used unchanged, and Class::Load itself comes from CPAN.

Synopsis

```
use Class::Load qw(is_class_loaded);   # takes this one when it can

print is_class_loaded('Some::Class') ? "loaded\n" : "not loaded\n";
print is_class_loaded('Some::Class', { -version => '1.2' }) ? "1.2␣
 ↪or later\n" : "older, or not loaded\n";
```

Differences from upstream

None.

Functions

Class::Load

is_class_loaded

Whether a package is loaded: it has a `$VERSION`, a non-empty `@ISA` or a sub; with `{ -version => $v }`, whether `$class->VERSION($v)` succeeds.

is_class_loaded

Whether a package is loaded: it has a `$VERSION`, a non-empty `@ISA` or a sub; with `{ -version => $v }`, whether `$class->VERSION($v)` succeeds.

Synopsis

```
Class::Load::XS::is_class_loaded('Foo');                    # true␣
↪or false
Class::Load::XS::is_class_loaded('Foo', { -version => 2 });  # Foo 2␣
↪or later, loaded
```

Edge cases

- An empty or non-string class name is not loaded.
- A second argument that is not a hash reference dies "Class::Load::XS::is_class_loaded: options is not a HASH reference".

Class::MOP

Modules

- `Class::MOP::Method`

Functions

Moose

`get_code_info`

The package and the name a sub was defined under.

Class::MOP::Method

Functions

Moose

`is_stub`

Whether a method is only declared: `sub foo;`, or a sub that was referred to but never defined.

is_stub

Whether a method is only declared: `sub foo;`, or a sub that was referred to but never defined.

Synopsis

```
my $method = Class::MOP::Class->initialize('Foo')->get_method('bar
↪');
print $method->is_stub ? "declared only\n" : "defined\n";
```

What you get back

True when the method's body has neither Perl code nor compiled code behind it, false otherwise. Dies when the method's body is not a code reference.

get_code_info

The package and the name a sub was defined under.

Synopsis

```
use Class::MOP;
my ($package, $name) = Class::MOP::get_code_info(\&Foo::bar);  # (
↪'Foo', 'bar')
```

What you get back

The name of the package of the glob the sub was defined in, and the sub's own name in it: __ANON__ for an anonymous sub. Nothing for what is not a code reference, or for a sub that is still being compiled.

Class::Struct

declare struct-like datatypes as Perl classes

SYNOPSIS

```
use Class::Struct;
        # declare struct, based on array:
struct( CLASS_NAME => [ ELEMENT_NAME => ELEMENT_TYPE, ... ]);
        # declare struct, based on hash:
struct( CLASS_NAME => { ELEMENT_NAME => ELEMENT_TYPE, ... });

package CLASS_NAME;
use Class::Struct;
        # declare struct, based on array, implicit class name:
struct( ELEMENT_NAME => ELEMENT_TYPE, ... );

# Declare struct at compile time
use Class::Struct CLASS_NAME => [ELEMENT_NAME => ELEMENT_TYPE, ...];
use Class::Struct CLASS_NAME => {ELEMENT_NAME => ELEMENT_TYPE, ...};

# declare struct at compile time, based on array, implicit
# class name:
package CLASS_NAME;
use Class::Struct ELEMENT_NAME => ELEMENT_TYPE, ... ;

package Myobj;
use Class::Struct;
```

(continues on next page)

(continued from previous page)

```
        # declare struct with four types of elements:
struct( s => '$', a => '@', h => '%', c => 'My_Other_Class' );

my $obj = Myobj->new;               # constructor

                                  # scalar type accessor:
my $element_value = $obj->s;            # element value
$obj->s('new value');                   # assign to element

                                  # array type accessor:
my $ary_ref = $obj->a;                  # reference to whole array
my $ary_element_value = $obj->a(2);     # array element value
$obj->a(2, 'new value');              # assign to array element

                                  # hash type accessor:
my $hash_ref = $obj->h;                 # reference to whole hash
my $hash_element_value = $obj->h('x'); # hash element value
$obj->h('x', 'new value');            # assign to hash element

                                  # class type accessor:
my $element_value = $obj->c;            # object reference
$obj->c->method(...);                 # call method of object
$obj->c(new My_Other_Class);          # assign a new object
```

DESCRIPTION

`Class::Struct` exports a single function, `struct`. Given a list of element names and types, and optionally a class name, `struct` creates a Perl 5 class that implements a "struct-like" data structure.

The new class is given a constructor method, `new`, for creating struct objects.

Each element in the struct data has an accessor method, which is used to assign to the element and to fetch its value. The default accessor can be overridden by declaring a `sub` of the same name in the package. (See Example 2.)

Each element's type can be scalar, array, hash, or class.

The `struct()` function

The `struct` function has three forms of parameter-list.

```
struct( CLASS_NAME => [ ELEMENT_LIST ]);
struct( CLASS_NAME => { ELEMENT_LIST });
struct( ELEMENT_LIST );
```

The first and second forms explicitly identify the name of the class being created. The third form assumes the current package name as the class name.

An object of a class created by the first and third forms is based on an array, whereas an object of a class created by the second form is based on a hash. The array-based forms

will be somewhat faster and smaller; the hash-based forms are more flexible.

The class created by `struct` must not be a subclass of another class other than `UNIVERSAL`.

It can, however, be used as a superclass for other classes. To facilitate this, the generated constructor method uses a two-argument blessing. Furthermore, if the class is hash-based, the key of each element is prefixed with the class name (see *Perl Cookbook*, Recipe 13.12).

A function named `new` must not be explicitly defined in a class created by `struct`.

The *ELEMENT_LIST* has the form

```
NAME => TYPE, ...
```

Each name-type pair declares one element of the struct. Each element name will be defined as an accessor method unless a method by that name is explicitly defined; in the latter case, a warning is issued if the warning flag (**-w**) is set.

Class Creation at Compile Time

`Class::Struct` can create your class at compile time. The main reason for doing this is obvious, so your class acts like every other class in Perl. Creating your class at compile time will make the order of events similar to using any other class (or Perl module).

There is no significant speed gain between compile time and run time class creation, there is just a new, more standard order of events.

Element Types and Accessor Methods

The four element types – scalar, array, hash, and class – are represented by strings – `'$'`, `'@'`, `'%'`, and a class name – optionally preceded by a `'*'`.

The accessor method provided by `struct` for an element depends on the declared type of the element.

- Scalar (`'$'` or `'*$'`)

  The element is a scalar, and by default is initialized to `undef` (but see "Initializing with new").

  The accessor's argument, if any, is assigned to the element.

  If the element type is `'$'`, the value of the element (after assignment) is returned. If the element type is `'*$'`, a reference to the element is returned.

- Array (`'@'` or `'*@'`)

  The element is an array, initialized by default to `()`.

  With no argument, the accessor returns a reference to the element's whole array (whether or not the element was specified as `'@'` or `'*@'`).

  With one or two arguments, the first argument is an index specifying one element of the array; the second argument, if present, is assigned to the array element. If the element type is `'@'`, the accessor returns the array element value. If the element type is `'*@'`, a reference to the array element is returned.

As a special case, when the accessor is called with an array reference as the sole argument, this causes an assignment of the whole array element. The object reference is returned.

- Hash (`'%'` or `'*%'`)

  The element is a hash, initialized by default to `()`.

  With no argument, the accessor returns a reference to the element's whole hash (whether or not the element was specified as `'%'` or `'*%'`).

  With one or two arguments, the first argument is a key specifying one element of the hash; the second argument, if present, is assigned to the hash element. If the element type is `'%'`, the accessor returns the hash element value. If the element type is `'*%'`, a reference to the hash element is returned.

  As a special case, when the accessor is called with a hash reference as the sole argument, this causes an assignment of the whole hash element. The object reference is returned.

- Class (`'Class_Name'` or `'*Class_Name'`)

  The element's value must be a reference blessed to the named class or to one of its subclasses. The element is not initialized by default.

  The accessor's argument, if any, is assigned to the element. The accessor will `croak` if this is not an appropriate object reference.

  If the element type does not start with a `'*'`, the accessor returns the element value (after assignment). If the element type starts with a `'*'`, a reference to the element itself is returned.

Initializing with new

`struct` always creates a constructor called `new`. That constructor may take a list of initializers for the various elements of the new struct.

Each initializer is a pair of values: *element name*=>*value*. The initializer value for a scalar element is just a scalar value. The initializer for an array element is an array reference. The initializer for a hash is a hash reference.

The initializer for a class element is an object of the corresponding class, or of one of it's subclasses, or a reference to a hash containing named arguments to be passed to the element's constructor.

See Example 3 below for an example of initialization.

EXAMPLES

- Example 1

  Giving a struct element a class type that is also a struct is how structs are nested. Here, `Timeval` represents a time (seconds and microseconds), and `Rusage` has two elements, each of which is of type `Timeval`.

```
use Class::Struct;

struct( Rusage => {
    ru_utime => 'Timeval',  # user time used
    ru_stime => 'Timeval',  # system time used
});

struct( Timeval => [
    tv_secs  => '$',        # seconds
    tv_usecs => '$',        # microseconds
]);

# create an object:
my $t = Rusage->new(ru_utime=>Timeval->new(),
    ru_stime=>Timeval->new());

# $t->ru_utime and $t->ru_stime are objects of type Timeval.
# set $t->ru_utime to 100.0 sec and $t->ru_stime to 5.0 sec.
$t->ru_utime->tv_secs(100);
$t->ru_utime->tv_usecs(0);
$t->ru_stime->tv_secs(5);
$t->ru_stime->tv_usecs(0);
```

- Example 2

  An accessor function can be redefined in order to provide additional checking of values, etc. Here, we want the `count` element always to be nonnegative, so we redefine the `count` accessor accordingly.

```
package MyObj;
use Class::Struct;

# declare the struct
struct ( 'MyObj', { count => '$', stuff => '%' } );

# override the default accessor method for 'count'
sub count {
    my $self = shift;
    if ( @_ ) {
        die 'count must be nonnegative' if $_[0] < 0;
        $self->{'MyObj::count'} = shift;
        warn "Too many args to count" if @_;
    }
    return $self->{'MyObj::count'};
}

package main;
$x = new MyObj;
print "\$x->count(5) = ", $x->count(5), "\n";
                         # prints '$x->count(5) = 5'
```

(continues on next page)

(continued from previous page)

```
print "\$x->count = ", $x->count, "\n";
                       # prints '$x->count = 5'

print "\$x->count(-5) = ", $x->count(-5), "\n";
                       # dies due to negative argument!
```

- Example 3

  The constructor of a generated class can be passed a list of *element=>value* pairs, with which to initialize the struct. If no initializer is specified for a particular element, its default initialization is performed instead. Initializers for non-existent elements are silently ignored.

  Note that the initializer for a nested class may be specified as an object of that class, or as a reference to a hash of initializers that are passed on to the nested struct's constructor.

```
use Class::Struct;

struct Breed =>
{
    name  => '$',
    cross => '$',
};

struct Cat =>
[
    name     => '$',
    kittens  => '@',
    markings => '%',
    breed    => 'Breed',
];


my $cat = Cat->new( name => 'Socks',
           kittens  => ['Monica', 'Kenneth'],
           markings => { socks=>1, blaze=>"white" },
           breed    => Breed->new(name=>'short-hair', cross=>1),
      or:  breed    => {name=>'short-hair', cross=>1},
                 );

print "Once a cat called ", $cat->name, "\n";
print "(which was a ", $cat->breed->name, ")\n";
print "had 2 kittens: ", join(' and ', @{$cat->kittens}), "\n";
```

Author and Modification History

Modified by Damian Conway, 2001-09-10, v0.62.

```
Modified implicit construction of nested objects.
Now will also take an object ref instead of requiring a hash ref.
Also default initializes nested object attributes to undef, rather
than calling object constructor without args
Original over-helpfulness was fraught with problems:
    * the class's constructor might not be called 'new'
    * the class might not have a hash-like-arguments constructor
    * the class might not have a no-argument constructor
    * "recursive" data structures didn't work well:
            package Person;
            struct { mother => 'Person', father => 'Person'};
```

Modified by Casey West, 2000-11-08, v0.59.

```
Added the ability for compile time class creation.
```

Modified by Damian Conway, 1999-03-05, v0.58.

```
Added handling of hash-like arg list to class ctor.

Changed to two-argument blessing in ctor to support
derivation from created classes.

Added classname prefixes to keys in hash-based classes
(refer to "Perl Cookbook", Recipe 13.12 for rationale).

Corrected behaviour of accessors for '*@' and '*%' struct
elements.  Package now implements documented behaviour when
returning a reference to an entire hash or array element.
Previously these were returned as a reference to a reference
to the element.
```

Renamed to `Class::Struct` and modified by Jim Miner, 1997-04-02.

```
members() function removed.
Documentation corrected and extended.
Use of struct() in a subclass prohibited.
User definition of accessor allowed.
Treatment of '*' in element types corrected.
Treatment of classes as element types corrected.
Class name to struct() made optional.
Diagnostic checks added.
```

Originally `Class::Template` by Dean Roehrich.

```
# Template.pm   --- struct/member template builder
#   12mar95
#   Dean Roehrich
#
# changes/bugs fixed since 28nov94 version:
#  - podified
```

(continues on next page)

(continued from previous page)

```
# changes/bugs fixed since 21nov94 version:
#  - Fixed examples.
# changes/bugs fixed since 02sep94 version:
#  - Moved to Class::Template.
# changes/bugs fixed since 20feb94 version:
#  - Updated to be a more proper module.
#  - Added "use strict".
#  - Bug in build_methods, was using @var when @$var needed.
#  - Now using my() rather than local().
#
# Uses perl5 classes to create nested data types.
# This is offered as one implementation of Tom Christiansen's
# "structs.pl" idea.
```

Class::XSAccessor

`Class::XSAccessor` - fast accessors, getters, setters, predicates and constructors for hash- and array-based objects, without compiling any Perl at runtime.

```
package MyClass;
use Class::XSAccessor
    constructor => 'new',
    accessors   => { name => 'name' },      # $obj->name / $obj->
↪name($v)
    getters     => { get_age => 'age' },
    setters     => { set_age => 'age' },
    predicates  => { has_age => 'age' },    # defined $obj->{age}
    chained     => 1;                       # setters return $self

package MyArrayClass;
use Class::XSAccessor::Array
    constructor => 'new',
    accessors   => { name => 0 };           # $obj->[0]
```

Every generated method is a compiled subroutine that reads or writes one slot of the object directly. Moo and other object systems use it automatically when it is installed, for their plain attribute accessors.

This is a native implementation of the CPAN distribution Class-XSAccessor 1.19; its `Class/XSAccessor.pm`, `Class/XSAccessor/Array.pm` and `Class/XSAccessor/Heavy.pm` are used unchanged. Like the XS original it rewrites a method call site the first time a generated accessor runs there, so later calls from that site skip the general subroutine-call machinery.

Modules

- `Class::XSAccessor::Array` - Generate fast native accessors without runtime compilation
- `Class::XSAccessor::Heavy` - Guts you don't care about

Functions

Other Functions

`getter`

`$obj->name`: the value of the object's key, or undef.

`lvalue_accessor`

`$obj->name = $value`: the slot itself, assignable.

`setter`

`$obj->set_name($value)`: stores a copy, returns the new value.

`chained_setter`

`$obj->set_name($value)` under `chained => 1`: returns the object.

`accessor`

`$obj->name` / `$obj->name($value)`: get, or set and return the new value.

`chained_accessor`

The accessor under `chained => 1`: setting returns the object.

`exists_predicate`

`$obj->has_name` from `exists_predicates`: whether the key exists.

`defined_predicate`

`$obj->has_name` from `predicates` / `defined_predicates`: whether the value is defined.

`constructor`

`MyClass->new(key => value, ...)`: a hash-based object holding copies of the pairs; an odd count dies.

`constant_false`

A method from `false => [...]`: always false.

constant_true

A method from `true => [...]`: always true.

newxs_defined_predicate

`newxs_getter(name, key)` and its ALIASes newxs_lvalue_accessor (1), newxs_predicate (2), newxs_defined_predicate (3), newxs_exists_predicate (4): install one read-side method.

Also available as `Class::XSAccessor::newxs_exists_predicate`, `Class::XSAccessor::newxs_getter`, `Class::XSAccessor::newxs_lvalue_accessor`, `Class::XSAccessor::newxs_predicate`.

newxs_accessor

`newxs_setter(name, key, chained)` and its ALIAS newxs_accessor (1).

Also available as `Class::XSAccessor::newxs_setter`.

newxs_constructor

`newxs_constructor(name)`.

newxs_boolean

`newxs_boolean(name, truth)`.

newxs_test

`newxs_test(name, key)`.

array_setter_init

`array_setter_init`: the compat setter that optimises its call site.

array_setter

`array_setter`: what an optimised compat setter call site calls.

array_accessor_init

`array_accessor_init`: the compat accessor that optimises its call site.

array_accessor

`array_accessor`: what an optimised compat accessor call site calls.

END

`Class::XSAccessor::END`: does nothing. Freeing the key table here is commented out upstream - accessors may still run after END blocks.

Class::XSAccessor::Array

Generate fast native accessors without runtime compilation

SYNOPSIS

```
package MyClassUsingArraysAsInternalStorage;
use Class::XSAccessor::Array
  constructor => 'new',
  getters => {
    get_foo => 0, # 0 is the array index to access
    get_bar => 1,
  },
  setters => {
    set_foo => 0,
    set_bar => 1,
  },
  accessors => { # a mutator
    buz => 2,
  },
  predicates => { # test for definedness
    has_buz => 2,
  },
  lvalue_accessors => { # see below
    baz => 3,
  },
  true => [ 'is_token', 'is_whitespace' ],
  false => [ 'significant' ];

# The imported methods are implemented natively.

# normal class code here.
```

As of version 1.05, some alternative syntax forms are available:

```
package MyClass;

# Options can be passed as a HASH reference if you prefer it,
# which can also help PerlTidy to flow the statement correctly.
use Class::XSAccessor {
  getters => {
    get_foo => 0,
    get_bar => 1,
  },
};
```

DESCRIPTION

The module implements fast native accessors both for getting at and setting an object attribute. Additionally, the module supports mutators and simple predicates (`has_foo()` like tests for definedness of an attributes). The module works only with objects that are implemented as **arrays**. Using it on hash-based objects is bound to make your life miserable. Refer to *Class::XSAccessor* for an implementation that works with hash-based objects.

A simple benchmark showed a significant performance advantage over writing accessors in Perl.

Since version 0.10, the module can also generate simple constructors (implemented natively) for you. Simply supply the `constructor => 'constructor_name'` option or the `constructors => ['new', 'create', 'spawn']` option. These constructors do the equivalent of the following Perl code:

```
sub new {
  my $class = shift;
  return bless [], ref($class)||$class;
}
```

That means they can be called on objects and classes but will not clone objects entirely. Note that any parameters to new() will be discarded! If there is a better idiom for array-based objects, let me know.

While generally more obscure than hash-based objects, objects using blessed arrays as internal representation are a bit faster as its somewhat faster to access arrays than hashes. Accordingly, this module is slightly faster (~10-15%) than *Class::XSAccessor*, which works on hash-based objects.

The method names may be fully qualified. In the example of the synopsis, you could have written `MyClass::get_foo` instead of `get_foo`. This way, you can install methods in classes other than the current class. See also: The `class` option below.

Since version 1.01, you can generate extremely simple methods which just return true or false (and always do so). If that seems like a really superfluous thing to you, then think of a large class hierarchy with interfaces such as PPI. This is implemented as the `true` and `false` options, see synopsis.

OPTIONS

In addition to specifying the types and names of accessors, you can add options which modify behaviour. The options are specified as key/value pairs just as the accessor declaration. Example:

```
use Class::XSAccessor::Array
  getters => {
    get_foo => 0,
  },
  replace => 1;
```

The list of available options is:

replace

Set this to a true value to prevent `Class::XSAccessor::Array` from complaining about replacing existing subroutines.

chained

Set this to a true value to change the return value of setters and mutators (when called with an argument). If `chained` is enabled, the setters and accessors/mutators will return the object. Mutators called without an argument still return the value of the associated attribute.

As with the other options, `chained` affects all methods generated in the same `use Class::XSAccessor::Array ...` statement.

class

By default, the accessors are generated in the calling class. Using the `class` option, you can explicitly specify where the methods are to be generated.

LVALUES

Support for lvalue accessors via the keyword `lvalue_accessors` was added in version 1.08. At this point, **THEY ARE CONSIDERED HIGHLY EXPERIMENTAL**. Furthermore, their performance hasn't been benchmarked yet.

The following example demonstrates an lvalue accessor:

```
package Address;
use Class::XSAccessor
  constructor => 'new',
  lvalue_accessors => { zip_code => 0 };

package main;
my $address = Address->new(2);
print $address->zip_code, "\n"; # prints 2
$address->zip_code = 76135; # <--- This is it!
print $address->zip_code, "\n"; # prints 76135
```

CAVEATS

Probably wouldn't work if your objects are *tied*. But that's a strange thing to do anyway.

Note that compiling C code at run-time a la Inline::C is a no go.

Threading. With version 1.00, a memory leak has been **fixed** that would leak a small amount of memory if you loaded `Class::XSAccessor`-based classes in a subthread that hadn't been loaded in the "main" thread before. If the subthread then terminated, a hash key and an int per associated method used to be lost. Note that this mattered only if classes were **only** loaded in a sort of throw-away thread.

In the new implementation as of 1.00, the memory will not be released again either in the above situation. But it will be recycled when the same class or a similar class is

loaded again in **any** thread.

SEE ALSO

Class::XSAccessor

AutoXS

AUTHOR

Steffen Mueller smueller@cpan.org

chocolateboy chocolate@cpan.org

COPYRIGHT AND LICENSE

Copyright (C) 2008, 2009, 2010, 2011, 2012, 2013 by Steffen Mueller

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself, either Perl version 5.8 or, at your option, any later version of Perl 5 you may have available.

Functions

Other Functions

`getter`

Array `$obj->name`: element `index`, or undef.

`lvalue_accessor`

Array `$obj->name = $value`.

`setter`

Array `$obj->set_name($value)`: returns the new value.

`chained_setter`

Array setter under `chained => 1`: returns the object.

`accessor`

Array `$obj->name` / `$obj->name($value)`.

`chained_accessor`

Array accessor under `chained => 1`.

predicate

Array `$obj->has_name`: whether element `index` is defined.

constructor

Array `MyClass->new`: an empty array-based object; arguments are ignored.

newxs_getter

Array `newxs_getter(name, index)` and its ALIASes newxs_lvalue_accessor (1), newxs_predicate (2).

Also available as `Class::XSAccessor::Array::newxs_lvalue_accessor`, `Class::XSAccessor::Array::newxs_predicate`.

newxs_accessor

Array `newxs_setter(name, index, chained)` and its ALIAS newxs_accessor (1).

Also available as `Class::XSAccessor::Array::newxs_setter`.

newxs_constructor

Array `newxs_constructor(name)`.

Class::XSAccessor::Heavy

Guts you don't care about

SYNOPSIS

```
use Class::XSAccessor!
```

DESCRIPTION

Common guts for Class::XSAccessor and Class::XSAccessor::Array. No user-serviceable parts inside!

SEE ALSO

Class::XSAccessor Class::XSAccessor::Array

AUTHOR

Steffen Mueller, smueller@cpan.org

chocolateboy, chocolate@cpan.org

COPYRIGHT AND LICENSE

Copyright (C) 2008, 2009, 2010, 2011, 2012, 2013 by Steffen Mueller

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself, either Perl version 5.8 or, at your option, any later version of Perl 5 you may have available.

5.2.16 Clone

Make a deep, independent copy of any Perl value - hashes, arrays, scalars, references, blessed objects, tied variables, and weak references.

`clone` walks a data structure recursively and returns a fresh copy. Shared references inside the input become shared references inside the copy (no accidental duplication), circular structures are reproduced without infinite recursion, weak references come out still weak, and tied variables keep their tie. Modifying the clone never touches the original.

For pure-Perl deep copies, `Clone` is typically faster than `Storable::dclone` on shallow to medium-depth structures (roughly three levels or fewer).

Synopsis

```
use Clone qw(clone);

my $copy = clone($original);       # function form

package Foo;
use parent 'Clone';
my $other = $obj->clone;           # method form (inherited)
```

`clone($data, $depth)` copies only `$depth` levels and shares what lies below. This is a native implementation of Clone 0.50; its `Clone.pm` is used unchanged, and nothing is exported by default.

Differences from upstream

None.

Functions

Cloning

clone

Return a deep, independent copy of a Perl value.

clone

Return a deep, independent copy of a Perl value.

Synopsis

```
use Clone qw(clone);

my $copy = clone($thing);          # function form
my $copy = clone($thing, 5);       # limit recursion depth
my $copy = $obj->clone;            # method form (class inherits
 ↪from Clone)
```

What you get back

A value of the same kind as the input. The copy is fully independent: assigning into `$copy` - at any nesting level - never changes `$thing`.

- Shared references inside the input are de-duplicated: if two slots pointed at the same referent, the corresponding two slots in the copy point at *one* fresh clone of that referent, not two separate clones.
- Circular structures are reproduced with their cycles intact.
- Weak references (see `Scalar::Util::weaken`) come out weak in the copy.
- Tied variables keep their tie; the tie's internal object is itself cloned.
- Blessed objects are reblessed into the same package as the original.

Examples

Basic hash clone - the copy is independent:

```
use Clone qw(clone);
my $h = { name => 'Alice', tags => [qw(admin user)] };
my $c = clone($h);
push @{ $c->{tags} }, 'guest';
print "@{ $h->{tags} }\n";         # admin user
print "@{ $c->{tags} }\n";         # admin user guest
```

Shared references stay shared inside the clone:

```
my $shared = { count => 0 };
my $pair   = { a => $shared, b => $shared };
my $copy   = clone($pair);
$copy->{a}{count} = 42;
print $copy->{b}{count};           # 42  (a and b still point at one
 ↪hash)
print $pair->{a}{count};           # 0   (original untouched)
```

Circular structures:

```
my $a = { name => 'A' };
my $b = { name => 'B', peer => $a };
$a->{peer} = $b;                    # cycle
my $copy = clone($a);
print $copy->{peer}{peer}{name};    # A  (cycle reproduced)
```

Edge cases

- `undef` clones to `undef`.
- A non-reference scalar (`42`, `"hi"`) clones by value - you get a fresh scalar with the same contents.
- Code references are shared, not duplicated: the clone points at the same sub.
- Filehandles and `IO` objects are cloned but share the underlying file descriptor and file position.
- The depth argument caps recursion. `clone($thing, 0)` returns `$thing` unchanged (refcount bump, no copy). A negative or omitted depth means "unlimited"; the runtime still guards against stack overflow by falling back to an iterative cloner once rdepth passes 4000.

Differences from upstream

Fully compatible with upstream `Clone` 0.50.

See also

- `Storable::dclone` - serializer-based deep copy; typically faster for structures 4+ levels deep, slower for shallow ones.
- `Scalar::Util::weaken` - create the weak references that `clone` preserves.
- `Clone::PP` - pure-Perl implementation of the same interface.

5.2.17 Compress

Namespace containing the following modules:

- `Compress::Raw`
- `Compress::Zlib` - Interface to zlib compression library

Compress::Raw

Namespace containing the following modules:

- `Compress::Raw::Bunzip2`
- `Compress::Raw::Bzip2` - Low-level interface to the `bzip2` compression library.
- `Compress::Raw::Zlib` - Low-level interface to the `zlib` compression library.

Compress::Raw::Bunzip2

Functions

Other Functions

new

`Compress::Raw::Bunzip2->new([$appendOut, $consume, $small, $verbosity, $limitOutput])`: a decompressor. In list context also the status, `BZ_OK` on success.

DispStream

`$bunz->DispStream([$message])`: prints the stream's state to stdout, for debugging.

bzinflate

`$bunz->bzinflate($input, $output)`: decompresses `$input` into `$output`, consuming the input it used when the object was made with `consume`. Returns `BZ_OK` while more input is needed, `BZ_STREAM_END` at the end of the stream.

inflateCount

`$bunz->inflateCount()`: bytes the last `bzinflate` produced.

status

`$bunz->status()`: the status the last call returned.

total_in_lo32

`$bunz->total_in_lo32()`: compressed bytes taken in so far (low 32 bits).

total_out_lo32

`$bunz->total_out_lo32()`: bytes put out so far (low 32 bits).

compressedBytes

`$bunz->compressedBytes()`: compressed bytes consumed so far.

uncompressedBytes

`$bunz->uncompressedBytes()`: uncompressed bytes produced so far.

Compress::Raw::Bzip2

Low-level interface to the bzip2 compression library.

`Compress::Raw::Bzip2` is the layer underneath `IO::Compress::Bzip2` and `IO::Uncompress::Bunzip2`, and what Archive::Tar uses for `.tbz` archives. It gives direct access to libbz2's streaming compressor (`Compress::Raw::Bzip2`) and decompressor (`Compress::Raw::Bunzip2`) and the `BZ_*` status constants. Most programs are better served by `IO::Compress::Bzip2` and friends; this module is for code that needs the streams themselves.

```
use Compress::Raw::Bzip2;

my $data = "hello, world\n" x 100;

# Compress: one bzdeflate call per chunk, bzclose at the end.
my ($bz, $status) = Compress::Raw::Bzip2->new(1, 9);   # append,␣
 ↪900k blocks
$status == BZ_OK or die "new: $status";
my $packed = '';
$bz->bzdeflate($data, $packed) == BZ_RUN_OK or die;
$bz->bzclose($packed) == BZ_STREAM_END or die;
printf "%d -> %d bytes\n", $bz->uncompressedBytes, $bz->
 ↪compressedBytes;
# 1300 -> 66 bytes

# Uncompress: bzinflate eats its input and says BZ_STREAM_END at the␣
 ↪end.
my $bunz = Compress::Raw::Bunzip2->new;
my $out = '';
$status = $bunz->bzinflate($packed, $out);
print "status: $status, round trip ", ($out eq $data ? "ok" : "BAD
 ↪"), "\n";
# status: End of Stream, round trip ok
```

Every status is a dualvar: numeric `BZ_*` and the message text, so `$status == BZ_STREAM_END` and `"$status"` both work. `bzdeflate`, `bzflush` and `bzclose` append to the output buffer when the object was made with `appendOut`; `bzinflate` consumes its input when made with `consume` and, with `limitOutput`, decompresses at most one buffer's worth per call.

This is a native implementation of the XS half of Compress::Raw::Bzip2 2.218; `Compress/Raw/Bzip2.pm` is upstream's own. libbz2 is the system's `libbz2.so.1`, loaded when the module is first used.

Differences from upstream

None.

Functions

Other Functions

bzlibversion

`Compress::Raw::Bzip2::bzlibversion()`: the version string of the libbz2 this process loaded, such as "1.0.8, 13-Jul-2019".

new

`Compress::Raw::Bzip2->new([$appendOut, $blockSize100k, $workfactor, $verbosity])`: a compressor. In list context also the status, `BZ_OK` on success.

DispStream

`$bz->DispStream([$message])`: prints the stream's state to stdout, for debugging.

bzdeflate

`$bz->bzdeflate($input, $output)`: compresses `$input`, appending to (or replacing, without appendOut) `$output`. Returns `BZ_RUN_OK` on success.

bzclose

`$bz->bzclose($output)`: ends the stream, writing what is pending and the end-of-stream marker into `$output`. Returns `BZ_STREAM_END`.

bzflush

`$bz->bzflush($output)`: writes what is pending into `$output` without ending the stream. Returns `BZ_RUN_OK`.

total_in_lo32

`$bz->total_in_lo32()`: bytes taken in so far (low 32 bits).

total_out_lo32

`$bz->total_out_lo32()`: bytes put out so far (low 32 bits).

compressedBytes

`$bz->compressedBytes()`: compressed bytes produced so far.

uncompressedBytes

`$bz->uncompressedBytes()`: uncompressed bytes taken in so far.

Compress::Raw::Zlib

Low-level interface to the `zlib` compression library.

`Compress::Raw::Zlib` is the layer underneath `Compress::Zlib` and the whole `IO::Compress::*` / `IO::Uncompress::*` family. It gives direct access to zlib's streaming compressor and decompressor, the `crc32` and `adler32` checksums, and the zlib constants. Most programs are better served by `IO::Compress::Gzip` and friends; this module is for code that needs the streams themselves.

```
use Compress::Raw::Zlib;

my $data = "hello, world\n" x 100;

# Compress: one deflate call per chunk, flush at the end.
my ($d, $status) = Compress::Raw::Zlib::Deflate->new(
    -Level => Z_BEST_COMPRESSION, -AppendOutput => 1, -CRC32 => 1);
$status == Z_OK or die "deflateInit: $status";
my $packed = '';
$d->deflate($data, $packed) == Z_OK or die $d->msg;
$d->flush($packed) == Z_OK or die $d->msg;
printf "%d -> %d bytes, crc32 %08x\n", $d->total_in, $d->total_out,
 ↪$d->crc32;
# 1300 -> 32 bytes, crc32 6d00e0f5

# Uncompress: inflate eats its input and says Z_STREAM_END at the␣
 ↪end.
my $i = Compress::Raw::Zlib::Inflate->new(-AppendOutput => 1);
my $out = '';
$status = $i->inflate($packed, $out);
print "status: $status, round trip ", ($out eq $data ? "ok" : "BAD
 ↪"), "\n";
# status: stream end, round trip ok

# The checksums, one-off or running, and merged.
my $crc = crc32("hello, ");
$crc = crc32("world", $crc);
printf "%08x %08x\n", $crc,
    Compress::Raw::Zlib::crc32_combine(crc32("hello, "), crc32(
 ↪"world"), 5);
# ffab723a ffab723a
```

This is a native implementation of `Compress::Raw::Zlib` 2.222; its `Compress/Raw/Zlib.pm` (the option parser, the constructors, `deflateParams`) is used unchanged. `use Compress::Raw::Zlib` exports `crc32`, `adler32` and every constant: the `Z_*` status, flush, level and strategy values, `MAX_WBITS`, `MAX_MEM_LEVEL`, `WANT_GZIP`, `WANT_GZIP_OR_ZLIB`, `OS_CODE`, `ZLIB_VERSION`, `ZLIB_VERNUM` and the `ZLIBNG_*` version parts.

Status values

The constructors and the stream methods that do work (`deflate`, `flush`, `deflateParams`, `inflate`, `inflateSync`, the resets) return a dual value (`deflateTune` returns a plain number): it compares numerically with the `Z_*` constants (`Z_OK`, `Z_STREAM_END`, `Z_BUF_ERROR`, `Z_DATA_ERROR`, ...) and prints as zlib's text for it ("stream end", "data error"). `$stream->status` repeats the last one as a plain number, and `$stream->msg` returns zlib's message for the last error.

Compressing: Compress::Raw::Zlib::Deflate

`my ($d, $status) = Compress::Raw::Zlib::Deflate->new(OPTIONS)` returns the stream and `Z_OK`, or `undef` and the zlib error code. In scalar context it returns the stream only. The options, all optional, are given as `-Name => value` pairs (or one hash reference):

- `-Level`: 0 to 9, or `Z_NO_COMPRESSION`, `Z_BEST_SPEED`, `Z_BEST_COMPRESSION`, `Z_DEFAULT_COMPRESSION` (the default).
- `-Method`: `Z_DEFLATED`, the only method and the default.
- `-WindowBits`: 8 to 15 for a zlib (RFC 1950) stream, `-MAX_WBITS` for raw deflate (RFC 1951), `WANT_GZIP` for gzip (RFC 1952). The default is `MAX_WBITS`.
- `-MemLevel`: zlib's `memLevel`, default `MAX_MEM_LEVEL`.
- `-Strategy`: `Z_DEFAULT_STRATEGY` (the default), `Z_FILTERED`, `Z_RLE`, `Z_FIXED` or `Z_HUFFMAN_ONLY`.
- `-Dictionary`: a preset dictionary; `$d->dict_adler` is its Adler-32.
- `-Bufsize`: the initial output buffer size and its growth step, default 4096.
- `-AppendOutput`: when true, output is appended to the output argument; when false (the default) the argument is overwritten.
- `-CRC32`, `-ADLER32`: keep a running checksum of the uncompressed data, read with `$d->crc32` / `$d->adler32`.

The methods:

- `$d->deflate($input, $output)` compresses all of `$input` (a scalar or a scalar reference) into `$output` and returns `Z_OK`. zlib may hold data back, so an empty `$output` after a successful call is not an error.
- `$d->flush($output [, $flush_type])` writes out what is pending. The default `Z_FINISH` ends the stream; `Z_SYNC_FLUSH`, `Z_FULL_FLUSH`, `Z_PARTIAL_FLUSH` and `Z_NO_FLUSH` keep it open.
- `$d->deflateReset` starts a new stream on the same object, after a finishing flush.
- `$d->deflateParams(-Level => ..., -Strategy => ..., -BufSize => ...)` changes those settings between calls; options not given stay.
- `$d->deflateTune($good_length, $max_lazy, $nice_length, $max_chain)` is zlib's `deflateTune`.
- `$d->get_Level`, `$d->get_Strategy`, `$d->get_Bufsize` return the settings in force.

- `$d->total_in` and `$d->total_out` count the bytes taken and given by zlib for the current stream; `$d->uncompressedBytes` and `$d->compressedBytes` are the same counts kept by the binding.
- `$d->crc32`, `$d->adler32`, `$d->dict_adler` return the checksums described above (0 when the option is off); `$d->status` the status of the last call; `$d->msg` zlib's last error message.

Uncompressing: Compress::Raw::Zlib::Inflate

`my ($i, $status) = Compress::Raw::Zlib::Inflate->new(OPTIONS)` takes `-WindowBits` (as for deflate, plus `WANT_GZIP_OR_ZLIB` to accept either header), `-Bufsize`, `-Dictionary`, `-AppendOutput`, `-CRC32`, `-ADLER32`, and:

- `-ConsumeInput` (default true): `inflate` removes the compressed data it used from its input argument. What is left after `Z_STREAM_END` is whatever followed the stream - the trailer of a container format, for example.
- `-LimitOutput`: `inflate` stops when the output buffer is full instead of uncompressing all of its input, and returns `Z_BUF_ERROR` or `Z_OK` with input left over; call it again until the input is used up. It turns `-ConsumeInput` on.

The methods:

- `$i->inflate($input, $output [, $eof])` uncompresses `$input` into `$output` and returns `Z_OK`, or `Z_STREAM_END` at the end of the compressed data. `$eof` matters only to zlib older than 1.2.0.
- `$i->inflateSync($input)` skips forward to the next full flush point (made by `flush(Z_FULL_FLUSH)`), to resume after corrupt data.
- `$i->inflateReset` starts a new stream on the same object.
- `$i->set_Append($bool)` switches `-AppendOutput` and returns the old setting.
- `$i->total_in`, `$i->total_out`, `$i->compressedBytes`, `$i->uncompressedBytes`, `$i->inflateCount`, `$i->get_Bufsize`, `$i->crc32`, `$i->adler32`, `$i->dict_adler`, `$i->status` and `$i->msg` report as their deflate counterparts do.

`Compress::Raw::Zlib::InflateScan->new` makes an `inflateScanStream`, which `IO::Compress::Zip` uses to append to an existing deflate stream: `scan`, `getEndOffset`, `getLastBlockOffset`, `getLastBufferOffset`, `resetLastBlockByte`, `inflateReset`, `inflateCount`, `createDeflateStream`, and the same counters as above. Upstream does not document it, and neither does this page. Every stream class also has `DispStream`, which prints the stream's internal state for debugging.

The zlib it uses

zlib is part of the binary, as upstream builds its own copy into the module: no library is loaded, and the module works wherever pperl runs. The implementation is zlib-rs, which has the zlib 1.3 API. `ZLIB_VERSION` reads `1.3.0-zlib-rs-` and the zlib-rs version; `ZLIB_VERNUM` is `0x1300`.

Differences from upstream

- **Compressed bytes.** A stream deflated here is a valid stream that any zlib inflates, and any zlib's stream inflates here, but it is not byte-for-byte the stream perl5 writes for the same input (level 0 excepted), and its size differs slightly. Upstream promises no particular bytes either: they change with the zlib it is built on.
- **Version.** `ZLIB_VERSION` and `ZLIB_VERNUM` are as above, where perl 5.44 with its bundled zlib reports `1.3.2` and `0x1320`.
- There is no zlib-ng mode: the four `is_zlib*` functions answer as for plain zlib.

Functions

Checksums

adler32

`adler32($buffer [, $adler])`: the Adler-32 checksum of `$buffer`. Pass the previous result as `$adler` to continue a running checksum; without it, or with `undef`, the checksum starts afresh. `$buffer` may be a scalar reference. Exported by default.

crc32

`crc32($buffer [, $crc [, $offset]])`: the CRC-32 checksum of `$buffer`, as gzip and zip use it. Pass the previous result as `$crc` to continue a running checksum; without it, or with `undef`, it starts afresh. `$offset` skips that many leading bytes; an offset past the end dies "Offset out of range in Compress::Raw::Zlib::crc32". `$buffer` may be a scalar reference. Exported by default.

crc32_combine

`Compress::Raw::Zlib::crc32_combine($crc1, $crc2, $len2)`: the CRC-32 of two pieces joined, from the CRC-32 of each and the length of the second - without the data. Not exported.

adler32_combine

`Compress::Raw::Zlib::adler32_combine($adler1, $adler2, $len2)`: as `crc32_combine`, for Adler-32. Not exported.

Library info

zlib_version

`Compress::Raw::Zlib::zlib_version()`: the version string of the zlib in use: `1.3.0-zlib-rs-` and the zlib-rs version.

zlibng_version

`Compress::Raw::Zlib::zlibng_version()`: the zlib-ng version string when the library is zlib-ng, and an empty string for zlib.

is_zlib_native

`Compress::Raw::Zlib::is_zlib_native()`: true when the library is zlib itself rather than zlib-ng. With `is_zlibng`, `is_zlibng_compat` and `is_zlibng_native` it tells which library the streams run on.

is_zlibng_native

`Compress::Raw::Zlib::is_zlibng_native()`: true when the binding uses zlib-ng's own API. Always false here: the library is reached through the zlib API.

is_zlibng_compat

`Compress::Raw::Zlib::is_zlibng_compat()`: true when the library is a zlib-ng built with the zlib-compatible API. Always false here.

is_zlibng

`Compress::Raw::Zlib::is_zlibng()`: true when the library is zlib-ng, in either mode.

ZLIB_VERNUM

`ZLIB_VERNUM`: zlib's version as a number, one hex digit per part: `0x1320` for 1.3.2. Exported by default.

zlibCompileFlags

`Compress::Raw::Zlib::zlibCompileFlags()`: zlib's `zlibCompileFlags`, the bit set describing how the library was built (type sizes, debug, assembler code); zlib.h documents the bits. 0 for zlib before 1.2.1.

ZLIBNG_VER_STATUS

`ZLIBNG_VER_STATUS`: the release status part of the zlib-ng version as a string; `"0"` when the library is zlib. Exported by default.

adler32

`adler32($buffer [, $adler])`: the Adler-32 checksum of `$buffer`. Pass the previous result as `$adler` to continue a running checksum; without it, or with `undef`, the checksum starts afresh. `$buffer` may be a scalar reference. Exported by default.

```
my $a = adler32("hello, ");
$a = adler32("world", $a);        # same as adler32("hello, world")
```

A string with characters above 255 that cannot be downgraded dies “Wide character in Compress::Raw::Zlib::adler32”.

Compress::Zlib

Interface to zlib compression library

SYNOPSIS

```
use Compress::Zlib ;

($d, $status) = deflateInit( [OPT] ) ;
$status = $d->deflate($input, $output) ;
$status = $d->flush([$flush_type]) ;
$d->deflateParams(OPTS) ;
$d->deflateTune(OPTS) ;
$d->dict_adler() ;
$d->crc32() ;
$d->adler32() ;
$d->total_in() ;
$d->total_out() ;
$d->msg() ;
$d->get_Strategy();
$d->get_Level();
$d->get_BufSize();

($i, $status) = inflateInit( [OPT] ) ;
$status = $i->inflate($input, $output [, $eof]) ;
$status = $i->inflateSync($input) ;
$i->dict_adler() ;
$d->crc32() ;
$d->adler32() ;
$i->total_in() ;
$i->total_out() ;
$i->msg() ;
$d->get_BufSize();

$dest = compress($source) ;
$dest = uncompress($source) ;

$gz = gzopen($filename or filehandle, $mode) ;
$bytesread = $gz->gzread($buffer [,$size]) ;
$bytesread = $gz->gzreadline($line) ;
$byteswritten = $gz->gzwrite($buffer) ;
$status = $gz->gzflush($flush) ;
$offset = $gz->gztell() ;
$status = $gz->gzseek($offset, $whence) ;
$status = $gz->gzclose() ;
$status = $gz->gzeof() ;
$status = $gz->gzsetparams($level, $strategy) ;
```

(continues on next page)

(continued from previous page)

```
$errstring = $gz->gzerror() ;
$gzerrno

$dest = Compress::Zlib::memGzip($buffer) ;
$dest = Compress::Zlib::memGunzip($buffer) ;

$crc = adler32($buffer [,$crc]) ;
$crc = crc32($buffer [,$crc]) ;

$crc = crc32_combine($crc1, $crc2, $len2);
$adler = adler32_combine($adler1, $adler2, $len2);

my $version = Compress::Raw::Zlib::zlib_version();
```

DESCRIPTION

The *Compress::Zlib* module provides a Perl interface to the *zlib* compression library (see "AUTHOR" for details about where to get *zlib*).

The `Compress::Zlib` module can be split into two general areas of functionality, namely a simple read/write interface to *gzip* files and a low-level in-memory compression/decompression interface.

Each of these areas will be discussed in the following sections.

Notes for users of Compress::Zlib version 1

The main change in `Compress::Zlib` version 2.x is that it does not now interface directly to the zlib library. Instead it uses the `IO::Compress::Gzip` and `IO::Uncompress::Gunzip` modules for reading/writing gzip files, and the `Compress::Raw::Zlib` module for some low-level zlib access.

The interface provided by version 2 of this module should be 100% backward compatible with version 1. If you find a difference in the expected behaviour please contact the author (See "AUTHOR"). See "GZIP INTERFACE"

With the creation of the `IO::Compress` and `IO::Uncompress` modules no new features are planned for `Compress::Zlib` - the new modules do everything that `Compress::Zlib` does and then some. Development on `Compress::Zlib` will be limited to bug fixes only.

If you are writing new code, your first port of call should be one of the new `IO::Compress` or `IO::Uncompress` modules.

GZIP INTERFACE

A number of functions are supplied in *zlib* for reading and writing *gzip* files that conform to RFC 1952. This module provides an interface to most of them.

If you have previously used `Compress::Zlib` 1.x, the following enhancements/changes have been made to the `gzopen` interface:

1. If you want to open either STDIN or STDOUT with `gzopen`, you can now optionally use the special filename "`-`" as a synonym for `\*STDIN` and `\*STDOUT`.

2. In `Compress::Zlib` version 1.x, `gzopen` used the zlib library to open the underlying file. This made things especially tricky when a Perl filehandle was passed to `gzopen`. Behind the scenes the numeric C file descriptor had to be extracted from the Perl filehandle and this passed to the zlib library.

   Apart from being non-portable to some operating systems, this made it difficult to use `gzopen` in situations where you wanted to extract/create a gzip data stream that is embedded in a larger file, without having to resort to opening and closing the file multiple times.

   It also made it impossible to pass a perl filehandle that wasn't associated with a real filesystem file, like, say, an `IO::String`.

   In `Compress::Zlib` version 2.x, the `gzopen` interface has been completely rewritten to use the *IO::Compress::Gzip* for writing gzip files and *IO::Uncompress::Gunzip* for reading gzip files. None of the limitations mentioned above apply.

3. Addition of `gzseek` to provide a restricted `seek` interface.

4. Added `gztell`.

A more complete and flexible interface for reading/writing gzip files/buffers is included with the module `IO-Compress-Zlib`. See *IO::Compress::Gzip* and *IO::Uncompress::Gunzip* for more details.

- **$gz = gzopen($filename, $mode)**
- **$gz = gzopen($filehandle, $mode)**

  This function opens either the *gzip* file `$filename` for reading or writing or attaches to the opened filehandle, `$filehandle`. It returns an object on success and `undef` on failure.

  When writing a gzip file this interface will *always* create the smallest possible gzip header (exactly 10 bytes). If you want greater control over what gets stored in the gzip header (like the original filename or a comment) use *IO::Compress::Gzip* instead. Similarly if you want to read the contents of the gzip header use *IO::Uncompress::Gunzip*.

  The second parameter, `$mode`, is used to specify whether the file is opened for reading or writing and to optionally specify a compression level and compression strategy when writing. The format of the `$mode` parameter is similar to the mode parameter to the 'C' function `fopen`, so "rb" is used to open for reading, "wb" for writing and "ab" for appending (writing at the end of the file).

  To specify a compression level when writing, append a digit between 0 and 9 to the mode string – 0 means no compression and 9 means maximum compression. If no compression level is specified Z_DEFAULT_COMPRESSION is used.

  To specify the compression strategy when writing, append 'f' for filtered data, 'h' for Huffman only compression, or 'R' for run-length encoding. If no strategy is specified Z_DEFAULT_STRATEGY is used.

  So, for example, "wb9" means open for writing with the maximum compression using the default strategy and "wb4R" means open for writing with compression level 4 and run-length encoding.

Refer to the *zlib* documentation for the exact format of the `$mode` parameter.

- **$bytesread = $gz->gzread($buffer [, $size]) ;**

  Reads `$size` bytes from the compressed file into `$buffer`. If `$size` is not specified, it will default to 4096. If the scalar `$buffer` is not large enough, it will be extended automatically.

  Returns the number of bytes actually read. On EOF it returns 0 and in the case of an error, -1.

- **$bytesread = $gz->gzreadline($line) ;**

  Reads the next line from the compressed file into `$line`.

  Returns the number of bytes actually read. On EOF it returns 0 and in the case of an error, -1.

  It is legal to intermix calls to `gzread` and `gzreadline`.

  To maintain backward compatibility with version 1.x of this module `gzreadline` ignores the `$/` variable - it *always* uses the string `"\n"` as the line delimiter.

  If you want to read a gzip file a line at a time and have it respect the `$/` variable (or `$INPUT_RECORD_SEPARATOR`, or `$RS` when `English` is in use) see *IO::Uncompress::Gunzip*.

- **$byteswritten = $gz->gzwrite($buffer) ;**

  Writes the contents of `$buffer` to the compressed file. Returns the number of bytes actually written, or 0 on error.

- **$status = $gz->gzflush($flush_type) ;**

  Flushes all pending output into the compressed file.

  This method takes an optional parameter, `$flush_type`, that controls how the flushing will be carried out. By default the `$flush_type` used is `Z_FINISH`. Other valid values for `$flush_type` are `Z_NO_FLUSH`, `Z_SYNC_FLUSH`, `Z_FULL_FLUSH` and `Z_BLOCK`. It is strongly recommended that you only set the `flush_type` parameter if you fully understand the implications of what it does - overuse of `flush` can seriously degrade the level of compression achieved. See the `zlib` documentation for details.

  Returns 0 on success.

- **$offset = $gz->gztell() ;**

  Returns the uncompressed file offset.

- **$status = $gz->gzseek($offset, $whence) ;**

  Provides a sub-set of the `seek` functionality, with the restriction that it is only legal to seek forward in the compressed file. It is a fatal error to attempt to seek backward.

  When opened for writing, empty parts of the file will have NULL (0x00) bytes written to them.

  The `$whence` parameter should be one of SEEK_SET, SEEK_CUR or SEEK_END.

  Returns 1 on success, 0 on failure.

- **$gz->gzclose**

  Closes the compressed file. Any pending data is flushed to the file before it is closed.

  Returns 0 on success.

- **$gz->gzsetparams($level, $strategy**

  Change settings for the deflate stream `$gz`.

  The list of the valid options is shown below. Options not specified will remain unchanged.

  Note: This method is only available if you are running zlib 1.0.6 or better.

  - **$level**

    Defines the compression level. Valid values are 0 through 9, `Z_NO_COMPRESSION`, `Z_BEST_SPEED`, `Z_BEST_COMPRESSION`, and `Z_DEFAULT_COMPRESSION`.

  - **$strategy**

    Defines the strategy used to tune the compression. The valid values are `Z_DEFAULT_STRATEGY`, `Z_FILTERED` and `Z_HUFFMAN_ONLY`.

- **$gz->gzerror**

  Returns the *zlib* error message or number for the last operation associated with `$gz`. The return value will be the *zlib* error number when used in a numeric context and the *zlib* error message when used in a string context. The *zlib* error number constants, shown below, are available for use.

```
Z_OK
Z_STREAM_END
Z_ERRNO
Z_STREAM_ERROR
Z_DATA_ERROR
Z_MEM_ERROR
Z_BUF_ERROR
```

- **$gzerrno**

  The `$gzerrno` scalar holds the error code associated with the most recent *gzip* routine. Note that unlike `gzerror()`, the error is *not* associated with a particular file.

  As with `gzerror()` it returns an error number in numeric context and an error message in string context. Unlike `gzerror()` though, the error message will correspond to the *zlib* message when the error is associated with *zlib* itself, or the UNIX error message when it is not (i.e. *zlib* returned `Z_ERRORNO`).

  As there is an overlap between the error numbers used by *zlib* and UNIX, `$gzerrno` should only be used to check for the presence of *an* error in numeric context. Use `gzerror()` to check for specific *zlib* errors. The *gzcat* example below shows how the variable can be used safely.

Examples

Here is an example script which uses the interface. It implements a *gzcat* function.

```
use strict ;
use warnings ;

use Compress::Zlib ;

# use stdin if no files supplied
@ARGV = '-' unless @ARGV ;

foreach my $file (@ARGV) {
    my $buffer ;

    my $gz = gzopen($file, "rb")
         or die "Cannot open $file: $gzerrno\n" ;

    print $buffer while $gz->gzread($buffer) > 0 ;

    die "Error reading from $file: $gzerrno" . ($gzerrno+0) . "\n"
        if $gzerrno != Z_STREAM_END ;

    $gz->gzclose() ;
}
```

Below is a script which makes use of `gzreadline`. It implements a very simple *grep* like script.

```
use strict ;
use warnings ;

use Compress::Zlib ;

die "Usage: gzgrep pattern [file...]\n"
    unless @ARGV >= 1;

my $pattern = shift ;

# use stdin if no files supplied
@ARGV = '-' unless @ARGV ;

foreach my $file (@ARGV) {
    my $gz = gzopen($file, "rb")
         or die "Cannot open $file: $gzerrno\n" ;

    while ($gz->gzreadline($_) > 0) {
        print if /$pattern/ ;
    }

    die "Error reading from $file: $gzerrno\n"
```

(continues on next page)

(continued from previous page)

```
        if $gzerrno != Z_STREAM_END ;

    $gz->gzclose() ;
}
```

This script, *gzstream*, does the opposite of the *gzcat* script above. It reads from standard input and writes a gzip data stream to standard output.

```
use strict ;
use warnings ;

use Compress::Zlib ;

binmode STDOUT;  # gzopen only sets it on the fd

my $gz = gzopen(\*STDOUT, "wb")
      or die "Cannot open stdout: $gzerrno\n" ;

while (<>) {
    $gz->gzwrite($_)
      or die "error writing: $gzerrno\n" ;
}

$gz->gzclose ;
```

Compress::Zlib::memGzip

This function is used to create an in-memory gzip file with the minimum possible gzip header (exactly 10 bytes).

```
$dest = Compress::Zlib::memGzip($buffer)
    or die "Cannot compress: $gzerrno\n";
```

If successful, it returns the in-memory gzip file. Otherwise it returns `undef` and the `$gzerrno` variable will store the zlib error code.

The `$buffer` parameter can either be a scalar or a scalar reference.

See *IO::Compress::Gzip* for an alternative way to carry out in-memory gzip compression.

Compress::Zlib::memGunzip

This function is used to uncompress an in-memory gzip file.

```
$dest = Compress::Zlib::memGunzip($buffer)
    or die "Cannot uncompress: $gzerrno\n";
```

If successful, it returns the uncompressed gzip file. Otherwise it returns `undef` and the `$gzerrno` variable will store the zlib error code.

The `$buffer` parameter can either be a scalar or a scalar reference. The contents of the `$buffer` parameter are destroyed after calling this function.

If `$buffer` consists of multiple concatenated gzip data streams only the first will be uncompressed. Use `gunzip` with the `MultiStream` option in the `IO::Uncompress::Gunzip` module if you need to deal with concatenated data streams.

See *IO::Uncompress::Gunzip* for an alternative way to carry out in-memory gzip uncompression.

COMPRESS/UNCOMPRESS

Two functions are provided to perform in-memory compression/uncompression of RFC 1950 data streams. They are called `compress` and `uncompress`.

- **$dest = compress($source [, $level]) ;**

  Compresses `$source`. If successful it returns the compressed data. Otherwise it returns *undef*.

  The source buffer, `$source`, can either be a scalar or a scalar reference.

  The `$level` parameter defines the compression level. Valid values are 0 through 9, `Z_NO_COMPRESSION`, `Z_BEST_SPEED`, `Z_BEST_COMPRESSION`, and `Z_DEFAULT_COMPRESSION`. If `$level` is not specified `Z_DEFAULT_COMPRESSION` will be used.

- **$dest = uncompress($source) ;**

  Uncompresses `$source`. If successful it returns the uncompressed data. Otherwise it returns *undef*.

  The source buffer can either be a scalar or a scalar reference.

Please note: the two functions defined above are *not* compatible with the Unix commands of the same name.

See IO::Deflate and IO::Inflate included with this distribution for an alternative interface for reading/writing RFC 1950 files/buffers.

Deflate Interface

This section defines an interface that allows in-memory compression using the *deflate* interface provided by zlib.

Here is a definition of the interface available:

($d, $status) = deflateInit([OPT])

Initialises a deflation stream.

It combines the features of the *zlib* functions `deflateInit`, `deflateInit2` and `deflateSetDictionary`.

If successful, it will return the initialised deflation stream, `$d` and `$status` of `Z_OK` in a list context. In scalar context it returns the deflation stream, `$d`, only.

If not successful, the returned deflation stream (`$d`) will be *undef* and `$status` will hold the exact *zlib* error code.

The function optionally takes a number of named options specified as `-Name=>value` pairs. This allows individual options to be tailored without having to specify them all in the parameter list.

For backward compatibility, it is also possible to pass the parameters as a reference to a hash containing the name=>value pairs.

The function takes one optional parameter, a reference to a hash. The contents of the hash allow the deflation interface to be tailored.

Here is a list of the valid options:

- **-Level**

  Defines the compression level. Valid values are 0 through 9, `Z_NO_COMPRESSION`, `Z_BEST_SPEED`, `Z_BEST_COMPRESSION`, and `Z_DEFAULT_COMPRESSION`.

  The default is Z_DEFAULT_COMPRESSION.

- **-Method**

  Defines the compression method. The only valid value at present (and the default) is Z_DEFLATED.

- **-WindowBits**

  To create an RFC 1950 data stream, set `WindowBits` to a positive number.

  To create an RFC 1951 data stream, set `WindowBits` to `-MAX_WBITS`.

  For a full definition of the meaning and valid values for `WindowBits` refer to the *zlib* documentation for *deflateInit2*.

  Defaults to MAX_WBITS.

- **-MemLevel**

  For a definition of the meaning and valid values for `MemLevel` refer to the *zlib* documentation for *deflateInit2*.

  Defaults to MAX_MEM_LEVEL.

- **-Strategy**

  Defines the strategy used to tune the compression. The valid values are `Z_DEFAULT_STRATEGY`, `Z_FILTERED` and `Z_HUFFMAN_ONLY`.

  The default is Z_DEFAULT_STRATEGY.

- **-Dictionary**

  When a dictionary is specified *Compress::Zlib* will automatically call `deflateSetDictionary` directly after calling `deflateInit`. The Adler32 value for the dictionary can be obtained by calling the method `$d->dict_adler()`.

  The default is no dictionary.

- **-Bufsize**

  Sets the initial size for the deflation buffer. If the buffer has to be reallocated to increase the size, it will grow in increments of `Bufsize`.

  The default is 4096.

Here is an example of using the `deflateInit` optional parameter list to override the default buffer size and compression level. All other options will take their default values.

```
deflateInit( -Bufsize => 300,
             -Level => Z_BEST_SPEED  ) ;
```

($out, $status) = $d->deflate($buffer)

Deflates the contents of `$buffer`. The buffer can either be a scalar or a scalar reference. When finished, `$buffer` will be completely processed (assuming there were no errors). If the deflation was successful it returns the deflated output, `$out`, and a status value, `$status`, of `Z_OK`.

On error, `$out` will be *undef* and `$status` will contain the *zlib* error code.

In a scalar context `deflate` will return `$out` only.

As with the *deflate* function in *zlib*, it is not necessarily the case that any output will be produced by this method. So don't rely on the fact that `$out` is empty for an error test.

($out, $status) = $d->flush()

=head2 **($out, $status)** = **$d->flush($flush_type)**

Typically used to finish the deflation. Any pending output will be returned via `$out`. `$status` will have a value `Z_OK` if successful.

In a scalar context `flush` will return `$out` only.

Note that flushing can seriously degrade the compression ratio, so it should only be used to terminate a decompression (using `Z_FINISH`) or when you want to create a *full flush point* (using `Z_FULL_FLUSH`).

By default the `flush_type` used is `Z_FINISH`. Other valid values for `flush_type` are `Z_NO_FLUSH`, `Z_PARTIAL_FLUSH`, `Z_SYNC_FLUSH` and `Z_FULL_FLUSH`. It is strongly recommended that you only set the `flush_type` parameter if you fully understand the implications of what it does. See the `zlib` documentation for details.

$status = $d->deflateParams([OPT])

Change settings for the deflate stream `$d`.

The list of the valid options is shown below. Options not specified will remain unchanged.

- **-Level**

  Defines the compression level. Valid values are 0 through 9, `Z_NO_COMPRESSION`, `Z_BEST_SPEED`, `Z_BEST_COMPRESSION`, and `Z_DEFAULT_COMPRESSION`.

- **-Strategy**

  Defines the strategy used to tune the compression. The valid values are `Z_DEFAULT_STRATEGY`, `Z_FILTERED` and `Z_HUFFMAN_ONLY`.

$d->dict_adler()

Returns the adler32 value for the dictionary.

$d->msg()

Returns the last error message generated by zlib.

$d->total_in()

Returns the total number of bytes uncompressed bytes input to deflate.

$d->total_out()

Returns the total number of compressed bytes output from deflate.

Example

Here is a trivial example of using `deflate`. It simply reads standard input, deflates it and writes it to standard output.

```
use strict ;
use warnings ;

use Compress::Zlib ;

binmode STDIN;
binmode STDOUT;
my $x = deflateInit()
   or die "Cannot create a deflation stream\n" ;

my ($output, $status) ;
while (<>)
{
    ($output, $status) = $x->deflate($_) ;

    $status == Z_OK
        or die "deflation failed\n" ;

    print $output ;
}

($output, $status) = $x->flush() ;

$status == Z_OK
    or die "deflation failed\n" ;

print $output ;
```

Inflate Interface

This section defines the interface available that allows in-memory uncompression using the *deflate* interface provided by zlib.

Here is a definition of the interface:

($i, $status) = inflateInit()

Initialises an inflation stream.

In a list context it returns the inflation stream, `$i`, and the *zlib* status code in `$status`. In a scalar context it returns the inflation stream only.

If successful, `$i` will hold the inflation stream and `$status` will be `Z_OK`.

If not successful, `$i` will be *undef* and `$status` will hold the *zlib* error code.

The function optionally takes a number of named options specified as `-Name=>value` pairs. This allows individual options to be tailored without having to specify them all in the parameter list.

For backward compatibility, it is also possible to pass the parameters as a reference to a hash containing the name=>value pairs.

The function takes one optional parameter, a reference to a hash. The contents of the hash allow the deflation interface to be tailored.

Here is a list of the valid options:

- **-WindowBits**

  To uncompress an RFC 1950 data stream, set `WindowBits` to a positive number.

  To uncompress an RFC 1951 data stream, set `WindowBits` to `-MAX_WBITS`.

  For a full definition of the meaning and valid values for `WindowBits` refer to the *zlib* documentation for *inflateInit2*.

  Defaults to MAX_WBITS.

- **-Bufsize**

  Sets the initial size for the inflation buffer. If the buffer has to be reallocated to increase the size, it will grow in increments of `Bufsize`.

  Default is 4096.

- **-Dictionary**

  The default is no dictionary.

Here is an example of using the `inflateInit` optional parameter to override the default buffer size.

```
inflateInit( -Bufsize => 300 ) ;
```

($out, $status) = $i->inflate($buffer)

Inflates the complete contents of `$buffer`. The buffer can either be a scalar or a scalar reference.

Returns `Z_OK` if successful and `Z_STREAM_END` if the end of the compressed data has been successfully reached. If not successful, `$out` will be *undef* and `$status` will hold the *zlib* error code.

The `$buffer` parameter is modified by `inflate`. On completion it will contain what remains of the input buffer after inflation. This means that `$buffer` will be an empty string when the return status is `Z_OK`. When the return status is `Z_STREAM_END` the `$buffer` parameter will contains what (if anything) was stored in the input buffer after the deflated data stream.

This feature is useful when processing a file format that encapsulates a compressed data stream (e.g. gzip, zip).

$status = $i->inflateSync($buffer)

Scans `$buffer` until it reaches either a *full flush point* or the end of the buffer.

If a *full flush point* is found, `Z_OK` is returned and `$buffer` will be have all data up to the flush point removed. This can then be passed to the `deflate` method.

Any other return code means that a flush point was not found. If more data is available, `inflateSync` can be called repeatedly with more compressed data until the flush point is found.

$i->dict_adler()

Returns the adler32 value for the dictionary.

$i->msg()

Returns the last error message generated by zlib.

$i->total_in()

Returns the total number of bytes compressed bytes input to inflate.

$i->total_out()

Returns the total number of uncompressed bytes output from inflate.

Example

Here is an example of using `inflate`.

```
use strict ;
use warnings ;
```

(continues on next page)

(continued from previous page)

```
use Compress::Zlib ;

my $x = inflateInit()
   or die "Cannot create a inflation stream\n" ;

my $input = '' ;
binmode STDIN;
binmode STDOUT;

my ($output, $status) ;
while (read(STDIN, $input, 4096))
{
    ($output, $status) = $x->inflate(\$input) ;

    print $output
        if $status == Z_OK or $status == Z_STREAM_END ;

    last if $status != Z_OK ;
}

die "inflation failed\n"
    unless $status == Z_STREAM_END ;
```

CHECKSUM FUNCTIONS

Two functions are provided by *zlib* to calculate checksums. For the Perl interface, the order of the two parameters in both functions has been reversed. This allows both running checksums and one off calculations to be done.

```
$crc = adler32($buffer [,$crc]) ;
$crc = crc32($buffer [,$crc]) ;
```

The buffer parameters can either be a scalar or a scalar reference.

If the $crc parameters is undef, the crc value will be reset.

If you have built this module with zlib 1.2.3 or better, two more CRC-related functions are available.

```
$crc = crc32_combine($crc1, $crc2, $len2);
$adler = adler32_combine($adler1, $adler2, $len2);
```

These functions allow checksums to be merged. Refer to the *zlib* documentation for more details.

Misc

my $version = Compress::Zlib::zlib_version();

Returns the version of the zlib library.

CONSTANTS

All the *zlib* constants are automatically imported when you make use of *Compress::Zlib*.

SUPPORT

General feedback/questions/bug reports should be sent to https://github.com/pmqs/IO-Compress/issues (preferred) or https://rt.cpan.org/Public/Dist/Display.html?Name=IO-Compress.

SEE ALSO

IO::Compress::Gzip, *IO::Uncompress::Gunzip*, *IO::Compress::Deflate*, *IO::Uncompress::Inflate*, *IO::Compress::RawDeflate*, *IO::Uncompress::RawInflate*, *IO::Compress::Bzip2*, *IO::Uncompress::Bunzip2*, IO::Compress::Lzma, IO::Uncompress::UnLzma, IO::Compress::Xz, IO::Uncompress::UnXz, IO::Compress::Lzip, IO::Uncompress::UnLzip, IO::Compress::Lzop, IO::Uncompress::UnLzop, IO::Compress::Lzf, IO::Uncompress::UnLzf, IO::Compress::Zstd, IO::Uncompress::UnZstd, *IO::Uncompress::AnyInflate*, *IO::Uncompress::AnyUncompress*

IO::Compress::FAQ

File::GlobMapper, Archive::Zip, *Archive::Tar*, *IO::Zlib*

For RFC 1950, 1951 and 1952 see https://datatracker.ietf.org/doc/html/rfc1950, https://datatracker.ietf.org/doc/html/rfc1951 and https://datatracker.ietf.org/doc/html/rfc1952

The *zlib* compression library was written by Jean-loup Gailly `gzip@prep.ai.mit.edu` and Mark Adler `madler@alumni.caltech.edu`.

The primary site for the *zlib* compression library is http://www.zlib.org.

The primary site for the *zlib-ng* compression library is https://github.com/zlib-ng/zlib-ng.

The primary site for gzip is http://www.gzip.org.

AUTHOR

This module was written by Paul Marquess, `pmqs@cpan.org`.

MODIFICATION HISTORY

See the Changes file.

COPYRIGHT AND LICENSE

Copyright (c) 1995-2026 Paul Marquess. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

5.2.18 Config

Look up the build-time configuration of the running Perl interpreter.

`Config` exposes the values recorded when Perl itself was compiled - platform, compiler flags, install paths, feature toggles. Scripts consult it to make portability decisions, locate installed libraries, or report interpreter characteristics.

Two surfaces are provided:

- The tied hash `%Config`, accessed element-wise as `$Config{key}`. Lookup, existence, and iteration go through the tie hooks. The hash is read-only; `STORE`, `DELETE`, and `CLEAR` croak.
- Programmatic queries: `config_re`, `config_sh`, `config_vars`, `myconfig`, `compile_date`, `bincompat_options`, `non_bincompat_options`, `local_patches`, `header_files`, and the `perl -V` backend `_V`.

Under pperl, every value reflects pperl's own build, not any upstream perl's. Callers using `%Config` entries to drive portability decisions (compiler name, archname, install prefix, feature defines) get answers that match the interpreter they are actually running.

```
use Config;

printf "%s, IVs of %d bytes, threads %s\n",
    $Config{archname}, $Config{ivsize}, $Config{useithreads} // 'no
 ↪';
# x86_64-linux, IVs of 8 bytes, threads no
print "can fork\n" if $Config{d_fork};
eval { $Config{cc} = 'gcc' };        # dies: %Config::Config is␣
 ↪read-only
print Config::myconfig();            # the summary perl -V prints
```

`use Config` exports `%Config`; `myconfig`, `config_sh`, `config_vars`, `config_re` and the other functions are imported by name. `$Config::VERSION` is the interpreter's version, as `$]` is.

Differences from upstream

The values describe pperl, as upstream's describe the perl they were built with: `version` is 5.44.99, `cc` is `rustc`, and `myconfig` starts "Summary of pperl X.Y.Z (perl 5.44 compatible) configuration" with pperl's own version. The keys that describe a C toolchain (`cppstdin`, `d_attribute_format`, `libsfound`, ...) are empty or undef: pperl's build has none. The keys, the functions and the read-only tied hash are upstream's. Three functions are pperl's own: `Config::lib_file` says under which path the interpreter sees a library file, the files inside the binary included, `Config::lib_files` lists the files inside the binary, and `Config::jit_counts` says how much the JIT has compiled. Every key perl 5.44's `%Config` has is here; `build.rs` derives the bulk of them at build time (`src/build_configure.rs`).

Modules

- `Config::Extensions` - hash lookup of which core extensions were built.
- `Config::Perl`

Functions

Querying

`myconfig`

Return a human-readable summary of the interpreter's build.

`config_sh`

Return the full configuration as one shell-assignment block.

`config_vars`

Print selected configuration entries to `STDOUT`.

`config_re`

Grep configuration entries whose key matches a pattern.

Introspection

`lib_file`

Say under which path this interpreter sees a library file.

`lib_files`

`jit_counts`

`compile_date`

Return a one-line description of when and with what the interpreter was built.

`local_patches`

Return the list of local patches applied to the interpreter.

`bincompat_options`

Return the compile-time options that affect binary compatibility.

non_bincompat_options

Return compile-time options that do not affect binary compatibility.

header_files

Return the canonical list of Perl C header filenames.

Config::Extensions

hash lookup of which core extensions were built.

SYNOPSIS

```
use Config::Extensions '%Extensions';
if ($Extensions{PerlIO::via}) {
    # This perl has PerlIO::via built
}
```

DESCRIPTION

The Config::Extensions module provides a hash `%Extensions` containing all the core extensions that were enabled for this perl. The hash is keyed by extension name, with each entry having one of 3 possible values:

- dynamic

  The extension is dynamically linked

- nonxs

  The extension is pure perl, so doesn't need linking to the perl executable

- static

  The extension is statically linked to the perl binary

As all values evaluate to true, a simple `if` test is good enough to determine whether an extension is present.

All the data uses to generate the `%Extensions` hash is already present in the `Config` module, but not in such a convenient format to quickly reference.

AUTHOR

Nicholas Clark nick@ccl4.org

Config::Perl

Namespace containing the following modules:

- `Config::Perl::V` - Structured data retrieval of perl -V output

Config::Perl::V

Structured data retrieval of perl -V output

SYNOPSIS

```
use Config::Perl::V;

my $local_config = Config::Perl::V::myconfig ();
print $local_config->{config}{osname};
```

DESCRIPTION

$conf = myconfig ()

This function will collect the data described in "The hash structure" below, and return that as a hash reference. It optionally accepts an option to include more entries from %ENV. See "environment" below.

Note that this will not work on uninstalled perls when called with `-I/path/to/uninstalled/perl/lib`, but it works when that path is in `$PERL5LIB` or in `$PERL5OPT`, as paths passed using `-I` are not known when the `-V` information is collected.

$conf = plv2hash ($text [, ...])

Convert a sole 'perl -V' text block, or list of lines, to a complete myconfig hash. All unknown entries are defaulted.

$info = summary ([$conf])

Return an arbitrary selection of the information. If no `$conf` is given, `myconfig ()` is used instead.

$md5 = signature ([$conf])

Return the MD5 of the info returned by `summary ()` without the `config_args` entry.

If `Digest::MD5` is not available, it return a string with only `0`'s.

The hash structure

The returned hash consists of 4 parts:

- build

  This information is extracted from the second block that is emitted by `perl -V`, and usually looks something like

  ```
  Characteristics of this binary (from libperl):
    Compile-time options: DEBUGGING USE_64_BIT_INT USE_LARGE_FILES
    Locally applied patches:
  ```

(continues on next page)

(continued from previous page)

```
        defined-or
        MAINT24637
  Built under linux
  Compiled at Jun 13 2005 10:44:20
  @INC:
    /usr/lib/perl5/5.8.7/i686-linux-64int
    /usr/lib/perl5/5.8.7
    /usr/lib/perl5/site_perl/5.8.7/i686-linux-64int
    /usr/lib/perl5/site_perl/5.8.7
    /usr/lib/perl5/site_perl
    .
```

or

```
Characteristics of this binary (from libperl):
  Compile-time options: DEBUGGING MULTIPLICITY
                        PERL_DONT_CREATE_GVSV PERL_IMPLICIT_
↪CONTEXT
                        PERL_MALLOC_WRAP PERL_TRACK_MEMPOOL
                        PERL_USE_SAFE_PUTENV USE_ITHREADS
                        USE_LARGE_FILES USE_PERLIO
                        USE_REENTRANT_API
  Built under linux
  Compiled at Jan 28 2009 15:26:59
```

This information is not available anywhere else, including `%Config`, but it is the information that is only known to the perl binary.

The extracted information is stored in 5 entries in the `build` hash:

- osname

  This is most likely the same as `$Config{osname}`, and was the name known when perl was built. It might be different if perl was cross-compiled.

  The default for this field, if it cannot be extracted, is to copy `$Config{osname}`. The two may be differing in casing (OpenBSD vs openbsd).

- stamp

  This is the time string for which the perl binary was compiled. The default value is 0.

- options

  This is a hash with all the known defines as keys. The value is either 0, which means unknown or unset, or 1, which means defined.

- derived

  As some variables are reported by a different name in the output of `perl -V` than their actual name in `%Config`, I decided to leave the `config` entry as close to reality as possible, and put in the entries that might have been guessed by the printed output in a separate block.

- patches

This is a list of optionally locally applied patches. Default is an empty list.

- environment

  By default this hash is only filled with the environment variables out of %ENV that start with PERL, but you can pass the env option to myconfig to get more

```
my $conf = Config::Perl::V::myconfig ({ env => qr/^ORACLE/ });
my $conf = Config::Perl::V::myconfig ([ env => qr/^ORACLE/ ]);
```

- config

  This hash is filled with the variables that `perl -V` fills its report with, and it has the same variables that `Config::myconfig` returns from `%Config`.

- inc

  This is the list of default @INC.

REASONING

This module was written to be able to return the configuration for the currently used perl as deeply as needed for the CPANTESTERS framework. Up until now they used the output of myconfig as a single text blob, and so it was missing the vital binary characteristics of the running perl and the optional applied patches.

BUGS

Please feedback what is wrong

TODO

```
* Implement retrieval functions/methods
* Documentation
* Error checking
* Tests
```

AUTHOR

H.Merijn Brand h.m.brand@xs4all.nl

COPYRIGHT AND LICENSE

Copyright (C) 2009-2026 H.Merijn Brand

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

myconfig

Return a human-readable summary of the interpreter's build.

Synopsis

```
use Config;
print Config::myconfig();
```

What you get back

A single multi-line string, suitable for printing verbatim. Groups the values under `Platform`, `Compiler`, and `Linker and Libraries` headings, matching the shape expected by tools that scrape `perl -V`-style output.

Examples

```
use Config;
my $summary = Config::myconfig();
print $summary if $summary =~ /osname=linux/;
```

Edge cases

- Returns a fresh scalar on every call; cache it if used repeatedly in a hot path.

Differences from upstream

- Platform, compiler, and library values describe pperl's build (Rust toolchain, pperl version line) rather than a C-compiled perl. The layout and key names match upstream so existing parsers keep working.

config_sh

Return the full configuration as one shell-assignment block.

Synopsis

```
use Config;
my $sh = Config::config_sh();
```

What you get back

A single scalar containing one line per entry in the form `key='value'`, newline-terminated. The output is stable across calls and suitable for feeding into a shell `eval` or a line-based parser.

Examples

```
use Config;
for my $line (split /^/, Config::config_sh()) {
    print $line if $line =~ /^archname=/;
}
```

Edge cases

- Entries whose value is undef render as `key=''`, matching how the tied hash reports them via `EXISTS` (true) and `FETCH` (undef).

Differences from upstream

Fully compatible with upstream.

config_vars

Print selected configuration entries to `STDOUT`.

Synopsis

```
use Config;
Config::config_vars(qw(archname osname ivsize));
```

What you get back

Nothing. Each named key is written to `STDOUT` on its own line as `key='value';`. Unknown keys render as `key='UNKNOWN';`, keys whose recorded value is undef as `key='undef';`. This is the backend of the `-V:key` command-line switch (see perlrun), including its query syntax: a leading `:` suppresses the `key=` tag, a trailing `:` ends the line with a space instead of `;\n`, and a query containing non-word characters is treated as a regex and answered from `config_re`.

Examples

```
use Config;
Config::config_vars('archname');           # archname='x86_64-linux
 ↪';
Config::config_vars('archname', 'osname'); # two lines
Config::config_vars('ivsize|nvsize');      # regex: both entries
Config::config_vars(':archname');          # 'x86_64-linux'; (no␣
 ↪tag)
```

Edge cases

- Unknown keys print `'UNKNOWN'`; undef values print `'undef'`.
- Regex queries with no match print `pattern:  not found`.
- Output goes straight to the OS `STDOUT`; redirecting with `local *STDOUT` in Perl does not affect it.

Differences from upstream

- Regex queries share `config_re`'s pattern approximation (see `config_re` "Differences from upstream").

config_re

Grep configuration entries whose key matches a pattern.

Synopsis

```
use Config;
my @hits = Config::config_re('^cc');
```

What you get back

A list of strings in `key='value'` form, one per matching entry. The pattern is anchored at the start of each `key='value'` line and must be followed by `=`, so in practice it matches the key; values are included verbatim in the returned strings.

Examples

```
use Config;
my @arch = Config::config_re('arch\w*');  # archname, archlibexp, ..
 ↪.
my @d    = Config::config_re('d_\w+');    # every d_* feature flag
```

Edge cases

- An empty pattern returns no results.
- A pattern that does not compile returns no results and leaves its error in `$@`.

Differences from upstream

Fully compatible with upstream.

lib_file

Say under which path this interpreter sees a library file.

Synopsis

```
use Config;
my $path = Config::lib_file("strict.pm");
my $path = Config::lib_file("ExtUtils/typemap");
my $path = Config::lib_file("Foo/Bar.pm", \@dirs);
```

What you get back

The path of the file a `require` of that name would load, or `undef` when there is none. The name is a path below a library directory, with `/` between the parts, as in `%INC`. The path can be opened.

A file that is part of the pperl binary is found first and reported under `$Config{privlibexp}`, the directory that holds the binary's library: nothing elsewhere can take its place. Any other file is looked for in each directory of `@INC` in turn, and the first one that exists and is not a directory is reported.

With a second argument, a reference to an array of directories, those are searched in place of `@INC`. The pperl binary's own files count only when `$Config{privlibexp}` is among them.

A built-in module written in Rust has no Perl source. Opened under the path reported here it reads as three lines: its package, its `$VERSION` and a true value.

Why it exists

Perl's own way of asking is to walk `@INC` with `-e` or `-r` and take the first hit. That walk works on pperl too, but the first file test of the binary's library writes it to disk, and a walk cannot see that a built-in module wins over a copy in an earlier directory. A tool that must know where a module is, or which version would be loaded, asks here, and nothing is written.

Examples

```
use Config;
use ExtUtils::MakeMaker ();
my $file = Config::lib_file("List/Util.pm");
print MM->parse_version($file), "\n" if defined $file;
```

lib_files

List the library files that are built into this pperl.

Synopsis

```
use Config;
my @names = Config::lib_files();
```

What you get back

The names of all built-in library files, sorted, in the form `require` takes: `strict.pm`, `File/Temp.pm`, `ExtUtils/typemap`. Each is a name `Config::lib_file` answers with a path under `$Config{privlibexp}`. Files installed elsewhere are not listed: walk `@INC` for those.

Examples

```
my @modules = map { s{/}{::}gr =~ s/\.pm\z//r } grep { /\.pm\z/ }␣
 ↪Config::lib_files();
print scalar(@modules), " modules are built in\n";
```

Differences from upstream

pperl's own; perl has no such function.

jit_counts

Say how much code the JIT has compiled so far.

Synopsis

```
use Config;
my ($compiled, $seen) = Config::jit_counts();
```

What you get back

Two numbers, both counted since the program started: the loops the JIT has compiled to machine code, and the loops it has seen and counts the trips of. Neither goes down when code is freed again. An empty list when the JIT is off or this pperl was built without it.

Examples

```
my ($before) = Config::jit_counts();
my $sum = 0;
$sum += $_ * $_ for 1 .. 1_000_000;
my ($after) = Config::jit_counts();
print "the loop was compiled\n" if defined $after && $after >
 ↪$before;
```

Differences from upstream

pperl's own; perl has no such function.

compile_date

Return a one-line description of when and with what the interpreter was built.

Synopsis

```
use Config;
print Config::compile_date(), "\n";
```

What you get back

A string such as `Compiled at Oct 8 2026 20:41:20`: when this pperl binary was built, in the layout perl uses, which `Config::Perl::V` and similar tools parse. `pperl -V` prints the same line. The compiler that built it is `$Config{ccversion}` and friends.

local_patches

Return the list of local patches applied to the interpreter.

Synopsis

```
use Config;
my @patches = Config::local_patches();
```

What you get back

A list of strings, one per local patch. The list is empty when the interpreter was built from pristine sources.

Examples

```
use Config;
print "vanilla build\n" unless Config::local_patches();
```

Differences from upstream

- Always returns an empty list under pperl. pperl is not a patched fork of upstream perl, so there is no patch trail to report.

bincompat_options

Return the compile-time options that affect binary compatibility.

Synopsis

```
use Config;
my @opts = Config::bincompat_options();
```

What you get back

A list of option names (`USE_64_BIT_INT`, `USE_PERLIO`, and so on) representing decisions baked into the interpreter that determine its binary interface. Two builds that disagree on any of them are not binary compatible.

Examples

```
use Config;
print "64-bit ints\n"
    if grep { $_ eq 'USE_64_BIT_INT' } Config::bincompat_options();
```

Differences from upstream

- The list reports the options that truthfully apply to pperl's build. Upstream-specific switches that do not exist in pperl are omitted rather than forged.

non_bincompat_options

Return compile-time options that do not affect binary compatibility.

Synopsis

```
use Config;
my @opts = Config::non_bincompat_options();
```

What you get back

A list of option names (`PERL_COPY_ON_WRITE`, `USE_LOCALE`, and so on) describing behavioural switches that do not change the ABI. Useful for feature-probing scripts that want to know whether a particular semantics is enabled without caring about binary compatibility.

Differences from upstream

- Reports only the switches that meaningfully apply to pperl's build.

header_files

Return the canonical list of Perl C header filenames.

Synopsis

```
use Config;
my @hdrs = Config::header_files();
```

What you get back

A fixed list of bare header filenames (`perl.h`, `sv.h`, `op.h`, and so on) - no paths, no extensions other than `.h`. The call exists so tooling that probes for the interpreter's C build prerequisites keeps working.

Edge cases

- The filenames name headers from the upstream perl C sources. pperl does not ship them as separate files; callers that read the headers from disk will fail to find them.

Differences from upstream

- pperl does not have a C-level public header layout, so the list is informational only. It matches upstream's roster so scripts that merely inspect the names (without opening the files) behave identically.

5.2.19 Coro

the only real threads in perl

> **Note**
>
> This is upstream Coro's documentation. Under pperl, Coro's compiled half is native and `Coro.pm` and every `Coro::*` module run unchanged; see *Coro::State* for what that means.

SYNOPSIS

```
use Coro;

async {
   # some asynchronous thread of execution
   print "2\n";
   cede; # yield back to main
   print "4\n";
};
print "1\n";
cede; # yield to coro
```

(continues on next page)

(continued from previous page)

```
print "3\n";
cede; # and again

# use locking
my $lock = new Coro::Semaphore;
my $locked;

$lock->down;
$locked = 1;
$lock->up;
```

DESCRIPTION

For a tutorial-style introduction, please read the Coro::Intro manpage. This manpage mainly contains reference information.

This module collection manages continuations in general, most often in the form of cooperative threads (also called coros, or simply "coro" in the documentation). They are similar to kernel threads but don't (in general) run in parallel at the same time even on SMP machines. The specific flavor of thread offered by this module also guarantees you that it will not switch between threads unless necessary, at easily-identified points in your program, so locking and parallel access are rarely an issue, making thread programming much safer and easier than using other thread models.

Unlike the so-called "Perl threads" (which are not actually real threads but only the windows process emulation (see section of same name for more details) ported to UNIX, and as such act as processes), Coro provides a full shared address space, which makes communication between threads very easy. And coro threads are fast, too: disabling the Windows process emulation code in your perl and using Coro can easily result in a two to four times speed increase for your programs. A parallel matrix multiplication benchmark (very communication-intensive) runs over 300 times faster on a single core than perls pseudo-threads on a quad core using all four cores.

Coro achieves that by supporting multiple running interpreters that share data, which is especially useful to code pseudo-parallel processes and for event-based programming, such as multiple HTTP-GET requests running concurrently. See *Coro::AnyEvent* to learn more on how to integrate Coro into an event-based environment.

In this module, a thread is defined as "callchain + lexical variables + some package variables + C stack), that is, a thread has its own callchain, its own set of lexicals and its own set of perls most important global variables (see *Coro::State* for more configuration and background info).

See also the SEE ALSO section at the end of this document - the Coro module family is quite large.

CORO THREAD LIFE CYCLE

During the long and exciting (or not) life of a coro thread, it goes through a number of states:

1. Creation

The first thing in the life of a coro thread is its creation - obviously. The typical way to create a thread is to call the `async BLOCK` function:

```
async {
   # thread code goes here
};
```

You can also pass arguments, which are put in `@_`:

```
async {
   print $_[1]; # prints 2
} 1, 2, 3;
```

This creates a new coro thread and puts it into the ready queue, meaning it will run as soon as the CPU is free for it.

`async` will return a Coro object - you can store this for future reference or ignore it - a thread that is running, ready to run or waiting for some event is alive on its own.

Another way to create a thread is to call the `new` constructor with a code-reference:

```
new Coro sub {
   # thread code goes here
}, @optional_arguments;
```

This is quite similar to calling `async`, but the important difference is that the new thread is not put into the ready queue, so the thread will not run until somebody puts it there. `async` is, therefore, identical to this sequence:

```
my $coro = new Coro sub {
   # thread code goes here
};
$coro->ready;
return $coro;
```

2. Startup

   When a new coro thread is created, only a copy of the code reference and the arguments are stored, no extra memory for stacks and so on is allocated, keeping the coro thread in a low-memory state.

   Only when it actually starts executing will all the resources be finally allocated.

   The optional arguments specified at coro creation are available in `@_`, similar to function calls.

3. Running / Blocking

   A lot can happen after the coro thread has started running. Quite usually, it will not run to the end in one go (because you could use a function instead), but it will give up the CPU regularly because it waits for external events.

   As long as a coro thread runs, its Coro object is available in the global variable `$Coro::current`.

The low-level way to give up the CPU is to call the scheduler, which selects a new coro thread to run:

```
Coro::schedule;
```

Since running threads are not in the ready queue, calling the scheduler without doing anything else will block the coro thread forever - you need to arrange either for the coro to put woken up (readied) by some other event or some other thread, or you can put it into the ready queue before scheduling:

```
# this is exactly what Coro::cede does
$Coro::current->ready;
Coro::schedule;
```

All the higher-level synchronisation methods (Coro::Semaphore, Coro::rouse_*...) are actually implemented via `->ready` and `Coro::schedule`.

While the coro thread is running it also might get assigned a C-level thread, or the C-level thread might be unassigned from it, as the Coro runtime wishes. A C-level thread needs to be assigned when your perl thread calls into some C-level function and that function in turn calls perl and perl then wants to switch coroutines. This happens most often when you run an event loop and block in the callback, or when perl itself calls some function such as `AUTOLOAD` or methods via the `tie` mechanism.

4. Termination

   Many threads actually terminate after some time. There are a number of ways to terminate a coro thread, the simplest is returning from the top-level code reference:

```
async {
   # after returning from here, the coro thread is terminated
};

async {
   return if 0.5 <  rand; # terminate a little earlier, maybe
   print "got a chance to print this\n";
   # or here
};
```

   Any values returned from the coroutine can be recovered using `->join`:

```
my $coro = async {
   "hello, world\n" # return a string
};

my $hello_world = $coro->join;

print $hello_world;
```

   Another way to terminate is to call `Coro::terminate`, which at any subroutine call nesting level:

```
async {
   Coro::terminate "return value 1", "return value 2";
};
```

Yet another way is to `->cancel` (or `->safe_cancel`) the coro thread from another thread:

```
my $coro = async {
   exit 1;
};

$coro->cancel; # also accepts values for ->join to retrieve
```

Cancellation *can* be dangerous - it's a bit like calling `exit` without actually exiting, and might leave C libraries and native modules in a weird state. Unlike other thread implementations, however, Coro is exceptionally safe with regards to cancellation, as perl will always be in a consistent state, and for those cases where you want to do truly marvellous things with your coro while it is being cancelled - that is, make sure all cleanup code is executed from the thread being cancelled - there is even a `->safe_cancel` method.

So, cancelling a thread that runs in a native event loop might not be the best idea, but any other combination that deals with perl only (cancelling when a thread is in a `tie` method or an `AUTOLOAD` for example) is safe.

Last not least, a coro thread object that isn't referenced is `->cancel`'ed automatically - just like other objects in Perl. This is not such a common case, however - a running thread is referencedy by `$Coro::current`, a thread ready to run is referenced by the ready queue, a thread waiting on a lock or semaphore is referenced by being in some wait list and so on. But a thread that isn't in any of those queues gets cancelled:

```
async {
   schedule; # cede to other coros, don't go into the ready␣
↪queue
};

cede;
# now the async above is destroyed, as it is not referenced by␣
↪anything.
```

A slightly embellished example might make it clearer:

```
async {
   my $guard = Guard::guard { print "destroyed\n" };
   schedule while 1;
};

cede;
```

Superficially one might not expect any output - since the `async` implements an endless loop, the `$guard` will not be cleaned up. However, since the thread object returned by `async` is not stored anywhere, the thread is initially referenced

because it is in the ready queue, when it runs it is referenced by `$Coro::current`, but when it calls `schedule`, it gets `canceled` causing the guard object to be destroyed (see the next section), and printing its message.

If this seems a bit drastic, remember that this only happens when nothing references the thread anymore, which means there is no way to further execute it, ever. The only options at this point are leaking the thread, or cleaning it up, which brings us to...

5. Cleanup

   Threads will allocate various resources. Most but not all will be returned when a thread terminates, during clean-up.

   Cleanup is quite similar to throwing an uncaught exception: perl will work its way up through all subroutine calls and blocks. On its way, it will release all `my` variables, undo all `local`'s and free any other resources truly local to the thread.

   So, a common way to free resources is to keep them referenced only by my variables:

```
async {
   my $big_cache = new Cache ...;
};
```

   If there are no other references, then the `$big_cache` object will be freed when the thread terminates, regardless of how it does so.

   What it does `NOT` do is unlock any Coro::Semaphores or similar resources, but that's where the `guard` methods come in handy:

```
my $sem = new Coro::Semaphore;

async {
   my $lock_guard = $sem->guard;
   # if we return, or die or get cancelled, here,
   # then the semaphore will be "up"ed.
};
```

   The `Guard::guard` function comes in handy for any custom cleanup you might want to do (but you cannot switch to other coroutines from those code blocks):

```
async {
   my $window = new Gtk2::Window "toplevel";
   # The window will not be cleaned up automatically, even when
↪$window
   # gets freed, so use a guard to ensure its destruction
   # in case of an error:
   my $window_guard = Guard::guard { $window->destroy };

   # we are safe here
};
```

   Last not least, `local` can often be handy, too, e.g. when temporarily replacing the coro thread description:

```
sub myfunction {
   local $Coro::current->{desc} = "inside myfunction(@_)";

   # if we return or die here, the description will be restored
}
```

6. Viva La Zombie Muerte

   Even after a thread has terminated and cleaned up its resources, the Coro object still is there and stores the return values of the thread.

   When there are no other references, it will simply be cleaned up and freed.

   If there areany references, the Coro object will stay around, and you can call `->join` as many times as you wish to retrieve the result values:

```
async {
   print "hi\n";
   1
};

# run the async above, and free everything before returning
# from Coro::cede:
Coro::cede;

{
   my $coro = async {
      print "hi\n";
      1
   };

   # run the async above, and clean up, but do not free the coro
   # object:
   Coro::cede;

   # optionally retrieve the result values
   my @results = $coro->join;

   # now $coro goes out of scope, and presumably gets freed
};
```

GLOBAL VARIABLES

- $Coro::main

  This variable stores the Coro object that represents the main program. While you can `ready` it and do most other things you can do to coro, it is mainly useful to compare again `$Coro::current`, to see whether you are running in the main program or not.

- $Coro::current

  The Coro object representing the current coro (the last coro that the Coro scheduler

switched to). The initial value is `$Coro::main` (of course).

This variable is **strictly** *read-only*. You can take copies of the value stored in it and use it as any other Coro object, but you must not otherwise modify the variable itself.

- $Coro::idle

  This variable is mainly useful to integrate Coro into event loops. It is usually better to rely on *Coro::AnyEvent* or *Coro::EV*, as this is pretty low-level functionality.

  This variable stores a Coro object that is put into the ready queue when there are no other ready threads (without invoking any ready hooks).

  The default implementation dies with "FATAL: deadlock detected.", followed by a thread listing, because the program has no other way to continue.

  This hook is overwritten by modules such as `Coro::EV` and `Coro::AnyEvent` to wait on an external event that hopefully wakes up a coro so the scheduler can run it.

  See *Coro::EV* or *Coro::AnyEvent* for examples of using this technique.

SIMPLE CORO CREATION

- async { ... } [@args...]

  Create a new coro and return its Coro object (usually unused). The coro will be put into the ready queue, so it will start running automatically on the next scheduler run.

  The first argument is a codeblock/closure that should be executed in the coro. When it returns argument returns the coro is automatically terminated.

  The remaining arguments are passed as arguments to the closure.

  See the `Coro::State::new` constructor for info about the coro environment in which coro are executed.

  Calling `exit` in a coro will do the same as calling exit outside the coro. Likewise, when the coro dies, the program will exit, just as it would in the main program.

  If you do not want that, you can provide a default `die` handler, or simply avoid dieing (by use of `eval`).

  Example: Create a new coro that just prints its arguments.

```
async {
   print "@_\n";
} 1,2,3,4;
```

- async_pool { ... } [@args...]

  Similar to `async`, but uses a coro pool, so you should not call terminate or join on it (although you are allowed to), and you get a coro that might have executed other code already (which can be good or bad :).

  On the plus side, this function is about twice as fast as creating (and destroying) a completely new coro, so if you need a lot of generic coros in quick successsion, use `async_pool`, not `async`.

The code block is executed in an `eval` context and a warning will be issued in case of an exception instead of terminating the program, as `async` does. As the coro is being reused, stuff like `on_destroy` will not work in the expected way, unless you call terminate or cancel, which somehow defeats the purpose of pooling (but is fine in the exceptional case).

The priority will be reset to `0` after each run, all `swap_sv` calls will be undone, tracing will be disabled, the description will be reset and the default output filehandle gets restored, so you can change all these. Otherwise the coro will be re-used "as-is": most notably if you change other per-coro global stuff such as `$/` you *must needs* revert that change, which is most simply done by using local as in: `local $/`.

The idle pool size is limited to `8` idle coros (this can be adjusted by changing $Coro::POOL_SIZE), but there can be as many non-idle coros as required.

If you are concerned about pooled coros growing a lot because a single `async_pool` used a lot of stackspace you can e.g. `async_pool { terminate }` once per second or so to slowly replenish the pool. In addition to that, when the stacks used by a handler grows larger than 32kb (adjustable via $Coro::POOL_RSS) it will also be destroyed.

STATIC METHODS

Static methods are actually functions that implicitly operate on the current coro.

- schedule

  Calls the scheduler. The scheduler will find the next coro that is to be run from the ready queue and switches to it. The next coro to be run is simply the one with the highest priority that is longest in its ready queue. If there is no coro ready, it will call the `$Coro::idle` hook.

  Please note that the current coro will *not* be put into the ready queue, so calling this function usually means you will never be called again unless something else (e.g. an event handler) calls `->ready`, thus waking you up.

  This makes `schedule` *the* generic method to use to block the current coro and wait for events: first you remember the current coro in a variable, then arrange for some callback of yours to call `->ready` on that once some event happens, and last you call `schedule` to put yourself to sleep. Note that a lot of things can wake your coro up, so you need to check whether the event indeed happened, e.g. by storing the status in a variable.

  See **HOW TO WAIT FOR A CALLBACK**, below, for some ways to wait for callbacks.

- cede

  "Cede" to other coros. This function puts the current coro into the ready queue and calls `schedule`, which has the effect of giving up the current "timeslice" to other coros of the same or higher priority. Once your coro gets its turn again it will automatically be resumed.

  This function is often called `yield` in other languages.

- Coro::cede_notself

Works like cede, but is not exported by default and will cede to *any* coro, regardless of priority. This is useful sometimes to ensure progress is made.

- terminate [arg...]

  Terminates the current coro with the given status values (see cancel). The values will not be copied, but referenced directly.

- Coro::on_enter BLOCK, Coro::on_leave BLOCK

  These function install enter and leave winders in the current scope. The enter block will be executed when on_enter is called and whenever the current coro is re-entered by the scheduler, while the leave block is executed whenever the current coro is blocked by the scheduler, and also when the containing scope is exited (by whatever means, be it exit, die, last etc.).

  Neither invoking the scheduler, nor exceptions, are allowed within those BLOCKs. That means: do not even think about calling `die` without an eval, and do not even think of entering the scheduler in any way.

  Since both BLOCKs are tied to the current scope, they will automatically be removed when the current scope exits.

  These functions implement the same concept as `dynamic-wind` in scheme does, and are useful when you want to localise some resource to a specific coro.

  They slow down thread switching considerably for coros that use them (about 40% for a BLOCK with a single assignment, so thread switching is still reasonably fast if the handlers are fast).

  These functions are best understood by an example: The following function will change the current timezone to "Antarctica/South_Pole", which requires a call to `tzset`, but by using `on_enter` and `on_leave`, which remember/change the current timezone and restore the previous value, respectively, the timezone is only changed for the coro that installed those handlers.

```
use POSIX qw(tzset);

async {
   my $old_tz; # store outside TZ value here

   Coro::on_enter {
      $old_tz = $ENV{TZ}; # remember the old value

      $ENV{TZ} = "Antarctica/South_Pole";
      tzset; # enable new value
   };

   Coro::on_leave {
      $ENV{TZ} = $old_tz;
      tzset; # restore old value
   };

   # at this place, the timezone is Antarctica/South_Pole,
```

(continues on next page)

(continued from previous page)

```
   # without disturbing the TZ of any other coro.
};
```

This can be used to localise about any resource (locale, uid, current working directory etc.) to a block, despite the existence of other coros.

Another interesting example implements time-sliced multitasking using interval timers (this could obviously be optimised, but does the job):

```
# "timeslice" the given block
sub timeslice(&) {
   use Time::HiRes ();

   Coro::on_enter {
      # on entering the thread, we set an VTALRM handler to cede
      $SIG{VTALRM} = sub { cede };
      # and then start the interval timer
      Time::HiRes::setitimer &Time::HiRes::ITIMER_VIRTUAL, 0.01,
↪ 0.01;
   };
   Coro::on_leave {
      # on leaving the thread, we stop the interval timer again
      Time::HiRes::setitimer &Time::HiRes::ITIMER_VIRTUAL, 0, 0;
   };

   &{+shift};
}

# use like this:
timeslice {
   # The following is an endless loop that would normally
   # monopolise the process. Since it runs in a timesliced
   # environment, it will regularly cede to other threads.
   while () { }
};
```

- killall

  Kills/terminates/cancels all coros except the currently running one.

  Note that while this will try to free some of the main interpreter resources if the calling coro isn't the main coro, but one cannot free all of them, so if a coro that is not the main coro calls this function, there will be some one-time resource leak.

CORO OBJECT METHODS

These are the methods you can call on coro objects (or to create them).

- new Coro \&sub [, @args...]

  Create a new coro and return it. When the sub returns, the coro automatically terminates as if `terminate` with the returned values were called. To make the coro run you must first put it into the ready queue by calling the ready method.

See `async` and `Coro::State::new` for additional info about the coro environment.

- $success = $coro->ready

  Put the given coro into the end of its ready queue (there is one queue for each priority) and return true. If the coro is already in the ready queue, do nothing and return false.

  This ensures that the scheduler will resume this coro automatically once all the coro of higher priority and all coro of the same priority that were put into the ready queue earlier have been resumed.

- $coro->suspend

  Suspends the specified coro. A suspended coro works just like any other coro, except that the scheduler will not select a suspended coro for execution.

  Suspending a coro can be useful when you want to keep the coro from running, but you don't want to destroy it, or when you want to temporarily freeze a coro (e.g. for debugging) to resume it later.

  A scenario for the former would be to suspend all (other) coros after a fork and keep them alive, so their destructors aren't called, but new coros can be created.

- $coro->resume

  If the specified coro was suspended, it will be resumed. Note that when the coro was in the ready queue when it was suspended, it might have been unreadied by the scheduler, so an activation might have been lost.

  To avoid this, it is best to put a suspended coro into the ready queue unconditionally, as every synchronisation mechanism must protect itself against spurious wakeups, and the one in the Coro family certainly do that.

- $state->is_new

  Returns true iff this Coro object is "new", i.e. has never been run yet. Those states basically consist of only the code reference to call and the arguments, but consumes very little other resources. New states will automatically get assigned a perl interpreter when they are transferred to.

- $state->is_zombie

  Returns true iff the Coro object has been cancelled, i.e. its resources freed because they were `cancel`'ed, `terminate`'d, `safe_cancel`'ed or simply went out of scope.

  The name "zombie" stems from UNIX culture, where a process that has exited and only stores and exit status and no other resources is called a "zombie".

- $is_ready = $coro->is_ready

  Returns true iff the Coro object is in the ready queue. Unless the Coro object gets destroyed, it will eventually be scheduled by the scheduler.

- $is_running = $coro->is_running

  Returns true iff the Coro object is currently running. Only one Coro object can ever be in the running state (but it currently is possible to have multiple running Coro::States).

- $is_suspended = $coro->is_suspended

  Returns true iff this Coro object has been suspended. Suspended Coros will not ever be scheduled.

- $coro->cancel ($arg...)

  Terminate the given Coro thread and make it return the given arguments as status (default: an empty list). Never returns if the Coro is the current Coro.

  This is a rather brutal way to free a coro, with some limitations - if the thread is inside a C callback that doesn't expect to be canceled, bad things can happen, or if the cancelled thread insists on running complicated cleanup handlers that rely on its thread context, things will not work.

  Any cleanup code being run (e.g. from `guard` blocks, destructors and so on) will be run without a thread context, and is not allowed to switch to other threads. A common mistake is to call `->cancel` from a destructor called by die'ing inside the thread to be cancelled for example.

  On the plus side, `->cancel` will always clean up the thread, no matter what. If your cleanup code is complex or you want to avoid cancelling a C-thread that doesn't know how to clean up itself, it can be better to `->throw` an exception, or use `->safe_cancel`.

  The arguments to `->cancel` are not copied, but instead will be referenced directly (e.g. if you pass `$var` and after the call change that variable, then you might change the return values passed to e.g. `join`, so don't do that).

  The resources of the Coro are usually freed (or destructed) before this call returns, but this can be delayed for an indefinite amount of time, as in some cases the manager thread has to run first to actually destruct the Coro object.

- $coro->safe_cancel ($arg...)

  Works mostly like `->cancel`, but is inherently "safer", and consequently, can fail with an exception in cases the thread is not in a cancellable state. Essentially, `->safe_cancel` is a `->cancel` with extra checks before canceling.

  It works a bit like throwing an exception that cannot be caught - specifically, it will clean up the thread from within itself, so all cleanup handlers (e.g. `guard` blocks) are run with full thread context and can block if they wish. The downside is that there is no guarantee that the thread can be cancelled when you call this method, and therefore, it might fail. It is also considerably slower than `cancel` or `terminate`.

  A thread is in a safe-cancellable state if it either has never been run yet, has already been canceled/terminated or otherwise destroyed, or has no C context attached and is inside an SLF function.

  The first two states are trivial - a thread that hasnot started or has already finished is safe to cancel.

  The last state basically means that the thread isn't currently inside a perl callback called from native code (usually a native module such as EV) and isn't currently executing inside native code itself.

  This call returns true when it could cancel the thread, or croaks with an error otherwise (i.e. it either returns true or doesn't return at all).

Why the weird interface? Well, there are two common models on how and when to cancel things. In the first, you have the expectation that your coro thread can be cancelled when you want to cancel it - if the thread isn't cancellable, this would be a bug somewhere, so `->safe_cancel` croaks to notify of the bug.

In the second model you sometimes want to ask nicely to cancel a thread, but if it's not a good time, well, then don't cancel. This can be done relatively easy like this:

```
if (! eval { $coro->safe_cancel }) {
   warn "unable to cancel thread: $@";
}
```

However, what you never should do is first try to cancel "safely" and if that fails, cancel the "hard" way with `->cancel`. That makes no sense: either you rely on being able to execute cleanup code in your thread context, or you don't. If you do, then `->safe_cancel` is the only way, and if you don't, then `->cancel` is always faster and more direct.

- $coro->schedule_to

  Puts the current coro to sleep (like `Coro::schedule`), but instead of continuing with the next coro from the ready queue, always switch to the given coro object (regardless of priority etc.). The readyness state of that coro isn't changed.

  This is an advanced method for special cases - I'd love to hear about any uses for this one.

- $coro->cede_to

  Like `schedule_to`, but puts the current coro into the ready queue. This has the effect of temporarily switching to the given coro, and continuing some time later.

  This is an advanced method for special cases - I'd love to hear about any uses for this one.

- $coro->throw ([$scalar])

  If `$throw` is specified and defined, it will be thrown as an exception inside the coro at the next convenient point in time. Otherwise clears the exception object.

  Coro will check for the exception each time a schedule-like-function returns, i.e. after each `schedule`, `cede`, `Coro::Semaphore->down`, `Coro::Handle->readable` and so on. Most of those functions (all that are part of Coro itself) detect this case and return early in case an exception is pending.

  The exception object will be thrown "as is" with the specified scalar in `$@`, i.e. if it is a string, no line number or newline will be appended (unlike with `die`).

  This can be used as a softer means than either `cancel` or `safe_cancel` to ask a coro to end itself, although there is no guarantee that the exception will lead to termination, and if the exception isn't caught it might well end the whole program.

  You might also think of `throw` as being the moral equivalent of `killing` a coro with a signal (in this case, a scalar).

- $coro->join

  Wait until the coro terminates and return any values given to the `terminate` or `cancel` functions. `join` can be called concurrently from multiple threads, and all

will be resumed and given the status return once the `$coro` terminates.

- $coro->on_destroy (\&cb)

  Registers a callback that is called when this coro thread gets destroyed, that is, after its resources have been freed but before it is joined. The callback gets passed the terminate/cancel arguments, if any, and *must not* die, under any circumstances.

  There can be any number of `on_destroy` callbacks per coro, and there is currently no way to remove a callback once added.

- $oldprio = $coro->prio ($newprio)

  Sets (or gets, if the argument is missing) the priority of the coro thread. Higher priority coro get run before lower priority coros. Priorities are small signed integers (currently -4 .. +3), that you can refer to using PRIO_xxx constants (use the import tag :prio to get then):

```
PRIO_MAX > PRIO_HIGH > PRIO_NORMAL > PRIO_LOW > PRIO_IDLE > PRIO_MIN
   3    >     1     >      0      >    -1    >    -3     >    -4

# set priority to HIGH
current->prio (PRIO_HIGH);
```

  The idle coro thread ($Coro::idle) always has a lower priority than any existing coro.

  Changing the priority of the current coro will take effect immediately, but changing the priority of a coro in the ready queue (but not running) will only take effect after the next schedule (of that coro). This is a bug that will be fixed in some future version.

- $newprio = $coro->nice ($change)

  Similar to `prio`, but subtract the given value from the priority (i.e. higher values mean lower priority, just as in UNIX's nice command).

- $olddesc = $coro->desc ($newdesc)

  Sets (or gets in case the argument is missing) the description for this coro thread. This is just a free-form string you can associate with a coro.

  This method simply sets the `$coro->{desc}` member to the given string. You can modify this member directly if you wish, and in fact, this is often preferred to indicate major processing states that can then be seen for example in a *Coro::Debug* session:

```
sub my_long_function {
   local $Coro::current->{desc} = "now in my_long_function";
   ...
   $Coro::current->{desc} = "my_long_function: phase 1";
   ...
   $Coro::current->{desc} = "my_long_function: phase 2";
   ...
}
```

GLOBAL FUNCTIONS

- Coro::nready

  Returns the number of coro that are currently in the ready state, i.e. that can be switched to by calling `schedule` directory or indirectly. The value `0` means that the only runnable coro is the currently running one, so `cede` would have no effect, and `schedule` would cause a deadlock unless there is an idle handler that wakes up some coro.

- my $guard = Coro::guard { ... }

  This function still exists, but is deprecated. Please use the `Guard::guard` function instead.

- unblock_sub { ... }

  This utility function takes a BLOCK or code reference and "unblocks" it, returning a new coderef. Unblocking means that calling the new coderef will return immediately without blocking, returning nothing, while the original code ref will be called (with parameters) from within another coro.

  The reason this function exists is that many event libraries (such as the venerable *Event* module) are not thread-safe (a weaker form of reentrancy). This means you must not block within event callbacks, otherwise you might suffer from crashes or worse. The only event library currently known that is safe to use without `unblock_sub` is *EV* (but you might still run into deadlocks if all event loops are blocked).

  Coro will try to catch you when you block in the event loop ("FATAL: $Coro::idle blocked itself"), but this is just best effort and only works when you do not run your own event loop.

  This function allows your callbacks to block by executing them in another coro where it is safe to block. One example where blocking is handy is when you use the *Coro::AIO* functions to save results to disk, for example.

  In short: simply use `unblock_sub { ... }` instead of `sub { ... }` when creating event callbacks that want to block.

  If your handler does not plan to block (e.g. simply sends a message to another coro, or puts some other coro into the ready queue), there is no reason to use `unblock_sub`.

  Note that you also need to use `unblock_sub` for any other callbacks that are indirectly executed by any C-based event loop. For example, when you use a module that uses *AnyEvent* (and you use *Coro::AnyEvent*) and it provides callbacks that are the result of some event callback, then you must not block either, or use `unblock_sub`.

- $cb = rouse_cb

  Create and return a "rouse callback". That's a code reference that, when called, will remember a copy of its arguments and notify the owner coro of the callback.

  Only the first invocation will store agruments and signal any waiter - further calls will effectively be ignored, but it is ok to try.

  Also see the next function.

- @args = rouse_wait [$cb]

  Wait for the specified rouse callback to be invoked (or if the argument is missing, use the most recently created callback in the current coro).

  As soon as the callback is invoked (or when the callback was invoked before `rouse_wait`), it will return the arguments originally passed to the rouse callback. In scalar context, that means you get the *last* argument, just as if `rouse_wait` had a `return ($a1, $a2, $a3...)` statement at the end.

  You are only allowed to wait once for a given rouse callback.

  See the section **HOW TO WAIT FOR A CALLBACK** for an actual usage example.

  As of Coro 6.57, you can reliably wait for a rouse callback in a different thread than from where it was created.

HOW TO WAIT FOR A CALLBACK

It is very common for a coro to wait for some callback to be called. This occurs naturally when you use coro in an otherwise event-based program, or when you use event-based libraries.

These typically register a callback for some event, and call that callback when the event occurred. In a coro, however, you typically want to just wait for the event, simplyifying things.

For example `AnyEvent->child` registers a callback to be called when a specific child has exited:

```
my $child_watcher = AnyEvent->child (pid => $pid, cb => sub { ... }
↪);
```

But from within a coro, you often just want to write this:

```
my $status = wait_for_child $pid;
```

Coro offers two functions specifically designed to make this easy, `rouse_cb` and `rouse_wait`.

The first function, `rouse_cb`, generates and returns a callback that, when invoked, will save its arguments and notify the coro that created the callback.

The second function, `rouse_wait`, waits for the callback to be called (by calling `schedule` to go to sleep) and returns the arguments originally passed to the callback.

Using these functions, it becomes easy to write the `wait_for_child` function mentioned above:

```
sub wait_for_child($) {
   my ($pid) = @_;

   my $watcher = AnyEvent->child (pid => $pid, cb => rouse_cb);

   my ($rpid, $rstatus) = rouse_wait;
   $rstatus
}
```

In the case where `rouse_cb` and `rouse_wait` are not flexible enough, you can roll your own, using `schedule` and `ready`:

```
sub wait_for_child($) {
   my ($pid) = @_;

   # store the current coro in $current,
   # and provide result variables for the closure passed to ->child
   my $current = $Coro::current;
   my ($done, $rstatus);

   # pass a closure to ->child
   my $watcher = AnyEvent->child (pid => $pid, cb => sub {
      $rstatus = $_[1]; # remember rstatus
      $done = 1;        # mark $rstatus as valid
      $current->ready;  # wake up the waiting thread
   });

   # wait until the closure has been called
   schedule while !$done;

   $rstatus
}
```

BUGS/LIMITATIONS

- fork with pthread backend

  When Coro is compiled using the pthread backend (which isn't recommended but required on many BSDs as their libcs are completely broken), then coro will not survive a fork. There is no known workaround except to fix your libc and use a saner backend.

- perl process emulation ("threads")

  This module is not perl-pseudo-thread-safe. You should only ever use this module from the first thread (this requirement might be removed in the future to allow per-thread schedulers, but Coro::State does not yet allow this). I recommend disabling thread support and using processes, as having the windows process emulation enabled under unix roughly halves perl performance, even when not used.

  Attempts to use threads created in another emulated process will crash ("cleanly", with a null pointer exception).

- coro switching is not signal safe

  You must not switch to another coro from within a signal handler (only relevant with %SIG - most event libraries provide safe signals), *unless* you are sure you are not interrupting a Coro function.

  That means you *MUST NOT* call any function that might "block" the current coro - `cede`, `schedule` `Coro::Semaphore->down` or anything that calls those. Everything else, including calling `ready`, works.

WINDOWS PROCESS EMULATION

A great many people seem to be confused about ithreads (for example, Chip Salzenberg called me unintelligent, incapable, stupid and gullible, while in the same mail making rather confused statements about perl ithreads (for example, that memory or files would be shared), showing his lack of understanding of this area - if it is hard to understand for Chip, it is probably not obvious to everybody).

What follows is an ultra-condensed version of my talk about threads in scripting languages given on the perl workshop 2009:

The so-called “ithreads” were originally implemented for two reasons: first, to (badly) emulate unix processes on native win32 perls, and secondly, to replace the older, real thread model (“5.005-threads”).

It does that by using threads instead of OS processes. The difference between processes and threads is that threads share memory (and other state, such as files) between threads within a single process, while processes do not share anything (at least not semantically). That means that modifications done by one thread are seen by others, while modifications by one process are not seen by other processes.

The “ithreads” work exactly like that: when creating a new ithreads process, all state is copied (memory is copied physically, files and code is copied logically). Afterwards, it isolates all modifications. On UNIX, the same behaviour can be achieved by using operating system processes, except that UNIX typically uses hardware built into the system to do this efficiently, while the windows process emulation emulates this hardware in software (rather efficiently, but of course it is still much slower than dedicated hardware).

As mentioned before, loading code, modifying code, modifying data structures and so on is only visible in the ithreads process doing the modification, not in other ithread processes within the same OS process.

This is why “ithreads” do not implement threads for perl at all, only processes. What makes it so bad is that on non-windows platforms, you can actually take advantage of custom hardware for this purpose (as evidenced by the forks module, which gives you the (i-) threads API, just much faster).

Sharing data is in the i-threads model is done by transferring data structures between threads using copying semantics, which is very slow - shared data simply does not exist. Benchmarks using i-threads which are communication-intensive show extremely bad behaviour with i-threads (in fact, so bad that Coro, which cannot take direct advantage of multiple CPUs, is often orders of magnitude faster because it shares data using real threads, refer to my talk for details).

As summary, i-threads *use* threads to implement processes, while the compatible forks module *uses* processes to emulate, uhm, processes. I-threads slow down every perl program when enabled, and outside of windows, serve no (or little) practical purpose, but disadvantages every single-threaded Perl program.

This is the reason that I try to avoid the name “ithreads”, as it is misleading as it implies that it implements some kind of thread model for perl, and prefer the name “windows process emulation”, which describes the actual use and behaviour of it much better.

SEE ALSO

Event-Loop integration: *Coro::AnyEvent*, *Coro::EV*, *Coro::Event*.

Debugging: *Coro::Debug*.

Support/Utility: *Coro::Specific*, *Coro::Util*.

Locking and IPC: *Coro::Signal*, *Coro::Channel*, *Coro::Semaphore*, *Coro::SemaphoreSet*, *Coro::RWLock*.

I/O and Timers: *Coro::Timer*, *Coro::Handle*, *Coro::Socket*, *Coro::AIO*.

Compatibility with other modules: *Coro::LWP* (but see also AnyEvent::HTTP for a better-working alternative), *Coro::BDB*, *Coro::Storable*, *Coro::Select*.

Low level Configuration, Thread Environment, Continuations: *Coro::State*.

AUTHOR/SUPPORT/CONTACT

```
Marc A. Lehmann <schmorp@schmorp.de>
http://software.schmorp.de/pkg/Coro.html
```

Modules

- `Coro::AIO` - truly asynchronous file and directory I/O
- `Coro::AnyEvent` - integrate threads into AnyEvent
- `Coro::BDB` - truly asynchronous bdb access
- `Coro::Channel` - message queues
- `Coro::Debug` - various functions that help debugging Coro programs
- `Coro::EV` - run Coro on the EV event loop
- `Coro::Event` - Event watchers that block the calling coroutine
- `Coro::Handle` - non-blocking I/O with a blocking interface.
- `Coro::LWP` - make LWP non-blocking - as much as possible
- `Coro::RWLock` - reader/write locks
- `Coro::Select` - a (slow but coro-aware) replacement for CORE::select
- `Coro::Semaphore` - counting semaphores
- `Coro::SemaphoreSet` - efficient set of counting semaphores
- `Coro::Signal` - thread signals (binary semaphores)
- `Coro::Socket` - non-blocking socket-I/O
- `Coro::Specific` - manage coroutine-specific variables.
- `Coro::State` - the engine under Coro: Perl-level threads
- `Coro::Storable` - offer a more fine-grained Storable interface
- `Coro::Timer` - timers and timeouts, independent of any event loop

- `Coro::Util` - various utility functions.

Coro::AIO

truly asynchronous file and directory I/O

SYNOPSIS

```
use Coro::AIO;

# can now use any of the aio requests your IO::AIO module supports.

# read 1MB of /etc/passwd, without blocking other coroutines
my $fh = aio_open "/etc/passwd", O_RDONLY, 0
   or die "/etc/passwd: $!";
aio_read $fh, 0, 1_000_000, my $buf, 0
   or die "aio_read: $!";
aio_close $fh;
```

DESCRIPTION

This module is an *AnyEvent* user, you need to make sure that you use and run a supported event loop.

This module implements a thin wrapper around IO::AIO. All of the functions that expect a callback are being wrapped by this module.

The API is exactly the same as that of the corresponding IO::AIO routines, except that you have to specify *all* arguments, even the ones optional in IO::AIO, *except* the callback argument. Instead of calling a callback, the routines return the values normally passed to the callback. Everything else, including `$!` and perls stat cache, are set as expected after these functions return.

You can mix calls to `IO::AIO` functions with calls to this module. You *must not*, however, call these routines from within IO::AIO callbacks, as this causes a deadlock. Start a coro inside the callback instead.

This module also loads AnyEvent::AIO to integrate into the event loop in use, so please refer to its (and *AnyEvent*'s) documentation on how it selects an appropriate event module.

All other functions exported by default by IO::AIO (e.g. `aioreq_pri`) will be exported by default by Coro::AIO, too.

Functions that can be optionally imported from IO::AIO can be imported from Coro::AIO or can be called directly, e.g. `Coro::AIO::nreqs`.

You cannot specify priorities with `aioreq_pri` if your coroutine has a non-zero priority, as this module overwrites the request priority with the current coroutine priority in that case.

For your convenience, here are the changed function signatures for most of the requests, for documentation of these functions please have a look at IO::AIO. Note that requests added by newer versions of IO::AIO will be automatically wrapped as well.

- @results = aio_wait $req

  This is not originally an IO::AIO request: what it does is to wait for `$req` to finish and return the results. This is most useful with `aio_group` requests.

  Is currently implemented by replacing the `$req` callback (and is very much like a wrapper around `$req->cb ()`).

- $fh = aio_open $pathname, $flags, $mode
- $status = aio_close $fh
- $retval = aio_read $fh,$offset,$length, $data,$dataoffset
- $retval = aio_write $fh,$offset,$length, $data,$dataoffset
- $retval = aio_sendfile $out_fh, $in_fh, $in_offset, $length
- $retval = aio_readahead $fh,$offset,$length
- $status = aio_stat $fh_or_path
- $status = aio_lstat $fh
- $status = aio_unlink $pathname
- $status = aio_rmdir $pathname
- $entries = aio_readdir $pathname
- ($dirs, $nondirs) = aio_scandir $path, $maxreq
- $status = aio_fsync $fh
- $status = aio_fdatasync $fh
- ... = aio_xxx ...

  Any additional aio requests follow the same scheme: same parameters except you must not specify a callback but instead get the callback arguments as return values.

SEE ALSO

Coro::Socket and *Coro::Handle* for non-blocking socket operation.

AUTHOR/SUPPORT/CONTACT

```
Marc A. Lehmann <schmorp@schmorp.de>
http://software.schmorp.de/pkg/Coro.html
```

Coro::AnyEvent

integrate threads into AnyEvent

SYNOPSIS

```
use Coro;
use AnyEvent;
# using both Coro and AnyEvent will automatically load
 ↪Coro::AnyEvent

# or load it manually for its utility functions:
use Coro::AnyEvent;

Coro::AnyEvent::sleep 5;     # block current thread for 5s
Coro::AnyEvent::poll;        # poll for new events once
Coro::AnyEvent::idle;        # block until process no longer busy
Coro::AnyEvent::idle_upto 5; # same, but only up to 5 seconds

Coro::AnyEvent::readable $fh, 60
   or die "fh didn't become readable within 60 seconds\n";
```

DESCRIPTION

When one naively starts to use threads in Perl, one will quickly run into the problem that threads which block on a syscall (sleeping, reading from a socket etc.) will block all threads.

If one then uses an event loop, the problem is that the event loop has no knowledge of threads and will not run them before it polls for new events, again blocking the whole process.

This module integrates threads into any event loop supported by AnyEvent, combining event-based programming with coroutine-based programming in a natural way.

As of Coro 5.21 and newer, this module gets loaded automatically when AnyEvent initialises itself and Coro is used in the same process, thus there is no need to load it manually if you just want your threads to coexist with AnyEvent.

If you want to use any functions from this module, you of course still need to `use Coro::AnyEvent`, just as with other perl modules.

Also, this module autodetects the event loop used (by relying on *AnyEvent*) and will either automatically defer to the high-performance *Coro::EV* or *Coro::Event* modules, or will use a generic integration method that should work with any event loop supported by *AnyEvent*.

USAGE

RUN AN EVENT LOOP - OR NOT?

For performance reasons, it is recommended that the main program or something else run the event loop of the event model you use, i.e.

```
use Gtk2; # <- the event model
use AnyEvent;
```

(continues on next page)

(continued from previous page)

```
use Coro:

# initialise stuff
async { ... };

# now run mainloop of Gtk2
main Gtk2;
```

You can move the event loop into a thread as well, although this tends to get confusing:

```
use Gtk2;
use AnyEvent;
use Coro:

async { main Gtk2 };

# do other things...
while () {
   use Coro::AnyEvent;
   Coro::AnyEvent::sleep 1;
   print "ping...\n";
}
```

You can also do nothing, in which case Coro::AnyEvent will invoke the event loop as needed, which is less efficient, but sometimes very convenient.

What you *MUST NOT EVER DO* is to block inside an event loop callback. The reason is that most event loops are not reentrant and this can cause a deadlock at best and corrupt memory at worst.

Coro will try to catch you when you block in the event loop ("FATAL: $Coro::IDLE blocked itself"), but this is just best effort and only works when you do not run your own event loop.

To avoid this problem, start a new thread (e.g. with `Coro::async_pool`) or use `Coro::unblock_sub` to run blocking tasks.

INVERSION OF CONTROL

If you need to wait for a single event, the rouse functions will come in handy (see the Coro manpage for details):

```
# wait for single SIGINT
{
   my $int_w = AnyEvent->signal (signal => "INT", cb => Coro::rouse_
↪cb);
   Coro::rouse_wait;
}
```

EVENT MODULES OTHER THAN ANYEVENT

Keep in mind that, as shipped, Coro and Coro::AnyEvent only work with AnyEvent, and only when AnyEvent is actually used (i.e. initialised), so this will not work:

```
# does not work: EV without AnyEvent is not recognised
use EV;
use Coro;

EV::loop;
```

And neither does this, unless you actually *use* AnyEvent for something:

```
# does not work: AnyEvent must be initialised (e.g. by creating␣
↪watchers)
use EV;
use AnyEvent;
use Coro;

EV::loop;
```

This does work, however, because you create a watcher (condvars work, too), thus forcing AnyEvent to initialise itself:

```
# does work: AnyEvent is actually used
use EV;
use AnyEvent;
use Coro;

my $timer = AE::timer 1, 1, sub { };

EV::loop;
```

And if you want to use AnyEvent just to bridge between Coro and your event model of choice, you can simply force it to initialise itself, like this:

```
# does work: AnyEvent is initialised manually
use POE;
use AnyEvent;
use Coro;

AnyEvent::detect; # force AnyEvent to integrate Coro into POE
POE::Kernel->run;
```

FUNCTIONS

Coro::AnyEvent also offers a few functions that might be useful.

- Coro::AnyEvent::poll

  This call will block the current thread until the event loop has polled for potential new events and instructs the event loop to poll for new events once, without blocking.

Note that this call will not actually execute the poll, nor will it wait until there are some events, just block until the event loop has polled for new events, so other threads will have a chance to run.

This is useful when you have a thread that does some computations, but you still want to poll for new events from time to time. Simply call `poll` from time to time:

```
my $long_calc = async {
   for (1..10000) {
      Coro::AnyEvent::poll;
      # do some stuff, make sure it takes at least 0.001s or so
   }
}
```

Although you should also consider `idle` or `idle_upto` in such cases.

- Coro::AnyEvent::sleep $seconds

  This blocks the current thread for at least the given number of seconds.

- Coro::AnyEvent::idle

  This call is similar to `poll` in that it will also poll for events. Unlike `poll`, it will only resume the thread once there are no events to handle anymore, i.e. when the process is otherwise idle.

  This is good for background threads that shouldn't use CPU time when foreground jobs are ready to run.

- Coro::AnyEvent::idle_upto $seconds

  Like `idle`, but with a maximum waiting time.

  If your process is busy handling events, calling `idle` can mean that your thread will never be resumed. To avoid this, you can use `idle_upto` and specify a timeout, after which your thread will be resumed even if the process is completely busy.

- Coro::AnyEvent::readable $fh_or_fileno[, $timeout]
- Coro::AnyEvent::writable $fh_or_fileno[, $timeout]

  Blocks the current thread until the given file handle (or file descriptor) becomes readable (or writable), or the given timeout has elapsed, whichever happens first. No timeout counts as infinite timeout.

  Returns true when the file handle became ready, false when a timeout occurred.

  Note that these functions are quite inefficient as compared to using a single watcher (they recreate watchers on every invocation) or compared to using Coro::Handle.

  Note also that they only work for sources that have reasonable non-blocking behaviour (e.g. not files).

  Example: wait until STDIN becomes readable, then quit the program.

```
use Coro::AnyEvent;
print "press enter to quit...\n";
Coro::AnyEvent::readable *STDIN;
exit 0;
```

IMPLEMENTATION DETAILS

Unfortunately, few event loops (basically only *EV* and *Event*) support the kind of integration required for smooth operations well, and consequently, AnyEvent cannot completely offer the functionality required by this module, so we need to improvise.

Here is what this module does when it has to work with other event loops:

- run ready threads before blocking the process

  Each time a thread is put into the ready queue (and there are no other threads in the ready queue), a timer with an `after` value of `0` is registered with AnyEvent.

  This creates something similar to an *idle* watcher, i.e. a watcher that keeps the event loop from blocking but still polls for new events. (Unfortunately, some badly designed event loops (e.g. Event::Lib) don't support a timeout of `0` and will always block for a bit).

  The callback for that timer will `cede` to other threads of the same or higher priority for as long as such threads exists. This has the effect of running all threads that have work to do until all threads block to wait for external events.

  If no threads of equal or higher priority are ready, it will cede to any thread, but only once. This has the effect of running lower-priority threads as well, but it will not keep higher priority threads from receiving new events.

  The priority used is simply the priority of the thread that runs the event loop, usually the main program, which usually has a priority of `0`. Note that Coro::AnyEvent does *not* run an event loop for you, so unless the main program runs one, there will simply be no event loop to `cede` to (event handling will still work, somewhat inefficiently, but any thread will have a higher priority than event handling in that case).

- provide a suitable idle callback.

  In addition to hooking into `ready`, this module will also provide a `$Coro::idle` handler that runs the event loop. It is best not to take advantage of this too often, as this is rather inefficient, but it should work perfectly fine.

- provide overrides for AnyEvent's condvars

  This module installs overrides for AnyEvent's condvars. That is, when the module is loaded it will provide its own condition variables. This makes them coroutine-safe, i.e. you can safely block on them from within a coroutine.

- lead to data corruption or worse

  As `unblock_sub` cannot be used by this module (as it is the module that implements it, basically), you must not call into the event loop recursively from any coroutine. This is not usually a difficult restriction to live with, just use condvars, `unblock_sub` or other means of inter-coroutine-communications.

  If you use a module that supports AnyEvent (or uses the same event loop as AnyEvent, making it implicitly compatible), and it offers callbacks of any kind, then you must not block in them, either (or use e.g. `unblock_sub`), see the description of `unblock_sub` in the *Coro* module.

  This also means that you should load the module as early as possible, as only condvars created after this module has been loaded will work correctly.

SEE ALSO

AnyEvent, to see which event loops are supported, *Coro::EV* and *Coro::Event* for more efficient and more correct solutions (they will be used automatically if applicable).

AUTHOR/SUPPORT/CONTACT

```
Marc A. Lehmann <schmorp@schmorp.de>
http://software.schmorp.de/pkg/Coro.html
```

Coro::BDB

truly asynchronous bdb access

SYNOPSIS

```
use Coro::BDB;
use BDB;

# can now use any of the bdb requests
```

DESCRIPTION

This module is an *AnyEvent* user, you need to make sure that you use and run a supported event loop.

This module implements a thin wrapper around the BDB module: Each BDB request that could block and doesn't get passed a callback will normally block all coroutines. after loading this module, this will no longer be the case (it provides a suitable sync prepare callback).

It will also register an AnyEvent watcher as soon as AnyEvent chooses an event loop.

The AnyEvent watcher can be disabled by executing `undef $Coro::BDB::WATCHER`. Please notify the author of when and why you think this was necessary.

This module does not export anything (unlike *Coro::AIO*), as BDB already supports leaving out the callback. (Unfortunately, it ties a C context to each coroutine executing such a callback, so in the future, it might export more efficient wrappers).

SEE ALSO

BDB of course.

AUTHOR/SUPPORT/CONTACT

```
Marc A. Lehmann <schmorp@schmorp.de>
http://software.schmorp.de/pkg/Coro.html
```

Coro::Channel

message queues

SYNOPSIS

```
use Coro;

$q1 = new Coro::Channel <maxsize>;

$q1->put ("xxx");
print $q1->get;

die unless $q1->size;
```

DESCRIPTION

A Coro::Channel is the equivalent of a unix pipe (and similar to amiga message ports): you can put things into it on one end and read things out of it from the other end. If the capacity of the Channel is maxed out writers will block. Both ends of a Channel can be read/written from by as many coroutines as you want concurrently.

You don't have to load `Coro::Channel` manually, it will be loaded automatically when you `use Coro` and call the `new` constructor.

- $q = new Coro:Channel $maxsize

  Create a new channel with the given maximum size (practically unlimited if `maxsize` is omitted or zero). Giving a size of one gives you a traditional channel, i.e. a queue that can store only a single element (which means there will be no buffering, and `put` will wait until there is a corresponding `get` call). To buffer one element you have to specify 2, and so on.

- $q->put ($scalar)

  Put the given scalar into the queue.

- $q->get

  Return the next element from the queue, waiting if necessary.

- $q->shutdown

  Shuts down the Channel by pushing a virtual end marker onto it: This changes the behaviour of the Channel when it becomes or is empty to return `undef`, almost as if infinitely many `undef` elements had been put into the queue.

  Specifically, this function wakes up any pending `get` calls and lets them return `undef`, the same on future `get` calls. `size` will return the real number of stored elements, though.

  Another way to describe the behaviour is that `get` calls will not block when the queue becomes empty but immediately return `undef`. This means that calls to `put` will work normally and the data will be returned on subsequent `get` calls.

This method is useful to signal the end of data to any consumers, quite similar to an end of stream on e.g. a tcp socket: You have one or more producers that `put` data into the Channel and one or more consumers who `get` them. When all producers have finished producing data, a call to `shutdown` signals this fact to any consumers.

A common implementation uses one or more threads that `get` from a channel until it returns `undef`. To clean everything up, first `shutdown` the channel, then `join` the threads.

- $q->size

  Return the number of elements waiting to be consumed. Please note that:

```
if ($q->size) {
   my $data = $q->get;
   ...
}
```

  is *not* a race condition but instead works just fine. Note that the number of elements that wait can be larger than `$maxsize`, as it includes any coroutines waiting to put data into the channel (but not any shutdown condition).

  This means that the number returned is *precisely* the number of calls to `get` that will succeed instantly and return some data. Calling `shutdown` has no effect on this number.

AUTHOR/SUPPORT/CONTACT

```
Marc A. Lehmann <schmorp@schmorp.de>
http://software.schmorp.de/pkg/Coro.html
```

Coro::Debug

various functions that help debugging Coro programs

SYNOPSIS

```
use Coro::Debug;

our $server = new_unix_server Coro::Debug "/tmp/socketpath";

$ socat readline unix:/tmp/socketpath
```

DESCRIPTION

This module is an *AnyEvent* user, you need to make sure that you use and run a supported event loop.

This module provides some debugging facilities. Most will, if not handled carefully, severely compromise the security of your program, so use it only for debugging (or take other precautions).

It mainly implements a very primitive debugger that is very easy to integrate in your program:

```
our $server = new_unix_server Coro::Debug "/tmp/somepath";
# see new_unix_server, below, for more info
```

It lets you list running coroutines:

```
         state (rUnning, Ready, New or neither)
         |cctx allocated
         ||  resident set size (octets)
         ||  |   scheduled this many times
> ps     ||  |   |
     PID SC  RSS USES Description              Where
14572344 UC  62k 128k [main::]                 [dm-support.ext:47]
14620056 -- 2260   13 [coro manager]           [Coro.pm:358]
14620128 -- 2260  166 [unblock_sub scheduler]  [Coro.pm:358]
17764008 N-  152    0 [EV idle process]        -
13990784 -- 2596  10k timeslot manager         [cf.pm:454]
81424176 --  18k 4758 [async pool idle]         [Coro.pm:257]
23513336 -- 2624    1 follow handler           [follow.ext:52]
40548312 --  15k 5597 player scheduler          [player-scheduler.
 ↪ext:13]
29138032 -- 2548  431 music scheduler          [player-env.ext:77]
43449808 -- 2260 3493 worldmap updater         [item-worldmap.
 ↪ext:115]
33352488 --  19k 2845 [async pool idle]         [Coro.pm:257]
81530072 --  13k  43k map scheduler            [map-scheduler.
 ↪ext:65]
30751144 --  15k 2204 [async pool idle]         [Coro.pm:257]
```

Lets you do backtraces on about any coroutine:

```
> bt 18334288
coroutine is at /opt/cf/ext/player-env.ext line 77
        eval {...} called at /opt/cf/ext/player-env.ext line 77
        ext::player_env::__ANON__ called at -e line 0
        Coro::_run_coro called at -e line 0
```

Or lets you eval perl code:

```
> 5+7
12
```

Or lets you eval perl code within other coroutines:

```
> eval 18334288 caller(1); $DB::args[0]->method
1
```

It can also trace subroutine entry/exits for most coroutines (those not having recursed into a C function), resulting in output similar to:

```
> loglevel 5
> trace 94652688
2007-09-27Z20:30:25.1368 (5) [94652688] enter Socket::sockaddr_in␣
 ↪with (8481,\x{7f}\x{00}\x{00}\x{01})
2007-09-27Z20:30:25.1369 (5) [94652688] leave Socket::sockaddr_in␣
 ↪returning (\x{02}\x{00}...)
2007-09-27Z20:30:25.1370 (5) [94652688] enter Net::FCP::Util::touc␣
 ↪with (client_get)
2007-09-27Z20:30:25.1371 (5) [94652688] leave Net::FCP::Util::touc␣
 ↪returning (ClientGet)
2007-09-27Z20:30:25.1372 (5) [94652688] enter␣
 ↪AnyEvent::Impl::Event::io with (AnyEvent,fh,GLOB(0x9256250),poll,
 ↪w,cb,CODE(0x8c963a0))
2007-09-27Z20:30:25.1373 (5) [94652688] enter Event::Watcher::__
 ↪ANON__ with (Event,poll,w,fd,GLOB(0x9256250),cb,CODE(0x8c963a0))
2007-09-27Z20:30:25.1374 (5) [94652688] enter Event::io::new with␣
 ↪(Event::io,poll,w,fd,GLOB(0x9256250),cb,CODE(0x8c963a0))
2007-09-27Z20:30:25.1375 (5) [94652688] enter Event::Watcher::init␣
 ↪with (Event::io=HASH(0x8bfb120),HASH(0x9b7940))
```

If your program uses the Coro::Debug::log facility:

```
Coro::Debug::log 0, "important message";
Coro::Debug::log 9, "unimportant message";
```

Then you can even receive log messages in any debugging session:

```
> loglevel 5
2007-09-26Z02:22:46 (9) unimportant message
```

Other commands are available in the shell, use the `help` command for a list.

FUNCTIONS

None of the functions are being exported.

- log $level, $msg

  Log a debug message of the given severity level (0 is highest, higher is less important) to all interested parties.

- stderr_loglevel $level

  Set the loglevel for logging to stderr (defaults to the value of the environment variable PERL_CORO_STDERR_LOGLEVEL, or -1 if missing).

- session_loglevel $level

  Set the default loglevel for new coro debug sessions (defaults to the value of the environment variable PERL_CORO_DEFAULT_LOGLEVEL, or -1 if missing).

- trace $coro, $loglevel

  Enables tracing the given coroutine at the given loglevel. If loglevel is omitted, use 5. If coro is omitted, trace the current coroutine. Tracing incurs a very high

runtime overhead.

It is not uncommon to enable tracing on oneself by simply calling `Coro::Debug::trace`.

A message will be logged at the given loglevel if it is not possible to enable tracing.

- untrace $coro

  Disables tracing on the given coroutine.

- command $string

  Execute a debugger command, sending any output to STDOUT. Used by `session`, below.

- session $fh

  Run an interactive debugger session on the given filehandle. Each line entered is simply passed to `command` (with a few exceptions).

- $server = new_unix_server Coro::Debug $path

  Creates a new unix domain socket that listens for connection requests and runs `session` on any connection. Normal unix permission checks and umask applies, so you can protect your socket by puttint it into a protected directory.

  The `socat` utility is an excellent way to connect to this socket:

```
socat readline /path/to/socket
```

  Socat also offers history support:

```
socat readline:history=/tmp/hist.corodebug /path/to/socket
```

  The server accepts connections until it is destroyed, so you must keep the return value around as long as you want the server to stay available.

- $server = new_tcp_server Coro::Debug $port

  Similar to `new_unix_server`, but binds on a TCP port. *Note that this is usually results in a gaping security hole.*

  Currently, only a TCPv4 socket is created, in the future, a TCPv6 socket might also be created.

AUTHOR/SUPPORT/CONTACT

```
Marc A. Lehmann <schmorp@schmorp.de>
http://software.schmorp.de/pkg/Coro.html
```

Coro::EV

`Coro::EV` - run Coro on the EV event loop.

```
use Coro;
use Coro::EV;

async { Coro::EV::timer_once 0.5; print "half a second later\n" };
EV::run;                        # or let Coro::AnyEvent drive it
```

Loading `Coro::EV` makes EV's loop and Coro's scheduler cooperate: before EV blocks for events it runs every ready coroutine, and when a coroutine becomes ready while EV sleeps, EV wakes up for it. That is what lets code in the linear Coro style wait on sockets and timers through EV - directly, or through `Coro::AnyEvent`, `Coro::Handle` and `Coro::Socket`, which use it when EV is the event loop.

`Coro::EV::timer_once $seconds` suspends the current coroutine for that long; `Coro::EV::timed_io_once $fh, $events, $timeout` waits for a filehandle to become readable or writable (`EV::READ`, `EV::WRITE`) and returns the events that occurred, or 0 on timeout.

This is a native implementation of the XS half of Coro 6.57's `Coro::EV`, on the native *EV* module; its `Coro/EV.pm` is used unchanged.

Differences from upstream

None.

Coro::Event

`Coro::Event` - Event watchers that block the calling coroutine instead of running a callback.

```
use Coro;
use Coro::Event;

async {
    my $w = Coro::Event->timer(interval => 1);
    while () {
        $w->next;                # this coroutine sleeps, others run
        print "tick\n";
    }
};

Event::loop;
```

Every Event watcher type has a Coro::Event constructor of the same name (`Coro::Event->io(...)`, `->timer(...)`, `->signal(...)`, ...) that takes Event's arguments without a callback. `$w->next` suspends the current coroutine until the watcher's next event and returns it; `Coro::Event::do_io(...)` and friends wait for a single event. While Event's loop waits it lets every ready coroutine run first, so programs written in the linear Coro style and callback code driven by Event share one loop. Run `Event::loop` in the main program or in a coroutine of its own.

This is a native implementation of the XS half of Coro 6.57's `Coro::Event`, on the native *Event* module; its `Coro/Event.pm` is used unchanged.

Differences from upstream

None.

Coro::Handle

non-blocking I/O with a blocking interface.

SYNOPSIS

```
use Coro::Handle;
```

DESCRIPTION

This module is an *AnyEvent* user, you need to make sure that you use and run a supported event loop.

This module implements IO-handles in a coroutine-compatible way, that is, other coroutines can run while reads or writes block on the handle.

It does so by using *AnyEvent* to wait for readable/writable data, allowing other coroutines to run while one coroutine waits for I/O.

Coro::Handle does NOT inherit from IO::Handle but uses tied objects.

If at all possible, you should *always* prefer method calls on the handle object over invoking tied methods, i.e.:

```
$fh->print ($str);          # NOT print $fh $str;
my $line = $fh->readline;   # NOT my $line = <$fh>;
```

The reason is that perl recurses within the interpreter when invoking tie magic, forcing the (temporary) allocation of a (big) stack. If you have lots of socket connections and they happen to wait in e.g. <$fh>, then they would all have a costly C coroutine associated with them.

- $fh = new_from_fh Coro::Handle $fhandle [, arg => value...]

  Create a new non-blocking io-handle using the given perl-filehandle. Returns `undef` if no filehandle is given. The only other supported argument is "timeout", which sets a timeout for each operation.

- $fh = unblock $fh

  This is a convenience function that just calls `new_from_fh` on the given filehandle. Use it to replace a normal perl filehandle by a non-(coroutine-)blocking equivalent.

- $fh->writable, $fh->readable

  Wait until the filehandle is readable or writable (and return true) or until an error condition happens (and return false).

- $fh->readline ([$terminator])

  Similar to the builtin of the same name, but allows you to specify the input record separator in a coroutine-safe manner (i.e. not using a global variable). Paragraph mode is not supported, use "\n\n" to achieve the same effect.

- $fh->autoflush ([...])

  Always returns true, arguments are being ignored (exists for compatibility only). Might change in the future.

- $fh->fileno, $fh->close, $fh->read, $fh->sysread, $fh->syswrite, $fh->print, $fh->printf

  Work like their function equivalents (except read, which works like sysread. You should not use the read function with Coro::Handle's, it will work but it's not efficient).

- connect, listen, bind, getsockopt, setsockopt, send, recv, peername, sockname, shutdown, peerport, peerhost

  Do the same thing as the perl builtins or IO::Socket methods (but return true on EINPROGRESS). Remember that these must be method calls.

- peeraddr, peerhost, peerport

  Return the peer host (as numericla IP address) and peer port (as integer).

- ($fh, $peername) = $listen_fh->accept

  In scalar context, returns the newly accepted socket (or undef) and in list context return the ($fh, $peername) pair (or nothing).

- $fh->timeout ([...])

  The optional argument sets the new timeout (in seconds) for this handle. Returns the current (new) value.

  `0` is a valid timeout, use `undef` to disable the timeout.

- $fh->fh

  Returns the "real" (non-blocking) filehandle. Use this if you want to do operations on the file handle you cannot do using the Coro::Handle interface.

- $fh->rbuf

  Returns the current contents of the read buffer (this is an lvalue, so you can change the read buffer if you like).

  You can use this function to implement your own optimized reader when neither readline nor sysread are viable candidates, like this:

```
# first get the _real_ non-blocking filehandle
# and fetch a reference to the read buffer
my $nb_fh = $fh->fh;
my $buf = \$fh->rbuf;

while () {
   # now use buffer contents, modifying
   # if necessary to reflect the removed data

   last if $$buf ne ""; # we have leftover data

   # read another buffer full of data
```

(continues on next page)

(continued from previous page)

```
   $fh->readable or die "end of file";
   sysread $nb_fh, $$buf, 8192;
}
```

BUGS

```
- Perl's IO-Handle model is THE bug.
```

AUTHOR/SUPPORT/CONTACT

```
Marc A. Lehmann <schmorp@schmorp.de>
http://software.schmorp.de/pkg/Coro.html
```

Coro::LWP

make LWP non-blocking - as much as possible

SYNOPSIS

```
use Coro::LWP; # afterwards LWP should not block
```

ALTERNATIVES

Over the years, a number of less-invasive alternatives have popped up, which you might find more acceptable than this rather invasive and fragile module. All of them only support HTTP (and sometimes HTTPS).

- AnyEvent::HTTP

  Works fine without Coro. Requires using a very different API than LWP. Probably the best choice *iff* you can do with a completely different event-based API.

- LWP::Protocol::AnyEvent::http

  Makes LWP use AnyEvent::HTTP. Does not make LWP event-based, but allows Coro threads to schedule unimpeded through its AnyEvent integration.

  Lets you use the LWP API normally.

- LWP::Protocol::Coro::http

  Basically the same as above, distinction unclear. :)

- AnyEvent::HTTP::LWP::UserAgent

  A different user agent implementation, not completely transparent to users, requires Coro.

DESCRIPTION

This module is an *AnyEvent* user, you need to make sure that you use and run a supported event loop.

This module tries to make LWP non-blocking with respect to other coroutines as much as possible, and with whatever means it takes.

LWP really tries very hard to be blocking (and relies on a lot of undocumented functionality in IO::Socket), so this module had to be very invasive and must be loaded very early to take the proper effect.

Note that the module AnyEvent::HTTP might offer an alternative to the full LWP that is designed to be non-blocking.

Here is what it currently does (future versions of LWP might require different tricks):

- It loads Coro::Select, overwriting the perl `select` builtin *globally*.

  This is necessary because LWP calls select quite often for timeouts and who-knows-what.

  Impact: everybody else uses this (slower) version of select, too. It should be quite compatible to perls builtin select, though.

- It overwrites Socket::inet_aton with Coro::Util::inet_aton.

  This is necessary because LWP might (and does) try to resolve hostnames this way.

  Impact: some code might not expect coroutine semantics, for example, when you fork you might prefer the blocking variant because other coroutines shouldn't actually run.

- It replaces the base class of Net::HTTP, Net::FTP, Net::NNTP.

  This is necessary because LWP does not always use select to see whether a filehandle can be read/written without blocking, so the base class `IO::Socket::INET` needs to be replaced by `Coro::Socket`.

  Impact: Coro::Socket is not at all compatible to IO::Socket::INET. While it duplicates some undocumented functionality required by LWP, it does not have all the methods of IO::Socket::INET and might act quite differently in practise. Also, protocols other than the above mentioned will still block, at least some of the time.

All this likely makes other libraries than just LWP not block, but that's just a side effect you cannot rely on.

Increases parallelism is not supported by all libraries, some might cache data globally.

AUTHOR/SUPPORT/CONTACT

```
Marc A. Lehmann <schmorp@schmorp.de>
http://software.schmorp.de/pkg/Coro.html
```

Coro::RWLock

reader/write locks

SYNOPSIS

```
use Coro;

$lck = new Coro::RWLock;

$lck->rdlock; # acquire read lock
$lck->unlock; # unlock lock again

# or:
$lck->wrlock; # acquire write lock
$lck->unlock; # unlock lock again

# try a readlock
if ($lck->tryrdlock) {
   ...;
   $l->unlock;
}

# try a write lock
if ($lck->trywrlock) {
   ...;
   $l->unlock;
}
```

DESCRIPTION

This module implements reader/write locks. A read can be acquired for read by many coroutines in parallel as long as no writer has locked it (shared access). A single write lock can be acquired when no readers exist. RWLocks basically allow many concurrent readers (without writers) OR a single writer (but no readers).

You don't have to load `Coro::RWLock` manually, it will be loaded automatically when you `use Coro` and call the `new` constructor.

- $l = new Coro::RWLock;

  Create a new reader/writer lock.

- $l->rdlock

  Acquire a read lock.

- $l->tryrdlock

  Try to acquire a read lock.

- $l->wrlock

  Acquire a write lock.

- $l->trywrlock

  Try to acquire a write lock.

- $l->unlock

  Give up a previous `rdlock` or `wrlock`.

AUTHOR/SUPPORT/CONTACT

```
Marc A. Lehmann <schmorp@schmorp.de>
http://software.schmorp.de/pkg/Coro.html
```

Coro::Select

a (slow but coro-aware) replacement for CORE::select

SYNOPSIS

```
use Coro::Select;          # replace select globally (be careful,
 ↪see below)
use Coro::Select 'select'; # only in this module
use Coro::Select ();       # use Coro::Select::select
```

DESCRIPTION

This module tries to create a fully working replacement for perl's `select` built-in, using `AnyEvent` watchers to do the job, so other threads can run in parallel to any select user. As many libraries that only have a blocking API do not use global variables and often use select (or IO::Select), this effectively makes most such libraries "somewhat" non-blocking w.r.t. other threads.

This implementation works fastest when only very few bits are set in the fd set(s).

To be effective globally, this module must be `use`'d before any other module that uses `select`, so it should generally be the first module `use`'d in the main program. Note that overriding `select` globally might actually cause problems, as some `AnyEvent` backends use `select` themselves, and asking AnyEvent to use Coro::Select, which in turn asks AnyEvent will not quite work.

You can also invoke it from the commandline as `perl -MCoro::Select`.

To override select only for a single module (e.g. `Net::DBus::Reactor`), use a code fragment like this to load it:

```
{
   package Net::DBus::Reactor;
   use Coro::Select qw(select);
   use Net::DBus::Reactor;
}
```

Some modules (notably POE::Loop::Select) directly call `CORE::select`. For these modules, we need to patch the opcode table by sandwiching it between calls to `Coro::Select::patch_pp_sselect` and `Coro::Select::unpatch_pp_sselect`:

```
BEGIN {
   use Coro::Select ();
   Coro::Select::patch_pp_sselect;
   require evil_poe_module_using_CORE::SELECT;
   Coro::Select::unpatch_pp_sselect;
}
```

BUGS

For performance reasons, Coro::Select's select function might not properly detect bad file descriptors (but relying on EBADF is inherently non-portable).

SEE ALSO

Coro::LWP.

AUTHOR/SUPPORT/CONTACT

```
Marc A. Lehmann <schmorp@schmorp.de>
http://software.schmorp.de/pkg/Coro.html
```

Coro::Semaphore

counting semaphores

SYNOPSIS

```
use Coro;

$sig = new Coro::Semaphore [initial value];

$sig->down; # wait for signal

# ... some other "thread"

$sig->up;
```

DESCRIPTION

This module implements counting semaphores. You can initialize a mutex with any level of parallel users, that is, you can initialize a sempahore that can be downed more than once until it blocks. There is no owner associated with semaphores, so one thread can down it while another can up it (or vice versa), up can be called before down and so on: the semaphore is really just an integer counter that optionally blocks when it is 0.

Counting semaphores are typically used to coordinate access to resources, with the semaphore count initialized to the number of free resources. Threads then increment the count when resources are added and decrement the count when resources are removed.

You don't have to load `Coro::Semaphore` manually, it will be loaded automatically when you `use Coro` and call the `new` constructor.

- new [initial count]

  Creates a new sempahore object with the given initial lock count. The default lock count is 1, which means it is unlocked by default. Zero (or negative values) are also allowed, in which case the semaphore is locked by default.

- $sem->count

  Returns the current semaphore count. The semaphore can be down'ed without blocking when the count is strictly higher than `0`.

- $sem->adjust ($diff)

  Atomically adds the amount given to the current semaphore count. If the count becomes positive, wakes up any waiters. Does not block if the count becomes negative, however.

- $sem->down

  Decrement the counter, therefore "locking" the semaphore. This method waits until the semaphore is available if the counter is zero or less.

- $sem->wait

  Similar to `down`, but does not actually decrement the counter. Instead, when this function returns, a following call to `down` or `try` is guaranteed to succeed without blocking, until the next thread switch (`cede` etc.).

  Note that using `wait` is much less efficient than using `down`, so try to prefer `down` whenever possible.

- $sem->wait ($callback)

  If you pass a callback argument to `wait`, it will not wait, but immediately return. The callback will be called as soon as the semaphore becomes available (which might be instantly), and gets passed the semaphore as first argument.

  The callback might `down` the semaphore exactly once, might wake up other threads, but is *NOT* allowed to block (switch to other threads).

- $sem->up

  Unlock the semaphore again.

- $sem->try

  Try to `down` the semaphore. Returns true when this was possible, otherwise return false and leave the semaphore unchanged.

- $sem->waiters

  In scalar context, returns the number of threads waiting for this semaphore. Might accidentally cause WW3 if called in other contexts, so don't use these.

- $guard = $sem->guard

  This method calls down and then creates a guard object. When the guard object is destroyed it automatically calls up.

AUTHOR/SUPPORT/CONTACT

```
Marc A. Lehmann <schmorp@schmorp.de>
http://software.schmorp.de/pkg/Coro.html
```

Coro::SemaphoreSet

efficient set of counting semaphores

SYNOPSIS

```
use Coro;

$sig = new Coro::SemaphoreSet [initial value];

$sig->down ("semaphoreid"); # wait for signal

# ... some other "thread"

$sig->up ("semaphoreid");
```

DESCRIPTION

This module implements sets of counting semaphores (see *Coro::Semaphore*). It is nothing more than a hash with normal semaphores as members, but is more efficiently managed.

This is useful if you want to allow parallel tasks to run in parallel but not on the same problem. Just use a SemaphoreSet and lock on the problem identifier.

You don't have to load `Coro::SemaphoreSet` manually, it will be loaded automatically when you `use Coro` and call the new constructor.

- new [initial count]

  Creates a new semaphore set with the given initial lock count for each individual semaphore. See *Coro::Semaphore*.

- $semset->down ($id)

  Decrement the counter, therefore "locking" the named semaphore. This method waits until the semaphore is available if the counter is zero.

- $semset->up ($id)

  Unlock the semaphore again. If the semaphore reaches the default count for this set and has no waiters, the space allocated for it will be freed.

- $semset->try ($id)

  Try to down the semaphore. Returns true when this was possible, otherwise return false and leave the semaphore unchanged.

- $semset->count ($id)

  Return the current semaphore count for the specified semaphore.

- $semset->waiters ($id)

  Returns the number (in scalar context) or list (in list context) of waiters waiting on the specified semaphore.

- $semset->wait ($id)

  Same as Coro::Semaphore::wait on the specified semaphore.

- $guard = $semset->guard ($id)

  This method calls down and then creates a guard object. When the guard object is destroyed it automatically calls up.

- $semaphore = $semset->sem ($id)

  This SemaphoreSet version is based on Coro::Semaphore's. This function creates (if necessary) the underlying Coro::Semaphore object and returns it. You may legally call any Coro::Semaphore method on it, but note that calling `$semset->up` can invalidate the returned semaphore.

AUTHOR/SUPPORT/CONTACT

```
Marc A. Lehmann <schmorp@schmorp.de>
http://software.schmorp.de/pkg/Coro.html
```

Coro::Signal

thread signals (binary semaphores)

SYNOPSIS

```
use Coro;

my $sig = new Coro::Signal;

$sig->wait; # wait for signal

# ... some other "thread"

$sig->send;
```

DESCRIPTION

This module implements signals/binary semaphores/condition variables (basically all the same thing). You can wait for a signal to occur or send it, in which case it will wake up one waiter, or it can be broadcast, waking up all waiters.

It is recommended not to mix `send` and `broadcast` calls on the same `Coro::Signal` without some deep thinking: while it should work as documented, it can easily confuse you :->

You don't have to load `Coro::Signal` manually, it will be loaded automatically when you `use Coro` and call the `new` constructor.

- $sig = new Coro::Signal;

  Create a new signal.

- $sig->wait

  Wait for the signal to occur (via either `send` or `broadcast`). Returns immediately if the signal has been sent before.

- $sig->wait ($callback)

  If you pass a callback argument to `wait`, it will not wait, but immediately return. The callback will be called under the same conditions as `wait` without arguments would continue the thrad.

  The callback might wake up any number of threads, but is *NOT* allowed to block (switch to other threads).

- $sig->send

  Send the signal, waking up *one* waiting process or remember the signal if no process is waiting.

- $sig->broadcast

  Send the signal, waking up *all* waiting process. If no process is waiting the signal is lost.

- $sig->awaited

  Return true when the signal is being awaited by some process.

AUTHOR/SUPPORT/CONTACT

```
Marc A. Lehmann <schmorp@schmorp.de>
http://software.schmorp.de/pkg/Coro.html
```

Coro::Socket

non-blocking socket-I/O

SYNOPSIS

```
use Coro::Socket;

# listen on an ipv4 socket
my $socket = new Coro::Socket PeerHost => "localhost",
                              PeerPort => 'finger';

# listen on any other type of socket
my $socket = Coro::Socket->new_from_fh
                (IO::Socket::UNIX->new
                    Local  => "/tmp/socket",
                    Type   => SOCK_STREAM,
                );
```

DESCRIPTION

This module is an *AnyEvent* user, you need to make sure that you use and run a supported event loop.

This module implements socket-handles in a coroutine-compatible way, that is, other coroutines can run while reads or writes block on the handle. See *Coro::Handle*, especially the note about prefering method calls.

IPV6 WARNING

This module was written to imitate the *IO::Socket::INET* API, and derive from it. Since IO::Socket::INET does not support IPv6, this module does neither.

Therefore it is not recommended to use Coro::Socket in new code. Instead, use *AnyEvent::Socket* and *Coro::Handle*, e.g.:

```
use Coro;
use Coro::Handle;
use AnyEvent::Socket;

# use tcp_connect from AnyEvent::Socket
# and call Coro::Handle::unblock on it.

tcp_connect "www.google.com", 80, Coro::rouse_cb;
my $fh = unblock +(Coro::rouse_wait)[0];

# now we have a perfectly thread-safe socket handle in $fh
print $fh "GET / HTTP/1.0\015\012\015\012";
local $/;
print <$fh>;
```

Using `AnyEvent::Socket::tcp_connect` gives you transparent IPv6, multi-homing, SRV-record etc. support.

For listening sockets, use `AnyEvent::Socket::tcp_server`.

- $fh = new Coro::Socket param => value, ...

Create a new non-blocking tcp handle and connect to the given host and port. The parameter names and values are mostly the same as for IO::Socket::INET (as ugly as I think they are).

The parameters officially supported currently are: `ReuseAddr`, `LocalPort`, `LocalHost`, `PeerPort`, `PeerHost`, `Listen`, `Timeout`, `SO_RCVBUF`, `SO_SNDBUF`.

```
$fh = new Coro::Socket PeerHost => "localhost", PeerPort =>
↪'finger';
```

AUTHOR/SUPPORT/CONTACT

```
Marc A. Lehmann <schmorp@schmorp.de>
http://software.schmorp.de/pkg/Coro.html
```

Coro::Specific

manage coroutine-specific variables.

SYNOPSIS

```
use Coro::Specific;

my $ref = new Coro::Specific;

$$ref = 5;
print $$ref;
```

DESCRIPTION

This module can be used to create variables (or better: references to them) that are specific to the currently executing coroutine. This module does not automatically load the Coro module (so the overhead will be small when no coroutines are used).

A much faster method is to store extra keys into `%$Coro::current` - all you have to do is to make sure that the key is unique (e.g. by prefixing it with your module name). You can even store data there before loading the *Coro* module - when Coro is loaded, the keys stored in `%$Coro::current` are automatically attached to the coro thread executing the main program.

You don't have to load `Coro::Specific` manually, it will be loaded automatically when you `use Coro` and call the new constructor.

- new

  Create a new coroutine-specific scalar and return a reference to it. The scalar is guaranteed to be "undef". Once such a scalar has been allocated you cannot deallocate it (yet), so allocate only when you must.

BUGS

The actual coroutine specific values do not automatically get destroyed when the Coro::Specific object gets destroyed.

AUTHOR/SUPPORT/CONTACT

```
Marc A. Lehmann <schmorp@schmorp.de>
http://software.schmorp.de/pkg/Coro.html
```

Coro::State

`Coro::State` - the engine under Coro: Perl-level threads ("coroutines") with their own call stacks, switched cooperatively.

```
use Coro;

# Three coroutines take turns: each cede lets the next ready one␣
 ↪run.
my @workers = map {
    my $id = $_;
    async {
        for my $step (1 .. 2) {
            print "worker $id, step $step\n";
            cede;
        }
        return "worker $id done";
    };
} 1 .. 3;

print $_->join, "\n" for @workers;     # wait for each, collect its␣
 ↪result

# A semaphore makes the others wait while one holds it.
my $lock = Coro::Semaphore->new(1);
my $t = async { my $g = $lock->guard; print "got the lock\n" };
{
    my $g = $lock->guard;
    cede;                              # $t runs, and blocks on the␣
 ↪lock
    print "releasing, ", scalar($lock->waiters), " waiting\n";
}
$t->join;

# The engine underneath: switch between two states by hand.
my ($main, $other);
$other = Coro::State->new(sub { print "in other\n"; $other->
 ↪transfer($main) });
$main  = Coro::State->new;
$main->transfer($other);
```

(continues on next page)

(continued from previous page)

```
print "back in main\n";
```

The program prints `worker 1, step 1` to `worker 3, step 1`, then the three step 2 lines, the three done lines, `releasing, 1 waiting, got the lock, in other` and `back in main`.

Most programs use `Coro` and never touch `Coro::State` directly. `Coro::State` is the machinery: a coroutine for a sub, and switching from one to another. Coro's scheduler, its coroutine objects, `Coro::Semaphore` and `Coro::Signal` are built on it, and their XS half lives here too.

Each coroutine has its own `$_`, `@_`, `$@`, `$/`, `$1...`, `$^H` and `%^H`, die and warn hooks (`$Coro::State::DIEHOOK` and `WARNHOOK` are the defaults), default output handle, `local` values and lexicals. A coroutine that switches away from inside a sort block, a tie method, a `DESTROY` or any other callback keeps a C stack of its own until it gets back out; the size of those stacks is `Coro::State::cctx_stacksize`, in pointers (default 16384, which is 128 KiB).

This is a native implementation of the XS half of Coro 6.57; its `Coro.pm` and every `Coro::*` module are used unchanged. `use Coro` exports `async`, `async_pool`, `cede`, `schedule`, `terminate`, `current`, `unblock_sub`, `rouse_cb` and `rouse_wait`; the `PRIO_*` constants and `nready` on request. `Coro::State` exports nothing. `Coro::State::BACKEND` names the stack-switch method: `asm` on x86-64, `sjlj` (setjmp/longjmp) elsewhere - what Coro's own build picks on Linux.

Coroutines: the Coro functions

- `async { ... } @args` creates a coroutine running the block with `@args`, puts it in the ready queue and returns its `Coro` object. Returning from the block ends it; an uncaught `die` ends the program, as in the main program.
- `async_pool { ... } @args` is the same with a coroutine from a pool of reused ones: faster, runs the block in an `eval` (a `die` warns), and never terminates, so it is not to be joined.
- `Coro->new(\&sub, @args)` creates a coroutine without readying it.
- `cede` puts the current coroutine at the end of the ready queue and runs the next one of the same or higher priority; `Coro::cede_notself` cedes to any ready one, whatever its priority.
- `schedule` runs the next ready coroutine without readying the current one: the way to block until something calls `->ready`.
- `terminate @values` ends the current coroutine with those values.
- `Coro::nready` is the number of coroutines in the ready queue.
- `rouse_cb` returns a callback; `rouse_wait` blocks until that callback (or the one passed to it) has been called, and returns the arguments it was called with. They turn a callback API into a blocking call.
- `Coro::on_enter BLOCK` and `Coro::on_leave BLOCK` run their block whenever the current coroutine is switched to or away from, while the enclosing scope lasts.

The methods of a `Coro` object `$coro`:

- `$coro->ready` queues it; false when it already was. `$coro->suspend` keeps the scheduler from running it until `$coro->resume`.
- `$coro->join` waits for it to end and returns its values (the block's return value, or what `terminate` or `cancel` gave).
- `$coro->cancel(@values)` ends it at once, from outside; `$coro->safe_cancel(@values)` ends it from inside, so its cleanup runs in its own context, and dies when it is not in a state where that can be done. `$coro->throw($exception)` makes it die with `$exception` when it next runs.
- `$coro->on_destroy(\&cb)` calls `cb` with its values when it ends.
- `$coro->schedule_to` and `$coro->cede_to` switch to it directly, bypassing the queue; `cede_to` readies the current coroutine first.
- `$coro->prio($new)` and `$coro->nice($change)` set its priority (from `PRIO_MIN`, -4, to `PRIO_MAX`, 3; `nice` subtracts) and return the previous one.
- `$coro->is_new`, `is_ready`, `is_running`, `is_suspended`, `is_zombie` (ended) and `is_destroyed` (the same) report its state.

`Coro::Semaphore->new($count)` has down (wait until the count is positive, then take one), up, `try` (take one if available), `count`, `adjust($delta)`, `waiters` (the waiting coroutines, or their number in scalar context) and, from Coro's Perl half, guard. `Coro::Signal->new` has `wait`, `send` (wake one waiter, or the next to wait), `broadcast` (wake every waiter) and `awaited`.

Coro::State

- `Coro::State->new(\&sub, @args)` makes a state that runs `sub` when first transferred to; `Coro::State->new` with no sub makes an empty one to save the current state in. A sub that returns ends the program.
- `$prev->transfer($next)` saves the running state in `$prev` and continues `$next`.
- `$state->call(\&code)` and `$state->eval($string)` run code in the context of another state (for debuggers; the code must not switch).
- `$state->swap_defsv` and `swap_defav` exchange the current `$_` or `@_` with the state's; `$state->swap_sv(\$var, \$private)` makes `$var` hold `$private`'s contents whenever that state runs (calling it again with the same two undoes it).
- `$state->cancel` destroys the state.
- `$state->throw`, `is_new`, `is_zombie` are as for Coro objects.
- `$state->rss` is the memory its stacks use, in bytes; `usecount` the number of transfers to it.
- `Coro::State::enable_times($bool)` turns on per-coroutine time accounting and returns the previous setting; `$state->times` then returns its real and CPU seconds.
- `Coro::State::list` returns every state, Coro objects included.
- `$state->trace($flags)` traces sub calls and statements in it, as `Coro::Debug` uses; `$state->is_traced` reads the flags.

- C stacks: `$state->has_cctx` says whether it holds one; `Coro::State::force_cctx` gives the running state one; `Coro::State::cctx_count` and `cctx_idle` count those in use and cached; `cctx_max_idle($n)` and `cctx_stacksize($n)` set the cache size (default 4) and the size of new stacks, each returning the old value.
- `$state->clone` dies "Coro::State->clone has not been configured into this installation of Coro", as in a default Coro build.

The other Coro modules (`Coro::Channel`, `Coro::RWLock`, `Coro::Specific`, `Coro::Timer`, `Coro::AnyEvent`, `Coro::Handle`, `Coro::Debug`, ...) are Coro's Perl modules, unchanged.

Differences from upstream

Two, both where perl 5.44 with XS Coro crashes (t/81-xs-native/Coro/390-or-better.t):

- `Coro::State::is_ready(bless {}, 'X')` and the other methods called on a blessed hash that is not a coroutine die "Coro::State object required", as they do for every other non-coroutine. Coro reads the magic of a hash that has none and crashes.
- `binmode $fh, ':cede'` with no interval uses the 0.01-second interval Coro's layer falls back to. Coro reads the missing argument and crashes.

Coro::Storable

offer a more fine-grained Storable interface

SYNOPSIS

```
use Coro::Storable;
```

DESCRIPTION

This module implements a few functions from the Storable module in a way so that it cede's more often. Some applications (such as the Deliantra game server) sometimes need to load large Storable objects without blocking the server for a long time.

This is being implemented by using a perlio layer that feeds only small amounts of data (4096 bytes per call) into Storable, and `Coro::cede`'ing regularly (at most 100 times per second by default, though).

As Storable is not reentrant, this module also wraps most functions of the Storable module so that only one freeze or thaw is done at any one moment (and recursive invocations are not currently supported).

FUNCTIONS

- $ref = thaw $pst

Retrieve an object from the given $pst, which must have been created with `Coro::Storable::freeze` or `Storable::store_fd`/`Storable::store` (sorry, but Storable uses incompatible formats for disk/mem objects).

This function will cede regularly.

- $pst = freeze $ref

  Freeze the given scalar into a Storable object. It uses the same format as `Storable::store_fd`.

  This functino will cede regularly.

- $pst = nfreeze $ref

  Same as `freeze` but is compatible to `Storable::nstore_fd` (note the n).

- $pst = blocking_freeze $ref

  Same as `freeze` but is guaranteed to block. This is useful e.g. in `Coro::Util::fork_eval` when you want to serialise a data structure for use with the `thaw` function for this module. You cannot use `Storable::freeze` for this as Storable uses incompatible formats for memory and file images, and this module uses file images.

- $pst = blocking_nfreeze $ref

  Same as `blocking_freeze` but uses `nfreeze` internally.

- $guard = guard

  Acquire the Storable lock, for when you want to call Storable yourself.

  Note that this module already wraps all Storable functions, so there is rarely the need to do this yourself.

AUTHOR/SUPPORT/CONTACT

```
Marc A. Lehmann <schmorp@schmorp.de>
http://software.schmorp.de/pkg/Coro.html
```

Coro::Timer

timers and timeouts, independent of any event loop

SYNOPSIS

```
# This package is mostly obsoleted by Coro::AnyEvent.

use Coro::Timer qw(timeout);
# nothing exported by default
```

DESCRIPTION

This package has been mostly obsoleted by *Coro::AnyEvent*, the only really useful function left in here is `timeout`.

- $flag = timeout $seconds

  This function will wake up the current coroutine after $seconds seconds and sets $flag to true (it is false initially). If $flag goes out of scope earlier then nothing happens.

  This is used by Coro itself to implement the `timed_down`, `timed_wait` etc. primitives. It is used like this:

```
sub timed_wait {
   my $timeout = Coro::Timer::timeout 60;

   while (condition false) {
      Coro::schedule; # wait until woken up or timeout
      return 0 if $timeout; # timed out
   }

   return 1; # condition satisfied
}
```

AUTHOR/SUPPORT/CONTACT

```
Marc A. Lehmann <schmorp@schmorp.de>
http://software.schmorp.de/pkg/Coro.html
```

Coro::Util

various utility functions.

SYNOPSIS

```
use Coro::Util;
```

DESCRIPTION

This module implements various utility functions, mostly replacing perl functions by non-blocking counterparts.

Many of these functions exist for the sole purpose of emulating existing interfaces, no matter how bad or limited they are (e.g. no IPv6 support).

This module is an AnyEvent user. Refer to the *AnyEvent* documentation to see how to integrate it into your own programs.

- $ipn = Coro::Util::inet_aton $hostname || $ip

  Works almost exactly like its `Socket::inet_aton` counterpart, except that it does not block other coroutines.

Does not handle multihomed hosts or IPv6 - consider using `AnyEvent::Socket::resolve_sockaddr` with the *Coro* rouse functions instead.

- gethostbyname, gethostbyaddr

  Work similarly to their Perl counterparts, but do not block. Uses `AnyEvent::Util::inet_aton` internally.

  Does not handle multihomed hosts or IPv6 - consider using `AnyEvent::Socket::resolve_sockaddr` or `AnyEvent::DNS::reverse_lookup` with the *Coro* rouse functions instead.

- @result = Coro::Util::fork_eval { ... }, @args

  Executes the given code block or code reference with the given arguments in a separate process, returning the results. The return values must be serialisable with Coro::Storable. It may, of course, block.

  Note that using event handling in the sub is not usually a good idea as you will inherit a mixed set of watchers from the parent.

  Exceptions will be correctly forwarded to the caller.

  This function is useful for pushing cpu-intensive computations into a different process, for example to take advantage of multiple CPU's. Its also useful if you want to simply run some blocking functions (such as `system()`) and do not care about the overhead enough to code your own pid watcher etc.

  This function might keep a pool of processes in some future version, as fork can be rather slow in large processes.

  You should also look at `AnyEvent::Util::fork_eval`, which is newer and more compatible to totally broken Perl implementations such as the one from ActiveState.

  Example: execute some external program (convert image to rgba raw form) and add a long computation (extract the alpha channel) in a separate process, making sure that never more then $NUMCPUS processes are being run.

```
my $cpulock = new Coro::Semaphore $NUMCPUS;

sub do_it {
   my ($path) = @_;

   my $guard = $cpulock->guard;

   Coro::Util::fork_eval {
      open my $fh, "convert -depth 8 \Q$path\E rgba:"
         or die "$path: $!";

      local $/;
      # make my eyes hurt
      pack "C*", unpack "(xxxC)*", <$fh>
   }
}
```

(continues on next page)

(continued from previous page)

```
my $alphachannel = do_it "/tmp/img.png";
```

AUTHOR/SUPPORT/CONTACT

```
Marc A. Lehmann <schmorp@schmorp.de>
http://software.schmorp.de/pkg/Coro.html
```

5.2.20 Crypt

Namespace containing the following modules:

- `Crypt::OpenSSL`

Crypt::OpenSSL

Namespace containing the following modules:

- `Crypt::OpenSSL::Random` - Draw bytes from OpenSSL's cryptographically secure random-number generator, and seed it.

Crypt::OpenSSL::Random

Draw bytes from OpenSSL's cryptographically secure random-number generator, and seed it.

`random_bytes($n)` returns `$n` random bytes, or false when the generator cannot supply them; `random_pseudo_bytes($n)` is the same (OpenSSL 3 has one generator). `random_seed($data)` mixes `$data` into the generator's state and returns `random_status()`: 1 when the generator is seeded well enough, which OpenSSL 3 does by itself from the operating system. Nothing is exported by default.

The generator is the one in the system's libcrypto 3, reached through the built-in Net::SSLeay, so both modules draw from the same state. `random_egd` is not provided: OpenSSL 3 is built without support for an entropy-gathering daemon, as Random.xs then compiles none.

This is a native implementation of Crypt::OpenSSL::Random 0.17's `Random.xs`; its `Crypt/OpenSSL/Random.pm` is used unchanged.

Synopsis

```
use Crypt::OpenSSL::Random qw(random_bytes random_status);

random_status() or die "the generator is not seeded";
my $key = random_bytes(32);        # 32 bytes for a key
printf "%d\n", length $key;        # 32
```

Differences from upstream

Two, each where perl 5.44 with Random.xs fails (t/81-xs-native/Crypt-OpenSSL-Random/150-or-better.t)

- A count of 0 returns the empty string. perl5 returns bytes of uninitialised memory, measured with `strlen`.
- A negative count returns false, as a refused request does. perl5 tries to allocate that many bytes and exits "Out of memory".

Functions

Random bytes

`random_bytes`

`random_bytes($n)`: `$n` bytes from the generator, or false when it cannot supply them.

`random_pseudo_bytes`

`random_pseudo_bytes($n)`: the same as `random_bytes`.

Seeding

`random_seed`

`random_seed($data)`: mix the bytes of `$data` into the generator; returns `random_status()`.

`random_status`

`random_status()`: 1 when the generator is seeded well enough, else 0.

5.2.21 Cwd

Report the current working directory and resolve paths to their absolute, symlink-free form. This module provides the four "where am I" functions (`cwd`, `getcwd`, `fastcwd`, `fastgetcwd`), the three "make this path absolute" functions (`abs_path`, `realpath`, `fast_abs_path`), and a `chdir` that keeps `$ENV{PWD}` in sync.

Two groups of tradeoffs matter when choosing between them.

Current directory. `getcwd` is the most portable and safe call - it asks the kernel via the POSIX `getcwd(3)` interface and never leaves the process anywhere surprising. `fastcwd` walks the tree with repeated `chdir('..')` and is faster on deep paths, but if a parent directory disappears mid-walk (or is unreadable) it returns `undef` and may leave the process in a different directory than it started. `cwd` picks the most natural form for the platform; on Linux it is effectively `getcwd`. `fastgetcwd` is a synonym for `cwd`.

Path resolution. `abs_path` canonicalises a path, resolving symlinks and `./..` components. `realpath` is its POSIX-style alias - same function, different name. `fast_abs_path` is the faster variant that drops the "non-existent leaf" fallback: it returns `undef` the moment any component fails to resolve, where `abs_path` will still return a usable path when only the final leaf is missing.

`chdir` changes directory and, on success, updates `$ENV{PWD}` so child processes inherit a consistent view.

Cwd's Perl half

These come from Cwd.pm itself, unchanged:

- `cwd` - the output of `/bin/pwd`; `fastgetcwd` is the same function.
- `fast_abs_path($path)` (alias `fast_realpath`) - resolves a directory by changing into it and back; `undef` with `$!` set to `ENOENT` when the path does not exist, missing leaf included.
- `chdir($dir)` - `CORE::chdir`, then keeps `$ENV{PWD}` LOGICAL: after entering a symlink it names the link, not its target, and `..` components are folded textually. Returns 1 or 0.
- `realpath` - the same function as `abs_path`.

This is Cwd 3.95 (PathTools); `Cwd.pm` is upstream's. `getcwd`, `cwd`, `fastcwd` and `fastgetcwd` are exported by default, the rest on request.

Synopsis

```
use Cwd qw(getcwd abs_path);

my $here = getcwd();                          # e.g. /home/alice/
 ↪project
print abs_path('/usr/../usr/./bin'), "\n";    # /usr/bin
print abs_path('some-symlink'), "\n";         # where it points,␣
 ↪resolved
print defined abs_path('/no/such/dir/file') ? "" : "undef\n";   #␣
 ↪undef
```

Differences from upstream

None: upstream's tests pass unchanged (t/81-xs-native/Cwd, the `upstream` files), and the rest there checks against perl 5.44.

Functions

Current directory

getcwd

Return the absolute path of the process's current working directory.

cwd

Return the absolute path of the process's current working directory.

chdir

Change the process's current working directory, and update `$ENV{PWD}` to match.

Path resolution

abs_path

Canonicalise a path: resolve symlinks, `.`, and `..`, and return the absolute form.

fast_abs_path

Canonicalise a path strictly: resolve symlinks and `./..`, or return `undef` if any component is missing.

getcwd

Return the absolute path of the process's current working directory.

Synopsis

```
use Cwd;
my $dir = getcwd();
my $dir = cwd();
my $dir = fastcwd();
my $dir = fastgetcwd();
```

What you get back

A plain Perl string: the absolute path, no trailing newline, no trailing slash (except when you really are at the root `/`). On failure `getcwd` and `fastcwd` return `undef` and set `$!`.

Examples

Print where the script is running from:

```
use Cwd;
print getcwd(), "\n";     # e.g. /home/alice/project
```

Save and restore the working directory around a block that `chdir`s:

```
use Cwd;
my $saved = getcwd();
chdir '/tmp' or die $!;
# ... do work in /tmp ...
chdir $saved or die $!;
```

Compare two paths by their canonical form:

```
use Cwd;
chdir '/var/log' or die $!;
print getcwd() eq '/var/log' ? "yes\n" : "no\n";    # yes
```

Edge cases

- If the current directory has been removed by another process, `getcwd` and `fastcwd` return `undef` with `$!` set to `ENOENT`, while `cwd` (the output of `/bin/pwd`) is the empty string.
- Symlinks in the path are already resolved by the kernel's `getcwd(3)`; the returned string is the real path.
- All four names are provided by default when you `use Cwd` with no import list.
- `fastcwd` takes no arguments.
- Under taint mode the result is tainted.

See also

- `abs_path` - canonicalise an arbitrary path, not just the CWD
- `realpath` - POSIX-style alias for `abs_path`

abs_path

Canonicalise a path: resolve symlinks, `.`, and `..`, and return the absolute form.

Synopsis

```
use Cwd qw(abs_path realpath);
my $abs = abs_path($path);
my $abs = realpath($path);    # same function, POSIX alias
my $abs = abs_path();         # defaults to current directory
```

What you get back

A plain Perl string: the absolute, symlink-free path. On unresolvable input returns `undef` and sets `$!`. If the leaf component does not exist but its parent does, the parent is canonicalised and the leaf is appended verbatim - useful for computing where a file *would* be written.

Examples

Resolve a relative path against the current directory:

```
use Cwd qw(abs_path);
chdir '/home/alice' or die $!;
print abs_path('project/Makefile'), "\n";
    # /home/alice/project/Makefile
```

Follow a symlink to its target:

```
use Cwd qw(abs_path);
symlink '/etc/hostname', '/tmp/link';
print abs_path('/tmp/link'), "\n";     # /etc/hostname
```

Compute the absolute path a file would have before creating it:

```
use Cwd qw(abs_path);
print abs_path('output.new'), "\n";
    # e.g. /home/alice/project/output.new - even if output.new
    # does not yet exist, as long as the directory does
```

Use `realpath` when the surrounding code expects the POSIX name:

```
use Cwd qw(realpath);
print realpath('.'), "\n";      # absolute path of current dir
```

Edge cases

- No argument - treated as `.`. An empty string resolves to the current directory too.
- Leaf missing, parent present - returns parent's canonical path joined with the leaf name. `fast_abs_path` returns `undef` in this situation; use it when "doesn't exist" should be an error.
- An inner component missing - returns `undef`.
- A chain of more than 21 symlinks - `undef`, `$!` set to `ELOOP`.
- The path is taken as bytes: a name that is not UTF-8 comes back unchanged.
- Under taint mode the result is tainted.

See also

- `fast_abs_path` - same canonicalisation, no leaf fallback
- `getcwd` - absolute path of the current directory only

cwd

Return the absolute path of the process's current working directory.

Synopsis

```
use Cwd;
my $dir = getcwd();
my $dir = cwd();
my $dir = fastcwd();
my $dir = fastgetcwd();
```

What you get back

A plain Perl string: the absolute path, no trailing newline, no trailing slash (except when you really are at the root `/`). On failure `getcwd` and `fastcwd` return `undef` and set `$!`; `cwd` and `fastgetcwd` run `/bin/pwd` and return an empty string.

Examples

Print where the script is running from:

```
use Cwd;
print getcwd(), "\n";     # e.g. /home/alice/project
```

Save and restore the working directory around a block that `chdir`s:

```
use Cwd;
my $saved = getcwd();
chdir '/tmp' or die $!;
# ... do work in /tmp ...
chdir $saved or die $!;
```

Compare two paths by their canonical form:

```
use Cwd;
chdir '/var/log' or die $!;
print getcwd() eq '/var/log' ? "yes\n" : "no\n";    # yes
```

Edge cases

- If the current directory has been removed by another process, `getcwd` and `fastcwd` return `undef` with `$!` set to `ENOENT`; `cwd` and `fastgetcwd` return an empty string, and `/bin/pwd` reports the failure on STDERR.
- Symlinks in the path are already resolved by the kernel's `getcwd(3)`; the returned string is the real path.
- All four names are provided by default when you `use Cwd` with no import list.

Differences from upstream

Fully compatible with upstream.

See also

- `abs_path` - canonicalise an arbitrary path, not just the CWD
- `chdir` - change directory and keep `$ENV{PWD}` synchronised
- `realpath` - POSIX-style alias for `abs_path`

fast_abs_path

Canonicalise a path strictly: resolve symlinks and `./..`, or return `undef` if any component is missing.

Synopsis

```
use Cwd qw(fast_abs_path);
my $abs = fast_abs_path($path);
my $abs = fast_abs_path();      # defaults to current directory
```

What you get back

A plain Perl string with the absolute, symlink-free path, or `undef` if any part of the path cannot be resolved (including the leaf). Unlike `abs_path`, there is no parent-plus-leaf fallback - a missing final component is treated as an error.

Examples

Verify that a path exists and get its canonical form in one step:

```
use Cwd qw(fast_abs_path);
my $abs = fast_abs_path('/etc/passwd');
defined $abs or die "not found: $!";
print $abs, "\n";                # /etc/passwd
```

Reject non-existent files up front:

```
use Cwd qw(fast_abs_path);
print defined fast_abs_path('/no/such/file') ? "ok" : "missing";
    # missing
```

Resolve the current directory quickly:

```
use Cwd qw(fast_abs_path);
print fast_abs_path(), "\n";      # equivalent to getcwd()
```

Edge cases

- No argument - the current directory.
- A path that does not exist, including the empty string - `undef`, with `$!` set to `ENOENT`. Use `abs_path` when a missing leaf should still produce a path.
- A directory is resolved by changing into it and back; a file by resolving its directory, following the file itself when it is a symlink.

See also

- `abs_path` - same resolution, with a parent-plus-leaf fallback
- `realpath` - POSIX-style alias for `abs_path`
- `getcwd` - absolute path of the current directory only

chdir

Change the process's current working directory, and update `$ENV{PWD}` to match.

Synopsis

```
use Cwd qw(chdir);
chdir($path) or die $!;
```

What you get back

1 on success, `0` on failure. `$!` is set when the call fails so you can distinguish errors.

Examples

Change directory and keep `$ENV{PWD}` consistent for child processes:

```
use Cwd qw(chdir);
chdir('/tmp') or die "chdir: $!";
system('printenv', 'PWD');       # /tmp
```

Guard against a missing target:

```
use Cwd qw(chdir);
unless (chdir('/does/not/exist')) {
    warn "cannot enter directory: $!";
}
```

Switch directories around a block and return:

```
use Cwd qw(getcwd chdir);
my $saved = getcwd();
chdir('/var/log') or die $!;
# ... work in /var/log ...
chdir($saved) or die $!;
```

Edge cases

- Called with no argument - returns `0` (unlike the core `chdir` builtin, which chdirs to `$ENV{HOME}`). Pass the target explicitly.
- Target exists but is not a directory - returns `0` with `$!` set to `ENOTDIR`.

- `$ENV{PWD}` is updated only on success, through the `%ENV` tied hash, so `setenv(3)` is called as well.

Differences from upstream

Fully compatible with upstream.

See also

- `getcwd` - absolute path of the current directory
- `abs_path` - canonicalise a path without changing directory

5.2.22 DB

programmatic interface to the Perl debugging API

SYNOPSIS

```
package CLIENT;
use DB;
@ISA = qw(DB);

# these (inherited) methods can be called by the client

CLIENT->register()      # register a client package name
CLIENT->done()          # de-register from the debugging API
CLIENT->skippkg('hide::hide')  # ask DB not to stop in this package
CLIENT->cont([WHERE])       # run some more (until BREAK or
                            # another breakpoint)
CLIENT->step()              # single step
CLIENT->next()              # step over
CLIENT->ret()               # return from current subroutine
CLIENT->backtrace()         # return the call stack description
CLIENT->ready()             # call when client setup is done
CLIENT->trace_toggle()      # toggle subroutine call trace mode
CLIENT->subs([SUBS])        # return subroutine information
CLIENT->files()             # return list of all files known to DB
CLIENT->lines()             # return lines in currently loaded file
CLIENT->loadfile(FILE,LINE) # load a file and let other clients know
CLIENT->lineevents()        # return info on lines with actions
CLIENT->set_break([WHERE],[COND])
CLIENT->set_tbreak([WHERE])
CLIENT->clr_breaks([LIST])
CLIENT->set_action(WHERE,ACTION)
CLIENT->clr_actions([LIST])
CLIENT->evalcode(STRING)  # eval STRING in executing code's context
CLIENT->prestop([STRING]) # execute in code context before stopping
CLIENT->poststop([STRING])# execute in code context before resuming
```

(continues on next page)

(continued from previous page)

```
# These methods will be called at the appropriate times.
# Stub versions provided do nothing.
# None of these can block.

CLIENT->init()          # called when debug API inits itself
CLIENT->stop(FILE,LINE) # when execution stops
CLIENT->idle()          # while stopped (can be a client event loop)
CLIENT->cleanup()       # just before exit
CLIENT->output(LIST)    # called to print any output that
                        # the API must show
```

DESCRIPTION

Perl debug information is frequently required not just by debuggers, but also by modules that need some "special" information to do their job properly, like profilers.

This module abstracts and provides all of the hooks into Perl internal debugging functionality, so that various implementations of Perl debuggers (or packages that want to simply get at the "privileged" debugging data) can all benefit from the development of this common code. Currently used by Swat, the perl/Tk GUI debugger.

Note that multiple "front-ends" can latch into this debugging API simultaneously. This is intended to facilitate things like debugging with a command line and GUI at the same time, debugging debuggers etc. [Sounds nice, but this needs some serious support – GSAR]

In particular, this API does **not** provide the following functions:

- data display
- command processing
- command alias management
- user interface (tty or graphical)

These are intended to be services performed by the clients of this API.

This module attempts to be squeaky clean w.r.t `use strict;` and when warnings are enabled.

Global Variables

The following "public" global names can be read by clients of this API. Beware that these should be considered "readonly".

- $DB::sub

  Name of current executing subroutine.

- The keys of this hash are the names of all the known subroutines. Each value is an encoded string that has the sprintf(3) format `("%s:%d-%d", filename, fromline, toline)`.

- $DB::single

Single-step flag. Will be true if the API will stop at the next statement.

- $DB::signal

  Signal flag. Will be set to a true value if a signal was caught. Clients may check for this flag to abort time-consuming operations.

- $DB::trace

  This flag is set to true if the API is tracing through subroutine calls.

- @DB::args

  Contains the arguments of current subroutine, or the `@ARGV` array if in the toplevel context.

- @DB::dbline

  List of lines in currently loaded file.

- Actions in current file (keys are line numbers). The values are strings that have the sprintf(3) format `("%s\000%s", breakcondition, actioncode)`.

- $DB::package

  Package namespace of currently executing code.

- $DB::filename

  Currently loaded filename.

- $DB::subname

  Fully qualified name of currently executing subroutine.

- $DB::lineno

  Line number that will be executed next.

API Methods

The following are methods in the DB base class. A client must access these methods by inheritance (*not* by calling them directly), since the API keeps track of clients through the inheritance mechanism.

- CLIENT->register()

  register a client object/package

- CLIENT->evalcode(STRING)

  eval STRING in executing code context

- CLIENT->skippkg('D::hide')

  ask DB not to stop in these packages

- CLIENT->run()

  run some more (until a breakpt is reached)

- CLIENT->step()

  single step

- CLIENT->next()

  step over

- CLIENT->done()

  de-register from the debugging API

Client Callback Methods

The following "virtual" methods can be defined by the client. They will be called by the API at appropriate points. Note that unless specified otherwise, the debug API only defines empty, non-functional default versions of these methods.

- CLIENT->init()

  Called after debug API inits itself.

- CLIENT->prestop([STRING])

  Usually inherited from DB package. If no arguments are passed, returns the prestop action string.

- CLIENT->stop()

  Called when execution stops (w/ args file, line).

- CLIENT->idle()

  Called while stopped (can be a client event loop).

- CLIENT->poststop([STRING])

  Usually inherited from DB package. If no arguments are passed, returns the poststop action string.

- CLIENT->evalcode(STRING)

  Usually inherited from DB package. Ask for a STRING to be `eval`-ed in executing code context.

- CLIENT->cleanup()

  Called just before exit.

- CLIENT->output(LIST)

  Called when API must show a message (warnings, errors etc.).

BUGS

The interface defined by this module is missing some of the later additions to perl's debugging functionality. As such, this interface should be considered highly experimental and subject to change.

AUTHOR

Gurusamy Sarathy gsar@activestate.com

This code heavily adapted from an early version of perl5db.pl attributable to Larry Wall and the Perl Porters.

5.2.23 DBD

Namespace containing the following modules:

- `DBD::DBM` - a DBI driver for DBM & MLDBM files
- `DBD::File` - Base class for writing file based DBI drivers
- `DBD::Gofer` - A stateless-proxy driver for communicating with a remote DBI
- `DBD::Mem` - a DBI driver for Mem & MLMem files
- `DBD::Proxy` - A proxy driver for the DBI
- `DBD::SQLite`
- `DBD::Sponge` - Create a DBI statement handle from Perl data

DBD::DBM

a DBI driver for DBM & MLDBM files

SYNOPSIS

```
use DBI;
$dbh = DBI->connect('dbi:DBM:');                    # defaults to␣
↪SDBM_File
$dbh = DBI->connect('DBI:DBM(RaiseError=1):');      # defaults to␣
↪SDBM_File
$dbh = DBI->connect('dbi:DBM:dbm_type=DB_File');    # defaults to␣
↪DB_File
$dbh = DBI->connect('dbi:DBM:dbm_mldbm=Storable');  # MLDBM with␣
↪SDBM_File

# or
$dbh = DBI->connect('dbi:DBM:', undef, undef);
$dbh = DBI->connect('dbi:DBM:', undef, undef, {
    f_ext              => '.db/r',
    f_dir              => '/path/to/dbfiles/',
    f_lockfile         => '.lck',
    dbm_type           => 'BerkeleyDB',
    dbm_mldbm          => 'FreezeThaw',
    dbm_store_metadata => 1,
    dbm_berkeley_flags => {
        '-Cachesize' => 1000, # set a ::Hash flag
    },
});
```

and other variations on connect() as shown in the *DBI* docs, *DBD::File metadata* and "Metadata" shown below.

Use standard DBI prepare, execute, fetch, placeholders, etc., see "QUICK START" for an example.

DESCRIPTION

DBD::DBM is a database management system that works right out of the box. If you have a standard installation of Perl and DBI you can begin creating, accessing, and modifying simple database tables without any further modules. You can add other modules (e.g., SQL::Statement, DB_File etc) for improved functionality.

The module uses a DBM file storage layer. DBM file storage is common on many platforms and files can be created with it in many programming languages using different APIs. That means, in addition to creating files with DBI/SQL, you can also use DBI/SQL to access and modify files created by other DBM modules and programs and vice versa. **Note** that in those cases it might be necessary to use a common subset of the provided features.

DBM files are stored in binary format optimized for quick retrieval when using a key field. That optimization can be used advantageously to make DBD::DBM SQL operations that use key fields very fast. There are several different "flavors" of DBM which use different storage formats supported by perl modules such as SDBM_File and MLDBM. This module supports all of the flavors that perl supports and, when used with MLDBM, supports tables with any number of columns and insertion of Perl objects into tables.

DBD::DBM has been tested with the following DBM types: SDBM_File, NDBM_File, ODBM_File, GDBM_File, DB_File, BerkeleyDB. Each type was tested both with and without MLDBM and with the Data::Dumper, Storable, FreezeThaw, YAML and JSON serializers using the DBI::SQL::Nano or the SQL::Statement engines.

QUICK START

DBD::DBM operates like all other DBD drivers - it's basic syntax and operation is specified by DBI. If you're not familiar with DBI, you should start by reading *DBI* and the documents it points to and then come back and read this file. If you are familiar with DBI, you already know most of what you need to know to operate this module. Just jump in and create a test script something like the one shown below.

You should be aware that there are several options for the SQL engine underlying DBD::DBM, see "Supported SQL syntax". There are also many options for DBM support, see especially the section on "Adding multi-column support with MLDBM".

But here's a sample to get you started.

```
use DBI;
my $dbh = DBI->connect('dbi:DBM:');
$dbh->{RaiseError} = 1;
for my $sql( split /;\n+/,"
    CREATE TABLE user ( user_name TEXT, phone TEXT );
    INSERT INTO user VALUES ('Fred Bloggs','233-7777');
    INSERT INTO user VALUES ('Sanjay Patel','777-3333');
    INSERT INTO user VALUES ('Junk','xxx-xxxx');
    DELETE FROM user WHERE user_name = 'Junk';
    UPDATE user SET phone = '999-4444' WHERE user_name = 'Sanjay␣
↪Patel';
    SELECT * FROM user
"){
```

(continues on next page)

(continued from previous page)

```
    my $sth = $dbh->prepare($sql);
    $sth->execute;
    $sth->dump_results if $sth->{NUM_OF_FIELDS};
}
$dbh->disconnect;
```

USAGE

This section will explain some usage cases in more detail. To get an overview about the available attributes, see “Metadata”.

Specifying Files and Directories

DBD::DBM will automatically supply an appropriate file extension for the type of DBM you are using. For example, if you use SDBM_File, a table called “fruit” will be stored in two files called “fruit.pag” and “fruit.dir”. You should **never** specify the file extensions in your SQL statements.

DBD::DBM recognizes following default extensions for following types:

- .pag/r

  Chosen for dbm_type `SDBM_File`, `ODBM_File` and `NDBM_File` when an implementation is detected which wraps `-ldbm` for `NDBM_File` (e.g. Solaris, AIX, ...).

  For those types, the `.dir` extension is recognized, too (for being deleted when dropping a table).

- .db/r

  Chosen for dbm_type `NDBM_File` when an implementation is detected which wraps BerkeleyDB 1.x for `NDBM_File` (typically BSD’s, Darwin).

`GDBM_File`, `DB_File` and `BerkeleyDB` don’t usually use a file extension.

If your DBM type uses an extension other than one of the recognized types of extensions, you should set the *f_ext* attribute to the extension **and** file a bug report as described in DBI with the name of the implementation and extension so we can add it to DBD::DBM. Thanks in advance for that :-).

```
$dbh = DBI->connect('dbi:DBM:f_ext=.db');  # .db extension is used
$dbh = DBI->connect('dbi:DBM:f_ext=');     # no extension is used

# or
$dbh->{f_ext}='.db';                       # global setting
$dbh->{f_meta}->{'qux'}->{f_ext}='.db';    # setting for table 'qux'
```

By default files are assumed to be in the current working directory. To use other directories specify the *f_dir* attribute in either the connect string or by setting the database handle attribute.

For example, this will look for the file /foo/bar/fruit (or /foo/bar/fruit.pag for DBM types that use that extension)

```
my $dbh = DBI->connect('dbi:DBM:f_dir=/foo/bar');
# and this will too:
my $dbh = DBI->connect('dbi:DBM:');
$dbh->{f_dir} = '/foo/bar';
# but this is recommended
my $dbh = DBI->connect('dbi:DBM:', undef, undef, { f_dir => '/foo/
 ↪bar' } );

# now you can do
my $ary = $dbh->selectall_arrayref(q{ SELECT x FROM fruit });
```

You can also use delimited identifiers to specify paths directly in SQL statements. This looks in the same place as the two examples above but without setting *f_dir*:

```
my $dbh = DBI->connect('dbi:DBM:');
my $ary = $dbh->selectall_arrayref(q{
    SELECT x FROM "/foo/bar/fruit"
});
```

You can also tell DBD::DBM to use a specified path for a specific table:

```
$dbh->{dbm_tables}->{f}->{file} = q(/foo/bar/fruit);
```

Please be aware that you cannot specify this during connection.

If you have SQL::Statement installed, you can use table aliases:

```
my $dbh = DBI->connect('dbi:DBM:');
my $ary = $dbh->selectall_arrayref(q{
    SELECT f.x FROM "/foo/bar/fruit" AS f
});
```

See the "GOTCHAS AND WARNINGS" for using DROP on tables.

Table locking and flock()

Table locking is accomplished using a lockfile which has the same basename as the table's file but with the file extension '.lck' (or a lockfile extension that you supply, see below). This lock file is created with the table during a CREATE and removed during a DROP. Every time the table itself is opened, the lockfile is flocked(). For SELECT, this is a shared lock. For all other operations, it is an exclusive lock (except when you specify something different using the *f_lock* attribute).

Since the locking depends on flock(), it only works on operating systems that support flock(). In cases where flock() is not implemented, DBD::DBM will simply behave as if the flock() had occurred although no actual locking will happen. Read the documentation for flock() for more information.

Even on those systems that do support flock(), locking is only advisory - as is always the case with flock(). This means that if another program tries to access the table file while DBD::DBM has the table locked, that other program will *succeed* at opening unless it is also using flock on the '.lck' file. As a result DBD::DBM's locking only really applies to

other programs using DBD::DBM or other program written to cooperate with DBD::DBM locking.

Specifying the DBM type

Each "flavor" of DBM stores its files in a different format and has different capabilities and limitations. See *AnyDBM_File* for a comparison of DBM types.

By default, DBD::DBM uses the `SDBM_File` type of storage since `SDBM_File` comes with Perl itself. If you have other types of DBM storage available, you can use any of them with DBD::DBM. It is strongly recommended to use at least `DB_File`, because `SDBM_File` has quirks and limitations and `ODBM_file`, `NDBM_File` and `GDBM_File` are not always available.

You can specify the DBM type using the *dbm_type* attribute which can be set in the connection string or with `$dbh->{dbm_type}` and `$dbh->{f_meta}->{$table_name}->{type}` for per-table settings in cases where a single script is accessing more than one kind of DBM file.

In the connection string, just set `dbm_type=TYPENAME` where `TYPENAME` is any DBM type such as GDBM_File, DB_File, etc. Do *not* use MLDBM as your *dbm_type* as that is set differently, see below.

```
my $dbh=DBI->connect('dbi:DBM:');                # uses the default␣
 ↪SDBM_File
my $dbh=DBI->connect('dbi:DBM:dbm_type=GDBM_File'); # uses the GDBM_
 ↪File

# You can also use $dbh->{dbm_type} to set the DBM type for the␣
 ↪connection:
$dbh->{dbm_type} = 'DB_File';    # set the global DBM type
print $dbh->{dbm_type};          # display the global DBM type
```

If you have several tables in your script that use different DBM types, you can use the $dbh->{dbm_tables} hash to store different settings for the various tables. You can even use this to perform joins on files that have completely different storage mechanisms.

```
# sets global default of GDBM_File
my $dbh->('dbi:DBM:type=GDBM_File');

# overrides the global setting, but only for the tables called
# I<foo> and I<bar>
my $dbh->{f_meta}->{foo}->{dbm_type} = 'DB_File';
my $dbh->{f_meta}->{bar}->{dbm_type} = 'BerkeleyDB';

# prints the dbm_type for the table "foo"
print $dbh->{f_meta}->{foo}->{dbm_type};
```

Note that you must change the *dbm_type* of a table before you access it for first time.

Adding multi-column support with MLDBM

Most of the DBM types only support two columns and even if it would support more, DBD::DBM would only use two. However a CPAN module called MLDBM overcomes this limitation by allowing more than two columns. MLDBM does this by serializing the data - basically it puts a reference to an array into the second column. It can also put almost any kind of Perl object or even **Perl coderefs** into columns.

If you want more than two columns, you **must** install MLDBM. It's available for many platforms and is easy to install.

MLDBM is by default distributed with three serializers - Data::Dumper, Storable, and FreezeThaw. Data::Dumper is the default and Storable is the fastest. MLDBM can also make use of user-defined serialization methods or other serialization modules (e.g. YAML::MLDBM or MLDBM::Serializer::JSON. You select the serializer using the *dbm_mldbm* attribute.

Some examples:

```
$dbh=DBI->connect('dbi:DBM:dbm_mldbm=Storable');  # use MLDBM with␣
↪Storable
$dbh=DBI->connect(
   'dbi:DBM:dbm_mldbm=MySerializer' # use MLDBM with a user defined␣
↪module
);
$dbh=DBI->connect('dbi::dbm:', undef,
    undef, { dbm_mldbm => 'YAML' }); # use 3rd party serializer
$dbh->{dbm_mldbm} = 'YAML'; # same as above
print $dbh->{dbm_mldbm} # show the MLDBM serializer
$dbh->{f_meta}->{foo}->{dbm_mldbm}='Data::Dumper';   # set␣
↪Data::Dumper for table "foo"
print $dbh->{f_meta}->{foo}->{mldbm}; # show serializer for table
↪"foo"
```

MLDBM works on top of other DBM modules so you can also set a DBM type along with setting dbm_mldbm. The examples above would default to using SDBM_File with MLDBM. If you wanted GDBM_File instead, here's how:

```
# uses DB_File with MLDBM and Storable
$dbh = DBI->connect('dbi:DBM:', undef, undef, {
    dbm_type  => 'DB_File',
    dbm_mldbm => 'Storable',
});
```

SDBM_File, the default *dbm_type* is quite limited, so if you are going to use MLDBM, you should probably use a different type, see *AnyDBM_File*.

See below for some "GOTCHAS AND WARNINGS" about MLDBM.

Support for Berkeley DB

The Berkeley DB storage type is supported through two different Perl modules - DB_File (which supports only features in old versions of Berkeley DB) and BerkeleyDB (which supports all versions). DBD::DBM supports specifying either "DB_File" or "BerkeleyDB" as a *dbm_type*, with or without MLDBM support.

The "BerkeleyDB" dbm_type is experimental and it's interface is likely to change. It currently defaults to BerkeleyDB::Hash and does not currently support ::Btree or ::Recno.

With BerkeleyDB, you can specify initialization flags by setting them in your script like this:

```
use BerkeleyDB;
my $env = new BerkeleyDB::Env -Home => $dir;  # and/or other Env
↪flags
$dbh = DBI->connect('dbi:DBM:', undef, undef, {
    dbm_type  => 'BerkeleyDB',
    dbm_mldbm => 'Storable',
    dbm_berkeley_flags => {
        'DB_CREATE'  => DB_CREATE,  # pass in constants
        'DB_RDONLY'  => DB_RDONLY,  # pass in constants
        '-Cachesize' => 1000,       # set a ::Hash flag
        '-Env'       => $env,       # pass in an environment
    },
});
```

Do *not* set the -Flags or -Filename flags as those are determined and overwritten by the SQL (e.g. -Flags => DB_RDONLY is set automatically when you issue a SELECT statement).

Time has not permitted us to provide support in this release of DBD::DBM for further Berkeley DB features such as transactions, concurrency, locking, etc. We will be working on these in the future and would value suggestions, patches, etc.

See *DB_File* and BerkeleyDB for further details.

Optimizing the use of key fields

Most "flavors" of DBM have only two physical columns (but can contain multiple logical columns as explained above in "Adding multi-column support with MLDBM"). They work similarly to a Perl hash with the first column serving as the key. Like a Perl hash, DBM files permit you to do quick lookups by specifying the key and thus avoid looping through all records (supported by DBI::SQL::Nano only). Also like a Perl hash, the keys must be unique. It is impossible to create two records with the same key. To put this more simply and in SQL terms, the key column functions as the *PRIMARY KEY* or UNIQUE INDEX.

In DBD::DBM, you can take advantage of the speed of keyed lookups by using DBI::SQL::Nano and a WHERE clause with a single equal comparison on the key field. For example, the following SQL statements are optimized for keyed lookup:

```
CREATE TABLE user ( user_name TEXT, phone TEXT);
INSERT INTO user VALUES ('Fred Bloggs','233-7777');
```

(continues on next page)

(continued from previous page)

```
# ... many more inserts
SELECT phone FROM user WHERE user_name='Fred Bloggs';
```

The “user_name” column is the key column since it is the first column. The SELECT statement uses the key column in a single equal comparison - “user_name=’Fred Bloggs’” - so the search will find it very quickly without having to loop through all the names which were inserted into the table.

In contrast, these searches on the same table are not optimized:

```
1. SELECT phone FROM user WHERE user_name < 'Fred';
2. SELECT user_name FROM user WHERE phone = '233-7777';
```

In #1, the operation uses a less-than (<) comparison rather than an equals comparison, so it will not be optimized for key searching. In #2, the key field “user_name” is not specified in the WHERE clause, and therefore the search will need to loop through all rows to find the requested row(s).

Note that the underlying DBM storage needs to loop over all *key/value* pairs when the optimized fetch is used. SQL::Statement has a massively improved where clause evaluation which costs around 15% of the evaluation in DBI::SQL::Nano - combined with the loop in the DBM storage the speed improvement isn’t so impressive.

Even if lookups are faster by around 50%, DBI::SQL::Nano and SQL::Statement can benefit from the key field optimizations on updating and deleting rows - and here the improved where clause evaluation of SQL::Statement might beat DBI::SQL::Nano every time the where clause contains not only the key field (or more than one).

Updatable Keys

The default behavior is that updating a key to the same value of an existing key will succeed, and the existing row will be overwritten.

There may be cases where this is undesirable.

This behavior can now be configured, by setting `dbm_updatable_key` attribute. To emit a warning when this happens, set it to 1:

```
$dbh = DBI->connect('dbi:DBM:', undef, undef, {
    dbm_updatable_key => 1,
} );
```

To instead die when that happens, set it to 2:

```
$dbh = DBI->connect('dbi:DBM:', undef, undef, {
    dbm_updatable_key => 2,
} );
```

Supported SQL syntax

DBD::DBM uses a subset of SQL. The robustness of that subset depends on what other modules you have installed. Both options support basic SQL operations including CREATE TABLE, DROP TABLE, INSERT, DELETE, UPDATE, and SELECT.

Option #1: By default, this module inherits its SQL support from DBI::SQL::Nano that comes with DBI. Nano is, as its name implies, a *very* small SQL engine. Although limited in scope, it is faster than option #2 for some operations (especially single *primary key* lookups). See *DBI::SQL::Nano* for a description of the SQL it supports and comparisons of it with option #2.

Option #2: If you install the pure Perl CPAN module SQL::Statement, DBD::DBM will use it instead of Nano. This adds support for table aliases, functions, joins, and much more. If you're going to use DBD::DBM for anything other than very simple tables and queries, you should install SQL::Statement. You don't have to change DBD::DBM or your scripts in any way, simply installing SQL::Statement will give you the more robust SQL capabilities without breaking scripts written for DBI::SQL::Nano. See SQL::Statement for a description of the SQL it supports.

To find out which SQL module is working in a given script, you can use the dbm_versions() method or, if you don't need the full output and version numbers, just do this:

```
print $dbh->{sql_handler}, "\n";
```

That will print out either "SQL::Statement" or "DBI::SQL::Nano".

Baring the section about optimized access to the DBM storage in mind, comparing the benefits of both engines:

```
# DBI::SQL::Nano is faster
$sth = $dbh->prepare( "update foo set value='new' where key=15" );
$sth->execute();
$sth = $dbh->prepare( "delete from foo where key=27" );
$sth->execute();
$sth = $dbh->prepare( "select * from foo where key='abc'" );

# SQL::Statement might faster (depending on DB size)
$sth = $dbh->prepare( "update foo set value='new' where key=?" );
$sth->execute(15);
$sth = $dbh->prepare( "update foo set value=? where key=15" );
$sth->execute('new');
$sth = $dbh->prepare( "delete from foo where key=?" );
$sth->execute(27);

# SQL::Statement is faster
$sth = $dbh->prepare( "update foo set value='new' where value='old'"
 ↪);
$sth->execute();
# must be expressed using "where key = 15 or key = 27 or key = 42 or
 ↪key = 'abc'"
# in DBI::SQL::Nano
$sth = $dbh->prepare( "delete from foo where key in (15,27,42,'abc')
```

(continues on next page)

(continued from previous page)

```
 ↪" );
$sth->execute();
# must be expressed using "where key > 10 and key < 90" in␣
 ↪DBI::SQL::Nano
$sth = $dbh->prepare( "select * from foo where key between (10,90)"␣
 ↪);
$sth->execute();

# only SQL::Statement can handle
$sth->prepare( "select * from foo,bar where foo.name = bar.name" );
$sth->execute();
$sth->prepare( "insert into foo values ( 1, 'foo' ), ( 2, 'bar' )"␣
 ↪);
$sth->execute();
```

Specifying Column Names

DBM files don't have a standard way to store column names. DBD::DBM gets around this issue with a DBD::DBM specific way of storing the column names. **If you are working only with DBD::DBM and not using files created by or accessed with other DBM programs, you can ignore this section.**

DBD::DBM stores column names as a row in the file with the key *_metadata \0*. So this code

```
my $dbh = DBI->connect('dbi:DBM:');
$dbh->do("CREATE TABLE baz (foo CHAR(10), bar INTEGER)");
$dbh->do("INSERT INTO baz (foo,bar) VALUES ('zippy',1)");
```

Will create a file that has a structure something like this:

```
_metadata \0 | <dbd_metadata><schema></schema><col_names>foo,bar</
 ↪col_names></dbd_metadata>
zippy        | 1
```

The next time you access this table with DBD::DBM, it will treat the *_metadata \0* row as a header rather than as data and will pull the column names from there. However, if you access the file with something other than DBD::DBM, the row will be treated as a regular data row.

If you do not want the column names stored as a data row in the table you can set the *dbm_store_metadata* attribute to 0.

```
my $dbh = DBI->connect('dbi:DBM:', undef, undef, { dbm_store_
 ↪metadata => 0 });

# or
$dbh->{dbm_store_metadata} = 0;

# or for per-table setting
$dbh->{f_meta}->{qux}->{dbm_store_metadata} = 0;
```

By default, DBD::DBM assumes that you have two columns named "k" and "v" (short for "key" and "value"). So if you have *dbm_store_metadata* set to 1 and you want to use alternate column names, you need to specify the column names like this:

```
my $dbh = DBI->connect('dbi:DBM:', undef, undef, {
    dbm_store_metadata => 0,
    dbm_cols => [ qw(foo bar) ],
});

# or
$dbh->{dbm_store_metadata} = 0;
$dbh->{dbm_cols}           = 'foo,bar';

# or to set the column names on per-table basis, do this:
# sets the column names only for table "qux"
$dbh->{f_meta}->{qux}->{dbm_store_metadata} = 0;
$dbh->{f_meta}->{qux}->{col_names}          = [qw(foo bar)];
```

If you have a file that was created by another DBM program or created with *dbm_store_metadata* set to zero and you want to convert it to using DBD::DBM's column name storage, just use one of the methods above to name the columns but *without* specifying *dbm_store_metadata* as zero. You only have to do that once - thereafter you can get by without setting either *dbm_store_metadata* or setting *dbm_cols* because the names will be stored in the file.

DBI database handle attributes

Metadata

Statement handle ($sth) attributes and methods

Most statement handle attributes such as NAME, NUM_OF_FIELDS, etc. are available only after an execute. The same is true of $sth->rows which is available after the execute but does *not* require a fetch.

Driver handle ($dbh) attributes

It is not supported anymore to use dbm-attributes without the dbm_-prefix. Currently, if an DBD::DBM private attribute is accessed without an underscore in it's name, dbm_ is prepended to that attribute and it's processed further. If the resulting attribute name is invalid, an error is thrown.

dbm_cols

Contains a comma separated list of column names or an array reference to the column names.

dbm_type

Contains the DBM storage type. Currently known supported type are `ODBM_File`, `NDBM_File`, `SDBM_File`, `GDBM_File`, `DB_File` and `BerkeleyDB`. It is not recommended to use one of the first three types - even if `SDBM_File` is the most commonly available *dbm_type*.

dbm_mldbm

Contains the serializer for DBM storage (value column). Requires the CPAN module MLDBM installed. Currently known supported serializers are:

- Data::Dumper

  Default serializer. Deployed with Perl core.

- Storable

  Faster serializer. Deployed with Perl core.

- FreezeThaw

  Pure Perl serializer, requires FreezeThaw to be installed.

- YAML

  Portable serializer (between languages but not architectures). Requires YAML::MLDBM installation.

- JSON

  Portable, fast serializer (between languages but not architectures). Requires MLDBM::Serializer::JSON installation.

dbm_store_metadata

Boolean value which determines if the metadata in DBM is stored or not.

dbm_berkeley_flags

Hash reference with additional flags for BerkeleyDB::Hash instantiation.

dbm_version

Readonly attribute containing the version of DBD::DBM.

f_meta

In addition to the attributes *DBD::File* recognizes, DBD::DBM knows about the (public) attributes `col_names` (**Note** not *dbm_cols* here!), `dbm_type`, `dbm_mldbm`, `dbm_store_metadata` and `dbm_berkeley_flags`. As in DBD::File, there are undocumented, internal attributes in DBD::DBM. Be very careful when modifying attributes you do not know; the consequence might a destroyed or corrupted table.

dbm_tables

This attribute provides restricted access to the table meta data. See f_meta and *"f_meta" in DBD::File* for attribute details.

dbm_tables is a tied hash providing the internal table names as keys (accessing unknown tables might create an entry) and their meta data as another tied hash. The table meta storage is obtained via the `get_table_meta` method from the table implementation (see DBD::File::Developers). Attribute setting and getting within the table meta data is handled via the methods `set_table_meta_attr` and `get_table_meta_attr`.

Following attributes are no longer handled by DBD::DBM:

dbm_ext

This attribute is silently mapped to DBD::File's attribute *f_ext*. Later versions of DBI might show a depreciated warning when this attribute is used and eventually it will be removed.

dbm_lockfile

This attribute is silently mapped to DBD::File's attribute *f_lockfile*. Later versions of DBI might show a depreciated warning when this attribute is used and eventually it will be removed.

DBI database handle methods

The $dbh->dbm_versions() method

The private method dbm_versions() returns a summary of what other modules are being used at any given time. DBD::DBM can work with or without many other modules - it can use either SQL::Statement or DBI::SQL::Nano as its SQL engine, it can be run with DBI or DBI::PurePerl, it can use many kinds of DBM modules, and many kinds of serializers when run with MLDBM. The dbm_versions() method reports all of that and more.

```
print $dbh->dbm_versions;              # displays global settings
print $dbh->dbm_versions($table_name);  # displays per table␣
 ↪settings
```

An important thing to note about this method is that when it called with no arguments, it displays the *global* settings. If you override these by setting per-table attributes, these will *not* be shown unless you specify a table name as an argument to the method call.

Storing Objects

If you are using MLDBM, you can use DBD::DBM to take advantage of its serializing abilities to serialize any Perl object that MLDBM can handle. To store objects in columns, you should (but don't absolutely need to) declare it as a column of type BLOB (the type is *currently* ignored by the SQL engine, but it's good form).

EXTENSIBILITY

- `SQL::Statement`

  Improved SQL engine compared to the built-in DBI::SQL::Nano - see "Supported SQL syntax".

- `DB_File`

  Berkeley DB version 1. This database library is available on many systems without additional installation and most systems are supported.

- `GDBM_File`

  Simple dbm type (comparable to `DB_File`) under the GNU license. Typically not available (or requires extra installation) on non-GNU operating systems.

- `BerkeleyDB`

  Berkeley DB version up to v4 (and maybe higher) - requires additional installation but is easier than GDBM_File on non-GNU systems.

  db4 comes with a many tools which allow repairing and migrating databases. This is the **recommended** dbm type for production use.

- `MLDBM`

  Serializer wrapper to support more than one column for the files. Comes with serializers using `Data::Dumper`, `FreezeThaw` and `Storable`.

- `YAML::MLDBM`

  Additional serializer for MLDBM. YAML is very portable between languages.

- `MLDBM::Serializer::JSON`

  Additional serializer for MLDBM. JSON is very portable between languages, probably more than YAML.

GOTCHAS AND WARNINGS

Using the SQL DROP command will remove any file that has the name specified in the command with either '.pag' and '.dir', '.db' or your {f_ext} appended to it. So this be dangerous if you aren't sure what file it refers to:

```
$dbh->do(qq{DROP TABLE "/path/to/any/file"});
```

Each DBM type has limitations. SDBM_File, for example, can only store values of less than 1,000 characters. *You* as the script author must ensure that you don't exceed those bounds. If you try to insert a value that is larger than DBM can store, the results will be unpredictable. See the documentation for whatever DBM you are using for details.

Different DBM implementations return records in different orders. That means that you *should not* rely on the order of records unless you use an ORDER BY statement.

DBM data files are platform-specific. To move them from one platform to another, you'll need to do something along the lines of dumping your data to CSV on platform #1 and then dumping from CSV to DBM on platform #2. DBD::AnyData and DBD::CSV can help with that. There may also be DBM conversion tools for your platforms which would probably be quicker.

When using MLDBM, there is a very powerful serializer - it will allow you to store Perl code or objects in database columns. When these get de-serialized, they may be eval'ed - in other words MLDBM (or actually Data::Dumper when used by MLDBM) may take the values and try to execute them in Perl. Obviously, this can present dangers, so if you do not know what is in a file, be careful before you access it with MLDBM turned on!

See the entire section on "Table locking and flock()" for gotchas and warnings about the use of flock().

BUGS AND LIMITATIONS

This module uses hash interfaces of two column file databases. While none of supported SQL engines have support for indices, the following statements really do the same (even if they mean something completely different) for each dbm type which lacks `EXISTS` support:

```
$sth->do( "insert into foo values (1, 'hello')" );

# this statement does ...
$sth->do( "update foo set v='world' where k=1" );
# ... the same as this statement
$sth->do( "insert into foo values (1, 'world')" );
```

This is considered to be a bug and might change in a future release.

Known affected dbm types are `ODBM_File` and `NDBM_File`. We highly recommended you use a more modern dbm type such as `DB_File`.

GETTING HELP, MAKING SUGGESTIONS, AND REPORTING BUGS

If you need help installing or using DBD::DBM, please write to the DBI users mailing list at dbi-users@perl.org or to the comp.lang.perl.modules newsgroup on usenet. I cannot always answer every question quickly but there are many on the mailing list or in the newsgroup who can.

DBD developers for DBD's which rely on DBD::File or DBD::DBM or use one of them as an example are suggested to join the DBI developers mailing list at dbi-dev@perl.org and strongly encouraged to join our IRC channel at irc://irc.perl.org/dbi.

If you have suggestions, ideas for improvements, or bugs to report, please report a bug as described in DBI. Do not mail any of the authors directly, you might not get an answer.

When reporting bugs, please send the output of $dbh->dbm_versions($table) for a table that exhibits the bug and as small a sample as you can make of the code that produces the bug. And of course, patches are welcome, too :-).

If you need enhancements quickly, you can get commercial support as described at http://dbi.perl.org/support/ or you can contact Jens Rehsack at rehsack@googlemail.com for commercial support in Germany.

Please don't bother Jochen Wiedmann or Jeff Zucker for support - they handed over further maintenance to H.Merijn Brand and Jens Rehsack.

ACKNOWLEDGEMENTS

Many, many thanks to Tim Bunce for prodding me to write this, and for copious, wise, and patient suggestions all along the way. (Jeff Zucker)

I send my thanks and acknowledgements to H.Merijn Brand for his initial refactoring of DBD::File and his strong and ongoing support of SQL::Statement. Without him, the current progress would never have been made. And I have to name Martin J. Evans for each laugh (and correction) of all those funny word creations I (as non-native speaker) made to the documentation. And - of course - I have to thank all those unnamed contributors and testers from the Perl community. (Jens Rehsack)

AUTHOR AND COPYRIGHT

This module is written by Jeff Zucker < jzucker AT cpan.org >, who also maintained it till 2007. After that, in 2010, Jens Rehsack & H.Merijn Brand took over maintenance.

```
Copyright (c) 2004 by Jeff Zucker, all rights reserved.
Copyright (c) 2010-2013 by Jens Rehsack & H.Merijn Brand, all rights␣
↪reserved.
```

You may freely distribute and/or modify this module under the terms of either the GNU General Public License (GPL) or the Artistic License, as specified in the Perl README file.

SEE ALSO

DBI, SQL::Statement, *DBI::SQL::Nano*, *AnyDBM_File*, *DB_File*, BerkeleyDB, MLDBM, YAML::MLDBM, MLDBM::Serializer::JSON

DBD::File

Base class for writing file based DBI drivers

SYNOPSIS

This module is a base class for writing other *DBD*s. It is not intended to function as a DBD itself (though it is possible). If you want to access flat files, use DBD::AnyData, or DBD::CSV (both of which are subclasses of DBD::File).

DESCRIPTION

The DBD::File module is not a true *DBI* driver, but an abstract base class for deriving concrete DBI drivers from it. The implication is, that these drivers work with plain files, for example CSV files or INI files. The module is based on the SQL::Statement module, a simple SQL engine.

See *DBI* for details on DBI, SQL::Statement for details on SQL::Statement and DBD::CSV, *DBD::DBM* or DBD::AnyData for example drivers.

Metadata

The following attributes are handled by DBI itself and not by DBD::File, thus they all work as expected:

```
Active
ActiveKids
CachedKids
CompatMode             (Not used)
InactiveDestroy
AutoInactiveDestroy
Kids
PrintError
RaiseError
Warn                   (Not used)
```

The following DBI attributes are handled by DBD::File:

AutoCommit

Always on.

ChopBlanks

Works.

NUM_OF_FIELDS

Valid after `$sth->execute`.

NUM_OF_PARAMS

Valid after `$sth->prepare`.

NAME

Valid after `$sth->execute`; undef for Non-Select statements.

NULLABLE

Not really working, always returns an array ref of ones, except the affected table has been created in this session. Valid after `$sth->execute`; undef for non-select statements.

Unsupported DBI attributes and methods

- bind_param_inout
- CursorName
- LongReadLen

- LongTruncOk

DBD::File specific attributes

In addition to the DBI attributes, you can use the following dbh attributes:

f_dir

This attribute is used for setting the directory where the files are opened and it defaults to the current directory (`.`). Usually you set it on the dbh but it may be overridden per table (see f_meta).

When the value for `f_dir` is a relative path, it is converted into the appropriate absolute path name (based on the current working directory) when the dbh attribute is set.

```
f_dir => "/data/foo/csv",
```

If `f_dir` is set to a non-existing location, the connection will fail. See CVE-2014-10401 for reasoning. Because of this, folders to use cannot be created after the connection, but must exist before the connection is initiated.

See “KNOWN BUGS AND LIMITATIONS”.

f_dir_search

This optional attribute can be set to pass a list of folders to also find existing tables. It will **not** be used to create new files.

```
f_dir_search => [ "/data/bar/csv", "/dump/blargh/data" ],
```

f_ext

This attribute is used for setting the file extension. The format is:

```
extension{/flag}
```

where the /flag is optional and the extension is case-insensitive. `f_ext` allows you to specify an extension which:

```
f_ext => ".csv/r",
```

- makes DBD::File prefer `table.extension` over `table`.
- makes the table name the filename minus the extension.

```
DBI:CSV:f_dir=data;f_ext=.csv
```

In the above example and when `f_dir` contains both `table.csv` and `table`, DBD::File will open `table.csv` and the table will be named “table”. If `table.csv` does not exist but `table` does that file is opened and the table is also called “table”.

If `f_ext` is not specified and `table.csv` exists it will be opened and the table will be called “table.csv” which is probably not what you want.

NOTE: even though extensions are case-insensitive, table names are not.

```
DBI:CSV:f_dir=data;f_ext=.csv/r
```

The `r` flag means the file extension is required and any filename that does not match the extension is ignored.

Usually you set it on the dbh but it may be overridden per table (see f_meta).

f_schema

This will set the schema name and defaults to the owner of the directory in which the table file resides. You can set `f_schema` to `undef`.

```
my $dbh = DBI->connect ("dbi:CSV:", "", "", {
    f_schema => undef,
    f_dir    => "data",
    f_ext    => ".csv/r",
    }) or die $DBI::errstr;
```

By setting the schema you affect the results from the tables call:

```
my @tables = $dbh->tables ();

# no f_schema
"merijn".foo
"merijn".bar

# f_schema => "dbi"
"dbi".foo
"dbi".bar

# f_schema => undef
foo
bar
```

Defining `f_schema` to the empty string is equal to setting it to `undef` so the DSN can be `"dbi:CSV:f_schema=;f_dir=."`.

f_lock

The `f_lock` attribute is used to set the locking mode on the opened table files. Note that not all platforms support locking. By default, tables are opened with a shared lock for reading, and with an exclusive lock for writing. The supported modes are:

```
0: No locking at all.

1: Shared locks will be used.

2: Exclusive locks will be used.
```

But see KNOWN BUGS below.

f_lockfile

If you wish to use a lockfile extension other than `.lck`, simply specify the `f_lockfile` attribute:

```
$dbh = DBI->connect ("dbi:DBM:f_lockfile=.foo");
$dbh->{f_lockfile} = ".foo";
$dbh->{dbm_tables}{qux}{f_lockfile} = ".foo";
```

If you wish to disable locking, set the `f_lockfile` to `0`.

```
$dbh = DBI->connect ("dbi:DBM:f_lockfile=0");
$dbh->{f_lockfile} = 0;
$dbh->{dbm_tables}{qux}{f_lockfile} = 0;
```

f_encoding

With this attribute, you can set the encoding in which the file is opened. This is implemented using `binmode $fh, ":encoding(<f_encoding>)"`.

f_meta

Private data area aliasing *"sql_meta" in DBI::DBD::SqlEngine* which contains information about the tables this module handles. Table meta data might not be available until the table has been accessed for the first time e.g., by issuing a select on it however it is possible to pre-initialize attributes for each table you use.

DBD::File recognizes the (public) attributes `f_ext`, `f_dir`, `f_file`, `f_encoding`, `f_lock`, `f_lockfile`, `f_schema`, in addition to the attributes *"sql_meta" in DBI::DBD::SqlEngine* already supports. Be very careful when modifying attributes you do not know, the consequence might be a destroyed or corrupted table.

`f_file` is an attribute applicable to table meta data only and you will not find a corresponding attribute in the dbh. Whilst it may be reasonable to have several tables with the same column names, it is not for the same file name. If you need access to the same file using different table names, use `SQL::Statement` as the SQL engine and the `AS` keyword:

```
SELECT * FROM tbl AS t1, tbl AS t2 WHERE t1.id = t2.id
```

`f_file` can be an absolute path name or a relative path name but if it is relative, it is interpreted as being relative to the `f_dir` attribute of the table meta data. When `f_file` is set DBD::File will use `f_file` as specified and will not attempt to work out an alternative for `f_file` using the `table name` and `f_ext` attribute.

While `f_meta` is a private and readonly attribute (which means, you cannot modify it's values), derived drivers might provide restricted write access through another attribute. Well known accessors are `csv_tables` for DBD::CSV, `ad_tables` for DBD::AnyData and `dbm_tables` for *DBD::DBM*.

New opportunities for attributes from DBI::DBD::SqlEngine

sql_table_source

`$dbh->{sql_table_source}` can be set to *DBD::File::TableSource::FileSystem* (and is the default setting of DBD::File). This provides usual behaviour of previous DBD::File releases on

```
@ary = DBI->data_sources ($driver);
@ary = DBI->data_sources ($driver, \%attr);

@ary = $dbh->data_sources ();
@ary = $dbh->data_sources (\%attr);

@names = $dbh->tables ($catalog, $schema, $table, $type);

$sth = $dbh->table_info ($catalog, $schema, $table, $type);
$sth = $dbh->table_info ($catalog, $schema, $table, $type, \%attr);

$dbh->func ("list_tables");
```

sql_data_source

`$dbh->{sql_data_source}` can be set to either *DBD::File::DataSource::File*, which is default and provides the well known behavior of DBD::File releases prior to 0.41, or *DBD::File::DataSource::Stream*, which reuses already opened file-handle for operations.

Internally private attributes to deal with SQL backends

Do not modify any of these private attributes unless you understand the implications of doing so. The behavior of DBD::File and derived DBDs might be unpredictable when one or more of those attributes are modified.

sql_nano_version

Contains the version of loaded DBI::SQL::Nano.

sql_statement_version

Contains the version of loaded SQL::Statement.

sql_handler

Contains either the text ‘SQL::Statement’ or ‘DBI::SQL::Nano’.

sql_ram_tables

Contains optionally temporary tables.

sql_flags

Contains optional flags to instantiate the SQL::Parser parsing engine when SQL::Statement is used as SQL engine. See SQL::Parser for valid flags.

Driver private methods

Default DBI methods

data_sources

The `data_sources` method returns a list of subdirectories of the current directory in the form "dbi:CSV:f_dir=$dirname".

If you want to read the subdirectories of another directory, use

```
my ($drh)  = DBI->install_driver ("CSV");
my (@list) = $drh->data_sources (f_dir => "/usr/local/csv_data");
```

Additional methods

The following methods are only available via their documented name when DBD::File is used directly. Because this is only reasonable for testing purposes, the real names must be used instead. Those names can be computed by replacing the `f_` in the method name with the driver prefix.

f_versions

Signature:

```
sub f_versions (;$) {
  my ($table_name) = @_;
  $table_name ||= ".";
  ...
  }
```

Returns the versions of the driver, including the DBI version, the Perl version, DBI::PurePerl version (if DBI::PurePerl is active) and the version of the SQL engine in use.

```
my $dbh = DBI->connect ("dbi:File:");
my $f_versions = $dbh->func ("f_versions");
print "$f_versions\n";
__END__
# DBD::File               0.41 using IO::File (1.16)
#   DBI::DBD::SqlEngine   0.05 using SQL::Statement 1.406
# DBI                     1.623
# OS                      darwin (12.2.1)
# Perl                    5.017006 (darwin-thread-multi-ld-2level)
```

Called in list context, f_versions will return an array containing each line as single entry.

Some drivers might use the optional (table name) argument and modify version information related to the table (e.g. DBD::DBM provides storage backend information for the requested table, when it has a table name).

KNOWN BUGS AND LIMITATIONS

- This module uses flock () internally but flock is not available on all platforms. On MacOS and Windows 95 there is no locking at all (perhaps not so important on MacOS and Windows 95, as there is only a single user).
- The module stores details about the handled tables in a private area of the driver handle (`$drh`). This data area is not shared between different driver instances, so several `DBI->connect ()` calls will cause different table instances and private data areas.

  This data area is filled for the first time when a table is accessed, either via an SQL statement or via `table_info` and is not destroyed until the table is dropped or the driver handle is released. Manual destruction is possible via f_clear_meta.

  The following attributes are preserved in the data area and will evaluated instead of driver globals:

  - f_ext
  - f_dir
  - f_dir_search
  - f_lock
  - f_lockfile
  - f_encoding
  - f_schema
  - col_names
  - sql_identifier_case

  The following attributes are preserved in the data area only and cannot be set globally.

  - f_file

  The following attributes are preserved in the data area only and are computed when initializing the data area:

  - f_fqfn
  - f_fqbn
  - f_fqln
  - table_name

  For DBD::CSV tables this means, once opened “foo.csv” as table named “foo”, another table named “foo” accessing the file “foo.txt” cannot be opened. Accessing “foo” will always access the file “foo.csv” in memorized `f_dir`, locking `f_lockfile` via memorized `f_lock`.

You can use f_clear_meta or the `f_file` attribute for a specific table to work around this.

- When used with SQL::Statement and temporary tables e.g.,

```
CREATE TEMP TABLE ...
```

the table data processing bypasses DBD::File::Table. No file system calls will be made and there are no clashes with existing (file based) tables with the same name. Temporary tables are chosen over file tables, but they will not covered by `table_info`.

AUTHOR

This module is currently maintained by

H.Merijn Brand < perl5 at tux.freedom.nl > and Jens Rehsack < rehsack at google-mail.com >

The original author is Jochen Wiedmann.

COPYRIGHT AND LICENSE


```
Copyright (C) 2009-2026 by H.Merijn Brand & Jens Rehsack
Copyright (C) 2004-2009 by Jeff Zucker
Copyright (C) 1998-2004 by Jochen Wiedmann
```

All rights reserved.

You may freely distribute and/or modify this module under the terms of either the GNU General Public License (GPL) or the Artistic License, as specified in the Perl README file.

SEE ALSO

DBI, *DBD::DBM*, DBD::CSV, *Text::CSV*, Text::CSV_XS, SQL::Statement, and *DBI::SQL::Nano*

DBD::Gofer

A stateless-proxy driver for communicating with a remote DBI

SYNOPSIS

```
use DBI;

$original_dsn = "dbi:..."; # your original DBI Data Source Name

$dbh = DBI->connect("dbi:Gofer:transport=$transport;...;dsn=
 ↪$original_dsn",
                    $user, $passwd, \%attributes);

... use $dbh as if it was connected to $original_dsn ...
```

The `transport=$transport` part specifies the name of the module to use to transport the requests to the remote DBI. If $transport doesn't contain any double colons then it's prefixed with `DBD::Gofer::Transport::`.

The `dsn=$original_dsn` part *must be the last element* of the DSN because everything after `dsn=` is assumed to be the DSN that the remote DBI should use.

The `...` represents attributes that influence the operation of the Gofer driver or transport. These are described below or in the documentation of the transport module being used.

DESCRIPTION

DBD::Gofer is a DBI database driver that forwards requests to another DBI driver, usually in a separate process, often on a separate machine. It tries to be as transparent as possible so it appears that you are using the remote driver directly.

DBD::Gofer is very similar to DBD::Proxy. The major difference is that with DBD::Gofer no state is maintained on the remote end. That means every request contains all the information needed to create the required state. (So, for example, every request includes the DSN to connect to.) Each request can be sent to any available server. The server executes the request and returns a single response that includes all the data.

This is very similar to the way http works as a stateless protocol for the web. Each request from your web browser can be handled by a different web server process.

Use Cases

This may seem like pointless overhead but there are situations where this is a very good thing. Let's consider a specific case.

Imagine using DBD::Gofer with an http transport. Your application calls connect(), prepare("select * from table where foo=?"), bind_param(), and execute(). At this point DBD::Gofer builds a request containing all the information about the method calls. It then uses the httpd transport to send that request to an apache web server.

This 'dbi execute' web server executes the request (using DBI::Gofer::Execute and related modules) and builds a response that contains all the rows of data, if the statement returned any, along with all the attributes that describe the results, such as $sth->{NAME}. This response is sent back to DBD::Gofer which unpacks it and presents it to the application as if it had executed the statement itself.

Advantages

Okay, but you still don't see the point? Well let's consider what we've gained:

Connection Pooling and Throttling

The 'dbi execute' web server leverages all the functionality of web infrastructure in terms of load balancing, high-availability, firewalls, access management, proxying, caching.

At its most basic level you get a configurable pool of persistent database connections.

Simple Scaling

Got thousands of processes all trying to connect to the database? You can use DBD::Gofer to connect them to your smaller pool of 'dbi execute' web servers instead.

Caching

Client-side caching is as simple as adding "`cache=1`" to the DSN. This feature alone can be worth using DBD::Gofer for.

Fewer Network Round-trips

DBD::Gofer sends as few requests as possible (dependent on the policy being used).

Thin Clients / Unsupported Platforms

You no longer need drivers for your database on every system. DBD::Gofer is pure perl.

CONSTRAINTS

There are some natural constraints imposed by the DBD::Gofer 'stateless' approach. But not many:

You can't change database handle attributes after connect()

You can't change database handle attributes after you've connected. Use the connect() call to specify all the attribute settings you want.

This is because it's critical that when a request is complete the database handle is left in the same state it was when first connected.

An exception is made for attributes with names starting "`private_`": They can be set after connect() but the change is only applied locally.

You can't change statement handle attributes after prepare()

You can't change statement handle attributes after prepare.

An exception is made for attributes with names starting "`private_`": They can be set after prepare() but the change is only applied locally.

You can't use transactions

AutoCommit only. Transactions aren't supported.

(In theory transactions could be supported when using a transport that maintains a connection, like `stream` does. If you're interested in this please get in touch via dbi-dev@perl.org)

You can't call driver-private sth methods

But that's rarely needed anyway.

GENERAL CAVEATS

A few important things to keep in mind when using DBD::Gofer:

Temporary tables, locks, and other per-connection persistent state

You shouldn't expect any per-session state to persist between requests. This includes locks and temporary tables.

Because the server-side may execute your requests via a different database connections, you can't rely on any per-connection persistent state, such as temporary tables, being available from one request to the next.

This is an easy trap to fall into. A good way to check for this is to test your code with a Gofer policy package that sets the `connect_method` policy to 'connect' to force a new connection for each request. The `pedantic` policy does this.

Driver-private Database Handle Attributes

Some driver-private dbh attributes may not be available if the driver has not implemented the private_attribute_info() method (added in DBI 1.54).

Driver-private Statement Handle Attributes

Driver-private sth attributes can be set in the prepare() call. TODO

Some driver-private sth attributes may not be available if the driver has not implemented the private_attribute_info() method (added in DBI 1.54).

Multiple Resultsets

Multiple resultsets are supported only if the driver supports the more_results() method (an exception is made for DBD::Sybase).

Statement activity that also updates dbh attributes

Some drivers may update one or more dbh attributes after performing activity on a child sth. For example, DBD::mysql provides $dbh->{mysql_insertid} in addition to $sth->{mysql_insertid}. Currently mysql_insertid is supported via a hack but a more general mechanism is needed for other drivers to use.

Methods that report an error always return undef

With DBD::Gofer, a method that sets an error always return an undef or empty list. That shouldn't be a problem in practice because the DBI doesn't define any methods that return meaningful values while also reporting an error.

Subclassing only applies to client-side

The RootClass and DbTypeSubclass attributes are not passed to the Gofer server.

CAVEATS FOR SPECIFIC METHODS

last_insert_id

To enable use of last_insert_id you need to indicate to DBD::Gofer that you'd like to use it. You do that my adding a `go_last_insert_id_args` attribute to the do() or prepare() method calls. For example:

```
$dbh->do($sql, { go_last_insert_id_args => [...] });
```

or

```
$sth = $dbh->prepare($sql, { go_last_insert_id_args => [...] });
```

The array reference should contains the args that you want passed to the last_insert_id() method.

execute_for_fetch

The array methods bind_param_array() and execute_array() are supported. When execute_array() is called the data is serialized and executed in a single round-trip to the Gofer server. This makes it very fast, but requires enough memory to store all the serialized data.

The execute_for_fetch() method currently isn't optimised, it uses the DBI fallback behaviour of executing each tuple individually. (It could be implemented as a wrapper for execute_array() - patches welcome.)

TRANSPORTS

DBD::Gofer doesn't concern itself with transporting requests and responses to and fro. For that it uses special Gofer transport modules.

Gofer transport modules usually come in pairs: one for the 'client' DBD::Gofer driver to use and one for the remote 'server' end. They have very similar names:

```
DBD::Gofer::Transport::<foo>
DBI::Gofer::Transport::<foo>
```

Sometimes the transports on the DBD and DBI sides may have different names. For example DBD::Gofer::Transport::http is typically used with DBI::Gofer::Transport::mod_perl (DBD::Gofer::Transport::http and DBI::Gofer::Transport::mod_perl modules are part of the GoferTransport-http distribution).

Bundled Transports

Several transport modules are provided with DBD::Gofer:

null

The null transport is the simplest of them all. It doesn't actually transport the request anywhere. It just serializes (freezes) the request into a string, then thaws it back into a data structure before passing it to DBI::Gofer::Execute to execute. The same freeze and thaw is applied to the results.

The null transport is the best way to test if your application will work with Gofer. Just set the DBI_AUTOPROXY environment variable to "`dbi:Gofer:transport=null;policy=pedantic`" (see "Using DBI_AUTOPROXY" below) and run your application, or ideally its test suite, as usual.

It doesn't take any parameters.

pipeone

The pipeone transport launches a subprocess for each request. It passes in the request and reads the response.

The fact that a new subprocess is started for each request ensures that the server side is truly stateless. While this does make the transport *very* slow, it is useful as a way to test that your application doesn't depend on per-connection state, such as temporary tables, persisting between requests.

It's also useful both as a proof of concept and as a base class for the stream driver.

stream

The stream driver also launches a subprocess and writes requests and reads responses, like the pipeone transport. In this case, however, the subprocess is expected to handle more that one request. (Though it will be automatically restarted if it exits.)

This is the first transport that is truly useful because it can launch the subprocess on a remote machine using `ssh`. This means you can now use DBD::Gofer to easily access any databases that's accessible from any system you can login to. You also get all the benefits of ssh, including encryption and optional compression.

See "Using DBI_AUTOPROXY" below for an example.

Other Transports

Implementing a Gofer transport is *very* simple, and more transports are very welcome. Just take a look at any existing transports that are similar to your needs.

http

See the GoferTransport-http distribution on CPAN: http://search.cpan.org/dist/GoferTransport-http/

Gearman

I know Ask Bjørn Hansen has implemented a transport for the `gearman` distributed job system, though it's not on CPAN at the time of writing this.

CONNECTING

Simply prefix your existing DSN with "`dbi:Gofer:transport=$transport;dsn=`" where $transport is the name of the Gofer transport you want to use (see "TRANSPORTS"). The `transport` and `dsn` attributes must be specified and the `dsn` attributes must be last.

Other attributes can be specified in the DSN to configure DBD::Gofer and/or the Gofer transport module being used. The main attributes after `transport`, are `url` and `policy`. These and other attributes are described below.

Using DBI_AUTOPROXY

The simplest way to try out DBD::Gofer is to set the DBI_AUTOPROXY environment variable. In this case you don't include the `dsn=` part. For example:

```
export DBI_AUTOPROXY="dbi:Gofer:transport=null"
```

or, for a more useful example, try:

```
export DBI_AUTOPROXY="dbi:Gofer:transport=stream;
↪url=ssh:user@example.com"
```

Connection Attributes

These attributes can be specified in the DSN. They can also be passed in the \%attr parameter of the DBI connect method by adding a "go_" prefix to the name.

transport

Specifies the Gofer transport class to use. Required. See "TRANSPORTS" above.

If the value does not include `::` then "`DBD::Gofer::Transport::`" is prefixed.

The transport object can be accessed via $h->{go_transport}.

dsn

Specifies the DSN for the remote side to connect to. Required, and must be last.

url

Used to tell the transport where to connect to. The exact form of the value depends on the transport used.

policy

Specifies the policy to use. See "CONFIGURING BEHAVIOUR POLICY".

If the value does not include `::` then "`DBD::Gofer::Policy`" is prefixed.

The policy object can be accessed via $h->{go_policy}.

timeout

Specifies a timeout, in seconds, to use when waiting for responses from the server side.

retry_limit

Specifies the number of times a failed request will be retried. Default is 0.

retry_hook

Specifies a code reference to be called to decide if a failed request should be retried. The code reference is called like this:

```
$transport = $h->{go_transport};
$retry = $transport->go_retry_hook->($request, $response,
 ↪$transport);
```

If it returns true then the request will be retried, up to the `retry_limit`. If it returns a false but defined value then the request will not be retried. If it returns undef then the default behaviour will be used, as if `retry_hook` had not been specified.

The default behaviour is to retry requests where $request->is_idempotent is true, or the error message matches `/induced by DBI_GOFER_RANDOM/`.

cache

Specifies that client-side caching should be performed. The value is the name of a cache class to use.

Any class implementing get($key) and set($key, $value) methods can be used. That includes a great many powerful caching classes on CPAN, including the Cache and Cache::Cache distributions.

You can use "`cache=1`" is a shortcut for "`cache=DBI::Util::CacheMemory`". See *DBI::Util::CacheMemory* for a description of this simple fast default cache.

The cache object can be accessed via $h->go_cache. For example:

```
$dbh->go_cache->clear; # free up memory being used by the cache
```

The cache keys are the frozen (serialized) requests, and the values are the frozen responses.

The default behaviour is to only use the cache for requests where $request->is_idempotent is true (i.e., the dbh has the ReadOnly attribute set or the SQL statement is obviously a SELECT without a FOR UPDATE clause.)

For even more control you can use the `go_cache` attribute to pass in an instantiated cache object. Individual methods, including prepare(), can also specify alternative caches via the `go_cache` attribute. For example, to specify no caching for a particular query, you could use

```
$sth = $dbh->prepare( $sql, { go_cache => 0 } );
```

This can be used to implement different caching policies for different statements.

It's interesting to note that DBD::Gofer can be used to add client-side caching to any (gofer compatible) application, with no code changes and no need for a gofer server. Just set the DBI_AUTOPROXY environment variable like this:

```
DBI_AUTOPROXY='dbi:Gofer:transport=null;cache=1'
```

CONFIGURING BEHAVIOUR POLICY

DBD::Gofer supports a 'policy' mechanism that allows you to fine-tune the number of round-trips to the Gofer server. The policies are grouped into classes (which may be subclassed) and referenced by the name of the class.

The *DBD::Gofer::Policy::Base* class is the base class for all the policy packages and describes all the available policies.

Three policy packages are supplied with DBD::Gofer:

DBD::Gofer::Policy::pedantic is most 'transparent' but slowest because it makes more round-trips to the Gofer server.

DBD::Gofer::Policy::classic is a reasonable compromise - it's the default policy.

DBD::Gofer::Policy::rush is fastest, but may require code changes in your applications.

Generally the default `classic` policy is fine. When first testing an existing application with Gofer it is a good idea to start with the `pedantic` policy first and then switch to `classic` or a custom policy, for final testing.

AUTHOR

Tim Bunce, http://www.tim.bunce.name

LICENCE AND COPYRIGHT

Copyright (c) 2007, Tim Bunce, Ireland. All rights reserved.

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself. See perlartistic.

ACKNOWLEDGEMENTS

The development of DBD::Gofer and related modules was sponsored by Shopzilla.com (http://Shopzilla.com), where I currently work.

SEE ALSO

DBI::Gofer::Request, *DBI::Gofer::Response*, *DBI::Gofer::Execute*.

DBI::Gofer::Transport::Base, *DBD::Gofer::Policy::Base*.

DBI

Caveats for specific drivers

This section aims to record issues to be aware of when using Gofer with specific drivers. It usually only documents issues that are not natural consequences of the limitations of the Gofer approach - as documented above.

TODO

This is just a random brain dump... (There's more in the source of the Changes file, not the pod)

Document policy mechanism

Add mechanism for transports to list config params and for Gofer to apply any that match (and warn if any left over?)

Driver-private sth attributes - set via prepare() - change DBI spec

add hooks into transport base class for checking & updating a result set cache ie via a standard cache interface such as: http://search.cpan.org/~robm/Cache-FastMmap/FastMmap.pm http://search.cpan.org/~bradfitz/Cache-Memcached/lib/Cache/Memcached.pm http://search.cpan.org/~dclinton/Cache-Cache/ http://search.cpan.org/~cleishman/Cache/ Also caching instructions could be passed through the httpd transport layer in such a way that appropriate http cache headers are added to the results so that web caches (squid etc) could be used to implement the caching. (MUST require the use of GET rather than POST requests.)

Rework handling of installed_methods to not piggyback on dbh_attributes?

Perhaps support transactions for transports where it's possible (ie null and stream)? Would make stream transport (ie ssh) more useful to more people.

Make sth_result_attr more like dbh_attributes (using '*' etc)

Add @val = FETCH_many(@names) to DBI in C and use in Gofer/Execute?

Implement _new_sth in C.

Modules

- `DBD::Gofer::Policy`
- `DBD::Gofer::Transport`

DBD::Gofer::Policy

Namespace containing the following modules:

- `DBD::Gofer::Policy::Base` - Base class for DBD::Gofer policies
- `DBD::Gofer::Policy::classic` - The 'classic' policy for DBD::Gofer
- `DBD::Gofer::Policy::pedantic` - The 'pedantic' policy for DBD::Gofer
- `DBD::Gofer::Policy::rush` - The 'rush' policy for DBD::Gofer

DBD::Gofer::Policy::Base

Base class for DBD::Gofer policies

SYNOPSIS

```
$dbh = DBI->connect("dbi:Gofer:transport=...;policy=...", ...)
```

DESCRIPTION

DBD::Gofer can be configured via a 'policy' mechanism that allows you to fine-tune the number of round-trips to the Gofer server. The policies are grouped into classes (which may be subclassed) and referenced by the name of the class.

The *DBD::Gofer::Policy::Base* class is the base class for all the policy classes and describes all the individual policy items.

The Base policy is not used directly. You should use a policy class derived from it.

POLICY CLASSES

Three policy classes are supplied with DBD::Gofer:

DBD::Gofer::Policy::pedantic is most 'transparent' but slowest because it makes more round-trips to the Gofer server.

DBD::Gofer::Policy::classic is a reasonable compromise - it's the default policy.

DBD::Gofer::Policy::rush is fastest, but may require code changes in your applications.

Generally the default `classic` policy is fine. When first testing an existing application with Gofer it is a good idea to start with the `pedantic` policy first and then switch to `classic` or a custom policy, for final testing.

POLICY ITEMS

These are temporary docs: See the source code for list of policies and their defaults.

In a future version the policies and their defaults will be defined in the pod and parsed out at load-time.

See the source code to this module for more details.

POLICY CUSTOMIZATION

XXX This area of DBD::Gofer is subject to change.

There are three ways to customize policies:

Policy classes are designed to influence the overall behaviour of DBD::Gofer with existing, unaltered programs, so they work in a reasonably optimal way without requiring code changes. You can implement new policy classes as subclasses of existing policies.

In many cases individual policy items can be overridden on a case-by-case basis within your application code. You do this by passing a corresponding `<go_<policy_name>>` attribute into DBI methods by your application code. This lets you fine-tune the behaviour for special cases.

The policy items are implemented as methods. In many cases the methods are passed parameters relating to the DBD::Gofer code being executed. This means the policy can implement dynamic behaviour that varies depending on the particular circumstances, such as the particular statement being executed.

AUTHOR

Tim Bunce, http://www.tim.bunce.name

LICENCE AND COPYRIGHT

Copyright (c) 2007, Tim Bunce, Ireland. All rights reserved.

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself. See perlartistic.

DBD::Gofer::Policy::classic

The 'classic' policy for DBD::Gofer

SYNOPSIS

```
$dbh = DBI->connect("dbi:Gofer:transport=...;policy=classic", ...)
```

The `classic` policy is the default DBD::Gofer policy, so need not be included in the DSN.

DESCRIPTION

Temporary docs: See the source code for list of policies and their defaults.

In a future version the policies and their defaults will be defined in the pod and parsed out at load-time.

AUTHOR

Tim Bunce, http://www.tim.bunce.name

LICENCE AND COPYRIGHT

Copyright (c) 2007, Tim Bunce, Ireland. All rights reserved.

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself. See perlartistic.

DBD::Gofer::Policy::pedantic

The 'pedantic' policy for DBD::Gofer

SYNOPSIS

```
$dbh = DBI->connect("dbi:Gofer:transport=...;policy=pedantic", ...)
```

DESCRIPTION

The `pedantic` policy tries to be as transparent as possible. To do this it makes round-trips to the server for almost every DBI method call.

This is the best policy to use when first testing existing code with Gofer. Once it's working well you should consider moving to the `classic` policy or defining your own policy class.

Temporary docs: See the source code for list of policies and their defaults.

In a future version the policies and their defaults will be defined in the pod and parsed out at load-time.

AUTHOR

Tim Bunce, http://www.tim.bunce.name

LICENCE AND COPYRIGHT

Copyright (c) 2007, Tim Bunce, Ireland. All rights reserved.

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself. See perlartistic.

DBD::Gofer::Policy::rush

The 'rush' policy for DBD::Gofer

SYNOPSIS

```
$dbh = DBI->connect("dbi:Gofer:transport=...;policy=rush", ...)
```

DESCRIPTION

The `rush` policy tries to make as few round-trips as possible. It's the opposite end of the policy spectrum to the `pedantic` policy.

Temporary docs: See the source code for list of policies and their defaults.

In a future version the policies and their defaults will be defined in the pod and parsed out at load-time.

AUTHOR

Tim Bunce, http://www.tim.bunce.name

LICENCE AND COPYRIGHT

Copyright (c) 2007, Tim Bunce, Ireland. All rights reserved.

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself. See perlartistic.

DBD::Gofer::Transport

Namespace containing the following modules:

- `DBD::Gofer::Transport::Base` - base class for DBD::Gofer client transports
- `DBD::Gofer::Transport::corostream` - Async DBD::Gofer stream transport using Coro and AnyEvent
- `DBD::Gofer::Transport::null` - DBD::Gofer client transport for testing
- `DBD::Gofer::Transport::pipeone` - DBD::Gofer client transport for testing
- `DBD::Gofer::Transport::stream` - DBD::Gofer transport for stdio streaming

DBD::Gofer::Transport::Base

base class for DBD::Gofer client transports

SYNOPSIS

```
my $remote_dsn = "..."
DBI->connect("dbi:Gofer:transport=...;url=...;timeout=...;retry_
 ↪limit=...;dsn=$remote_dsn",...)
```

or, enable by setting the DBI_AUTOPROXY environment variable:

```
export DBI_AUTOPROXY='dbi:Gofer:transport=...;url=...'
```

which will force *all* DBI connections to be made via that Gofer server.

DESCRIPTION

This is the base class for all DBD::Gofer client transports.

ATTRIBUTES

Gofer transport attributes can be specified either in the attributes parameter of the connect() method call, or in the DSN string. When used in the DSN string, attribute names don't have the go_ prefix.

go_dsn

The full DBI DSN that the Gofer server should connect to on your behalf.

When used in the DSN it must be the last element in the DSN string.

go_timeout

A time limit for sending a request and receiving a response. Some drivers may implement sending and receiving as separate steps, in which case (currently) the timeout applies to each separately.

If a request needs to be resent then the timeout is restarted for each sending of a request and receiving of a response.

go_retry_limit

The maximum number of times an request may be retried. The default is 2.

go_retry_hook

This subroutine reference is called, if defined, for each response received where $response->err is true.

The subroutine is pass three parameters: the request object, the response object, and the transport object.

If it returns an undefined value then the default retry behaviour is used. See "RETRY ON ERROR" below.

If it returns a defined but false value then the request is not resent.

If it returns true value then the request is resent, so long as the number of retries does not exceed `go_retry_limit`.

RETRY ON ERROR

The default retry on error behaviour is:

```
- Retry if the error was due to DBI_GOFER_RANDOM. See L
↪<DBI::Gofer::Execute>.
```

(continues on next page)

(continued from previous page)

```
- Retry if $request->is_idempotent returns true. See L
↪<DBI::Gofer::Request>.
```

A retry won't be allowed if the number of previous retries has reached `go_retry_limit`.

TRACING

Tracing of gofer requests and responses can be enabled by setting the `DBD_GOFER_TRACE` environment variable. A value of 1 gives a reasonably compact summary of each request and response. A value of 2 or more gives a detailed, and voluminous, dump.

The trace is written using DBI->trace_msg() and so is written to the default DBI trace output, which is usually STDERR.

METHODS

This section is currently far from complete.

response_retry_preference

```
$retry = $transport->response_retry_preference($request, $response);
```

The response_retry_preference is called by DBD::Gofer when considering if a request should be retried after an error.

Returns true (would like to retry), false (must not retry), undef (no preference).

If a true value is returned in the form of a CODE ref then, if DBD::Gofer does decide to retry the request, it calls the code ref passing $retry_count, $retry_limit. Can be used for logging and/or to implement exponential back-off behaviour. Currently the called code must return using `return;` to allow for future extensions.

AUTHOR

Tim Bunce, http://www.tim.bunce.name

LICENCE AND COPYRIGHT

Copyright (c) 2007-2008, Tim Bunce, Ireland. All rights reserved.

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself. See perlartistic.

SEE ALSO

DBD::Gofer, *DBI::Gofer::Request*, *DBI::Gofer::Response*, *DBI::Gofer::Execute*.

and some example transports:

DBD::Gofer::Transport::stream

DBD::Gofer::Transport::http

DBI::Gofer::Transport::mod_perl

DBD::Gofer::Transport::corostream

Async DBD::Gofer stream transport using Coro and AnyEvent

SYNOPSIS

```
DBI_AUTOPROXY="dbi:Gofer:transport=corostream" perl some-perl-
↪script-using-dbi.pl
```

or

```
$dsn = ...; # the DSN for the driver and database you want to use
$dbh = DBI->connect("dbi:Gofer:transport=corostream;dsn=$dsn", ...);
```

DESCRIPTION

The *BIG WIN* from using *Coro* is that it enables the use of existing DBI frameworks like DBIx::Class.

KNOWN ISSUES AND LIMITATIONS

```
- Uses Coro::Select so alters CORE::select globally
  Parent class probably needs refactoring to enable a more␣
↪encapsulated approach.

- Doesn't prevent multiple concurrent requests
  Probably just needs a per-connection semaphore

- Coro has many caveats. Caveat emptor.
```

STATUS

THIS IS CURRENTLY JUST A PROOF-OF-CONCEPT IMPLEMENTATION FOR EXPERIMENTATION.

Please note that I have no plans to develop this code further myself. I'd very much welcome contributions. Interested? Let me know!

AUTHOR

Tim Bunce, http://www.tim.bunce.name

LICENCE AND COPYRIGHT

Copyright (c) 2010, Tim Bunce, Ireland. All rights reserved.

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself. See perlartistic.

SEE ALSO

DBD::Gofer::Transport::stream

DBD::Gofer

APPENDIX

Example code:

```
#!perl

use strict;
use warnings;
use Time::HiRes qw(time);

BEGIN { $ENV{PERL_ANYEVENT_STRICT} = 1; $ENV{PERL_ANYEVENT_VERBOSE}
 ↪= 1; }

use AnyEvent;

BEGIN { $ENV{DBI_TRACE} = 0; $ENV{DBI_GOFER_TRACE} = 0; $ENV{DBD_
 ↪GOFER_TRACE} = 0; };

use DBI;

$ENV{DBI_AUTOPROXY} = 'dbi:Gofer:transport=corostream';

my $ticker = AnyEvent->timer( after => 0, interval => 0.1, cb => sub
 ↪{
    warn sprintf "-tick- %.2f\n", time
} );

warn "connecting...\n";
my $dbh = DBI->connect("dbi:NullP:");
warn "...connected\n";

for (1..3) {
    warn "entering DBI...\n";
    $dbh->do("sleep 0.3"); # pseudo-sql understood by the DBD::NullP
 ↪driver
    warn "...returned\n";
}

warn "done.";
```

Example output:

```
$ perl corogofer.pl
connecting...
-tick- 1293631437.14
-tick- 1293631437.14
...connected
entering DBI...
-tick- 1293631437.25
-tick- 1293631437.35
-tick- 1293631437.45
-tick- 1293631437.55
...returned
entering DBI...
-tick- 1293631437.66
-tick- 1293631437.76
-tick- 1293631437.86
...returned
entering DBI...
-tick- 1293631437.96
-tick- 1293631438.06
-tick- 1293631438.16
...returned
done. at corogofer.pl line 39.
```

You can see that the timer callback is firing while the code 'waits' inside the do() method for the response from the database. Normally that would block.

DBD::Gofer::Transport::null

DBD::Gofer client transport for testing

SYNOPSIS

```
my $original_dsn = "..."
DBI->connect("dbi:Gofer:transport=null;dsn=$original_dsn",...)
```

or, enable by setting the DBI_AUTOPROXY environment variable:

```
export DBI_AUTOPROXY="dbi:Gofer:transport=null"
```

DESCRIPTION

Connect via DBD::Gofer but execute the requests within the same process.

This is a quick and simple way to test applications for compatibility with the (few) restrictions that DBD::Gofer imposes.

It also provides a simple, portable way for the DBI test suite to be used to test DBD::Gofer on all platforms with no setup.

Also, by measuring the difference in performance between normal connections and connections via `dbi:Gofer:transport=null` the basic cost of using DBD::Gofer can be measured. Furthermore, the additional cost of more advanced transports can be isolated by comparing their performance with the null transport.

The `t/85gofer.t` script in the DBI distribution includes a comparative benchmark.

AUTHOR

Tim Bunce, http://www.tim.bunce.name

LICENCE AND COPYRIGHT

Copyright (c) 2007, Tim Bunce, Ireland. All rights reserved.

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself. See perlartistic.

SEE ALSO

DBD::Gofer::Transport::Base

DBD::Gofer

DBD::Gofer::Transport::pipeone

DBD::Gofer client transport for testing

SYNOPSIS

```
$original_dsn = "...";
DBI->connect("dbi:Gofer:transport=pipeone;dsn=$original_dsn",...)
```

or, enable by setting the DBI_AUTOPROXY environment variable:

```
export DBI_AUTOPROXY="dbi:Gofer:transport=pipeone"
```

DESCRIPTION

Connect via DBD::Gofer and execute each request by starting executing a subprocess.

This is, as you might imagine, spectacularly inefficient!

It's only intended for testing. Specifically it demonstrates that the server side is completely stateless.

It also provides a base class for the much more useful *DBD::Gofer::Transport::stream* transport.

AUTHOR

Tim Bunce, http://www.tim.bunce.name

LICENCE AND COPYRIGHT

Copyright (c) 2007, Tim Bunce, Ireland. All rights reserved.

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself. See perlartistic.

SEE ALSO

DBD::Gofer::Transport::Base

DBD::Gofer

DBD::Gofer::Transport::stream

DBD::Gofer transport for stdio streaming

SYNOPSIS

```
DBI->connect('dbi:Gofer:transport=stream;url=ssh:username@host.
↪example.com;dsn=dbi:...',...)
```

or, enable by setting the DBI_AUTOPROXY environment variable:

```
export DBI_AUTOPROXY='dbi:Gofer:transport=stream;
↪url=ssh:username@host.example.com'
```

DESCRIPTION

Without the `url=` parameter it launches a subprocess as

```
perl -MDBI::Gofer::Transport::stream -e run_stdio_hex
```

and feeds requests into it and reads responses from it. But that's not very useful.

With a `url=ssh:username@host.example.com` parameter it uses ssh to launch the subprocess on a remote system. That's much more useful!

It gives you secure remote access to DBI databases on any system you can login to. Using ssh also gives you optional compression and many other features (see the ssh manual for how to configure that and many other options via ~/.ssh/config file).

The actual command invoked is something like:

```
ssh -xq ssh:username@host.example.com bash -c $setup $run
```

where $run is the command shown above, and $command is

```
. .bash_profile 2>/dev/null || . .bash_login 2>/dev/null || . .
↪profile 2>/dev/null; exec "$@"
```

which is trying (in a limited and fairly unportable way) to setup the environment (PATH, PERL5LIB etc) as it would be if you had logged in to that system.

The "`perl`" used in the command will default to the value of $^X when not using ssh. On most systems that's the full path to the perl that's currently executing.

PERSISTENCE

Currently gofer stream connections persist (remain connected) after all database handles have been disconnected. This makes later connections in the same process very fast.

Currently up to 5 different gofer stream connections (based on url) can persist. If more than 5 are in the cache when a new connection is made then the cache is cleared before adding the new connection. Simple but effective.

TO DO

Document go_perl attribute

Automatically reconnect (within reason) if there's a transport error.

Decide on default for persistent connection - on or off? limits? ttl?

AUTHOR

Tim Bunce, http://www.tim.bunce.name

LICENCE AND COPYRIGHT

Copyright (c) 2007, Tim Bunce, Ireland. All rights reserved.

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself. See perlartistic.

SEE ALSO

DBD::Gofer::Transport::Base

DBD::Gofer

DBD::Mem

a DBI driver for Mem & MLMem files

SYNOPSIS

```
use DBI;
$dbh = DBI->connect('dbi:Mem:', undef, undef, {});
$dbh = DBI->connect('dbi:Mem:', undef, undef, {RaiseError => 1});

# or
$dbh = DBI->connect('dbi:Mem:');
$dbh = DBI->connect('DBI:Mem(RaiseError=1):');
```

and other variations on connect() as shown in the *DBI* docs and <DBI::DBD::SqlEngine metadata|DBI::DBD::SqlEngine/Metadata>.

Use standard DBI prepare, execute, fetch, placeholders, etc.

DESCRIPTION

DBD::Mem is a database management system that works right out of the box. If you have a standard installation of Perl and DBI you can begin creating, accessing, and modifying simple database tables without any further modules. You can add other modules (e.g., SQL::Statement) for improved functionality.

DBD::Mem doesn't store any data persistently - all data has the lifetime of the instantiated $dbh. The main reason to use DBD::Mem is to use extended features of SQL::Statement where temporary tables are required. One can use DBD::Mem to simulate `VIEWS` or sub-queries.

Bundling `DBD::Mem` with *DBI* will allow us further compatibility checks of *DBI::DBD::SqlEngine* beyond the capabilities of *DBD::File* and *DBD::DBM*. This will ensure DBI provided basis for drivers like DBD::AnyData2 or DBD::Amazon are better prepared and tested for not-file based backends.

Metadata

There're no new meta data introduced by `DBD::Mem`. See *"Metadata" in DBI::DBD::SqlEngine* for full description.

GETTING HELP, MAKING SUGGESTIONS, AND REPORTING BUGS

If you need help installing or using DBD::Mem, please write to the DBI users mailing list at mailto:dbi-users@perl.org or to the comp.lang.perl.modules newsgroup on usenet. I cannot always answer every question quickly but there are many on the mailing list or in the newsgroup who can.

DBD developers for DBD's which rely on DBI::DBD::SqlEngine or DBD::Mem or use one of them as an example are suggested to join the DBI developers mailing list at mailto:dbi-dev@perl.org and strongly encouraged to join our IRC channel at irc://irc.perl.org/dbi.

If you have suggestions, ideas for improvements, or bugs to report, please report a bug as described in DBI. Do not mail any of the authors directly, you might not get an answer.

When reporting bugs, please send the output of `$dbh->mem_versions($table)` for a

table that exhibits the bug and as small a sample as you can make of the code that produces the bug. And of course, patches are welcome, too :-).

If you need enhancements quickly, you can get commercial support as described at http://dbi.perl.org/support/ or you can contact Jens Rehsack at rehsack@googlemail.com for commercial support.

AUTHOR AND COPYRIGHT

This module is written by Jens Rehsack < rehsack AT googlemail.com >.

```
Copyright (c) 2016- by Jens Rehsack, all rights reserved.
```

You may freely distribute and/or modify this module under the terms of either the GNU General Public License (GPL) or the Artistic License, as specified in the Perl README file.

SEE ALSO

DBI for the Database interface of the Perl Programming Language.

SQL::Statement and *DBI::SQL::Nano* for the available SQL engines.

SQL::Statement::RAM where the implementation is shamelessly stolen from to allow DBI bundled Pure-Perl drivers increase the test coverage.

DBD::SQLite using `dbname=:memory:` for an incredible fast in-memory database engine.

DBD::Proxy

A proxy driver for the DBI

SYNOPSIS

```
use DBI;

$dbh = DBI->connect("dbi:Proxy:hostname=$host;port=$port;dsn=$db",
                    $user, $passwd);

# See the DBI module documentation for full details
```

DESCRIPTION

DBD::Proxy is a Perl module for connecting to a database via a remote DBI driver. See *DBD::Gofer* for an alternative with different trade-offs.

This is of course not needed for DBI drivers which already support connecting to a remote database, but there are engines which don't offer network connectivity.

Another application is offering database access through a firewall, as the driver offers query based restrictions. For example you can restrict queries to exactly those that are used in a given CGI application.

Speaking of CGI, another application is (or rather, will be) to reduce the database connect/disconnect overhead from CGI scripts by using proxying the connect_cached

method. The proxy server will hold the database connections open in a cache. The CGI script then trades the database connect/disconnect overhead for the DBD::Proxy connect/disconnect overhead which is typically much less.

CONNECTING TO THE DATABASE

Before connecting to a remote database, you must ensure, that a Proxy server is running on the remote machine. There's no default port, so you have to ask your system administrator for the port number. See *DBI::ProxyServer* for details.

Say, your Proxy server is running on machine "alpha", port 3334, and you'd like to connect to an ODBC database called "mydb" as user "joe" with password "hello". When using DBD::ODBC directly, you'd do a

```
$dbh = DBI->connect("DBI:ODBC:mydb", "joe", "hello");
```

With DBD::Proxy this becomes

```
$dsn = "DBI:Proxy:hostname=alpha;port=3334;dsn=DBI:ODBC:mydb";
$dbh = DBI->connect($dsn, "joe", "hello");
```

You see, this is mainly the same. The DBD::Proxy module will create a connection to the Proxy server on "alpha" which in turn will connect to the ODBC database.

Refer to the *DBI* documentation on the `connect` method for a way to automatically use DBD::Proxy without having to change your code.

DBD::Proxy's DSN string has the format

```
$dsn = "DBI:Proxy:key1=val1; ... ;keyN=valN;dsn=valDSN";
```

In other words, it is a collection of key/value pairs. The following keys are recognized:

- hostname
- port

  Hostname and port of the Proxy server; these keys must be present, no defaults. Example:

  ```
  hostname=alpha;port=3334
  ```

- dsn

  The value of this attribute will be used as a dsn name by the Proxy server. Thus it must have the format `DBI:driver:...`, in particular it will contain colons. The *dsn* value may contain semicolons, hence this key *must* be the last and it's value will be the complete remaining part of the dsn. Example:

  ```
  dsn=DBI:ODBC:mydb
  ```

- cipher
- key
- usercipher

- userkey

  By using these fields you can enable encryption. If you set, for example,

```
cipher=$class;key=$key
```

  (note the semicolon) then DBD::Proxy will create a new cipher object by executing

```
$cipherRef = $class->new(pack("H*", $key));
```

  and pass this object to the RPC::PlClient module when creating a client. See RPC::PlClient. Example:

```
cipher=IDEA;key=97cd2375efa329aceef2098babdc9721
```

  The usercipher/userkey attributes allow you to use two phase encryption: The cipher/key encryption will be used in the login and authorisation phase. Once the client is authorised, he will change to usercipher/userkey encryption. Thus the cipher/key pair is a **host** based secret, typically less secure than the usercipher/userkey secret and readable by anyone. The usercipher/userkey secret is **your** private secret.

  Of course encryption requires an appropriately configured server. See "CONFIGURATION FILE" in DBD::ProxyServer.

- debug

  Turn on debugging mode

- stderr

  This attribute will set the corresponding attribute of the RPC::PlClient object, thus logging will not use syslog(), but redirected to stderr. This is the default under Windows.

```
stderr=1
```

- logfile

  Similar to the stderr attribute, but output will be redirected to the given file.

```
logfile=/dev/null
```

- RowCacheSize

  The DBD::Proxy driver supports this attribute (which is DBI standard, as of DBI 1.02). It's used to reduce network round-trips by fetching multiple rows in one go. The current default value is 20, but this may change.

- proxy_no_finish

  This attribute can be used to reduce network traffic: If the application is calling $sth->finish() then the proxy tells the server to finish the remote statement handle. Of course this slows down things quite a lot, but is perfectly good for reducing memory usage with persistent connections.

However, if you set the *proxy_no_finish* attribute to a TRUE value, either in the database handle or in the statement handle, then finish() calls will be suppressed. This is what you want, for example, in small and fast CGI applications.

- proxy_quote

  This attribute can be used to reduce network traffic: By default calls to $dbh->quote() are passed to the remote driver. Of course this slows down things quite a lot, but is the safest default behaviour.

  However, if you set the *proxy_quote* attribute to the value 'local' either in the database handle or in the statement handle, and the call to quote has only one parameter, then the local default DBI quote method will be used (which will be faster but may be wrong).

KNOWN ISSUES

Unproxied method calls

If a method isn't being proxied, try declaring a stub sub in the appropriate package (DBD::Proxy::db for a dbh method, and DBD::Proxy::st for an sth method). For example:

```
sub DBD::Proxy::db::selectall_arrayref;
```

That will enable selectall_arrayref to be proxied.

Currently many methods aren't explicitly proxied and so you get the DBI's default methods executed on the client.

Some of those methods, like selectall_arrayref, may then call other methods that are proxied (selectall_arrayref calls fetchall_arrayref which calls fetch which is proxied). So things may appear to work but operate more slowly than the could.

This may all change in a later version.

Complex handle attributes

Sometimes handles are having complex attributes like hash refs or array refs and not simple strings or integers. For example, with DBD::CSV, you would like to write something like

```
$dbh->{"csv_tables"}->{"passwd"} =
      { "sep_char" => ":", "eol" => "\n";
```

The above example would advice the CSV driver to assume the file "passwd" to be in the format of the /etc/passwd file: Colons as separators and a line feed without carriage return as line terminator.

Surprisingly this example doesn't work with the proxy driver. To understand the reasons, you should consider the following: The Perl compiler is executing the above example in two steps:

1. The first step is fetching the value of the key "csv_tables" in the handle $dbh. The value returned is complex, a hash ref.

2. The second step is storing some value (the right hand side of the assignment) as the key “passwd” in the hash ref from step 1.

This becomes a little bit clearer, if we rewrite the above code:

```
$tables = $dbh->{"csv_tables"};
$tables->{"passwd"} = { "sep_char" => ":", "eol" => "\n";
```

While the examples work fine without the proxy, the fail due to a subtle difference in step 1: By DBI magic, the hash ref $dbh->{‘csv_tables’} is returned from the server to the client. The client creates a local copy. This local copy is the result of step 1. In other words, step 2 modifies a local copy of the hash ref, but not the server’s hash ref.

The workaround is storing the modified local copy back to the server:

```
$tables = $dbh->{"csv_tables"};
$tables->{"passwd"} = { "sep_char" => ":", "eol" => "\n";
$dbh->{"csv_tables"} = $tables;
```

SECURITY WARNING

RPC::PlClient used underneath is not secure due to serializing and deserializing data with *Storable* module. Use the proxy driver only in trusted environment.

AUTHOR AND COPYRIGHT

This module is Copyright (c) 1997, 1998

```
Jochen Wiedmann
Am Eisteich 9
72555 Metzingen
Germany

Email: joe@ispsoft.de
Phone: +49 7123 14887
```

The DBD::Proxy module is free software; you can redistribute it and/or modify it under the same terms as Perl itself. In particular permission is granted to Tim Bunce for distributing this as a part of the DBI.

SEE ALSO

DBI, RPC::PlClient, *Storable*

DBD::SQLite

Namespace containing the following modules:

- `DBD::SQLite::Constants` - common SQLite constants

DBD::SQLite::Constants

common SQLite constants

SYNOPSIS

```
DBD::SQLite::Constants qw/:result_codes/;
```

DESCRIPTION

You can import necessary SQLite constants from this module. Available tags are `all`, `allowed_return_values_from_sqlite3_txn_state`, `authorizer_action_codes`, `authorizer_return_codes`, `version` (`compile_time_library_version_numbers`), `database_connection_configuration_options`, `dbd_sqlite_string_mode`, `extended_result_codes`, `file_open` (`flags_for_file_open_operations`), `flags_for_sqlite3_setlk_timeout`, `function_flags`, `datatypes` (`fundamental_datatypes`), `result_codes`, `run_time_limit_categories`. See http://sqlite.org/c3ref/constlist.html for the complete list of constants.

This module does not export anything by default.

CONSTANTS

allowed_return_values_from_sqlite3_txn_state

- SQLITE_TXN_NONE
- SQLITE_TXN_READ
- SQLITE_TXN_WRITE

authorizer_action_codes

- SQLITE_CREATE_INDEX
- SQLITE_CREATE_TABLE
- SQLITE_CREATE_TEMP_INDEX
- SQLITE_CREATE_TEMP_TABLE
- SQLITE_CREATE_TEMP_TRIGGER
- SQLITE_CREATE_TEMP_VIEW
- SQLITE_CREATE_TRIGGER
- SQLITE_CREATE_VIEW
- SQLITE_DELETE
- SQLITE_DROP_INDEX
- SQLITE_DROP_TABLE
- SQLITE_DROP_TEMP_INDEX
- SQLITE_DROP_TEMP_TABLE

- SQLITE_DROP_TEMP_TRIGGER
- SQLITE_DROP_TEMP_VIEW
- SQLITE_DROP_TRIGGER
- SQLITE_DROP_VIEW
- SQLITE_INSERT
- SQLITE_PRAGMA
- SQLITE_READ
- SQLITE_SELECT
- SQLITE_TRANSACTION
- SQLITE_UPDATE
- SQLITE_ATTACH
- SQLITE_DETACH
- SQLITE_ALTER_TABLE
- SQLITE_REINDEX
- SQLITE_ANALYZE
- SQLITE_CREATE_VTABLE
- SQLITE_DROP_VTABLE
- SQLITE_FUNCTION
- SQLITE_COPY
- SQLITE_SAVEPOINT
- SQLITE_RECURSIVE

authorizer_return_codes

- SQLITE_DENY
- SQLITE_IGNORE

version (compile_time_library_version_numbers)

- SQLITE_VERSION_NUMBER

database_connection_configuration_options

- SQLITE_DBCONFIG_LOOKASIDE
- SQLITE_DBCONFIG_ENABLE_FKEY
- SQLITE_DBCONFIG_ENABLE_TRIGGER
- SQLITE_DBCONFIG_ENABLE_FTS3_TOKENIZER
- SQLITE_DBCONFIG_ENABLE_LOAD_EXTENSION

- SQLITE_DBCONFIG_MAINDBNAME
- SQLITE_DBCONFIG_NO_CKPT_ON_CLOSE
- SQLITE_DBCONFIG_ENABLE_QPSG
- SQLITE_DBCONFIG_TRIGGER_EQP
- SQLITE_DBCONFIG_MAX
- SQLITE_DBCONFIG_RESET_DATABASE
- SQLITE_DBCONFIG_DEFENSIVE
- SQLITE_DBCONFIG_WRITABLE_SCHEMA
- SQLITE_DBCONFIG_LEGACY_ALTER_TABLE
- SQLITE_DBCONFIG_DQS_DML
- SQLITE_DBCONFIG_DQS_DDL
- SQLITE_DBCONFIG_ENABLE_VIEW
- SQLITE_DBCONFIG_LEGACY_FILE_FORMAT
- SQLITE_DBCONFIG_TRUSTED_SCHEMA
- SQLITE_DBCONFIG_STMT_SCANSTATUS
- SQLITE_DBCONFIG_REVERSE_SCANORDER
- SQLITE_DBCONFIG_ENABLE_ATTACH_CREATE
- SQLITE_DBCONFIG_ENABLE_ATTACH_WRITE
- SQLITE_DBCONFIG_ENABLE_COMMENTS

dbd_sqlite_string_mode

- DBD_SQLITE_STRING_MODE_PV
- DBD_SQLITE_STRING_MODE_BYTES
- DBD_SQLITE_STRING_MODE_UNICODE_NAIVE
- DBD_SQLITE_STRING_MODE_UNICODE_FALLBACK
- DBD_SQLITE_STRING_MODE_UNICODE_STRICT

extended_result_codes

- SQLITE_IOERR_LOCK
- SQLITE_IOERR_READ
- SQLITE_IOERR_SHORT_READ
- SQLITE_IOERR_WRITE
- SQLITE_IOERR_FSYNC
- SQLITE_IOERR_DIR_FSYNC
- SQLITE_IOERR_TRUNCATE

- SQLITE_IOERR_FSTAT
- SQLITE_IOERR_UNLOCK
- SQLITE_IOERR_RDLOCK
- SQLITE_IOERR_DELETE
- SQLITE_IOERR_BLOCKED
- SQLITE_IOERR_NOMEM
- SQLITE_IOERR_ACCESS
- SQLITE_IOERR_CHECKRESERVEDLOCK
- SQLITE_IOERR_CLOSE
- SQLITE_IOERR_DIR_CLOSE
- SQLITE_LOCKED_SHAREDCACHE
- SQLITE_IOERR_SHMOPEN
- SQLITE_IOERR_SHMSIZE
- SQLITE_IOERR_SHMLOCK
- SQLITE_BUSY_RECOVERY
- SQLITE_CANTOPEN_NOTEMPDIR
- SQLITE_IOERR_SHMMAP
- SQLITE_IOERR_SEEK
- SQLITE_CORRUPT_VTAB
- SQLITE_READONLY_RECOVERY
- SQLITE_READONLY_CANTLOCK
- SQLITE_ABORT_ROLLBACK
- SQLITE_CANTOPEN_ISDIR
- SQLITE_IOERR_DELETE_NOENT
- SQLITE_CANTOPEN_FULLPATH
- SQLITE_READONLY_ROLLBACK
- SQLITE_CONSTRAINT_CHECK
- SQLITE_CONSTRAINT_COMMITHOOK
- SQLITE_CONSTRAINT_FOREIGNKEY
- SQLITE_CONSTRAINT_FUNCTION
- SQLITE_CONSTRAINT_NOTNULL
- SQLITE_CONSTRAINT_PRIMARYKEY
- SQLITE_CONSTRAINT_TRIGGER
- SQLITE_CONSTRAINT_UNIQUE

- SQLITE_CONSTRAINT_VTAB
- SQLITE_IOERR_MMAP
- SQLITE_NOTICE_RECOVER_WAL
- SQLITE_NOTICE_RECOVER_ROLLBACK
- SQLITE_IOERR_GETTEMPPATH
- SQLITE_BUSY_SNAPSHOT
- SQLITE_WARNING_AUTOINDEX
- SQLITE_IOERR_CONVPATH
- SQLITE_CANTOPEN_CONVPATH
- SQLITE_CONSTRAINT_ROWID
- SQLITE_READONLY_DBMOVED
- SQLITE_AUTH_USER
- SQLITE_IOERR_VNODE
- SQLITE_IOERR_AUTH
- SQLITE_IOERR_BEGIN_ATOMIC
- SQLITE_IOERR_COMMIT_ATOMIC
- SQLITE_IOERR_ROLLBACK_ATOMIC
- SQLITE_ERROR_MISSING_COLLSEQ
- SQLITE_ERROR_RETRY
- SQLITE_READONLY_CANTINIT
- SQLITE_READONLY_DIRECTORY
- SQLITE_LOCKED_VTAB
- SQLITE_CORRUPT_SEQUENCE
- SQLITE_ERROR_SNAPSHOT
- SQLITE_CANTOPEN_DIRTYWAL
- SQLITE_CANTOPEN_SYMLINK
- SQLITE_CONSTRAINT_PINNED
- SQLITE_OK_SYMLINK
- SQLITE_IOERR_DATA
- SQLITE_BUSY_TIMEOUT
- SQLITE_CORRUPT_INDEX
- SQLITE_IOERR_CORRUPTFS
- SQLITE_CONSTRAINT_DATATYPE
- SQLITE_NOTICE_RBU

- SQLITE_IOERR_IN_PAGE
- SQLITE_ERROR_RESERVESIZE
- SQLITE_ERROR_KEY
- SQLITE_ERROR_UNABLE
- SQLITE_IOERR_BADKEY
- SQLITE_IOERR_CODEC

file_open (flags_for_file_open_operations)

- SQLITE_OPEN_READONLY
- SQLITE_OPEN_READWRITE
- SQLITE_OPEN_CREATE
- SQLITE_OPEN_NOMUTEX
- SQLITE_OPEN_FULLMUTEX
- SQLITE_OPEN_SHAREDCACHE
- SQLITE_OPEN_PRIVATECACHE
- SQLITE_OPEN_URI
- SQLITE_OPEN_MEMORY
- SQLITE_OPEN_NOFOLLOW
- SQLITE_OPEN_SUPER_JOURNAL
- SQLITE_OPEN_EXRESCODE

flags_for_sqlite3_setlk_timeout

- SQLITE_SETLK_BLOCK_ON_CONNECT

function_flags

- SQLITE_DETERMINISTIC
- SQLITE_DIRECTONLY
- SQLITE_SUBTYPE
- SQLITE_INNOCUOUS
- SQLITE_RESULT_SUBTYPE
- SQLITE_SELFORDER1

datatypes (fundamental_datatypes)

- SQLITE_INTEGER
- SQLITE_FLOAT
- SQLITE_BLOB
- SQLITE_NULL
- SQLITE_TEXT

result_codes

- SQLITE_OK
- SQLITE_ERROR
- SQLITE_INTERNAL
- SQLITE_PERM
- SQLITE_ABORT
- SQLITE_BUSY
- SQLITE_LOCKED
- SQLITE_NOMEM
- SQLITE_READONLY
- SQLITE_INTERRUPT
- SQLITE_IOERR
- SQLITE_CORRUPT
- SQLITE_NOTFOUND
- SQLITE_FULL
- SQLITE_CANTOPEN
- SQLITE_PROTOCOL
- SQLITE_EMPTY
- SQLITE_SCHEMA
- SQLITE_TOOBIG
- SQLITE_CONSTRAINT
- SQLITE_MISMATCH
- SQLITE_MISUSE
- SQLITE_NOLFS
- SQLITE_AUTH
- SQLITE_FORMAT
- SQLITE_RANGE

- SQLITE_NOTADB
- SQLITE_ROW
- SQLITE_DONE
- SQLITE_NOTICE
- SQLITE_WARNING

run_time_limit_categories

- SQLITE_LIMIT_LENGTH
- SQLITE_LIMIT_SQL_LENGTH
- SQLITE_LIMIT_COLUMN
- SQLITE_LIMIT_EXPR_DEPTH
- SQLITE_LIMIT_COMPOUND_SELECT
- SQLITE_LIMIT_VDBE_OP
- SQLITE_LIMIT_FUNCTION_ARG
- SQLITE_LIMIT_ATTACHED
- SQLITE_LIMIT_LIKE_PATTERN_LENGTH
- SQLITE_LIMIT_VARIABLE_NUMBER
- SQLITE_LIMIT_TRIGGER_DEPTH
- SQLITE_LIMIT_WORKER_THREADS

DBD::Sponge

Create a DBI statement handle from Perl data

SYNOPSIS

```
my $sponge = DBI->connect("dbi:Sponge:","","",{ RaiseError => 1 });
my $sth = $sponge->prepare($statement, {
        rows => $data,
        NAME => $names,
        %attr
    }
);
```

DESCRIPTION

DBD::Sponge is useful for making a Perl data structure accessible through a standard DBI statement handle. This may be useful to DBD module authors who need to transform data in this way.

METHODS

connect()

```
my $sponge = DBI->connect("dbi:Sponge:","","",{ RaiseError => 1 });
```

Here's a sample syntax for creating a database handle for the Sponge driver. No username and password are needed.

prepare()

```
my $sth = $sponge->prepare($statement, {
        rows => $data,
        NAME => $names,
        %attr
    }
);
```

- The `$statement` here is an arbitrary statement or name you want to provide as identity of your data. If you're using DBI::Profile it will appear in the profile data.

  Generally it's expected that you are preparing a statement handle as if a `select` statement happened.

- `$data` is a reference to the data you are providing, given as an array of arrays.

- `$names` is a reference an array of column names for the `$data` you are providing. The number and order should match the number and ordering of the `$data` columns.

- `%attr` is a hash of other standard DBI attributes that you might pass to a prepare statement.

  Currently only NAME, TYPE, and PRECISION are supported. TYPE defaults to SQL_VARCHAR. PRECISION will be lazily computed if not supplied.

BUGS

Using this module to prepare INSERT-like statements is not currently documented.

AUTHOR AND COPYRIGHT

This module is Copyright (c) 2003 Tim Bunce

Documentation initially written by Mark Stosberg

The DBD::Sponge module is free software; you can redistribute it and/or modify it under the same terms as Perl itself. In particular permission is granted to Tim Bunce for distributing this as a part of the DBI.

SEE ALSO

DBI

5.2.24 DBI

Database independent interface for Perl

SYNOPSIS

```
use DBI;

@driver_names = DBI->available_drivers;
%drivers      = DBI->installed_drivers;
@data_sources = DBI->data_sources($driver_name, \%attr);

$dbh = DBI->connect($data_source, $username, $auth, \%attr);

$rv  = $dbh->do($statement);
$rv  = $dbh->do($statement, \%attr);
$rv  = $dbh->do($statement, \%attr, @bind_values);

$ary_ref  = $dbh->selectall_arrayref($statement);
$hash_ref = $dbh->selectall_hashref($statement, $key_field);

$ary_ref  = $dbh->selectcol_arrayref($statement);
$ary_ref  = $dbh->selectcol_arrayref($statement, \%attr);

@row_ary  = $dbh->selectrow_array($statement);
$ary_ref  = $dbh->selectrow_arrayref($statement);
$hash_ref = $dbh->selectrow_hashref($statement);

$sth = $dbh->prepare($statement);
$sth = $dbh->prepare_cached($statement);

$rc = $sth->bind_param($p_num, $bind_value);
$rc = $sth->bind_param($p_num, $bind_value, $bind_type);
$rc = $sth->bind_param($p_num, $bind_value, \%attr);

$rv = $sth->execute;
$rv = $sth->execute(@bind_values);
$rv = $sth->execute_array(\%attr, ...);

$rc = $sth->bind_col($col_num, \$col_variable);
$rc = $sth->bind_columns(@list_of_refs_to_vars_to_bind);

@row_ary  = $sth->fetchrow_array;
$ary_ref  = $sth->fetchrow_arrayref;
$hash_ref = $sth->fetchrow_hashref;
```

(continues on next page)

(continued from previous page)

```
$ary_ref  = $sth->fetchall_arrayref;
$ary_ref  = $sth->fetchall_arrayref( $slice, $max_rows );

$hash_ref = $sth->fetchall_hashref( $key_field );

$rv  = $sth->rows;

$rc  = $dbh->begin_work;
$rc  = $dbh->commit;
$rc  = $dbh->rollback;

$quoted_string = $dbh->quote($string);

$rc  = $h->err;
$str = $h->errstr;
$rv  = $h->state;

$rc  = $dbh->disconnect;
```

The synopsis above only lists the major methods and parameters.

GETTING HELP

General

Before asking any questions, reread this document, consult the archives and read the DBI FAQ. The archives are listed at the end of this document and on the DBI home page http://dbi.perl.org/support/

You might also like to read the Advanced DBI Tutorial at http://www.slideshare.net/Tim.Bunce/dbi-advanced-tutorial-2007

To help you make the best use of the dbi-users mailing list, and any other lists or forums you may use, I recommend that you read "Getting Answers" by Mike Ash: http://mikeash.com/getting_answers.html.

Mailing Lists

If you have questions about DBI, or DBD driver modules, you can get help from the *dbi-users@perl.org* mailing list. This is the best way to get help. You don't have to subscribe to the list in order to post, though I'd recommend it. You can get help on subscribing and using the list by emailing *dbi-users-help@perl.org*.

Please note that Tim Bunce does not maintain the mailing lists or the web pages (generous volunteers do that). So please don't send mail directly to him; he just doesn't have the time to answer questions personally. The *dbi-users* mailing list has lots of experienced people who should be able to help you if you need it. If you do email Tim he is very likely to just forward it to the mailing list.

IRC

DBI IRC Channel: #dbi on irc.perl.org (irc://irc.perl.org/#dbi)

Online

StackOverflow has a DBI tag https://stackoverflow.com/questions/tagged/dbi with over 800 questions.

The DBI home page at https://dbi.perl.org/ might be worth a visit. It includes links to other resources, but *is rather out-dated*.

Reporting a Bug

If you think you've found a bug then please read "How to Report Bugs Effectively" by Simon Tatham: https://www.chiark.greenend.org.uk/~sgtatham/bugs.html.

If you think you've found a memory leak then read "Memory Leaks".

Your problem is most likely related to the specific DBD driver module you're using. If that's the case then click on the 'Bugs' link on the https://metacpan.org page for your driver. Only submit a bug report against the DBI itself if you're sure that your issue isn't related to the driver you're using.

NOTES

This is the DBI specification that corresponds to DBI version 1.655 (see DBI::Changes for details).

The DBI is evolving at a steady pace, so it's good to check that you have the latest copy.

The significant user-visible changes in each release are documented in the DBI::Changes module so you can read them by executing `perldoc DBI::Changes`.

Some DBI changes require changes in the drivers, but the drivers can take some time to catch up. Newer versions of the DBI have added features that may not yet be supported by the drivers you use. Talk to the authors of your drivers if you need a new feature that is not yet supported.

Features added after DBI 1.21 (February 2002) are marked in the text with the version number of the DBI release they first appeared in.

Extensions to the DBI API often use the `DBIx::*` namespace. See "Naming Conventions and Name Space". DBI extension modules can be found at https://metacpan.org/search?q=DBIx. And all modules related to the DBI can be found at https://metacpan.org/search?q=DBI.

DESCRIPTION

The DBI is a database access module for the Perl programming language. It defines a set of methods, variables, and conventions that provide a consistent database interface, independent of the actual database being used.

It is important to remember that the DBI is just an interface. The DBI is a layer of "glue" between an application and one or more database *driver* modules. It is the driver mod-

ules which do most of the real work. The DBI provides a standard interface and framework for the drivers to operate within.

This document often uses terms like *references*, *objects*, *methods*. If you're not familiar with those terms then it would be a good idea to read at least the following perl manuals first: perlreftut, perldsc, perllol, and perlboot.

Architecture of a DBI Application

```
           |<- Scope of DBI ->|
                .-.   .--------------.   .--------------.
.-------.       | |---| XYZ Driver   |---| XYZ Engine   |
| Perl  |       | |   `--------------'   `--------------'
| script|  |A|  |D|   .--------------.   .--------------.
| using |--|P|--|B|---|Oracle Driver |---|Oracle Engine|
| DBI   |  |I|  |I|   `--------------'   `--------------'
| API   |       | |...
|methods|       | |... Other drivers
`-------'       | |...
                `-'
```

The API, or Application Programming Interface, defines the call interface and variables for Perl scripts to use. The API is implemented by the Perl DBI extension.

The DBI "dispatches" the method calls to the appropriate driver for actual execution. The DBI is also responsible for the dynamic loading of drivers, error checking and handling, providing default implementations for methods, and many other non-database specific duties.

Each driver contains implementations of the DBI methods using the private interface functions of the corresponding database engine. Only authors of sophisticated/multi-database applications or generic library functions need be concerned with drivers.

Notation and Conventions

The following conventions are used in this document:

```
$dbh    Database handle object
$sth    Statement handle object
$drh    Driver handle object (rarely seen or used in applications)
$h      Any of the handle types above ($dbh, $sth, or $drh)
$rc     General Return Code  (boolean: true=ok, false=error)
$rv     General Return Value (typically an integer)
@ary    List of values returned from the database, typically a row
 ↪of data
$rows   Number of rows processed (if available, else -1)
$fh     A filehandle
undef   NULL values are represented by undefined values in Perl
\%attr  Reference to a hash of attribute values passed to methods
```

Note that Perl will automatically destroy database and statement handle objects if all references to them are deleted.

Outline Usage

To use DBI, first you need to load the DBI module:

```
use DBI;
use strict;
```

(The `use strict;` isn't required but is strongly recommended.)

Then you need to "connect" to your data source and get a *handle* for that connection:

```
$dbh = DBI->connect($dsn, $user, $password,
                    { RaiseError => 1, AutoCommit => 0 });
```

Since connecting can be expensive, you generally just connect at the start of your program and disconnect at the end.

Explicitly defining the required `AutoCommit` behaviour is strongly recommended and may become mandatory in a later version. This determines whether changes are automatically committed to the database when executed, or need to be explicitly committed later.

The DBI allows an application to "prepare" statements for later execution. A prepared statement is identified by a statement handle held in a Perl variable. We'll call the Perl variable `$sth` in our examples.

The typical method call sequence for a `SELECT` statement is:

```
prepare,
  execute, fetch, fetch, ...
  execute, fetch, fetch, ...
  execute, fetch, fetch, ...
```

for example:

```
$sth = $dbh->prepare("SELECT foo, bar FROM table WHERE baz=?");

$sth->execute( $baz );

while ( @row = $sth->fetchrow_array ) {
  print "@row\n";
}
```

For queries that are not executed many times at once, it is often cleaner to use the higher level select wrappers:

```
$row_hashref = $dbh->selectrow_hashref("SELECT foo, bar FROM table␣
↪WHERE baz=?", undef, $baz);

$arrayref_of_row_hashrefs = $dbh->selectall_arrayref(
  "SELECT foo, bar FROM table WHERE baz BETWEEN ? AND ?",
  { Slice => {} }, $baz_min, $baz_max);
```

The typical method call sequence for a *non*-`SELECT` statement is:

```
prepare,
  execute,
  execute,
  execute.
```

for example:

```
$sth = $dbh->prepare("INSERT INTO table(foo,bar,baz) VALUES (?,?,?)
 ↪");

while(<CSV>) {
  chomp;
  my ($foo,$bar,$baz) = split /,/;
      $sth->execute( $foo, $bar, $baz );
}
```

The `do()` method is a wrapper of prepare and execute that can be simpler for non repeated *non*-`SELECT` statements (or with drivers that don't support placeholders):

```
$rows_affected = $dbh->do("UPDATE your_table SET foo = foo + 1");

$rows_affected = $dbh->do("DELETE FROM table WHERE foo = ?", undef,
 ↪$foo);
```

To commit your changes to the database (when "AutoCommit" is off):

```
$dbh->commit;  # or call $dbh->rollback; to undo changes
```

Finally, when you have finished working with the data source, you should "disconnect" from it:

```
$dbh->disconnect;
```

General Interface Rules & Caveats

The DBI does not have a concept of a "current session". Every session has a handle object (i.e., a `$dbh`) returned from the `connect` method. That handle object is used to invoke database related methods.

Most data is returned to the Perl script as strings. (Null values are returned as `undef`.) This allows arbitrary precision numeric data to be handled without loss of accuracy. Beware that Perl may not preserve the same accuracy when the string is used as a number.

Dates and times are returned as character strings in the current default format of the corresponding database engine. Time zone effects are database/driver dependent.

Perl supports binary data in Perl strings, and the DBI will pass binary data to and from the driver without change. It is up to the driver implementors to decide how they wish to handle such binary data.

Perl supports two kinds of strings: Unicode (utf8 internally) and non-Unicode (defaults to iso-8859-1 if forced to assume an encoding). Drivers should accept both kinds of strings and, if required, convert them to the character set of the database being used.

Similarly, when fetching from the database character data that isn't iso-8859-1 the driver should convert it into utf8.

Multiple SQL statements may not be combined in a single statement handle (`$sth`), although some databases and drivers do support this (notably Sybase and SQL Server).

Non-sequential record reads are not supported in this version of the DBI. In other words, records can only be fetched in the order that the database returned them, and once fetched they are forgotten.

Positioned updates and deletes are not directly supported by the DBI. See the description of the `CursorName` attribute for an alternative.

Individual driver implementors are free to provide any private functions and/or handle attributes that they feel are useful. Private driver functions can be invoked using the DBI `func()` method. Private driver attributes are accessed just like standard attributes.

Many methods have an optional `\%attr` parameter which can be used to pass information to the driver implementing the method. Except where specifically documented, the `\%attr` parameter can only be used to pass driver specific hints. In general, you can ignore `\%attr` parameters or pass it as `undef`.

Naming Conventions and Name Space

The DBI package and all packages below it (`DBI::*`) are reserved for use by the DBI. Extensions and related modules use the `DBIx::` namespace (see http://www.perl.com/CPAN/modules/by-module/DBIx/). Package names beginning with `DBD::` are reserved for use by DBI database drivers. All environment variables used by the DBI or by individual DBDs begin with "`DBI_`" or "`DBD_`".

The letter case used for attribute names is significant and plays an important part in the portability of DBI scripts. The case of the attribute name is used to signify who defined the meaning of that name and its values.

```
Case of name  Has a meaning defined by
------------  ------------------------
UPPER_CASE    Standards, e.g.,  X/Open, ISO SQL92 etc (portable)
MixedCase     DBI API (portable), underscores are not used.
lower_case    Driver or database engine specific (non-portable)
```

It is of the utmost importance that Driver developers only use lowercase attribute names when defining private attributes. Private attribute names must be prefixed with the driver name or suitable abbreviation (e.g., "`ora_`" for Oracle, "`ing_`" for Ingres, etc).

SQL - A Query Language

Most DBI drivers require applications to use a dialect of SQL (Structured Query Language) to interact with the database engine. The "Standards Reference Information" section provides links to useful information about SQL.

The DBI itself does not mandate or require any particular language to be used; it is language independent. In ODBC terms, the DBI is in "pass-thru" mode, although individual drivers might not be. The only requirement is that queries and other statements must be expressed as a single string of characters passed as the first argument to the "prepare" or "do" methods.

For an interesting diversion on the *real* history of RDBMS and SQL, from the people who made it happen, see:

```
http://www.mcjones.org/System_R/SQL_Reunion_95/sqlr95.html
```

Follow the “Full Contents” then “Intergalactic dataspeak” links for the SQL history.

Placeholders and Bind Values

Some drivers support placeholders and bind values. *Placeholders*, also called parameter markers, are used to indicate values in a database statement that will be supplied later, before the prepared statement is executed. For example, an application might use the following to insert a row of data into the SALES table:

```
INSERT INTO sales (product_code, qty, price) VALUES (?, ?, ?)
```

or the following, to select the description for a product:

```
SELECT description FROM products WHERE product_code = ?
```

The ? characters are the placeholders. The association of actual values with placeholders is known as *binding*, and the values are referred to as *bind values*. Note that the ? is not enclosed in quotation marks, even when the placeholder represents a string.

Some drivers also allow placeholders like :*name* and :*N* (e.g., :1, :2, and so on) in addition to ?, but their use is not portable.

If the :*N* form of placeholder is supported by the driver you’re using, then you should be able to use either “bind_param” or “execute” to bind values. Check your driver documentation.

Some drivers allow you to prevent the recognition of a placeholder by placing a single backslash character (\) immediately before it. The driver will remove the backslash character and ignore the placeholder, passing it unchanged to the backend. If the driver supports this then “get_info”(9000) will return true.

With most drivers, placeholders can’t be used for any element of a statement that would prevent the database server from validating the statement and creating a query execution plan for it. For example:

```
"SELECT name, age FROM ?"         # wrong (will probably fail)
"SELECT name, ?   FROM people"    # wrong (but may not 'fail')
```

Also, placeholders can only represent single scalar values. For example, the following statement won’t work as expected for more than one value:

```
"SELECT name, age FROM people WHERE name IN (?)"    # wrong
"SELECT name, age FROM people WHERE name IN (?,?)"  # two names
```

When using placeholders with the SQL LIKE qualifier, you must remember that the placeholder substitutes for the whole string. So you should use “. . . LIKE ? . . .” and include any wildcard characters in the value that you bind to the placeholder.

NULL Values

Undefined values, or undef, are used to indicate NULL values. You can insert and update columns with a NULL value as you would a non-NULL value. These examples insert and update the column age with a NULL value:

```
$sth = $dbh->prepare(qq{
  INSERT INTO people (fullname, age) VALUES (?, ?)
});
$sth->execute("Joe Bloggs", undef);

$sth = $dbh->prepare(qq{
  UPDATE people SET age = ? WHERE fullname = ?
});
$sth->execute(undef, "Joe Bloggs");
```

However, care must be taken when trying to use NULL values in a WHERE clause. Consider:

```
SELECT fullname FROM people WHERE age = ?
```

Binding an undef (NULL) to the placeholder will *not* select rows which have a NULL age! At least for database engines that conform to the SQL standard. Refer to the SQL manual for your database engine or any SQL book for the reasons for this. To explicitly select NULLs you have to say "WHERE age IS NULL".

A common issue is to have a code fragment handle a value that could be either defined or undef (non-NULL or NULL) at runtime. A simple technique is to prepare the appropriate statement as needed, and substitute the placeholder for non-NULL cases:

```
$sql_clause = defined $age? "age = ?" : "age IS NULL";
$sth = $dbh->prepare(qq{
  SELECT fullname FROM people WHERE $sql_clause
});
$sth->execute(defined $age ? $age : ());
```

The following technique illustrates qualifying a WHERE clause with several columns, whose associated values (defined or undef) are in a hash %h:

```
for my $col ("age", "phone", "email") {
  if (defined $h{$col}) {
    push @sql_qual, "$col = ?";
    push @sql_bind, $h{$col};
  }
  else {
    push @sql_qual, "$col IS NULL";
  }
}
$sql_clause = join(" AND ", @sql_qual);
$sth = $dbh->prepare(qq{
    SELECT fullname FROM people WHERE $sql_clause
});
$sth->execute(@sql_bind);
```

The techniques above call prepare for the SQL statement with each call to execute. Be-

cause calls to prepare() can be expensive, performance can suffer when an application iterates many times over statements like the above.

A better solution is a single WHERE clause that supports both NULL and non-NULL comparisons. Its SQL statement would need to be prepared only once for all cases, thus improving performance. Several examples of WHERE clauses that support this are presented below. But each example lacks portability, robustness, or simplicity. Whether an example is supported on your database engine depends on what SQL extensions it provides, and where it supports the ? placeholder in a statement.

```
0)  age = ?
1)  NVL(age, xx) = NVL(?, xx)
2)  ISNULL(age, xx) = ISNULL(?, xx)
3)  DECODE(age, ?, 1, 0) = 1
4)  age = ? OR (age IS NULL AND ? IS NULL)
5)  age = ? OR (age IS NULL AND SP_ISNULL(?) = 1)
6)  age = ? OR (age IS NULL AND ? = 1)
```

Statements formed with the above WHERE clauses require execute statements as follows. The arguments are required, whether their values are defined or undef.

```
0,1,2,3)  $sth->execute($age);
4,5)      $sth->execute($age, $age);
6)        $sth->execute($age, defined($age) ? 0 : 1);
```

Example 0 should not work (as mentioned earlier), but may work on a few database engines anyway (e.g. Sybase). Example 0 is part of examples 4, 5, and 6, so if example 0 works, these other examples may work, even if the engine does not properly support the right hand side of the OR expression.

Examples 1 and 2 are not robust: they require that you provide a valid column value xx (e.g. '~') which is not present in any row. That means you must have some notion of what data won't be stored in the column, and expect clients to adhere to that.

Example 5 requires that you provide a stored procedure (SP_ISNULL in this example) that acts as a function: it checks whether a value is null, and returns 1 if it is, or 0 if not.

Example 6, the least simple, is probably the most portable, i.e., it should work with most, if not all, database engines.

Here is a table that indicates which examples above are known to work on various database engines:

```
                -----Examples------
                0  1  2  3  4  5  6
                -  -  -  -  -  -  -
Oracle 9        N  Y  N  Y  Y  ?  Y
Informix IDS 9  N  N  N  Y  N  Y  Y
MS SQL          N  N  Y  N  Y  ?  Y
Sybase          Y  N  N  N  N  N  Y
AnyData,DBM,CSV Y  N  N  N  Y  Y* Y
SQLite 3.3      N  N  N  N  Y  N  N
MSAccess        N  N  N  N  Y  N  Y
```

* Works only because Example 0 works.

DBI provides a sample perl script that will test the examples above on your database engine and tell you which ones work. It is located in the `ex/` subdirectory of the DBI source distribution, or here: https://github.com/perl5-dbi/dbi/blob/master/ex/perl_dbi_nulls_test.pl Please use the script to help us fill-in and maintain this table.

Performance

Without using placeholders, the insert statement shown previously would have to contain the literal values to be inserted and would have to be re-prepared and re-executed for each row. With placeholders, the insert statement only needs to be prepared once. The bind values for each row can be given to the `execute` method each time it's called. By avoiding the need to re-prepare the statement for each row, the application typically runs many times faster. Here's an example:

```
my $sth = $dbh->prepare(q{
  INSERT INTO sales (product_code, qty, price) VALUES (?, ?, ?)
}) or die $dbh->errstr;
while (<>) {
    chomp;
    my ($product_code, $qty, $price) = split /,/;
    $sth->execute($product_code, $qty, $price) or die $dbh->errstr;
}
$dbh->commit or die $dbh->errstr;
```

See "execute" and "bind_param" for more details.

The `q{...}` style quoting used in this example avoids clashing with quotes that may be used in the SQL statement. Use the double-quote like `qq{...}` operator if you want to interpolate variables into the string. See *"Quote and Quote-like Operators" in perlop* for more details.

See also the "bind_columns" method, which is used to associate Perl variables with the output columns of a `SELECT` statement.

THE DBI PACKAGE AND CLASS

In this section, we cover the DBI class methods, utility functions, and the dynamic attributes associated with generic DBI handles.

DBI Constants

Constants representing the values of the SQL standard types can be imported individually by name, or all together by importing the special `:sql_types` tag.

The names and values of all the defined SQL standard types can be produced like this:

```
foreach (@{ $DBI::EXPORT_TAGS{sql_types} }) {
  printf "%s=%d\n", $_, &{"DBI::$_"};
}
```

These constants are defined by SQL/CLI, ODBC or both. `SQL_BIGINT` has conflicting codes in SQL/CLI and ODBC, DBI uses the ODBC one.

See the "type_info", "type_info_all", and "bind_param" methods for possible uses.

Note that just because the DBI defines a named constant for a given data type doesn't mean that drivers will support that data type.

DBI Class Methods

The following methods are provided by the DBI class:

parse_dsn

```
($scheme, $driver, $attr_string, $attr_hash, $driver_dsn) = DBI->
↪parse_dsn($dsn)
    or die "Can't parse DBI DSN '$dsn'";
```

Breaks apart a DBI Data Source Name (DSN) and returns the individual parts. If $dsn doesn't contain a valid DSN then parse_dsn() returns an empty list.

$scheme is the first part of the DSN and is currently always 'dbi'. $driver is the driver name, possibly defaulted to $ENV{DBI_DRIVER}, and may be undefined. $attr_string is the contents of the optional attribute string, which may be undefined. If $attr_string is not empty then $attr_hash is a reference to a hash containing the parsed attribute names and values. $driver_dsn is the last part of the DBI DSN string. For example:

```
($scheme, $driver, $attr_string, $attr_hash, $driver_dsn)
    = DBI->parse_dsn("dbi:MyDriver(RaiseError=>1):db=test;port=42");
$scheme      = 'dbi';
$driver      = 'MyDriver';
$attr_string = 'RaiseError=>1';
$attr_hash   = { 'RaiseError' => '1' };
$driver_dsn  = 'db=test;port=42';
```

The parse_dsn() method was added in DBI 1.43.

connect

```
$dbh = DBI->connect($data_source, $username, $password)
          or die $DBI::errstr;
$dbh = DBI->connect($data_source, $username, $password, \%attr)
          or die $DBI::errstr;
```

Establishes a database connection, or session, to the requested `$data_source`. Returns a database handle object if the connection succeeds. Use `$dbh->disconnect` to terminate the connection.

If the connect fails (see below), it returns `undef` and sets both `$DBI::err` and `$DBI::errstr`. (It does *not* explicitly set `$!`.) You should generally test the return status of `connect` and `print $DBI::errstr` if it has failed.

Multiple simultaneous connections to multiple databases through multiple drivers can be made via the DBI. Simply make one `connect` call for each database and keep a copy of each returned database handle.

The `$data_source` value must begin with "`dbi:`*driver_name*`:`". The *driver_name* specifies the driver that will be used to make the connection. (Letter case is significant.)

As a convenience, if the `$data_source` parameter is undefined or empty, the DBI will substitute the value of the environment variable `DBI_DSN`. If just the *driver_name* part is empty (i.e., the `$data_source` prefix is "`dbi::`"), the environment variable `DBI_DRIVER` is used. If neither variable is set, then `connect` dies.

Examples of `$data_source` values are:

```
dbi:DriverName:database_name
dbi:DriverName:database_name@hostname:port
dbi:DriverName:database=database_name;host=hostname;port=port
```

There is *no standard* for the text following the driver name. Each driver is free to use whatever syntax it wants. The only requirement the DBI makes is that all the information is supplied in a single string. You must consult the documentation for the drivers you are using for a description of the syntax they require.

It is recommended that drivers support the ODBC style, shown in the last example above. It is also recommended that they support the three common names '`host`', '`port`', and '`database`' (plus 'db' as an alias for `database`). This simplifies automatic construction of basic DSNs: `"dbi:$driver:database=$db;host=$host;port=$port"`. Drivers should aim to 'do something reasonable' when given a DSN in this form, but if any part is meaningless for that driver (such as 'port' for Informix) it should generate an error if that part is not empty.

If the environment variable `DBI_AUTOPROXY` is defined (and the driver in `$data_source` is not "`Proxy`") then the connect request will automatically be changed to:

```
$ENV{DBI_AUTOPROXY};dsn=$data_source
```

`DBI_AUTOPROXY` is typically set as "`dbi:Proxy:hostname=...;port=...`". If $ENV{DBI_AUTOPROXY} doesn't begin with '`dbi:`' then "dbi:Proxy:" will be prepended to it first. See the DBD::Proxy documentation for more details.

If `$username` or `$password` are undefined (rather than just empty), then the DBI will substitute the values of the `DBI_USER` and `DBI_PASS` environment variables, respectively. The DBI will warn if the environment variables are not defined. However, the everyday use of these environment variables is not recommended for security reasons. The mechanism is primarily intended to simplify testing. See below for alternative way to specify the username and password.

`DBI->connect` automatically installs the driver if it has not been installed yet. Driver installation either returns a valid driver handle, or it *dies* with an error message that includes the string "`install_driver`" and the underlying problem. So `DBI->connect` will die on a driver installation failure and will only return `undef` on a connect failure, in which case `$DBI::errstr` will hold the error message. Use `eval` if you need to catch the "`install_driver`" error.

The `$data_source` argument (with the "`dbi:...:`" prefix removed) and the `$username` and `$password` arguments are then passed to the driver for processing. The DBI does not define any interpretation for the contents of these fields. The driver is free to interpret the `$data_source`, `$username`, and `$password` fields in any way, and supply whatever defaults are appropriate for the engine being accessed. (Oracle, for example, uses the ORACLE_SID and TWO_TASK environment variables if no `$data_source` is specified.)

The AutoCommit and PrintError attributes for each connection default to "on". (See "AutoCommit" and "PrintError" for more information.) However, it is strongly recommended that you explicitly define AutoCommit rather than rely on the default. The PrintWarn attribute defaults to true. The RaiseWarn attribute defaults to false.

The \%attr parameter can be used to alter the default settings of PrintError, RaiseError, AutoCommit, and other attributes. For example:

```
$dbh = DBI->connect($data_source, $user, $pass, {
      PrintError => 0,
      AutoCommit => 0
});
```

The username and password can also be specified using the attributes Username and Password, in which case they take precedence over the $username and $password parameters.

You can also define connection attribute values within the $data_source parameter. For example:

```
dbi:DriverName(PrintWarn=>0,PrintError=>0,Taint=>1):...
```

Individual attributes values specified in this way take precedence over any conflicting values specified via the \%attr parameter to connect.

The dbi_connect_method attribute can be used to specify which driver method should be called to establish the connection. The only useful values are 'connect', 'connect_cached', or some specialized case like 'Apache::DBI::connect' (which is automatically the default when running within Apache).

Where possible, each session ($dbh) is independent from the transactions in other sessions. This is useful when you need to hold cursors open across transactions–for example, if you use one session for your long lifespan cursors (typically read-only) and another for your short update transactions.

For compatibility with old DBI scripts, the driver can be specified by passing its name as the fourth argument to connect (instead of \%attr):

```
$dbh = DBI->connect($data_source, $user, $pass, $driver);
```

In this "old-style" form of connect, the $data_source should not start with "dbi:driver_name:". (If it does, the embedded driver_name will be ignored). Also note that in this older form of connect, the $dbh->{AutoCommit} attribute is *undefined*, the $dbh->{PrintError} attribute is off, and the old DBI_DBNAME environment variable is checked if DBI_DSN is not defined. Beware that this "old-style" connect will soon be withdrawn in a future version of DBI.

connect_cached

```
$dbh = DBI->connect_cached($data_source, $username, $password)
          or die $DBI::errstr;
$dbh = DBI->connect_cached($data_source, $username, $password, \
↪%attr)
          or die $DBI::errstr;
```

connect_cached is like "connect", except that the database handle returned is also stored in a hash associated with the given parameters. If another call is made to connect_cached with the same parameter values, then the corresponding cached $dbh will be returned if it is still valid. The cached database handle is replaced with a new connection if it has been disconnected or if the ping method fails.

Note that the behaviour of this method differs in several respects from the behaviour of persistent connections implemented by Apache::DBI. However, if Apache::DBI is loaded then connect_cached will use it.

Caching connections can be useful in some applications, but it can also cause problems, such as too many connections, and so should be used with care. In particular, avoid changing the attributes of a database handle created via connect_cached() because it will affect other code that may be using the same handle. When connect_cached() returns a handle the attributes will be reset to their initial values. This can cause problems, especially with the AutoCommit attribute.

Also, to ensure that the attributes passed are always the same, avoid passing references inline. For example, the Callbacks attribute is specified as a hash reference. Be sure to declare it external to the call to connect_cached(), such that the hash reference is not re-created on every call. A package-level lexical works well:

```
package MyDBH;
my $cb = {
    'connect_cached.reused' => sub { delete $_[4]->{AutoCommit} },
};

sub dbh {
    DBI->connect_cached( $dsn, $username, $auth, { Callbacks => $cb
 ↪});
}
```

Where multiple separate parts of a program are using connect_cached() to connect to the same database with the same (initial) attributes it is a good idea to add a private attribute to the connect_cached() call to effectively limit the scope of the caching. For example:

```
DBI->connect_cached(..., { private_foo_cachekey => "Bar", ... });
```

Handles returned from that connect_cached() call will only be returned by other connect_cached() call elsewhere in the code if those other calls also pass in the same attribute values, including the private one. (I've used private_foo_cachekey here as an example, you can use any attribute name with a private_ prefix.)

Taking that one step further, you can limit a particular connect_cached() call to return handles unique to that one place in the code by setting the private attribute to a unique value for that place:

```
DBI->connect_cached(..., { private_foo_cachekey => __FILE__.__LINE__
 ↪, ... });
```

By using a private attribute you still get connection caching for the individual calls to connect_cached() but, by making separate database connections for separate parts of

the code, the database handles are isolated from any attribute changes made to other handles.

The cache can be accessed (and cleared) via the "CachedKids" attribute:

```
my $CachedKids_hashref = $dbh->{Driver}->{CachedKids};
%$CachedKids_hashref = () if $CachedKids_hashref;
```

available_drivers

```
@ary = DBI->available_drivers;
@ary = DBI->available_drivers($quiet);
```

Returns a list of all available drivers by searching for `DBD::*` modules through the directories in `@INC`. By default, a warning is given if some drivers are hidden by others of the same name in earlier directories. Passing a true value for `$quiet` will inhibit the warning.

installed_drivers

```
%drivers = DBI->installed_drivers();
```

Returns a list of driver name and driver handle pairs for all drivers 'installed' (loaded) into the current process. The driver name does not include the 'DBD::' prefix.

To get a list of all drivers available in your perl installation you can use "available_drivers".

Added in DBI 1.49.

installed_versions

```
DBI->installed_versions;
@ary  = DBI->installed_versions;
$hash = DBI->installed_versions;
```

Calls available_drivers() and attempts to load each of them in turn using install_driver(). For each load that succeeds the driver name and version number are added to a hash. When running under *DBI::PurePerl* drivers which appear not be pure-perl are ignored.

When called in array context the list of successfully loaded drivers is returned (without the 'DBD::' prefix).

When called in scalar context an extra entry for the `DBI` is added (and `DBI::PurePerl` if appropriate) and a reference to the hash is returned.

When called in a void context the installed_versions() method will print out a formatted list of the hash contents, one per line, along with some other information about the DBI version and OS.

Due to the potentially high memory cost and unknown risks of loading in an unknown number of drivers that just happen to be installed on the system, this method is not recommended for general use. Use available_drivers() instead.

The installed_versions() method is primarily intended as a quick way to see from the command line what's installed. For example:

```
perl -MDBI -e 'DBI->installed_versions'
```

The installed_versions() method was added in DBI 1.38.

data_sources

```
@ary = DBI->data_sources($driver);
@ary = DBI->data_sources($driver, \%attr);
```

Returns a list of data sources (databases) available via the named driver. If `$driver` is empty or `undef`, then the value of the `DBI_DRIVER` environment variable is used.

The driver will be loaded if it hasn't been already. Note that if the driver loading fails then data_sources() *dies* with an error message that includes the string "`install_driver`" and the underlying problem.

Data sources are returned in a form suitable for passing to the "connect" method (that is, they will include the "`dbi:$driver:`" prefix).

Note that many drivers have no way of knowing what data sources might be available for it. These drivers return an empty or incomplete list or may require driver-specific attributes.

There is also a data_sources() method defined for database handles.

trace

```
DBI->trace($trace_setting)
DBI->trace($trace_setting, $trace_filename)
DBI->trace($trace_setting, $trace_filehandle)
$trace_setting = DBI->trace;
```

The `DBI->trace` method sets the *global default* trace settings and returns the *previous* trace settings. It can also be used to change where the trace output is sent.

There's a similar method, `$h->trace`, which sets the trace settings for the specific handle it's called on.

See the "TRACING" section for full details about the DBI's powerful tracing facilities.

visit_handles

```
DBI->visit_handles( $coderef );
DBI->visit_handles( $coderef, $info );
```

Where $coderef is a reference to a subroutine and $info is an arbitrary value which, if undefined, defaults to a reference to an empty hash. Returns $info.

For each installed driver handle, if any, $coderef is invoked as:

```
$coderef->($driver_handle, $info);
```

If the execution of $coderef returns a true value then "visit_child_handles" is called on that child handle and passed the returned value as $info.

For example:

```
my $info = $dbh->{Driver}->visit_child_handles(sub {
    my ($h, $info) = @_;
    ++$info->{ $h->{Type} }; # count types of handles (dr/db/st)
    return $info; # visit kids
});
```

See also "visit_child_handles".

DBI Utility Functions

In addition to the DBI methods listed in the previous section, the DBI package also provides several utility functions.

These can be imported into your code by listing them in the `use` statement. For example:

```
use DBI qw(neat data_diff);
```

Alternatively, all these utility functions (except hash) can be imported using the `:utils` import tag. For example:

```
use DBI qw(:utils);
```

data_string_desc

```
$description = data_string_desc($string);
```

Returns an informal description of the string. For example:

```
UTF8 off, ASCII, 42 characters 42 bytes
UTF8 off, non-ASCII, 42 characters 42 bytes
UTF8 on, non-ASCII, 4 characters 6 bytes
UTF8 on but INVALID encoding, non-ASCII, 4 characters 6 bytes
UTF8 off, undef
```

The initial `UTF8` on/off refers to Perl's internal SvUTF8 flag. If $string has the SvUTF8 flag set but the sequence of bytes it contains are not a valid UTF-8 encoding then data_string_desc() will report `UTF8 on but INVALID encoding`.

The `ASCII` vs non-`ASCII` portion shows `ASCII` if *all* the characters in the string are ASCII (have code points <= 127).

The data_string_desc() function was added in DBI 1.46.

data_string_diff

```
$diff = data_string_diff($a, $b);
```

Returns an informal description of the first character difference between the strings. If both $a and $b contain the same sequence of characters then data_string_diff() returns an empty string. For example:

```
Params a & b     Result
------------     ------
'aaa', 'aaa'     ''
'aaa', 'abc'     'Strings differ at index 2: a[2]=a, b[2]=b'
'aaa', undef     'String b is undef, string a has 3 characters'
'aaa', 'aa'      'String b truncated after 2 characters'
```

Unicode characters are reported in `\x{XXXX}` format. Unicode code points in the range U+0800 to U+08FF are unassigned and most likely to occur due to double-encoding. Characters in this range are reported as `\x{08XX}='C'` where `C` is the corresponding latin-1 character.

The data_string_diff() function only considers logical *characters* and not the underlying encoding. See "data_diff" for an alternative.

The data_string_diff() function was added in DBI 1.46.

data_diff

```
$diff = data_diff($a, $b);
$diff = data_diff($a, $b, $logical);
```

Returns an informal description of the difference between two strings. It calls "data_string_desc" and "data_string_diff" and returns the combined results as a multi-line string.

For example, `data_diff("abc", "ab\x{263a}")` will return:

```
a: UTF8 off, ASCII, 3 characters 3 bytes
b: UTF8 on, non-ASCII, 3 characters 5 bytes
Strings differ at index 2: a[2]=c, b[2]=\x{263A}
```

If $a and $b are identical in both the characters they contain *and* their physical encoding then data_diff() returns an empty string. If $logical is true then physical encoding differences are ignored (but are still reported if there is a difference in the characters).

The data_diff() function was added in DBI 1.46.

neat

```
$str = neat($value);
$str = neat($value, $maxlen);
```

Return a string containing a neat (and tidy) representation of the supplied value.

Strings will be quoted, although internal quotes will *not* be escaped. Values known to be numeric will be unquoted. Undefined (NULL) values will be shown as `undef` (without quotes).

If the string is flagged internally as utf8 then double quotes will be used, otherwise single quotes are used and unprintable characters will be replaced by dot (.).

For result strings longer than `$maxlen` the result string will be truncated to `$maxlen-4` and "`...'`" will be appended. If `$maxlen` is 0 or `undef`, it defaults to `$DBI::neat_maxlen` which, in turn, defaults to 400.

This function is designed to format values for human consumption. It is used internally by the DBI for "trace" output. It should typically *not* be used for formatting values for database use. (See also "quote".)

neat_list

```
$str = neat_list(\@listref, $maxlen, $field_sep);
```

Calls `neat` on each element of the list and returns a string containing the results joined with `$field_sep`. `$field_sep` defaults to `", "`.

looks_like_number

```
@bool = looks_like_number(@array);
```

Returns true for each element that looks like a number. Returns false for each element that does not look like a number. Returns `undef` for each element that is undefined or empty.

hash

```
$hash_value = DBI::hash($buffer, $type);
```

Return a 32-bit integer 'hash' value corresponding to the contents of $buffer. The $type parameter selects which kind of hash algorithm should be used.

For the technically curious, type 0 (which is the default if $type isn't specified) is based on the Perl 5.1 hash except that the value is forced to be negative (for obscure historical reasons). Type 1 is the better "Fowler / Noll / Vo" (FNV) hash. See http://www.isthe.com/chongo/tech/comp/fnv/ for more information. Both types are implemented in C and are very fast.

This function doesn't have much to do with databases, except that it can sometimes be handy to store such values in a database. It also doesn't have much to do with perl hashes, like %foo.

sql_type_cast

```
$sts = DBI::sql_type_cast($sv, $sql_type, $flags);
```

sql_type_cast attempts to cast `$sv` to the SQL type (see [DBI Constants](https://metacpan.org/pod/DBI Constants)) specified in `$sql_type`. At present only the SQL types `SQL_INTEGER`, `SQL_DOUBLE` and `SQL_NUMERIC` are supported.

For `SQL_INTEGER` the effect is similar to using the value in an expression that requires an integer. It gives the perl scalar an 'integer aspect'. (Technically the value gains an IV, or possibly a UV or NV if the value is too large for an IV.)

For `SQL_DOUBLE` the effect is similar to using the value in an expression that requires a general numeric value. It gives the perl scalar a 'numeric aspect'. (Technically the value gains an NV.)

`SQL_NUMERIC` is similar to `SQL_INTEGER` or `SQL_DOUBLE` but more general and more cautious. It will look at the string first and if it looks like an integer (that will fit in an IV or UV) it will act like `SQL_INTEGER`, if it looks like a floating point value it will act like `SQL_DOUBLE`, if it looks like neither then it will do nothing - and thereby avoid the warnings that would be generated by `SQL_INTEGER` and `SQL_DOUBLE` when given non-numeric data.

`$flags` may be:

- `DBIstcf_DISCARD_STRING`

  If this flag is specified then when the driver successfully casts the bound perl scalar to a non-string type then the string portion of the scalar will be discarded.

- `DBIstcf_STRICT`

  If `$sv` cannot be cast to the requested `$sql_type` then by default it is left untouched and no error is generated. If you specify `DBIstcf_STRICT` and the cast fails, this will generate an error.

The returned `$sts` value is:

```
-2 sql_type is not handled
-1 sv is undef so unchanged
 0 sv could not be cast cleanly and DBIstcf_STRICT was used
 1 sv could not be cast and DBIstcf_STRICT was not used
 2 sv was cast successfully
```

This method is exported by the :utils tag and was introduced in DBI 1.611.

DBI Dynamic Attributes

Dynamic attributes are always associated with the *last handle used* (that handle is represented by `$h` in the descriptions below).

Where an attribute is equivalent to a method call, then refer to the method call for all related documentation.

Warning: these attributes are provided as a convenience but they do have limitations. Specifically, they have a short lifespan: because they are associated with the last handle used, they should only be used *immediately* after calling the method that "sets" them. If in any doubt, use the corresponding method call.

$DBI::err

Equivalent to $h->err.

$DBI::errstr

Equivalent to $h->errstr.

$DBI::state

Equivalent to $h->state.

$DBI::rows

Equivalent to $h->rows. Please refer to the documentation for the “rows” method.

$DBI::lasth

Returns the DBI object handle used for the most recent DBI method call. If the last DBI method call was a DESTROY then $DBI::lasth will return the handle of the parent of the destroyed handle, if there is one.

METHODS COMMON TO ALL HANDLES

The following methods can be used by all types of DBI handles.

err

```
$rv = $h->err;
```

Returns the *native* database engine error code from the last driver method called. The code is typically an integer but you should not assume that.

The DBI resets $h->err to undef before almost all DBI method calls, so the value only has a short lifespan. Also, for most drivers, the statement handles share the same error variable as the parent database handle, so calling a method on one handle may reset the error on the related handles.

(Methods which don’t reset err before being called include err() and errstr(), obviously, state(), rows(), func(), trace(), trace_msg(), ping(), and the tied hash attribute FETCH() and STORE() methods.)

If you need to test for specific error conditions *and* have your program be portable to different database engines, then you’ll need to determine what the corresponding error codes are for all those engines and test for all of them.

The DBI uses the value of $DBI::stderr as the `err` value for internal errors. Drivers should also do likewise. The default value for $DBI::stderr is 2000000000.

A driver may return `0` from err() to indicate a warning condition after a method call. Similarly, a driver may return an empty string to indicate a ‘success with information’ condition. In both these cases the value is false but not undef. The errstr() and state() methods may be used to retrieve extra information in these cases.

See “set_err” for more information.

errstr

```
$str = $h->errstr;
```

Returns the native database engine error message from the last DBI method called. This has the same lifespan issues as the “err” method described above.

The returned string may contain multiple messages separated by newline characters.

The errstr() method should not be used to test for errors, use err() for that, because drivers may return ‘success with information’ or warning messages via errstr() for methods that have not ‘failed’.

See “set_err” for more information.

state

```
$str = $h->state;
```

Returns a state code in the standard SQLSTATE five character format. Note that the specific success code `00000` is translated to any empty string (false). If the driver does not support SQLSTATE (and most don’t), then state() will return `S1000` (General Error) for all errors.

The driver is free to return any value via `state`, e.g., warning codes, even if it has not declared an error by returning a true value via the “err” method described above.

The state() method should not be used to test for errors, use err() for that, because drivers may return a ‘success with information’ or warning state code via state() for methods that have not ‘failed’.

set_err

```
$rv = $h->set_err($err, $errstr);
$rv = $h->set_err($err, $errstr, $state);
$rv = $h->set_err($err, $errstr, $state, $method);
$rv = $h->set_err($err, $errstr, $state, $method, $rv);
```

Set the `err`, `errstr`, and `state` values for the handle. This method is typically only used by DBI drivers and DBI subclasses.

If the “HandleSetErr” attribute holds a reference to a subroutine it is called first. The subroutine can alter the $err, $errstr, $state, and $method values. See “HandleSetErr” for full details. If the subroutine returns a true value then the handle `err`, `errstr`, and `state` values are not altered and set_err() returns an empty list (it normally returns $rv which defaults to undef, see below).

Setting `err` to a *true* value indicates an error and will trigger the normal DBI error handling mechanisms, such as `RaiseError` and `HandleError`, if they are enabled, when execution returns from the DBI back to the application.

Setting `err` to `""` indicates an 'information' state, and setting it to `"0"` indicates a 'warning' state. Setting `err` to `undef` also sets `errstr` to undef, and `state` to `""`, irrespective of the values of the $errstr and $state parameters.

The $method parameter provides an alternate method name for the `RaiseError`/`PrintError`/`RaiseWarn`/`PrintWarn` error string instead of the fairly unhelpful '`set_err`'.

The `set_err` method normally returns undef. The $rv parameter provides an alternate return value.

Some special rules apply if the `err` or `errstr` values for the handle are *already* set...

If `errstr` is true then: "`[err was %s now %s]`" is appended if $err is true and `err` is already true and the new err value differs from the original one. Similarly "`[state was %s now %s]`" is appended if $state is true and `state` is already true and the new state value differs from the original one. Finally "`\n`" and the new $errstr are appended if $errstr differs from the existing errstr value. Obviously the `%s`'s above are replaced by the corresponding values.

The handle `err` value is set to $err if: $err is true; or handle `err` value is undef; or $err is defined and the length is greater than the handle `err` length. The effect is that an 'information' state only overrides undef; a 'warning' overrides undef or 'information', and an 'error' state overrides anything.

The handle `state` value is set to $state if $state is true and the handle `err` value was set (by the rules above).

Support for warning and information states was added in DBI 1.41.

trace

```
$h->trace($trace_settings);
$h->trace($trace_settings, $trace_filename);
$trace_settings = $h->trace;
```

The trace() method is used to alter the trace settings for a handle (and any future children of that handle). It can also be used to change where the trace output is sent.

There's a similar method, `DBI->trace`, which sets the global default trace settings.

See the "TRACING" section for full details about the DBI's powerful tracing facilities.

trace_msg

```
$h->trace_msg($message_text);
$h->trace_msg($message_text, $min_level);
```

Writes `$message_text` to the trace file if the trace level is greater than or equal to $min_level (which defaults to 1). Can also be called as `DBI->trace_msg($msg)`.

See "TRACING" for more details.

func

```
$h->func(@func_arguments, $func_name) or die ...;
```

The func method can be used to call private non-standard and non-portable methods implemented by the driver. Note that the function name is given as the *last* argument.

It's also important to note that the func() method does not clear a previous error ($DBI::err etc.) and it does not trigger automatic error detection (RaiseError etc.) so you must check the return status and/or $h->err to detect errors.

(This method is not directly related to calling stored procedures. Calling stored procedures is currently not defined by the DBI. Some drivers, such as DBD::Oracle, support it in non-portable ways. See driver documentation for more details.)

See also install_method() in *DBI::DBD* for how you can avoid needing to use func() and gain direct access to driver-private methods.

can

```
$is_implemented = $h->can($method_name);
```

Returns true if $method_name is implemented by the driver or a default method is provided by the DBI's driver base class. It returns false where a driver hasn't implemented a method and the default method is provided by the DBI's driver base class is just an empty stub.

parse_trace_flags

```
$trace_settings_integer = $h->parse_trace_flags($trace_settings);
```

Parses a string containing trace settings and returns the corresponding integer value used internally by the DBI and drivers.

The $trace_settings argument is a string containing a trace level between 0 and 15 and/or trace flag names separated by vertical bar ("|") or comma (",") characters. For example: `"SQL|3|foo"`.

It uses the parse_trace_flag() method, described below, to process the individual trace flag names.

The parse_trace_flags() method was added in DBI 1.42.

parse_trace_flag

```
$bit_flag = $h->parse_trace_flag($trace_flag_name);
```

Returns the bit flag corresponding to the trace flag name in $trace_flag_name. Drivers are expected to override this method and check if $trace_flag_name is a driver specific trace flags and, if not, then call the DBI's default parse_trace_flag().

The parse_trace_flag() method was added in DBI 1.42.

private_attribute_info

```
$hash_ref = $h->private_attribute_info();
```

Returns a reference to a hash whose keys are the names of driver-private handle attributes available for the kind of handle (driver, database, statement) that the method was called on.

For example, the return value when called with a DBD::Sybase $dbh could look like this:

```
{
    syb_dynamic_supported => undef,
    syb_oc_version => undef,
    syb_server_version => undef,
    syb_server_version_string => undef,
}
```

and when called with a DBD::Sybase $sth they could look like this:

```
{
    syb_types => undef,
    syb_proc_status => undef,
    syb_result_type => undef,
}
```

The values should be undef. Meanings may be assigned to particular values in future.

swap_inner_handle

```
$rc = $h1->swap_inner_handle( $h2 );
$rc = $h1->swap_inner_handle( $h2, $allow_reparent );
```

Brain transplants for handles. You don't need to know about this unless you want to become a handle surgeon.

A DBI handle is a reference to a tied hash. A tied hash has an *inner* hash that actually holds the contents. The swap_inner_handle() method swaps the inner hashes between two handles. The $h1 and $h2 handles still point to the same tied hashes, but what those hashes are tied to has been swapped. In effect $h1 *becomes* $h2 and vice-versa. This is powerful stuff, expect problems. Use with care.

As a small safety measure, the two handles, $h1 and $h2, have to share the same parent unless $allow_reparent is true.

The swap_inner_handle() method was added in DBI 1.44.

Here's a quick kind of 'diagram' as a worked example to help think about what's happening:

```
Original state:
        dbh1o -> dbh1i
        sthAo -> sthAi(dbh1i)
        dbh2o -> dbh2i
```

(continues on next page)

(continued from previous page)

```
swap_inner_handle dbh1o with dbh2o:
        dbh2o -> dbh1i
        sthAo -> sthAi(dbh1i)
        dbh1o -> dbh2i

create new sth from dbh1o:
        dbh2o -> dbh1i
        sthAo -> sthAi(dbh1i)
        dbh1o -> dbh2i
        sthBo -> sthBi(dbh2i)

swap_inner_handle sthAo with sthBo:
        dbh2o -> dbh1i
        sthBo -> sthAi(dbh1i)
        dbh1o -> dbh2i
        sthAo -> sthBi(dbh2i)
```

visit_child_handles

```
$h->visit_child_handles( $coderef );
$h->visit_child_handles( $coderef, $info );
```

Where $coderef is a reference to a subroutine and $info is an arbitrary value which, if undefined, defaults to a reference to an empty hash. Returns $info.

For each child handle of $h, if any, $coderef is invoked as:

```
$coderef->($child_handle, $info);
```

If the execution of $coderef returns a true value then `visit_child_handles` is called on that child handle and passed the returned value as $info.

For example:

```
# count database connections with names (DSN) matching a pattern
my $connections = 0;
$dbh->{Driver}->visit_child_handles(sub {
    my ($h, $info) = @_;
    ++$connections if $h->{Name} =~ /foo/;
    return 0; # don't visit kids
})
```

See also “visit_handles”.

ATTRIBUTES COMMON TO ALL HANDLES

These attributes are common to all types of DBI handles.

Some attributes are inherited by child handles. That is, the value of an inherited attribute in a newly created statement handle is the same as the value in the parent

database handle. Changes to attributes in the new statement handle do not affect the parent database handle and changes to the database handle do not affect existing statement handles, only future ones.

Attempting to set or get the value of an unknown attribute generates a warning, except for private driver specific attributes (which all have names starting with a lowercase letter).

Example:

```
$h->{AttributeName} = ...;    # set/write
... = $h->{AttributeName};    # get/read
```

Warn

Type: boolean, inherited

The Warn attribute enables useful warnings for certain bad practices. It is enabled by default and should only be disabled in rare circumstances. Since warnings are generated using the Perl warn function, they can be intercepted using the Perl $SIG{__WARN__} hook.

The Warn attribute is not related to the PrintWarn attribute.

Active

Type: boolean, read-only

The Active attribute is true if the handle object is "active". This is rarely used in applications. The exact meaning of active is somewhat vague at the moment. For a database handle it typically means that the handle is connected to a database ($dbh->disconnect sets Active off). For a statement handle it typically means that the handle is a SELECT that may have more data to fetch. (Fetching all the data or calling $sth->finish sets Active off.)

Executed

Type: boolean

The Executed attribute is true if the handle object has been "executed". Currently only the $dbh do() method and the $sth execute(), execute_array(), and execute_for_fetch() methods set the Executed attribute.

When it's set on a handle it is also set on the parent handle at the same time. So calling execute() on a $sth also sets the Executed attribute on the parent $dbh.

The Executed attribute for a database handle is cleared by the commit() and rollback() methods (even if they fail). The Executed attribute of a statement handle is not cleared by the DBI under any circumstances and so acts as a permanent record of whether the statement handle was ever used.

The Executed attribute was added in DBI 1.41.

Kids

Type: integer, read-only

For a driver handle, `Kids` is the number of currently existing database handles that were created from that driver handle. For a database handle, `Kids` is the number of currently existing statement handles that were created from that database handle. For a statement handle, the value is zero.

ActiveKids

Type: integer, read-only

Like `Kids`, but only counting those that are `Active` (as above).

CachedKids

Type: hash ref

For a database handle, `CachedKids` returns a reference to the cache (hash) of statement handles created by the "prepare_cached" method. For a driver handle, returns a reference to the cache (hash) of database handles created by the "connect_cached" method.

Type

Type: scalar, read-only

The `Type` attribute identifies the type of a DBI handle. Returns "dr" for driver handles, "db" for database handles and "st" for statement handles.

ChildHandles

Type: array ref

The ChildHandles attribute contains a reference to an array of all the handles created by this handle which are still accessible. The contents of the array are weak-refs and will become undef when the handle goes out of scope. (They're cleared out occasionally.)

`ChildHandles` returns undef if your perl version does not support weak references (check the *Scalar::Util* module). The referenced array returned should be treated as read-only.

For example, to enumerate all driver handles, database handles and statement handles:

```
sub show_child_handles {
    my ($h, $level) = @_;
    printf "%sh %s %s\n", $h->{Type}, "\t" x $level, $h;
    show_child_handles($_, $level + 1)
        for (grep { defined } @{$h->{ChildHandles}});
}

my %drivers = DBI->installed_drivers();
show_child_handles($_, 0) for (values %drivers);
```

CompatMode

Type: boolean, inherited

The CompatMode attribute is used by emulation layers (such as Oraperl) to enable compatible behaviour in the underlying driver (e.g., DBD::Oracle) for this handle. Not normally set by application code.

It also has the effect of disabling the 'quick FETCH' of attribute values from the handles attribute cache. So all attribute values are handled by the drivers own FETCH method. This makes them slightly slower but is useful for special-purpose drivers like DBD::Multiplex.

InactiveDestroy

Type: boolean

The default value, false, means a handle will be fully destroyed as normal when the last reference to it is removed, just as you'd expect.

If set true then the handle will be treated by the DESTROY as if it was no longer Active, and so the *database engine* related effects of DESTROYing a handle will be skipped. Think of the name as meaning 'treat the handle as not-Active in the DESTROY method'.

For a database handle, this attribute does not disable an *explicit* call to the disconnect method, only the implicit call from DESTROY that happens if the handle is still marked as Active.

This attribute is specifically designed for use in Unix applications that "fork" child processes. For some drivers, when the child process exits the destruction of inherited handles cause the corresponding handles in the parent process to cease working.

Either the parent or the child process, but not both, should set InactiveDestroy true on all their shared handles. Alternatively, and preferably, the "AutoInactiveDestroy" can be set in the parent on connect.

To help tracing applications using fork the process id is shown in the trace log whenever a DBI or handle trace() method is called. The process id also shown for *every* method call if the DBI trace level (not handle trace level) is set high enough to show the trace from the DBI's method dispatcher, e.g. >= 9.

AutoInactiveDestroy

Type: boolean, inherited

The "InactiveDestroy" attribute, described above, needs to be explicitly set in the child process after a fork(), on every active database and statement handle. This is a problem if the code that performs the fork() is not under your control, perhaps in a third-party module. Use AutoInactiveDestroy to get around this situation.

If set true, the DESTROY method will check the process id of the handle and, if different from the current process id, it will set the *InactiveDestroy* attribute. It is strongly recommended that AutoInactiveDestroy is enabled on all new code (it's only not enabled by default to avoid backwards compatibility problems).

This is the example it's designed to deal with:

```
my $dbh = DBI->connect(...);
some_code_that_forks(); # Perhaps without your knowledge
# Child process dies, destroying the inherited dbh
$dbh->do(...); # Breaks because parent $dbh is now broken
```

The AutoInactiveDestroy attribute was added in DBI 1.614.

PrintWarn

Type: boolean, inherited

The PrintWarn attribute controls the printing of warnings recorded by the driver. When set to a true value (the default) the DBI will check method calls to see if a warning condition has been set. If so, the DBI will effectively do a `warn("$class $method warning: $DBI::errstr")` where $class is the driver class and $method is the name of the method which failed. E.g.,

```
DBD::Oracle::db execute warning: ... warning text here ...
```

If desired, the warnings can be caught and processed using a $SIG{__WARN__} handler or modules like CGI::Carp and CGI::ErrorWrap.

See also “set_err” for how warnings are recorded and “HandleSetErr” for how to influence it.

Fetching the full details of warnings can require an extra round-trip to the database server for some drivers. In which case the driver may opt to only fetch the full details of warnings if the PrintWarn attribute is true. If PrintWarn is false then these drivers should still indicate the fact that there were warnings by setting the warning string to, for example: “3 warnings”.

PrintError

Type: boolean, inherited

The PrintError attribute can be used to force errors to generate warnings (using warn) in addition to returning error codes in the normal way. When set “on”, any method which results in an error occurring will cause the DBI to effectively do a `warn("$class $method failed: $DBI::errstr")` where $class is the driver class and $method is the name of the method which failed. E.g.,

```
DBD::Oracle::db prepare failed: ... error text here ...
```

By default, DBI->connect sets PrintError “on”.

If desired, the warnings can be caught and processed using a $SIG{__WARN__} handler or modules like CGI::Carp and CGI::ErrorWrap.

RaiseWarn

Type: boolean, inherited

The RaiseWarn attribute can be used to force warnings to raise exceptions rather then simply printing them. It is "off" by default. When set "on", any method which sets warning condition will cause the DBI to effectively do a `die("$class $method warning: $DBI::errstr")`, where `$class` is the driver class and `$method` is the name of the method that sets warning condition. E.g.,

```
DBD::Oracle::db execute warning: ... warning text here ...
```

If you turn RaiseWarn on then you'd normally turn PrintWarn off. If PrintWarn is also on, then the PrintWarn is done first (naturally).

This attribute was added in DBI 1.643.

RaiseError

Type: boolean, inherited

The RaiseError attribute can be used to force errors to raise exceptions rather than simply return error codes in the normal way. It is "off" by default. When set "on", any method which results in an error will cause the DBI to effectively do a `die("$class $method failed: $DBI::errstr")`, where `$class` is the driver class and `$method` is the name of the method that failed. E.g.,

```
DBD::Oracle::db prepare failed: ... error text here ...
```

If you turn RaiseError on then you'd normally turn PrintError off. If PrintError is also on, then the PrintError is done first (naturally).

Typically RaiseError is used in conjunction with `eval`, or a module like Try::Tiny or TryCatch, to catch the exception that's been thrown and handle it. For example:

```
use Try::Tiny;

try {
  ...
  $sth->execute();
  ...
} catch {
  # $sth->err and $DBI::err will be true if error was from DBI
  warn $_; # print the error (which Try::Tiny puts into $_)
  ... # do whatever you need to deal with the error
};
```

In the catch block the $DBI::lasth variable can be useful for diagnosis and reporting if you can't be sure which handle triggered the error. For example, $DBI::lasth->{Type} and $DBI::lasth->{Statement}.

See also "Transactions".

If you want to temporarily turn RaiseError off (inside a library function that is likely to fail, for example), the recommended way is like this:

```
{
  local $h->{RaiseError};  # localize and turn off for this block
  ...
}
```

The original value will automatically and reliably be restored by Perl, regardless of how the block is exited. The same logic applies to other attributes, including `PrintError`.

HandleError

Type: code ref, inherited

The `HandleError` attribute can be used to provide your own alternative behaviour in case of errors. If set to a reference to a subroutine then that subroutine is called when an error is detected (at the same point that `RaiseError` and `PrintError` are handled). It is called also when `RaiseWarn` is enabled and a warning is detected.

The subroutine is called with three parameters: the error message string that `RaiseError`, `RaiseWarn` or `PrintError` would use, the DBI handle being used (dbh, sth or drh as appropriate) and the first value being returned by the method that failed (typically undef).

If the subroutine returns a false value then the `RaiseError`, `RaiseWarn` and/or `PrintError` attributes are checked and acted upon as normal.

For example, to `die` with a full stack trace for any error:

```
use Carp;
$h->{HandleError} = sub { confess(shift) };
```

Or to turn errors into exceptions:

```
use Exception; # or your own favourite exception module
$h->{HandleError} = sub { Exception->new('DBI')->raise($_[0]) };
```

It is possible to 'stack' multiple HandleError handlers by using closures:

```
sub your_subroutine {
  my $previous_handler = $h->{HandleError};
  $h->{HandleError} = sub {
    return 1 if $previous_handler and &$previous_handler(@_);
    ... your code here ...
  };
}
```

Using a `my` inside a subroutine to store the previous `HandleError` value is important. See *perlsub* and perlref for more information about *closures*.

It is possible for `HandleError` to alter the error message that will be used by `RaiseError`, `RaiseWarn` and `PrintError` if it returns false. It can do that by altering the value of `$_[0]`. This example appends a stack trace to all errors and, unlike the previous example using Carp::confess, this will work `PrintError` as well as `RaiseError`:

```
$h->{HandleError} = sub { $_[0]=Carp::longmess($_[0]); 0; };
```

It is also possible for HandleError to hide an error, to a limited degree, by using "set_err" to reset $DBI::err and $DBI::errstr, and altering the return value of the failed method. For example:

```
$h->{HandleError} = sub {
  return 0 unless $_[0] =~ /^\S+ fetchrow_arrayref failed:/;
  return 0 unless $_[1]->err == 1234; # the error to 'hide'
  $h->set_err(undef,undef);   # turn off the error
  $_[2] = [ ... ];    # supply alternative return value
  return 1;
};
```

This only works for methods which return a single value and is hard to make reliable (avoiding infinite loops, for example) and so isn't recommended for general use! If you find a *good* use for it then please let me know.

HandleSetErr

Type: code ref, inherited

The HandleSetErr attribute can be used to intercept the setting of handle err, errstr, and state values. If set to a reference to a subroutine then that subroutine is called whenever set_err() is called, typically by the driver or a subclass.

The subroutine is called with five arguments, the first five that were passed to set_err(): the handle, the err, errstr, and state values being set, and the method name. These can be altered by changing the values in the @_ array. The return value affects set_err() behaviour, see "set_err" for details.

It is possible to 'stack' multiple HandleSetErr handlers by using closures. See "HandleError" for an example.

The HandleSetErr and HandleError subroutines differ in subtle but significant ways. HandleError is only invoked at the point where the DBI is about to return to the application with err set true. It's not invoked by the failure of a method that's been called by another DBI method. HandleSetErr, on the other hand, is called whenever set_err() is called with a defined err value, even if false. So it's not just for errors, despite the name, but also warn and info states. The set_err() method, and thus HandleSetErr, may be called multiple times within a method and is usually invoked from deep within driver code.

In theory a driver can use the return value from HandleSetErr via set_err() to decide whether to continue or not. If set_err() returns an empty list, indicating that the HandleSetErr code has 'handled' the 'error', the driver could then continue instead of failing (if that's a reasonable thing to do). This isn't excepted to be common and any such cases should be clearly marked in the driver documentation and discussed on the dbi-dev mailing list.

The HandleSetErr attribute was added in DBI 1.41.

ErrCount

Type: unsigned integer

The ErrCount attribute is incremented whenever the set_err() method records an error. It isn't incremented by warnings or information states. It is not reset by the DBI at any time.

The ErrCount attribute was added in DBI 1.41. Older drivers may not have been updated to use set_err() to record errors and so this attribute may not be incremented when using them.

ShowErrorStatement

Type: boolean, inherited

The ShowErrorStatement attribute can be used to cause the relevant Statement text to be appended to the error messages generated by the RaiseError, PrintError, RaiseWarn and PrintWarn attributes. Only applies to errors on statement handles plus the prepare(), do(), and the various `select*()` database handle methods. (The exact format of the appended text is subject to change.)

If `$h->{ParamValues}` returns a hash reference of parameter (placeholder) values then those are formatted and appended to the end of the Statement text in the error message.

TraceLevel

Type: integer, inherited

The TraceLevel attribute can be used as an alternative to the "trace" method to set the DBI trace level and trace flags for a specific handle. See "TRACING" for more details.

The TraceLevel attribute is especially useful combined with `local` to alter the trace settings for just a single block of code.

FetchHashKeyName

Type: string, inherited

The FetchHashKeyName attribute is used to specify whether the fetchrow_hashref() method should perform case conversion on the field names used for the hash keys. For historical reasons it defaults to 'NAME' but it is recommended to set it to 'NAME_lc' (convert to lower case) or 'NAME_uc' (convert to upper case) according to your preference. It can only be set for driver and database handles. For statement handles the value is frozen when prepare() is called.

ChopBlanks

Type: boolean, inherited

The ChopBlanks attribute can be used to control the trimming of trailing space characters from fixed width character (CHAR) fields. No other field types are affected, even where field values have trailing spaces.

The default is false (although it is possible that the default may change). Applications that need specific behaviour should set the attribute as needed.

Drivers are not required to support this attribute, but any driver which does not support it must arrange to return `undef` as the attribute value.

LongReadLen

Type: unsigned integer, inherited

The `LongReadLen` attribute may be used to control the maximum length of 'long' type fields (LONG, BLOB, CLOB, MEMO, etc.) which the driver will read from the database automatically when it fetches each row of data.

The `LongReadLen` attribute only relates to fetching and reading long values; it is not involved in inserting or updating them.

A value of 0 means not to automatically fetch any long data. Drivers may return undef or an empty string for long fields when `LongReadLen` is 0.

The default is typically 0 (zero) or 80 bytes but may vary between drivers. Applications fetching long fields should set this value to slightly larger than the longest long field value to be fetched.

Some databases return some long types encoded as pairs of hex digits. For these types, `LongReadLen` relates to the underlying data length and not the doubled-up length of the encoded string.

Changing the value of `LongReadLen` for a statement handle after it has been `prepare`'d will typically have no effect, so it's common to set `LongReadLen` on the `$dbh` before calling `prepare`.

For most drivers the value used here has a direct effect on the memory used by the statement handle while it's active, so don't be too generous. If you can't be sure what value to use you could execute an extra select statement to determine the longest value. For example:

```
$dbh->{LongReadLen} = $dbh->selectrow_array(qq{
    SELECT MAX(OCTET_LENGTH(long_column_name))
    FROM table WHERE ...
});
$sth = $dbh->prepare(qq{
    SELECT long_column_name, ... FROM table WHERE ...
});
```

You may need to take extra care if the table can be modified between the first select and the second being executed. You may also need to use a different function if OCTET_LENGTH() does not work for long types in your database. For example, for Sybase use DATALENGTH() and for Oracle use LENGTHB().

See also "LongTruncOk" for information on truncation of long types.

LongTruncOk

Type: boolean, inherited

The LongTruncOk attribute may be used to control the effect of fetching a long field value which has been truncated (typically because it's longer than the value of the LongReadLen attribute).

By default, LongTruncOk is false and so fetching a long value that needs to be truncated will cause the fetch to fail. (Applications should always be sure to check for errors after a fetch loop in case an error, such as a divide by zero or long field truncation, caused the fetch to terminate prematurely.)

If a fetch fails due to a long field truncation when LongTruncOk is false, many drivers will allow you to continue fetching further rows.

See also "LongReadLen".

TaintIn

Type: boolean, inherited

If the TaintIn attribute is set to a true value *and* Perl is running in taint mode (e.g., started with the -T option), then all the arguments to most DBI method calls are checked for being tainted. *This may change.*

The attribute defaults to off, even if Perl is in taint mode. See *perlsec* for more about taint mode. If Perl is not running in taint mode, this attribute has no effect.

When fetching data that you trust you can turn off the TaintIn attribute, for that statement handle, for the duration of the fetch loop.

The TaintIn attribute was added in DBI 1.31.

TaintOut

Type: boolean, inherited

If the TaintOut attribute is set to a true value *and* Perl is running in taint mode (e.g., started with the -T option), then most data fetched from the database is considered tainted. *This may change.*

The attribute defaults to off, even if Perl is in taint mode. See *perlsec* for more about taint mode. If Perl is not running in taint mode, this attribute has no effect.

When fetching data that you trust you can turn off the TaintOut attribute, for that statement handle, for the duration of the fetch loop.

Currently only fetched data is tainted. It is possible that the results of other DBI method calls, and the value of fetched attributes, may also be tainted in future versions. That change may well break your applications unless you take great care now. If you use DBI Taint mode, please report your experience and any suggestions for changes.

The TaintOut attribute was added in DBI 1.31.

Taint

Type: boolean, inherited

The `Taint` attribute is a shortcut for "TaintIn" and "TaintOut" (it is also present for backwards compatibility).

Setting this attribute sets both "TaintIn" and "TaintOut", and retrieving it returns a true value if and only if "TaintIn" and "TaintOut" are both set to true values.

Profile

Type: inherited

The `Profile` attribute enables the collection and reporting of method call timing statistics. See the *DBI::Profile* module documentation for *much* more detail.

The `Profile` attribute was added in DBI 1.24.

ReadOnly

Type: boolean, inherited

An application can set the `ReadOnly` attribute of a handle to a true value to indicate that it will not be attempting to make any changes using that handle or any children of it.

Note that the exact definition of 'read only' is rather fuzzy. For more details see the documentation for the driver you're using.

If the driver can make the handle truly read-only then it should (unless doing so would have unpleasant side effect, like changing the consistency level from per-statement to per-session). Otherwise the attribute is simply advisory.

A driver can set the `ReadOnly` attribute itself to indicate that the data it is connected to cannot be changed for some reason.

If the driver cannot ensure the `ReadOnly` attribute is adhered to it will record a warning. In this case reading the `ReadOnly` attribute back after it is set true will return true even if the underlying driver cannot ensure this (so any application knows the application declared itself ReadOnly).

Library modules and proxy drivers can use the attribute to influence their behavior. For example, the DBD::Gofer driver considers the `ReadOnly` attribute when making a decision about whether to retry an operation that failed.

The attribute should be set to 1 or 0 (or undef). Other values are reserved.

Callbacks

Type: hash ref

The DBI callback mechanism lets you intercept, and optionally replace, any method call on a DBI handle. At the extreme, it lets you become a puppet master, deceiving the application in any way you want.

The `Callbacks` attribute is a hash reference where the keys are DBI method names and the values are code references. For each key naming a method, the DBI will execute the associated code reference before executing the method.

The arguments to the code reference will be the same as to the method, including the invocant (a database handle or statement handle). For example, say that to callback to some code on a call to `prepare()`:

```
$dbh->{Callbacks} = {
    prepare => sub {
        my ($dbh, $query, $attrs) = @_;
        print "Preparing q{$query}\n"
    },
};
```

The callback would then be executed when you called the `prepare()` method:

```
$dbh->prepare('SELECT 1');
```

And the output of course would be:

```
Preparing q{SELECT 1}
```

Because callbacks are executed *before* the methods they're associated with, you can modify the arguments before they're passed on to the method call. For example, to make sure that all calls to `prepare()` are immediately prepared by DBD::Pg, add a callback that makes sure that the `pg_prepare_now` attribute is always set:

```
my $dbh = DBI->connect($dsn, $username, $auth, {
    Callbacks => {
        prepare => sub {
            $_[2] ||= {};
            $_[2]->{pg_prepare_now} = 1;
            return; # must return nothing
        },
    }
});
```

Note that we are editing the contents of `@_` directly. In this case we've created the attributes hash if it's not passed to the `prepare` call.

You can also prevent the associated method from ever executing. While a callback executes, `$_` holds the method name. (This allows multiple callbacks to share the same code reference and still know what method was called.) To prevent the method from executing, simply `undef $_`. For example, if you wanted to disable calls to `ping()`, you could do this:

```
$dbh->{Callbacks} = {
    ping => sub {
        # tell dispatch to not call the method:
        undef $_;
        # return this value instead:
        return "42 bells";
    }
};
```

As with other attributes, Callbacks can be specified on a handle or via the attributes to

`connect()`. Callbacks can also be applied to a statement methods on a statement handle. For example:

```
$sth->{Callbacks} = {
    execute => sub {
        print "Executing ", shift->{Statement}, "\n";
    }
};
```

The `Callbacks` attribute of a database handle isn't copied to any statement handles it creates. So setting callbacks for a statement handle requires you to set the `Callbacks` attribute on the statement handle yourself, as in the example above, or use the special `ChildCallbacks` key described below.

Special Keys in Callbacks Attribute

In addition to DBI handle method names, the `Callbacks` hash reference supports four additional keys.

The first is the `ChildCallbacks` key. When a statement handle is created from a database handle the `ChildCallbacks` key of the database handle's `Callbacks` attribute, if any, becomes the new `Callbacks` attribute of the statement handle. This allows you to define callbacks for all statement handles created from a database handle. For example, if you wanted to count how many times `execute` was called in your application, you could write:

```
my $exec_count = 0;
my $dbh = DBI->connect( $dsn, $username, $auth, {
    Callbacks => {
        ChildCallbacks => {
            execute => sub { $exec_count++; return; }
        }
    }
});

END {
    print "The execute method was called $exec_count times\n";
}
```

The other three special keys are `connect_cached.new`, `connect_cached.connected`, and `connect_cached.reused`. These keys define callbacks that are called when `connect_cached()` is called, but allow different behaviors depending on whether a new handle is created or a handle is returned. The callback is invoked with these arguments: `$dbh, $dsn, $user, $auth, $attr`.

For example, some applications uses `connect_cached()` to connect with `AutoCommit` enabled and then disable `AutoCommit` temporarily for transactions. If `connect_cached()` is called during a transaction, perhaps in a utility method, then it might select the same cached handle and then force `AutoCommit` on, forcing a commit of the transaction. See the "connect_cached" documentation for one way to deal with that. Here we'll describe an alternative approach using a callback.

Because the `connect_cached.new` and `connect_cached.reused` callbacks are invoked before `connect_cached()` has applied the connect attributes, you can use them

to edit the attributes that will be applied. To prevent a cached handle from having its transactions committed before it's returned, you can eliminate the `AutoCommit` attribute in a `connect_cached.reused` callback, like so:

```
my $cb = {
    'connect_cached.reused' => sub { delete $_[4]->{AutoCommit} },
};

sub dbh {
    my $self = shift;
    DBI->connect_cached( $dsn, $username, $auth, {
        PrintError => 0,
        RaiseError => 1,
        AutoCommit => 1,
        Callbacks  => $cb,
    });
}
```

The upshot is that new database handles are created with `AutoCommit` enabled, while cached database handles are left in whatever transaction state they happened to be in when retrieved from the cache.

Note that we've also used a lexical for the callbacks hash reference. This is because `connect_cached()` returns a new database handle if any of the attributes passed to is have changed. If we used an inline hash reference, `connect_cached()` would return a new database handle every time. Which would rather defeat the purpose.

A more common application for callbacks is setting connection state only when a new connection is made (by connect() or connect_cached()). Adding a callback to the connected method (when using `connect`) or via `connect_cached.connected` (when using connect_cached()>) makes this easy. The connected() method is a no-op by default (unless you subclass the DBI and change it). The DBI calls it to indicate that a new connection has been made and the connection attributes have all been set. You can give it a bit of added functionality by applying a callback to it. For example, to make sure that MySQL understands your application's ANSI-compliant SQL, set it up like so:

```
my $dbh = DBI->connect($dsn, $username, $auth, {
    Callbacks => {
        connected => sub {
            shift->do(q{
                SET SESSION sql_mode='ansi,strict_trans_tables,no_
↪auto_value_on_zero';
            });
            return;
        },
    }
});
```

If you're using `connect_cached()`, use the `connect_cached.connected` callback, instead. This is because `connected()` is called for both new and reused database handles, but you want to execute a callback only the when a new database handle is returned. For example, to set the time zone on connection to a PostgreSQL database, try this:

```
my $cb = {
    'connect_cached.connected' => sub {
        shift->do('SET timezone = UTC');
    }
};

sub dbh {
    my $self = shift;
    DBI->connect_cached( $dsn, $username, $auth, { Callbacks => $cb
 ↪});
}
```

One significant limitation with callbacks is that there can only be one per method per handle. This means it's easy for one use of callbacks to interfere with, or typically simply overwrite, another use of callbacks. For this reason modules using callbacks should document the fact clearly so application authors can tell if use of callbacks by the module will clash with use of callbacks by the application.

You might be able to work around this issue by taking a copy of the original callback and calling it within your own. For example:

```
my $prev_cb = $h->{Callbacks}{method_name};
$h->{Callbacks}{method_name} = sub {
  if ($prev_cb) {
      my @result = $prev_cb->(@_);
      return @result if not $_; # $prev_cb vetoed call
  }
  ... your callback logic here ...
};
```

private_your_module_name_*

The DBI provides a way to store extra information in a DBI handle as "private" attributes. The DBI will allow you to store and retrieve any attribute which has a name starting with "private_".

It is *strongly* recommended that you use just *one* private attribute (e.g., use a hash ref) *and* give it a long and unambiguous name that includes the module or application name that the attribute relates to (e.g., "private_YourFullModuleName_thingy").

Because of the way the Perl tie mechanism works you cannot reliably use the ||= operator directly to initialise the attribute, like this:

```
my $foo = $dbh->{private_yourmodname_foo} ||= { ... }; # WRONG
```

you should use a two step approach like this:

```
my $foo = $dbh->{private_yourmodname_foo};
$foo ||= $dbh->{private_yourmodname_foo} = { ... };
```

This attribute is primarily of interest to people sub-classing DBI, or for applications to piggy-back extra information onto DBI handles.

DBI DATABASE HANDLE OBJECTS

This section covers the methods and attributes associated with database handles.

Database Handle Methods

The following methods are specified for DBI database handles:

clone

```
$new_dbh = $dbh->clone(\%attr);
```

The clone method duplicates the $dbh connection by connecting with the same parameters ($dsn, $user, $password) as originally used.

The attributes for the cloned connect are the same as those used for the *original* connect, with any other attributes in \%attr merged over them. Effectively the same as doing:

```
%attributes_used = ( %original_attributes, %attr );
```

If \%attr is not given then it defaults to a hash containing all the attributes in the attribute cache of $dbh excluding any non-code references, plus the main boolean attributes (RaiseError, PrintError, AutoCommit, etc.). *This behaviour is unreliable and so use of clone without an argument is deprecated and may cause a warning in a future release.*

The clone method can be used even if the database handle is disconnected.

The clone method was added in DBI 1.33.

data_sources

```
@ary = $dbh->data_sources();
@ary = $dbh->data_sources(\%attr);
```

Returns a list of data sources (databases) available via the $dbh driver's data_sources() method, plus any extra data sources that the driver can discover via the connected $dbh. Typically the extra data sources are other databases managed by the same server process that the $dbh is connected to.

Data sources are returned in a form suitable for passing to the "connect" method (that is, they will include the "dbi:$driver:" prefix).

The data_sources() method, for a $dbh, was added in DBI 1.38.

do

```
$rows = $dbh->do($statement)           or die $dbh->errstr;
$rows = $dbh->do($statement, \%attr)   or die $dbh->errstr;
$rows = $dbh->do($statement, \%attr, @bind_values) or die ...
```

Prepare and execute a single statement. Returns the number of rows affected or undef on error. A return value of -1 means the number of rows is not known, not applicable, or not available.

This method is typically most useful for *non*-SELECT statements that either cannot be prepared in advance (due to a limitation of the driver) or do not need to be executed repeatedly. It should not be used for SELECT statements because it does not return a statement handle (so you can't fetch any data).

The default do method is logically similar to:

```
sub do {
    my($dbh, $statement, $attr, @bind_values) = @_;
    my $sth = $dbh->prepare($statement, $attr) or return undef;
    $sth->execute(@bind_values) or return undef;
    my $rows = $sth->rows;
    ($rows == 0) ? "0E0" : $rows; # always return true if no error
}
```

For example:

```
my $rows_deleted = $dbh->do(q{
    DELETE FROM table
    WHERE status = ?
}, undef, 'DONE') or die $dbh->errstr;
```

Using placeholders and @bind_values with the do method can be useful because it avoids the need to correctly quote any variables in the $statement. But if you'll be executing the statement many times then it's more efficient to prepare it once and call execute many times instead.

The q{...} style quoting used in this example avoids clashing with quotes that may be used in the SQL statement. Use the double-quote-like qq{...} operator if you want to interpolate variables into the string. See *"Quote and Quote-like Operators" in perlop* for more details.

Note drivers are free to avoid the overhead of creating an DBI statement handle for do(), especially if there are no parameters. In this case error handlers, if invoked during do(), will be passed the database handle.

last_insert_id

```
$rv = $dbh->last_insert_id();
$rv = $dbh->last_insert_id($catalog, $schema, $table, $field);
$rv = $dbh->last_insert_id($catalog, $schema, $table, $field, \
 ↪%attr);
```

Returns a value 'identifying' the row just inserted, if possible. Typically this would be a value assigned by the database server to a column with an *auto_increment* or *serial* type. Returns undef if the driver does not support the method or can't determine the value.

The $catalog, $schema, $table, and $field parameters may be required for some drivers (see below). If you don't know the parameter values and your driver does not need them, then use undef for each.

There are several caveats to be aware of with this method if you want to use it for portable applications:

* For some drivers the value may only be available immediately after the insert statement has executed (e.g., mysql, Informix).

* For some drivers the $catalog, $schema, $table, and $field parameters are required, for others they are ignored (e.g., mysql).

* Drivers may return an indeterminate value if no insert has been performed yet.

* For some drivers the value may only be available if placeholders have *not* been used (e.g., Sybase, MS SQL). In this case the value returned would be from the last non-placeholder insert statement.

* Some drivers may need driver-specific hints about how to get the value. For example, being told the name of the database 'sequence' object that holds the value. Any such hints are passed as driver-specific attributes in the \%attr parameter.

* If the underlying database offers nothing better, then some drivers may attempt to implement this method by executing "`select max($field) from $table`". Drivers using any approach like this should issue a warning if `AutoCommit` is true because it is generally unsafe - another process may have modified the table between your insert and the select. For situations where you know it is safe, such as when you have locked the table, you can silence the warning by passing `Warn` => 0 in \%attr.

* If no insert has been performed yet, or the last insert failed, then the value is implementation defined.

Given all the caveats above, it's clear that this method must be used with care.

The `last_insert_id` method was added in DBI 1.38.

selectrow_array

```
@row_ary = $dbh->selectrow_array($statement);
@row_ary = $dbh->selectrow_array($statement, \%attr);
@row_ary = $dbh->selectrow_array($statement, \%attr, @bind_values);
```

This utility method combines "prepare", "execute" and "fetchrow_array" into a single call. If called in a list context, it returns the first row of data from the statement. The `$statement` parameter can be a previously prepared statement handle, in which case the `prepare` is skipped.

If any method fails, and "RaiseError" is not set, `selectrow_array` will return an empty list.

If called in a scalar context for a statement handle that has more than one column, it is undefined whether the driver will return the value of the first column or the last. So don't do that. Also, in a scalar context, an `undef` is returned if there are no more rows or if an error occurred. That `undef` can't be distinguished from an `undef` returned because the first field value was NULL. For these reasons you should exercise some caution if you use `selectrow_array` in a scalar context, or just don't do that.

selectrow_arrayref

```
$ary_ref = $dbh->selectrow_arrayref($statement);
$ary_ref = $dbh->selectrow_arrayref($statement, \%attr);
$ary_ref = $dbh->selectrow_arrayref($statement, \%attr, @bind_
↪values);
```

This utility method combines "prepare", "execute" and "fetchrow_arrayref" into a single call. It returns the first row of data from the statement. The `$statement` parameter can be a previously prepared statement handle, in which case the `prepare` is skipped.

If any method fails, and "RaiseError" is not set, `selectrow_arrayref` will return undef.

selectrow_hashref

```
$hash_ref = $dbh->selectrow_hashref($statement);
$hash_ref = $dbh->selectrow_hashref($statement, \%attr);
$hash_ref = $dbh->selectrow_hashref($statement, \%attr, @bind_
↪values);
```

This utility method combines "prepare", "execute" and "fetchrow_hashref" into a single call. It returns the first row of data from the statement. The `$statement` parameter can be a previously prepared statement handle, in which case the `prepare` is skipped.

If any method fails, and "RaiseError" is not set, `selectrow_hashref` will return undef.

selectall_arrayref

```
$ary_ref = $dbh->selectall_arrayref($statement);
$ary_ref = $dbh->selectall_arrayref($statement, \%attr);
$ary_ref = $dbh->selectall_arrayref($statement, \%attr, @bind_
↪values);
```

This utility method combines "prepare", "execute" and "fetchall_arrayref" into a single call. It returns a reference to an array containing a reference to an array (or hash, see below) for each row of data fetched.

The `$statement` parameter can be a previously prepared statement handle, in which case the `prepare` is skipped. This is recommended if the statement is going to be executed many times.

If "RaiseError" is not set and any method except `fetchall_arrayref` fails then `selectall_arrayref` will return `undef`; if `fetchall_arrayref` fails then it will return with whatever data has been fetched thus far. You should check `$dbh->err` afterwards (or use the `RaiseError` attribute) to discover if the data is complete or was truncated due to an error.

The "fetchall_arrayref" method called by `selectall_arrayref` supports a $max_rows parameter. You can specify a value for $max_rows by including a '`MaxRows`' attribute in \%attr. In which case finish() is called for you after fetchall_arrayref() returns.

The "fetchall_arrayref" method called by `selectall_arrayref` also supports a $slice parameter. You can specify a value for $slice by including a '`Slice`' or '`Columns`' attribute in \%attr. The only difference between the two is that if `Slice` is not defined

and Columns is an array ref, then the array is assumed to contain column index values (which count from 1), rather than perl array index values. In which case the array is copied and each value decremented before passing to /fetchall_arrayref.

You may often want to fetch an array of rows where each row is stored as a hash. That can be done simply using:

```
my $emps = $dbh->selectall_arrayref(
    "SELECT ename FROM emp ORDER BY ename",
    { Slice => {} }
);
foreach my $emp ( @$emps ) {
    print "Employee: $emp->{ename}\n";
}
```

Or, to fetch into an array instead of an array ref:

```
@result = @{ $dbh->selectall_arrayref($sql, { Slice => {} }) };
```

See "fetchall_arrayref" method for more details.

selectall_array

```
@ary = $dbh->selectall_array($statement);
@ary = $dbh->selectall_array($statement, \%attr);
@ary = $dbh->selectall_array($statement, \%attr, @bind_values);
```

This is a convenience wrapper around "selectall_arrayref" that returns the rows directly as a list, rather than a reference to an array of rows.

Note that if "RaiseError" is not set then you can't tell the difference between returning no rows and an error. Using RaiseError is best practice.

The selectall_array method was added in DBI 1.635.

selectall_hashref

```
$hash_ref = $dbh->selectall_hashref($statement, $key_field);
$hash_ref = $dbh->selectall_hashref($statement, $key_field, \%attr);
$hash_ref = $dbh->selectall_hashref($statement, $key_field, \%attr,␣
↪@bind_values);
```

This utility method combines "prepare", "execute" and "fetchall_hashref" into a single call. It returns a reference to a hash containing one entry, at most, for each row, as returned by fetchall_hashref().

The $statement parameter can be a previously prepared statement handle, in which case the prepare is skipped. This is recommended if the statement is going to be executed many times.

The $key_field parameter defines which column, or columns, are used as keys in the returned hash. It can either be the name of a single field, or a reference to an array containing multiple field names. Using multiple names yields a tree of nested hashes.

If a row has the same key as an earlier row then it replaces the earlier row.

If any method except `fetchall_hashref` fails, and "RaiseError" is not set, `selectall_hashref` will return `undef`. If `fetchall_hashref` fails and "RaiseError" is not set, then it will return with whatever data it has fetched thus far. $DBI::err should be checked to catch that.

See fetchall_hashref() for more details.

selectcol_arrayref

```
$ary_ref = $dbh->selectcol_arrayref($statement);
$ary_ref = $dbh->selectcol_arrayref($statement, \%attr);
$ary_ref = $dbh->selectcol_arrayref($statement, \%attr, @bind_
 ↪values);
```

This utility method combines "prepare", "execute", and fetching one column from all the rows, into a single call. It returns a reference to an array containing the values of the first column from each row.

The `$statement` parameter can be a previously prepared statement handle, in which case the `prepare` is skipped. This is recommended if the statement is going to be executed many times.

If any method except `fetch` fails, and "RaiseError" is not set, `selectcol_arrayref` will return `undef`. If `fetch` fails and "RaiseError" is not set, then it will return with whatever data it has fetched thus far. $DBI::err should be checked to catch that.

The `selectcol_arrayref` method defaults to pushing a single column value (the first) from each row into the result array. However, it can also push another column, or even multiple columns per row, into the result array. This behaviour can be specified via a '`Columns`' attribute which must be a ref to an array containing the column number or numbers to use. For example:

```
# get array of id and name pairs:
my $ary_ref = $dbh->selectcol_arrayref("select id, name from table",␣
 ↪{ Columns=>[1,2] });
my %hash = @$ary_ref; # build hash from key-value pairs so $hash{
 ↪$id} => name
```

You can specify a maximum number of rows to fetch by including a '`MaxRows`' attribute in \%attr.

prepare

```
$sth = $dbh->prepare($statement)          or die $dbh->errstr;
$sth = $dbh->prepare($statement, \%attr)  or die $dbh->errstr;
```

Prepares a statement for later execution by the database engine and returns a reference to a statement handle object.

The returned statement handle can be used to get attributes of the statement and invoke the "execute" method. See "Statement Handle Methods".

Drivers for engines without the concept of preparing a statement will typically just store the statement in the returned handle and process it when `$sth->execute` is called. Such drivers are unlikely to give much useful information about the statement, such as `$sth->{NUM_OF_FIELDS}`, until after `$sth->execute` has been called. Portable applications should take this into account.

In general, DBI drivers do not parse the contents of the statement (other than simply counting any Placeholders). The statement is passed directly to the database engine, sometimes known as pass-thru mode. This has advantages and disadvantages. On the plus side, you can access all the functionality of the engine being used. On the downside, you're limited if you're using a simple engine, and you need to take extra care if writing applications intended to be portable between engines.

Portable applications should not assume that a new statement can be prepared and/or executed while still fetching results from a previous statement.

Some command-line SQL tools use statement terminators, like a semicolon, to indicate the end of a statement. Such terminators should not normally be used with the DBI.

prepare_cached

```
$sth = $dbh->prepare_cached($statement)
$sth = $dbh->prepare_cached($statement, \%attr)
$sth = $dbh->prepare_cached($statement, \%attr, $if_active)
```

Like "prepare" except that the statement handle returned will be stored in a hash associated with the `$dbh`. If another call is made to `prepare_cached` with the same `$statement` and `%attr` parameter values, then the corresponding cached `$sth` will be returned without contacting the database server. Be sure to understand the cautions and caveats noted below.

The `$if_active` parameter lets you adjust the behaviour if an already cached statement handle is still Active. There are several alternatives:

- **0**: A warning will be generated, and finish() will be called on the statement handle before it is returned. This is the default behaviour if $if_active is not passed.
- **1**: finish() will be called on the statement handle, but the warning is suppressed.
- **2**: Disables any checking.
- **3**: The existing active statement handle will be removed from the cache and a new statement handle prepared and cached in its place. This is the safest option because it doesn't affect the state of the old handle, it just removes it from the cache. [Added in DBI 1.40]

Here are some examples of `prepare_cached`:

```
sub insert_hash {
  my ($table, $field_values) = @_;
  # sort to keep field order, and thus sql, stable for prepare_
↪cached
  my @fields = sort keys %$field_values;
  my @values = @{$field_values}{@fields};
  my $sql = sprintf "insert into %s (%s) values (%s)",
```

(continues on next page)

(continued from previous page)

```
      $table, join(",", @fields), join(",", ("?")x@fields);
  my $sth = $dbh->prepare_cached($sql);
  return $sth->execute(@values);
}

sub search_hash {
  my ($table, $field_values) = @_;
  # sort to keep field order, and thus sql, stable for prepare_
↪cached
  my @fields = sort keys %$field_values;
  my @values = @{$field_values}{@fields};
  my $qualifier = "";
  $qualifier = "where ".join(" and ", map { "$_=?" } @fields) if␣
↪@fields;
  $sth = $dbh->prepare_cached("SELECT * FROM $table $qualifier");
  return $dbh->selectall_arrayref($sth, {}, @values);
}
```

Caveat emptor: This caching can be useful in some applications, but it can also cause problems and should be used with care. Here is a contrived case where caching would cause a significant problem:

```
my $sth = $dbh->prepare_cached('SELECT * FROM foo WHERE bar=?');
$sth->execute(...);
while (my $data = $sth->fetchrow_hashref) {

  # later, in some other code called within the loop...
  my $sth2 = $dbh->prepare_cached('SELECT * FROM foo WHERE bar=?');
  $sth2->execute(...);
  while (my $data2 = $sth2->fetchrow_arrayref) {
    do_stuff(...);
  }
}
```

In this example, since both handles are preparing the exact same statement, `$sth2` will not be its own statement handle, but a duplicate of `$sth` returned from the cache. The results will certainly not be what you expect. Typically the inner fetch loop will work normally, fetching all the records and terminating when there are no more, but now that $sth is the same as $sth2 the outer fetch loop will also terminate.

You'll know if you run into this problem because prepare_cached() will generate a warning by default (when $if_active is false).

The cache used by prepare_cached() is keyed by both the statement and any attributes so you can also avoid this issue by doing something like:

```
$sth = $dbh->prepare_cached("...", { dbi_dummy => __FILE__.__LINE__␣
↪});
```

which will ensure that prepare_cached only returns statements cached by that line of code in that source file.

Also, to ensure the attributes passed are always the same, avoid passing references inline. For example, the Slice attribute is specified as a reference. Be sure to declare it external to the call to prepare_cached(), such that a new hash reference is not created on every call. See "connect_cached" for more details and examples.

If you'd like the cache to managed intelligently, you can tie the hashref returned by `CachedKids` to an appropriate caching module, such as Tie::Cache::LRU:

```
my $cache;
tie %$cache, 'Tie::Cache::LRU', 500;
$dbh->{CachedKids} = $cache;
```

commit

```
$rc  = $dbh->commit     or die $dbh->errstr;
```

Commit (make permanent) the most recent series of database changes if the database supports transactions and AutoCommit is off.

If `AutoCommit` is on, then calling `commit` will issue a "commit ineffective with AutoCommit" warning.

See also "Transactions" in the "FURTHER INFORMATION" section below.

rollback

```
$rc  = $dbh->rollback   or die $dbh->errstr;
```

Rollback (undo) the most recent series of uncommitted database changes if the database supports transactions and AutoCommit is off.

If `AutoCommit` is on, then calling `rollback` will issue a "rollback ineffective with AutoCommit" warning.

See also "Transactions" in the "FURTHER INFORMATION" section below.

begin_work

```
$rc  = $dbh->begin_work   or die $dbh->errstr;
```

Enable transactions (by turning `AutoCommit` off) until the next call to `commit` or `rollback`. After the next `commit` or `rollback`, `AutoCommit` will automatically be turned on again.

If `AutoCommit` is already off when `begin_work` is called then it does nothing except return an error. If the driver does not support transactions then when `begin_work` attempts to set `AutoCommit` off the driver will trigger a fatal error.

See also "Transactions" in the "FURTHER INFORMATION" section below.

disconnect

```
$rc = $dbh->disconnect  or warn $dbh->errstr;
```

Disconnects the database from the database handle. `disconnect` is typically only used before exiting the program. The handle is of little use after disconnecting.

The transaction behaviour of the `disconnect` method is, sadly, undefined. Some database systems (such as Oracle and Ingres) will automatically commit any outstanding changes, but others (such as Informix) will rollback any outstanding changes. Applications not using `AutoCommit` should explicitly call `commit` or `rollback` before calling `disconnect`.

The database is automatically disconnected by the `DESTROY` method if still connected when there are no longer any references to the handle. The `DESTROY` method for each driver should implicitly call `rollback` to undo any uncommitted changes. This is vital behaviour to ensure that incomplete transactions don't get committed simply because Perl calls `DESTROY` on every object before exiting. Also, do not rely on the order of object destruction during "global destruction", as it is undefined.

Generally, if you want your changes to be committed or rolled back when you disconnect, then you should explicitly call "commit" or "rollback" before disconnecting.

If you disconnect from a database while you still have active statement handles (e.g., SELECT statement handles that may have more data to fetch), you will get a warning. The warning may indicate that a fetch loop terminated early, perhaps due to an uncaught error. To avoid the warning call the `finish` method on the active handles.

ping

```
$rc = $dbh->ping;
```

Attempts to determine, in a reasonably efficient way, if the database server is still running and the connection to it is still working. Individual drivers should implement this function in the most suitable manner for their database engine.

The current *default* implementation always returns true without actually doing anything. Actually, it returns "`0 but true`" which is true but zero. That way you can tell if the return value is genuine or just the default. Drivers should override this method with one that does the right thing for their type of database.

Few applications would have direct use for this method. See the specialized Apache::DBI module for one example usage.

get_info

```
$value = $dbh->get_info( $info_type );
```

Returns information about the implementation, i.e. driver and data source capabilities, restrictions etc. It returns `undef` for unknown or unimplemented information types. For example:

```
$database_version  = $dbh->get_info(  18 ); # SQL_DBMS_VER
$max_select_tables = $dbh->get_info( 106 ); # SQL_MAXIMUM_TABLES_IN_
↪SELECT
```

See “Standards Reference Information” for more detailed information about the information types and their meanings and possible return values.

The *DBI::Const::GetInfoType* module exports a %GetInfoType hash that can be used to map info type names to numbers. For example:

```
$database_version = $dbh->get_info( $GetInfoType{SQL_DBMS_VER} );
```

The names are a merging of the ANSI and ODBC standards (which differ in some cases). See *DBI::Const::GetInfoType* for more details.

Because some DBI methods make use of get_info(), drivers are strongly encouraged to support *at least* the following very minimal set of information types to ensure the DBI itself works properly:

```
Type  Name                        Example A     Example B
----  --------------------------  ------------  ----------------
  17  SQL_DBMS_NAME               'ACCESS'      'Oracle'
  18  SQL_DBMS_VER                '03.50.0000'  '08.01.0721 ...'
  29  SQL_IDENTIFIER_QUOTE_CHAR   '`'           '"'
  41  SQL_CATALOG_NAME_SEPARATOR  '.'           '@'
 114  SQL_CATALOG_LOCATION        1             2
```

Values from 9000 to 9999 for get_info are officially reserved for use by Perl DBI. Values in that range which have been assigned a meaning are defined here:

`9000`: true if a backslash character (\) before placeholder-like text (e.g. ?, :foo) will prevent it being treated as a placeholder by the driver. The backslash will be removed before the text is passed to the backend.

table_info

```
$sth = $dbh->table_info( $catalog, $schema, $table, $type );
$sth = $dbh->table_info( $catalog, $schema, $table, $type, \%attr );

# then $sth->fetchall_arrayref or $sth->fetchall_hashref etc
```

Returns an active statement handle that can be used to fetch information about tables and views that exist in the database.

The arguments $catalog, $schema and $table may accept search patterns according to the database/driver, for example: $table = ‘%FOO%’; Remember that the underscore character (’_’) is a search pattern that means match any character, so ‘FOO_%’ is the same as ‘FOO%’ and ‘FOO_BAR%’ will match names like ‘FOO1BAR’.

The value of $type is a comma-separated list of one or more types of tables to be returned in the result set. Each value may optionally be quoted, e.g.:

```
$type = "TABLE";
$type = "'TABLE','VIEW'";
```

In addition the following special cases may also be supported by some drivers:

- If the value of $catalog is '%' and $schema and $table name are empty strings, the result set contains a list of catalog names. For example:

  ```
  $sth = $dbh->table_info('%', '', '');
  ```

- If the value of $schema is '%' and $catalog and $table are empty strings, the result set contains a list of schema names.
- If the value of $type is '%' and $catalog, $schema, and $table are all empty strings, the result set contains a list of table types.

If your driver doesn't support one or more of the selection filter parameters then you may get back more than you asked for and can do the filtering yourself.

This method can be expensive, and can return a large amount of data. (For example, small Oracle installation returns over 2000 rows.) So it's a good idea to use the filters to limit the data as much as possible.

The statement handle returned has at least the following fields in the order show below. Other fields, after these, may also be present.

TABLE_CAT: Table catalog identifier. This field is NULL (`undef`) if not applicable to the data source, which is usually the case. This field is empty if not applicable to the table.

TABLE_SCHEM: The name of the schema containing the TABLE_NAME value. This field is NULL (`undef`) if not applicable to data source, and empty if not applicable to the table.

TABLE_NAME: Name of the table (or view, synonym, etc).

TABLE_TYPE: One of the following: "TABLE", "VIEW", "SYSTEM TABLE", "GLOBAL TEMPORARY", "LOCAL TEMPORARY", "ALIAS", "SYNONYM" or a type identifier that is specific to the data source.

REMARKS: A description of the table. May be NULL (`undef`).

Note that `table_info` might not return records for all tables. Applications can use any valid table regardless of whether it's returned by `table_info`.

See also "tables", "Catalog Methods" and "Standards Reference Information".

column_info

```
$sth = $dbh->column_info( $catalog, $schema, $table, $column );

# then $sth->fetchall_arrayref or $sth->fetchall_hashref etc
```

Returns an active statement handle that can be used to fetch information about columns in specified tables.

The arguments $schema, $table and $column may accept search patterns according to the database/driver, for example: $table = '%FOO%';

Note: The support for the selection criteria is driver specific. If the driver doesn't support one or more of them then you may get back more than you asked for and can do the filtering yourself.

Note: If your driver does not support column_info an undef is returned. This is distinct from asking for something which does not exist in a driver which supports column_info as a valid statement handle to an empty result-set will be returned in this case.

If the arguments don't match any tables then you'll still get a statement handle, it'll just return no rows.

The statement handle returned has at least the following fields in the order shown below. Other fields, after these, may also be present.

TABLE_CAT: The catalog identifier. This field is NULL (`undef`) if not applicable to the data source, which is often the case. This field is empty if not applicable to the table.

TABLE_SCHEM: The schema identifier. This field is NULL (`undef`) if not applicable to the data source, and empty if not applicable to the table.

TABLE_NAME: The table identifier. Note: A driver may provide column metadata not only for base tables, but also for derived objects like SYNONYMS etc.

COLUMN_NAME: The column identifier.

DATA_TYPE: The concise data type code.

TYPE_NAME: A data source dependent data type name.

COLUMN_SIZE: The column size. This is the maximum length in characters for character data types, the number of digits or bits for numeric data types or the length in the representation of temporal types. See the relevant specifications for detailed information.

BUFFER_LENGTH: The length in bytes of transferred data.

DECIMAL_DIGITS: The total number of significant digits to the right of the decimal point.

NUM_PREC_RADIX: The radix for numeric precision. The value is 10 or 2 for numeric data types and NULL (`undef`) if not applicable.

NULLABLE: Indicates if a column can accept NULLs. The following values are defined:

```
SQL_NO_NULLS          0
SQL_NULLABLE          1
SQL_NULLABLE_UNKNOWN  2
```

REMARKS: A description of the column.

COLUMN_DEF: The default value of the column, in a format that can be used directly in an SQL statement.

Note that this may be an expression and not simply the text used for the default value in the original CREATE TABLE statement. For example, given:

```
col1 char(30) default current_user    -- a 'function'
col2 char(30) default 'string'        -- a string literal
```

where “current_user” is the name of a function, the corresponding COLUMN_DEF values would be:

```
Database        col1                    col2
--------        ----                    ----
Oracle:         current_user            'string'
Postgres:       "current_user"()        'string'::text
MS SQL:         (user_name())           ('string')
```

SQL_DATA_TYPE: The SQL data type.

SQL_DATETIME_SUB: The subtype code for datetime and interval data types.

CHAR_OCTET_LENGTH: The maximum length in bytes of a character or binary data type column.

ORDINAL_POSITION: The column sequence number (starting with 1).

IS_NULLABLE: Indicates if the column can accept NULLs. Possible values are: ‘NO’, ‘YES’ and ‘’.

SQL/CLI defines the following additional columns:

```
CHAR_SET_CAT
CHAR_SET_SCHEM
CHAR_SET_NAME
COLLATION_CAT
COLLATION_SCHEM
COLLATION_NAME
UDT_CAT
UDT_SCHEM
UDT_NAME
DOMAIN_CAT
DOMAIN_SCHEM
DOMAIN_NAME
SCOPE_CAT
SCOPE_SCHEM
SCOPE_NAME
MAX_CARDINALITY
DTD_IDENTIFIER
IS_SELF_REF
```

Drivers capable of supplying any of those values should do so in the corresponding column and supply undef values for the others.

Drivers wishing to provide extra database/driver specific information should do so in extra columns beyond all those listed above, and use lowercase field names with the driver-specific prefix (i.e., ‘ora_...’). Applications accessing such fields should do so by name and not by column number.

The result set is ordered by TABLE_CAT, TABLE_SCHEM, TABLE_NAME and ORDINAL_POSITION.

Note: There is some overlap with statement handle attributes (in perl) and SQLDescribeCol (in ODBC). However, SQLColumns provides more metadata.

See also “Catalog Methods” and “Standards Reference Information”.

primary_key_info

```
$sth = $dbh->primary_key_info( $catalog, $schema, $table );

# then $sth->fetchall_arrayref or $sth->fetchall_hashref etc
```

Returns an active statement handle that can be used to fetch information about columns that make up the primary key for a table. The arguments don’t accept search patterns (unlike table_info()).

The statement handle will return one row per column, ordered by TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and KEY_SEQ. If there is no primary key then the statement handle will fetch no rows.

Note: The support for the selection criteria, such as $catalog, is driver specific. If the driver doesn’t support catalogs and/or schemas, it may ignore these criteria.

The statement handle returned has at least the following fields in the order shown below. Other fields, after these, may also be present.

TABLE_CAT: The catalog identifier. This field is NULL (undef) if not applicable to the data source, which is often the case. This field is empty if not applicable to the table.

TABLE_SCHEM: The schema identifier. This field is NULL (undef) if not applicable to the data source, and empty if not applicable to the table.

TABLE_NAME: The table identifier.

COLUMN_NAME: The column identifier.

KEY_SEQ: The column sequence number (starting with 1). Note: This field is named **ORDINAL_POSITION** in SQL/CLI.

PK_NAME: The primary key constraint identifier. This field is NULL (undef) if not applicable to the data source.

See also “Catalog Methods” and “Standards Reference Information”.

primary_key

```
@key_column_names = $dbh->primary_key( $catalog, $schema, $table );
```

Simple interface to the primary_key_info() method. Returns a list of the column names that comprise the primary key of the specified table. The list is in primary key column sequence order. If there is no primary key then an empty list is returned.

foreign_key_info

```
$sth = $dbh->foreign_key_info( $pk_catalog, $pk_schema, $pk_table
                             , $fk_catalog, $fk_schema, $fk_table );

$sth = $dbh->foreign_key_info( $pk_catalog, $pk_schema, $pk_table
```

(continues on next page)

(continued from previous page)

```
                                , $fk_catalog, $fk_schema, $fk_table
                                , \%attr );

# then $sth->fetchall_arrayref or $sth->fetchall_hashref etc
```

Returns an active statement handle that can be used to fetch information about foreign keys in and/or referencing the specified table(s). The arguments don't accept search patterns (unlike table_info()).

`$pk_catalog`, `$pk_schema`, `$pk_table` identify the primary (unique) key table (**PKT**).

`$fk_catalog`, `$fk_schema`, `$fk_table` identify the foreign key table (**FKT**).

If both **PKT** and **FKT** are given, the function returns the foreign key, if any, in table **FKT** that refers to the primary (unique) key of table **PKT**. (Note: In SQL/CLI, the result is implementation-defined.)

If only **PKT** is given, then the result set contains the primary key of that table and all foreign keys that refer to it.

If only **FKT** is given, then the result set contains all foreign keys in that table and the primary keys to which they refer. (Note: In SQL/CLI, the result includes unique keys too.)

For example:

```
$sth = $dbh->foreign_key_info( undef, $user, 'master');
$sth = $dbh->foreign_key_info( undef, undef,   undef , undef, $user,␣
↪'detail');
$sth = $dbh->foreign_key_info( undef, $user, 'master', undef, $user,␣
↪'detail');

# then $sth->fetchall_arrayref or $sth->fetchall_hashref etc
```

Note: The support for the selection criteria, such as `$catalog`, is driver specific. If the driver doesn't support catalogs and/or schemas, it may ignore these criteria.

The statement handle returned has the following fields in the order shown below. Because ODBC never includes unique keys, they define different columns in the result set than SQL/CLI. SQL/CLI column names are shown in parentheses.

PKTABLE_CAT (UK_TABLE_CAT): The primary (unique) key table catalog identifier. This field is NULL (`undef`) if not applicable to the data source, which is often the case. This field is empty if not applicable to the table.

PKTABLE_SCHEM (UK_TABLE_SCHEM): The primary (unique) key table schema identifier. This field is NULL (`undef`) if not applicable to the data source, and empty if not applicable to the table.

PKTABLE_NAME (UK_TABLE_NAME): The primary (unique) key table identifier.

PKCOLUMN_NAME (UK_COLUMN_NAME): The primary (unique) key column identifier.

FKTABLE_CAT (FK_TABLE_CAT): The foreign key table catalog identifier. This field is NULL (`undef`) if not applicable to the data source, which is often the case. This field is

empty if not applicable to the table.

FKTABLE_SCHEM (FK_TABLE_SCHEM): The foreign key table schema identifier. This field is NULL (undef) if not applicable to the data source, and empty if not applicable to the table.

FKTABLE_NAME (FK_TABLE_NAME): The foreign key table identifier.

FKCOLUMN_NAME (FK_COLUMN_NAME): The foreign key column identifier.

KEY_SEQ (ORDINAL_POSITION): The column sequence number (starting with 1).

UPDATE_RULE (UPDATE_RULE): The referential action for the UPDATE rule. The following codes are defined:

```
CASCADE              0
RESTRICT             1
SET NULL             2
NO ACTION            3
SET DEFAULT          4
```

DELETE_RULE (DELETE_RULE): The referential action for the DELETE rule. The codes are the same as for UPDATE_RULE.

FK_NAME (FK_NAME): The foreign key name.

PK_NAME (UK_NAME): The primary (unique) key name.

DEFERRABILITY (DEFERABILITY): The deferrability of the foreign key constraint. The following codes are defined:

```
INITIALLY DEFERRED   5
INITIALLY IMMEDIATE  6
NOT DEFERRABLE       7
```

** (UNIQUE_OR_PRIMARY)**: This column is necessary if a driver includes all candidate (i.e. primary and alternate) keys in the result set (as specified by SQL/CLI). The value of this column is UNIQUE if the foreign key references an alternate key and PRIMARY if the foreign key references a primary key, or it may be undefined if the driver doesn't have access to the information.

See also "Catalog Methods" and "Standards Reference Information".

statistics_info

```
$sth = $dbh->statistics_info( $catalog, $schema, $table, $unique_
 ↪only, $quick );

# then $sth->fetchall_arrayref or $sth->fetchall_hashref etc
```

Returns an active statement handle that can be used to fetch statistical information about a table and its indexes.

The arguments don't accept search patterns (unlike "table_info").

If the boolean argument $unique_only is true, only UNIQUE indexes will be returned in the result set, otherwise all indexes will be returned.

If the boolean argument $quick is set, the actual statistical information columns (CARDINALITY and PAGES) will only be returned if they are readily available from the server, and might not be current. Some databases may return stale statistics or no statistics at all with this flag set.

The statement handle will return at most one row per column name per index, plus at most one row for the entire table itself, ordered by NON_UNIQUE, TYPE, INDEX_QUALIFIER, INDEX_NAME, and ORDINAL_POSITION.

Note: The support for the selection criteria, such as $catalog, is driver specific. If the driver doesn't support catalogs and/or schemas, it may ignore these criteria.

The statement handle returned has at least the following fields in the order shown below. Other fields, after these, may also be present.

TABLE_CAT: The catalog identifier. This field is NULL (`undef`) if not applicable to the data source, which is often the case. This field is empty if not applicable to the table.

TABLE_SCHEM: The schema identifier. This field is NULL (`undef`) if not applicable to the data source, and empty if not applicable to the table.

TABLE_NAME: The table identifier.

NON_UNIQUE: Unique index indicator. Returns 0 for unique indexes, 1 for non-unique indexes

INDEX_QUALIFIER: Index qualifier identifier. The identifier that is used to qualify the index name when doing a `DROP INDEX`; NULL (`undef`) is returned if an index qualifier is not supported by the data source. If a non-NULL (defined) value is returned in this column, it must be used to qualify the index name on a `DROP INDEX` statement; otherwise, the TABLE_SCHEM should be used to qualify the index name.

INDEX_NAME: The index identifier.

TYPE: The type of information being returned. Can be any of the following values: 'table', 'btree', 'clustered', 'content', 'hashed', or 'other'.

In the case that this field is 'table', all fields other than TABLE_CAT, TABLE_SCHEM, TABLE_NAME, TYPE, CARDINALITY, and PAGES will be NULL (`undef`).

ORDINAL_POSITION: Column sequence number (starting with 1).

COLUMN_NAME: The column identifier.

ASC_OR_DESC: Column sort sequence. `A` for Ascending, `D` for Descending, or NULL (`undef`) if not supported for this index.

CARDINALITY: Cardinality of the table or index. For indexes, this is the number of unique values in the index. For tables, this is the number of rows in the table. If not supported, the value will be NULL (`undef`).

PAGES: Number of storage pages used by this table or index. If not supported, the value will be NULL (`undef`).

FILTER_CONDITION: The index filter condition as a string. If the index is not a filtered index, or it cannot be determined whether the index is a filtered index, this value is NULL (`undef`). If the index is a filtered index, but the filter condition cannot be determined, this value is the empty string `''`. Otherwise it will be the literal filter condition as a string, such as `SALARY <= 4500`.

See also “Catalog Methods” and “Standards Reference Information”.

tables

```
@names = $dbh->tables( $catalog, $schema, $table, $type );
@names = $dbh->tables;        # deprecated
```

Simple interface to table_info(). Returns a list of matching table names, possibly including a catalog/schema prefix.

See “table_info” for a description of the parameters.

If `$dbh->get_info(29)` returns true (29 is SQL_IDENTIFIER_QUOTE_CHAR) then the table names are constructed and quoted by “quote_identifier” to ensure they are usable even if they contain whitespace or reserved words etc. This means that the table names returned will include quote characters.

type_info_all

```
$type_info_all = $dbh->type_info_all;
```

Returns a reference to an array which holds information about each data type variant supported by the database and driver. The array and its contents should be treated as read-only.

The first item is a reference to an ‘index’ hash of `Name => Index` pairs. The items following that are references to arrays, one per supported data type variant. The leading index hash defines the names and order of the fields within the arrays that follow it. For example:

```
$type_info_all = [
  {   TYPE_NAME         => 0,
      DATA_TYPE         => 1,
      COLUMN_SIZE       => 2,     # was PRECISION originally
      LITERAL_PREFIX    => 3,
      LITERAL_SUFFIX    => 4,
      CREATE_PARAMS     => 5,
      NULLABLE          => 6,
      CASE_SENSITIVE    => 7,
      SEARCHABLE        => 8,
      UNSIGNED_ATTRIBUTE=> 9,
      FIXED_PREC_SCALE  => 10,    # was MONEY originally
      AUTO_UNIQUE_VALUE => 11,    # was AUTO_INCREMENT originally
      LOCAL_TYPE_NAME   => 12,
      MINIMUM_SCALE     => 13,
      MAXIMUM_SCALE     => 14,
      SQL_DATA_TYPE     => 15,
      SQL_DATETIME_SUB  => 16,
      NUM_PREC_RADIX    => 17,
      INTERVAL_PRECISION=> 18,
  },
```

(continues on next page)

(continued from previous page)

```
  [ 'VARCHAR', SQL_VARCHAR,
      undef, "'","'", undef,0, 1,1,0,0,0,undef,1,255, undef
  ],
  [ 'INTEGER', SQL_INTEGER,
      undef,  "", "", undef,0, 0,1,0,0,0,undef,0,  0, 10
  ],
];
```

More than one row may have the same value in the DATA_TYPE field if there are different ways to spell the type name and/or there are variants of the type with different attributes (e.g., with and without AUTO_UNIQUE_VALUE set, with and without UNSIGNED_ATTRIBUTE, etc).

The rows are ordered by DATA_TYPE first and then by how closely each type maps to the corresponding ODBC SQL data type, closest first.

The meaning of the fields is described in the documentation for the "type_info" method.

An 'index' hash is provided so you don't need to rely on index values defined above. However, using DBD::ODBC with some old ODBC drivers may return older names, shown as comments in the example above. Another issue with the index hash is that the lettercase of the keys is not defined. It is usually uppercase, as show here, but drivers may return names with any lettercase.

Drivers are also free to return extra driver-specific columns of information - though it's recommended that they start at column index 50 to leave room for expansion of the DBI/ODBC specification.

The type_info_all() method is not normally used directly. The "type_info" method provides a more usable and useful interface to the data.

type_info

```
@type_info = $dbh->type_info($data_type);
```

Returns a list of hash references holding information about one or more variants of $data_type. The list is ordered by DATA_TYPE first and then by how closely each type maps to the corresponding ODBC SQL data type, closest first. If called in a scalar context then only the first (best) element is returned.

If $data_type is undefined or SQL_ALL_TYPES, then the list will contain hashes for all data type variants supported by the database and driver.

If $data_type is an array reference then type_info returns the information for the *first* type in the array that has any matches.

The keys of the hash follow the same letter case conventions as the rest of the DBI (see "Naming Conventions and Name Space"). The following uppercase items should always exist, though may be undef:

- TYPE_NAME (string)

  Data type name for use in CREATE TABLE statements etc.

- DATA_TYPE (integer)

SQL data type number.

- COLUMN_SIZE (integer)

  For numeric types, this is either the total number of digits (if the NUM_PREC_RADIX value is 10) or the total number of bits allowed in the column (if NUM_PREC_RADIX is 2).

  For string types, this is the maximum size of the string in characters.

  For date and interval types, this is the maximum number of characters needed to display the value.

- LITERAL_PREFIX (string)

  Characters used to prefix a literal. A typical prefix is "`'`" for characters, or possibly "`0x`" for binary values passed as hexadecimal. NULL (`undef`) is returned for data types for which this is not applicable.

- LITERAL_SUFFIX (string)

  Characters used to suffix a literal. Typically "`'`" for characters. NULL (`undef`) is returned for data types where this is not applicable.

- CREATE_PARAMS (string)

  Parameter names for data type definition. For example, `CREATE_PARAMS` for a `DECIMAL` would be "`precision,scale`" if the DECIMAL type should be declared as `DECIMAL`(*precision*,*scale*) where *precision* and *scale* are integer values. For a `VARCHAR` it would be "`max length`". NULL (`undef`) is returned for data types for which this is not applicable.

- NULLABLE (integer)

  Indicates whether the data type accepts a NULL value: `0` or an empty string = no, 1 = yes, 2 = unknown.

- CASE_SENSITIVE (boolean)

  Indicates whether the data type is case sensitive in collations and comparisons.

- SEARCHABLE (integer)

  Indicates how the data type can be used in a WHERE clause, as follows:

```
0 - Cannot be used in a WHERE clause
1 - Only with a LIKE predicate
2 - All comparison operators except LIKE
3 - Can be used in a WHERE clause with any comparison operator
```

- UNSIGNED_ATTRIBUTE (boolean)

  Indicates whether the data type is unsigned. NULL (`undef`) is returned for data types for which this is not applicable.

- FIXED_PREC_SCALE (boolean)

  Indicates whether the data type always has the same precision and scale (such as a money type). NULL (`undef`) is returned for data types for which this is not applicable.

- AUTO_UNIQUE_VALUE (boolean)

  Indicates whether a column of this data type is automatically set to a unique value whenever a new row is inserted. NULL (undef) is returned for data types for which this is not applicable.

- LOCAL_TYPE_NAME (string)

  Localized version of the TYPE_NAME for use in dialog with users. NULL (undef) is returned if a localized name is not available (in which case TYPE_NAME should be used).

- MINIMUM_SCALE (integer)

  The minimum scale of the data type. If a data type has a fixed scale, then MAXIMUM_SCALE holds the same value. NULL (undef) is returned for data types for which this is not applicable.

- MAXIMUM_SCALE (integer)

  The maximum scale of the data type. If a data type has a fixed scale, then MINIMUM_SCALE holds the same value. NULL (undef) is returned for data types for which this is not applicable.

- SQL_DATA_TYPE (integer)

  This column is the same as the DATA_TYPE column, except for interval and datetime data types. For interval and datetime data types, the SQL_DATA_TYPE field will return SQL_INTERVAL or SQL_DATETIME, and the SQL_DATETIME_SUB field below will return the subcode for the specific interval or datetime data type. If this field is NULL, then the driver does not support or report on interval or datetime subtypes.

- SQL_DATETIME_SUB (integer)

  For interval or datetime data types, where the SQL_DATA_TYPE field above is SQL_INTERVAL or SQL_DATETIME, this field will hold the *subcode* for the specific interval or datetime data type. Otherwise it will be NULL (undef).

  Although not mentioned explicitly in the standards, it seems there is a simple relationship between these values:

```
DATA_TYPE == (10 * SQL_DATA_TYPE) + SQL_DATETIME_SUB
```

- NUM_PREC_RADIX (integer)

  The radix value of the data type. For approximate numeric types, NUM_PREC_RADIX contains the value 2 and COLUMN_SIZE holds the number of bits. For exact numeric types, NUM_PREC_RADIX contains the value 10 and COLUMN_SIZE holds the number of decimal digits. NULL (undef) is returned either for data types for which this is not applicable or if the driver cannot report this information.

- INTERVAL_PRECISION (integer)

  The interval leading precision for interval types. NULL is returned either for data types for which this is not applicable or if the driver cannot report this information.

For example, to find the type name for the fields in a select statement you can do:

```
@names = map { scalar $dbh->type_info($_)->{TYPE_NAME} } @{ $sth->
 ↪{TYPE} }
```

Since DBI and ODBC drivers vary in how they map their types into the ISO standard types you may need to search for more than one type. Here's an example looking for a usable type to store a date:

```
$my_date_type = $dbh->type_info( [ SQL_DATE, SQL_TIMESTAMP ] );
```

Similarly, to more reliably find a type to store small integers, you could use a list starting with `SQL_SMALLINT`, `SQL_INTEGER`, `SQL_DECIMAL`, etc.

See also "Standards Reference Information".

quote

```
$sql = $dbh->quote($value);
$sql = $dbh->quote($value, $data_type);
```

Quote a string literal for use as a literal value in an SQL statement, by escaping any special characters (such as quotation marks) contained within the string and adding the required type of outer quotation marks.

```
$sql = sprintf "SELECT foo FROM bar WHERE baz = %s",
               $dbh->quote("Don't");
```

For most database types, at least those that conform to SQL standards, quote would return `'Don''t'` (including the outer quotation marks). For others it may return something like `'Don\'t'`

An undefined `$value` value will be returned as the string `NULL` (without single quotation marks) to match how NULLs are represented in SQL.

If `$data_type` is supplied, it is used to try to determine the required quoting behaviour by using the information returned by "type_info". As a special case, the standard numeric types are optimized to return `$value` without calling `type_info`.

Quote will probably *not* be able to deal with all possible input (such as binary data or data containing newlines), and is not related in any way with escaping or quoting shell meta-characters.

It is valid for the quote() method to return an SQL expression that evaluates to the desired string. For example:

```
$quoted = $dbh->quote("one\ntwo\0three")
```

may return something like:

```
CONCAT('one', CHAR(12), 'two', CHAR(0), 'three')
```

The quote() method should *not* be used with "Placeholders and Bind Values".

quote_identifier

```
$sql = $dbh->quote_identifier( $name );
$sql = $dbh->quote_identifier( $catalog, $schema, $table, \%attr );
```

Quote an identifier (table name etc.) for use in an SQL statement, by escaping any special characters (such as double quotation marks) it contains and adding the required type of outer quotation marks.

Undefined names are ignored and the remainder are quoted and then joined together, typically with a dot (.) character. For example:

```
$id = $dbh->quote_identifier( undef, 'Her schema', 'My table' );
```

would, for most database types, return `"Her schema"."My table"` (including all the double quotation marks).

If three names are supplied then the first is assumed to be a catalog name and special rules may be applied based on what "get_info" returns for SQL_CATALOG_NAME_SEPARATOR (41) and SQL_CATALOG_LOCATION (114). For example, for Oracle:

```
$id = $dbh->quote_identifier( 'link', 'schema', 'table' );
```

would return `"schema"."table"@"link"`.

take_imp_data

```
$imp_data = $dbh->take_imp_data;
```

Leaves the $dbh in an almost dead, zombie-like, state and returns a binary string of raw implementation data from the driver which describes the current database connection. Effectively it detaches the underlying database API connection data from the DBI handle. After calling take_imp_data(), all other methods except `DESTROY` will generate a warning and return undef.

Why would you want to do this? You don't, forget I even mentioned it. Unless, that is, you're implementing something advanced like a multi-threaded connection pool like `DBI::Pool`.

The returned $imp_data can be passed as a `dbi_imp_data` attribute to a later connect() call, even in a separate thread in the same process, where the driver can use it to 'adopt' the existing connection that the implementation data was taken from.

Some things to keep in mind...

* the $imp_data holds the only reference to the underlying database API connection data. That connection is still 'live' and won't be cleaned up properly unless the $imp_data is used to create a new $dbh which is then allowed to disconnect() normally.

* using the same $imp_data to create more than one other new $dbh at a time may well lead to unpleasant problems. Don't do that.

Any child statement handles are effectively destroyed when take_imp_data() is called.

The `take_imp_data` method was added in DBI 1.36 but wasn't useful till 1.49.

Database Handle Attributes

This section describes attributes specific to database handles.

Changes to these database handle attributes do not affect any other existing or future database handles.

Attempting to set or get the value of an unknown attribute generates a warning, except for private driver-specific attributes (which all have names starting with a lowercase letter).

Example:

```
$h->{AutoCommit} = ...;       # set/write
... = $h->{AutoCommit};       # get/read
```

AutoCommit

Type: boolean

If true, then database changes cannot be rolled-back (undone). If false, then database changes automatically occur within a "transaction", which must either be committed or rolled back using the `commit` or `rollback` methods.

Drivers should always default to `AutoCommit` mode (an unfortunate choice largely forced on the DBI by ODBC and JDBC conventions.)

Attempting to set `AutoCommit` to an unsupported value is a fatal error. This is an important feature of the DBI. Applications that need full transaction behaviour can set `$dbh->{AutoCommit} = 0` (or set `AutoCommit` to 0 via "connect") without having to check that the value was assigned successfully.

For the purposes of this description, we can divide databases into three categories:

```
Databases which don't support transactions at all.
Databases in which a transaction is always active.
Databases in which a transaction must be explicitly started (C<
 ↪'BEGIN WORK'>).
```

*** Databases which don't support transactions at all**

For these databases, attempting to turn `AutoCommit` off is a fatal error. `commit` and `rollback` both issue warnings about being ineffective while `AutoCommit` is in effect.

*** Databases in which a transaction is always active**

These are typically mainstream commercial relational databases with "ANSI standard" transaction behaviour. If `AutoCommit` is off, then changes to the database won't have any lasting effect unless "commit" is called (but see also "disconnect"). If "rollback" is called then any changes since the last commit are undone.

If `AutoCommit` is on, then the effect is the same as if the DBI called `commit` automatically after every successful database operation. So calling `commit` or `rollback` explicitly while `AutoCommit` is on would be ineffective because the changes would have already been committed.

Changing `AutoCommit` from off to on will trigger a "commit".

For databases which don't support a specific auto-commit mode, the driver has to commit each statement automatically using an explicit `COMMIT` after it completes successfully (and roll it back using an explicit `ROLLBACK` if it fails). The error information reported to the application will correspond to the statement which was executed, unless it succeeded and the commit or rollback failed.

*** Databases in which a transaction must be explicitly started**

For these databases, the intention is to have them act like databases in which a transaction is always active (as described above).

To do this, the driver will automatically begin an explicit transaction when `AutoCommit` is turned off, or after a "commit" or "rollback" (or when the application issues the next database operation after one of those events).

In this way, the application does not have to treat these databases as a special case.

See "commit", "disconnect" and "Transactions" for other important notes about transactions.

Driver

Type: handle

Holds the handle of the parent driver. The only recommended use for this is to find the name of the driver using:

```
$dbh->{Driver}->{Name}
```

Name

Type: string

Holds the "name" of the database. Usually (and recommended to be) the same as the "`dbi:DriverName:...`" string used to connect to the database, but with the leading "`dbi:DriverName:`" removed.

Statement

Type: string, read-only

Returns the statement string passed to the most recent "prepare" or "do" method called in this database handle, even if that method failed. This is especially useful where `RaiseError` is enabled and the exception handler checks $@ and sees that a 'prepare' method call failed.

RowCacheSize

Type: integer

A hint to the driver indicating the size of the local row cache that the application would like the driver to use for future `SELECT` statements. If a row cache is not implemented, then setting `RowCacheSize` is ignored and getting the value returns `undef`.

Some `RowCacheSize` values have special meaning, as follows:

```
 0 - Automatically determine a reasonable cache size for each C
↪<SELECT>
 1 - Disable the local row cache
>1 - Cache this many rows
<0 - Cache as many rows that will fit into this much memory for each␣
↪C<SELECT>.
```

Note that large cache sizes may require a very large amount of memory (*cached rows* * *maximum size of row*). Also, a large cache will cause a longer delay not only for the first fetch, but also whenever the cache needs refilling.

See also the "RowsInCache" statement handle attribute.

Username

Type: string

Returns the username used to connect to the database.

DBI STATEMENT HANDLE OBJECTS

This section lists the methods and attributes associated with DBI statement handles.

Statement Handle Methods

The DBI defines the following methods for use on DBI statement handles:

bind_param

```
$sth->bind_param($p_num, $bind_value)
$sth->bind_param($p_num, $bind_value, \%attr)
$sth->bind_param($p_num, $bind_value, $bind_type)
```

The bind_param method takes a copy of $bind_value and associates it (binds it) with a placeholder, identified by $p_num, embedded in the prepared statement. Placeholders are indicated with question mark character (?). For example:

```
$dbh->{RaiseError} = 1;         # save having to check each method␣
↪call
$sth = $dbh->prepare("SELECT name, age FROM people WHERE name LIKE ?
↪");
$sth->bind_param(1, "John%");  # placeholders are numbered from 1
$sth->execute;
DBI::dump_results($sth);
```

See "Placeholders and Bind Values" for more information.

Data Types for Placeholders

The \%attr parameter can be used to hint at the data type the placeholder should have. This is rarely needed. Typically, the driver is only interested in knowing if the placeholder should be bound as a number or a string.

```
$sth->bind_param(1, $value, { TYPE => SQL_INTEGER });
```

As a short-cut for the common case, the data type can be passed directly, in place of the `\%attr` hash reference. This example is equivalent to the one above:

```
$sth->bind_param(1, $value, SQL_INTEGER);
```

The TYPE value indicates the standard (non-driver-specific) type for this parameter. To specify the driver-specific type, the driver may support a driver-specific attribute, such as `{ ora_type => 97 }`.

The SQL_INTEGER and other related constants can be imported using

```
use DBI qw(:sql_types);
```

See "DBI Constants" for more information.

The data type is 'sticky' in that bind values passed to execute() are bound with the data type specified by earlier bind_param() calls, if any. Portable applications should not rely on being able to change the data type after the first `bind_param` call.

Perl only has string and number scalar data types. All database types that aren't numbers are bound as strings and must be in a format the database will understand except where the bind_param() TYPE attribute specifies a type that implies a particular format. For example, given:

```
$sth->bind_param(1, $value, SQL_DATETIME);
```

the driver should expect $value to be in the ODBC standard SQL_DATETIME format, which is 'YYYY-MM-DD HH:MM:SS'. Similarly for SQL_DATE, SQL_TIME etc.

As an alternative to specifying the data type in the `bind_param` call, you can let the driver pass the value as the default type (VARCHAR). You can then use an SQL function to convert the type within the statement. For example:

```
INSERT INTO price(code, price) VALUES (?, CONVERT(MONEY,?))
```

The CONVERT function used here is just an example. The actual function and syntax will vary between different databases and is non-portable.

See also "Placeholders and Bind Values" for more information.

bind_param_inout

```
$rc = $sth->bind_param_inout($p_num, \$bind_value, $max_len)  or die
 ↪$sth->errstr;
$rv = $sth->bind_param_inout($p_num, \$bind_value, $max_len, \%attr)
 ↪    or ...
$rv = $sth->bind_param_inout($p_num, \$bind_value, $max_len, $bind_
 ↪type) or ...
```

This method acts like "bind_param", but also enables values to be updated by the statement. The statement is typically a call to a stored procedure. The `$bind_value` must be passed as a reference to the actual value to be used.

Note that unlike "bind_param", the `$bind_value` variable is not copied when `bind_param_inout` is called. Instead, the value in the variable is read at the time "execute" is called.

The additional `$max_len` parameter specifies the minimum amount of memory to allocate to `$bind_value` for the new value. If the value returned from the database is too big to fit, then the execution should fail. If unsure what value to use, pick a generous length, i.e., a length larger than the longest value that would ever be returned. The only cost of using a larger value than needed is wasted memory.

Undefined values or `undef` are used to indicate null values. See also "Placeholders and Bind Values" for more information.

bind_param_array

```
$rc = $sth->bind_param_array($p_num, $array_ref_or_value)
$rc = $sth->bind_param_array($p_num, $array_ref_or_value, \%attr)
$rc = $sth->bind_param_array($p_num, $array_ref_or_value, $bind_
↪type)
```

The `bind_param_array` method is used to bind an array of values to a placeholder embedded in the prepared statement which is to be executed with "execute_array". For example:

```
$dbh->{RaiseError} = 1;         # save having to check each method␣
↪call
$sth = $dbh->prepare("INSERT INTO staff (first_name, last_name,␣
↪dept) VALUES(?, ?, ?)");
$sth->bind_param_array(1, [ 'John', 'Mary', 'Tim' ]);
$sth->bind_param_array(2, [ 'Booth', 'Todd', 'Robinson' ]);
$sth->bind_param_array(3, "SALES"); # scalar will be reused for each␣
↪row
$sth->execute_array( { ArrayTupleStatus => \my @tuple_status } );
```

The `%attr` ($bind_type) argument is the same as defined for "bind_param". Refer to "bind_param" for general details on using placeholders.

(Note that bind_param_array() can *not* be used to expand a placeholder into a list of values for a statement like "SELECT foo WHERE bar IN (?)". A placeholder can only ever represent one value per execution.)

Scalar values, including `undef`, may also be bound by `bind_param_array`. In which case the same value will be used for each "execute" call. Driver-specific implementations may behave differently, e.g., when binding to a stored procedure call, some databases may permit mixing scalars and arrays as arguments.

The default implementation provided by DBI (for drivers that have not implemented array binding) is to iteratively call "execute" for each parameter tuple provided in the bound arrays. Drivers may provide more optimized implementations using whatever bulk operation support the database API provides. The default driver behaviour should match the default DBI behaviour, but always consult your driver documentation as there may be driver specific issues to consider.

Note that the default implementation currently only supports non-data returning statements (INSERT, UPDATE, but not SELECT). Also, bind_param_array and "bind_param" cannot be mixed in the same statement execution, and bind_param_array must be used with "execute_array"; using bind_param_array will have no effect for "execute".

The bind_param_array method was added in DBI 1.22.

execute

```
$rv = $sth->execute                 or die $sth->errstr;
$rv = $sth->execute(@bind_values)   or die $sth->errstr;
```

Perform whatever processing is necessary to execute the prepared statement. An undef is returned if an error occurs. A successful execute always returns true regardless of the number of rows affected, even if it's zero (see below). It is always important to check the return status of execute (and most other DBI methods) for errors if you're not using "RaiseError".

For a *non*-SELECT statement, execute returns the number of rows affected, if known. If no rows were affected, then execute returns "0E0", which Perl will treat as 0 but will regard as true. Note that it is *not* an error for no rows to be affected by a statement. If the number of rows affected is not known, then execute returns -1.

For SELECT statements, execute simply "starts" the query within the database engine. Use one of the fetch methods to retrieve the data after calling execute. The execute method does *not* return the number of rows that will be returned by the query (because most databases can't tell in advance), it simply returns a true value.

You can tell if the statement was a SELECT statement by checking if $sth->{NUM_OF_FIELDS} is greater than zero after calling execute.

If any arguments are given, then execute will effectively call "bind_param" for each value before executing the statement. Values bound in this way are usually treated as SQL_VARCHAR types unless the driver can determine the correct type (which is rare), or unless bind_param (or bind_param_inout) has already been used to specify the type.

Note that passing execute an empty array is the same as passing no arguments at all, which will execute the statement with previously bound values. That's probably not what you want.

If execute() is called on a statement handle that's still active ($sth->{Active} is true) then it should effectively call finish() to tidy up the previous execution results before starting this new execution.

execute_array

```
$tuples = $sth->execute_array(\%attr) or die $sth->errstr;
$tuples = $sth->execute_array(\%attr, @bind_values) or die $sth->
↪errstr;

($tuples, $rows) = $sth->execute_array(\%attr) or die $sth->errstr;
($tuples, $rows) = $sth->execute_array(\%attr, @bind_values) or die
↪$sth->errstr;
```

Execute the prepared statement once for each parameter tuple (group of values) provided either in the @bind_values, or by prior calls to "bind_param_array", or via a reference passed in \%attr.

When called in scalar context the execute_array() method returns the number of tuples executed, or `undef` if an error occurred. Like execute(), a successful execute_array() always returns true regardless of the number of tuples executed, even if it's zero. If there were any errors the ArrayTupleStatus array can be used to discover which tuples failed and with what errors.

When called in list context the execute_array() method returns two scalars; $tuples is the same as calling execute_array() in scalar context and $rows is the number of rows affected for each tuple, if available or -1 if the driver cannot determine this. NOTE, some drivers cannot determine the number of rows affected per tuple but can provide the number of rows affected for the batch. If you are doing an update operation the returned rows affected may not be what you expect if, for instance, one or more of the tuples affected the same row multiple times. Some drivers may not yet support list context, in which case $rows will be undef, or may not be able to provide the number of rows affected when performing this batch operation, in which case $rows will be -1.

Bind values for the tuples to be executed may be supplied row-wise by an `ArrayTupleFetch` attribute, or else column-wise in the `@bind_values` argument, or else column-wise by prior calls to "bind_param_array".

Where column-wise binding is used (via the `@bind_values` argument or calls to bind_param_array()) the maximum number of elements in any one of the bound value arrays determines the number of tuples executed. Placeholders with fewer values in their parameter arrays are treated as if padded with undef (NULL) values.

If a scalar value is bound, instead of an array reference, it is treated as a *variable* length array with all elements having the same value. It does not influence the number of tuples executed, so if all bound arrays have zero elements then zero tuples will be executed. If *all* bound values are scalars then one tuple will be executed, making execute_array() act just like execute().

The `ArrayTupleFetch` attribute can be used to specify a reference to a subroutine that will be called to provide the bind values for each tuple execution. The subroutine should return an reference to an array which contains the appropriate number of bind values, or return an undef if there is no more data to execute.

As a convenience, the `ArrayTupleFetch` attribute can also be used to specify a statement handle. In which case the fetchrow_arrayref() method will be called on the given statement handle in order to provide the bind values for each tuple execution.

The values specified via bind_param_array() or the @bind_values parameter may be either scalars, or arrayrefs. If any `@bind_values` are given, then `execute_array` will effectively call "bind_param_array" for each value before executing the statement. Values bound in this way are usually treated as `SQL_VARCHAR` types unless the driver can determine the correct type (which is rare), or unless `bind_param`, `bind_param_inout`, `bind_param_array`, or `bind_param_inout_array` has already been used to specify the type. See "bind_param_array" for details.

The `ArrayTupleStatus` attribute can be used to specify a reference to an array which will receive the execute status of each executed parameter tuple. Note the `ArrayTupleStatus` attribute was mandatory until DBI 1.38.

For tuples which are successfully executed, the element at the same ordinal position in the status array is the resulting rowcount (or -1 if unknown). If the execution of a tuple causes an error, then the corresponding status array element will be set to a reference to an array containing “err”, “errstr” and “state” set by the failed execution.

If **any** tuple execution returns an error, `execute_array` will return `undef`. In that case, the application should inspect the status array to determine which parameter tuples failed. Some databases may not continue executing tuples beyond the first failure. In this case the status array will either hold fewer elements, or the elements beyond the failure will be undef.

If all parameter tuples are successfully executed, `execute_array` returns the number tuples executed. If no tuples were executed, then execute_array() returns “`0E0`”, just like execute() does, which Perl will treat as 0 but will regard as true.

For example:

```
$sth = $dbh->prepare("INSERT INTO staff (first_name, last_name)␣
↪VALUES (?, ?)");
my $tuples = $sth->execute_array(
    { ArrayTupleStatus => \my @tuple_status },
    \@first_names,
    \@last_names,
);
if ($tuples) {
    print "Successfully inserted $tuples records\n";
}
else {
    for my $tuple (0..@last_names-1) {
        my $status = $tuple_status[$tuple];
        $status //= [0, "Skipped"];
        next unless ref $status;
        printf "Failed to insert (%s, %s): %s\n",
            $first_names[$tuple], $last_names[$tuple], $status->[1];
    }
}
```

Support for data returning statements such as SELECT is driver-specific and subject to change. At present, the default implementation provided by DBI only supports non-data returning statements.

Transaction semantics when using array binding are driver and database specific. If `AutoCommit` is on, the default DBI implementation will cause each parameter tuple to be individually committed (or rolled back in the event of an error). If `AutoCommit` is off, the application is responsible for explicitly committing the entire set of bound parameter tuples. Note that different drivers and databases may have different behaviours when some parameter tuples cause failures. In some cases, the driver or database may automatically rollback the effect of all prior parameter tuples that succeeded in the transaction; other drivers or databases may retain the effect of prior successfully executed parameter tuples. Be sure to check your driver and database for its specific behaviour.

Note that, in general, performance will usually be better with `AutoCommit` turned off, and using explicit `commit` after each `execute_array` call.

The execute_array method was added in DBI 1.22, and ArrayTupleFetch was added in 1.36.

execute_for_fetch

```
$tuples = $sth->execute_for_fetch($fetch_tuple_sub);
$tuples = $sth->execute_for_fetch($fetch_tuple_sub, \@tuple_status);

($tuples, $rows) = $sth->execute_for_fetch($fetch_tuple_sub);
($tuples, $rows) = $sth->execute_for_fetch($fetch_tuple_sub, \
↪@tuple_status);
```

The execute_for_fetch() method is used to perform bulk operations and although it is most often used via the execute_array() method you can use it directly. The main difference between execute_array and execute_for_fetch is the former does column or row-wise binding and the latter uses row-wise binding.

The fetch subroutine, referenced by $fetch_tuple_sub, is expected to return a reference to an array (known as a 'tuple') or undef.

The execute_for_fetch() method calls $fetch_tuple_sub, without any parameters, until it returns a false value. Each tuple returned is used to provide bind values for an $sth->execute(@$tuple) call.

In scalar context execute_for_fetch() returns undef if there were any errors and the number of tuples executed otherwise. Like execute() and execute_array() a zero is returned as "0E0" so execute_for_fetch() is only false on error. If there were any errors the @tuple_status array can be used to discover which tuples failed and with what errors.

When called in list context execute_for_fetch() returns two scalars; $tuples is the same as calling execute_for_fetch() in scalar context and $rows is the sum of the number of rows affected for each tuple, if available or -1 if the driver cannot determine this. If you are doing an update operation the returned rows affected may not be what you expect if, for instance, one or more of the tuples affected the same row multiple times. Some drivers may not yet support list context, in which case $rows will be undef, or may not be able to provide the number of rows affected when performing this batch operation, in which case $rows will be -1.

If \@tuple_status is passed then the execute_for_fetch method uses it to return status information. The tuple_status array holds one element per tuple. If the corresponding execute() did not fail then the element holds the return value from execute(), which is typically a row count. If the execute() did fail then the element holds a reference to an array containing ($sth->err, $sth->errstr, $sth->state).

If the driver detects an error that it knows means no further tuples can be executed then it may return, with an error status, even though $fetch_tuple_sub may still have more tuples to be executed.

Although each tuple returned by $fetch_tuple_sub is effectively used to call $sth->execute(@$tuple_array_ref) the exact timing may vary. Drivers are free to accumulate sets of tuples to pass to the database server in bulk group operations for more efficient execution. However, the $fetch_tuple_sub is specifically allowed to return the same array reference each time (which is what fetchrow_arrayref() usually does).

For example:

```
my $sel = $dbh1->prepare("select foo, bar from table1");
$sel->execute;

my $ins = $dbh2->prepare("insert into table2 (foo, bar) values (?,?)
 ↪");
my $fetch_tuple_sub = sub { $sel->fetchrow_arrayref };

my @tuple_status;
$rc = $ins->execute_for_fetch($fetch_tuple_sub, \@tuple_status);
my @errors = grep { ref $_ } @tuple_status;
```

Similarly, if you already have an array containing the data rows to be processed you'd use a subroutine to shift off and return each array ref in turn:

```
$ins->execute_for_fetch( sub { shift @array_of_arrays }, \@tuple_
 ↪status);
```

The `execute_for_fetch` method was added in DBI 1.38.

last_insert_id

```
$rv = $sth->last_insert_id();
$rv = $sth->last_insert_id($catalog, $schema, $table, $field);
$rv = $sth->last_insert_id($catalog, $schema, $table, $field, \
 ↪%attr);
```

Returns a value 'identifying' the row inserted by last execution of the statement `$sth`, if possible.

For some drivers the value may be 'identifying' the row inserted by the last executed statement, not by `$sth`.

See database handle method last_insert_id for all details.

The `last_insert_id` statement method was added in DBI 1.642.

fetchrow_arrayref

```
$ary_ref = $sth->fetchrow_arrayref;
$ary_ref = $sth->fetch;    # alias
```

Fetches the next row of data and returns a reference to an array holding the field values. Null fields are returned as `undef` values in the array. This is the fastest way to fetch data, particularly if used with `$sth->bind_columns`.

If there are no more rows or if an error occurs, then `fetchrow_arrayref` returns an `undef`. You should check `$sth->err` afterwards (or use the `RaiseError` attribute) to discover if the `undef` returned was due to an error.

Note that the same array reference is returned for each fetch, so don't store the reference and then use it after a later fetch. Also, the elements of the array are also reused for each row, so take care if you want to take a reference to an element. See also "bind_columns".

fetchrow_array

```
@ary = $sth->fetchrow_array;
```

An alternative to `fetchrow_arrayref`. Fetches the next row of data and returns it as a list containing the field values. Null fields are returned as `undef` values in the list.

If there are no more rows or if an error occurs, then `fetchrow_array` returns an empty list. You should check `$sth->err` afterwards (or use the `RaiseError` attribute) to discover if the empty list returned was due to an error.

If called in a scalar context for a statement handle that has more than one column, it is undefined whether the driver will return the value of the first column or the last. So don't do that. Also, in a scalar context, an `undef` is returned if there are no more rows or if an error occurred. That `undef` can't be distinguished from an `undef` returned because the first field value was NULL. For these reasons you should exercise some caution if you use `fetchrow_array` in a scalar context.

fetchrow_hashref

```
$hash_ref = $sth->fetchrow_hashref;
$hash_ref = $sth->fetchrow_hashref($name);
```

An alternative to `fetchrow_arrayref`. Fetches the next row of data and returns it as a reference to a hash containing field name and field value pairs. Null fields are returned as `undef` values in the hash.

If there are no more rows or if an error occurs, then `fetchrow_hashref` returns an `undef`. You should check `$sth->err` afterwards (or use the `RaiseError` attribute) to discover if the `undef` returned was due to an error.

The optional `$name` parameter specifies the name of the statement handle attribute. For historical reasons it defaults to "`NAME`", however using either "`NAME_lc`" or "`NAME_uc`" is recommended for portability.

The keys of the hash are the same names returned by `$sth->{$name}`. If more than one field has the same name, there will only be one entry in the returned hash for those fields, so statements like "`select foo, foo from bar`" will return only a single key from `fetchrow_hashref`. In these cases use column aliases or `fetchrow_arrayref`. Note that it is the database server (and not the DBD implementation) which provides the *name* for fields containing functions like "`count(*)`" or "`max(c_foo)`" and they may clash with existing column names (most databases don't care about duplicate column names in a result-set). If you want these to return as unique names that are the same across databases, use *aliases*, as in "`select count(*) as cnt`" or "`select max(c_foo) mx_foo, ...`" depending on the syntax your database supports.

Because of the extra work `fetchrow_hashref` and Perl have to perform, it is not as efficient as `fetchrow_arrayref` or `fetchrow_array`.

By default a reference to a new hash is returned for each row. It is likely that a future version of the DBI will support an attribute which will enable the same hash to be reused for each row. This will give a significant performance boost, but it won't be enabled by default because of the risk of breaking old code.

fetchall_arrayref

```
$tbl_ary_ref = $sth->fetchall_arrayref;
$tbl_ary_ref = $sth->fetchall_arrayref( $slice );
$tbl_ary_ref = $sth->fetchall_arrayref( $slice, $max_rows  );
```

The `fetchall_arrayref` method can be used to fetch all the data to be returned from a prepared and executed statement handle. It returns a reference to an array that contains one reference per row.

If called on an *inactive* statement handle, `fetchall_arrayref` returns undef.

If there are no rows left to return from an *active* statement handle, `fetchall_arrayref` returns a reference to an empty array. If an error occurs, `fetchall_arrayref` returns the data fetched thus far, which may be none. You should check `$sth->err` afterwards (or use the `RaiseError` attribute) to discover if the data is complete or was truncated due to an error.

If $slice is an array reference, `fetchall_arrayref` uses "fetchrow_arrayref" to fetch each row as an array ref. If the $slice array is not empty then it is used as a slice to select individual columns by perl array index number (starting at 0, unlike column and parameter numbers which start at 1).

With no parameters, or if $slice is undefined, `fetchall_arrayref` acts as if passed an empty array ref.

For example, to fetch just the first column of every row:

```
$tbl_ary_ref = $sth->fetchall_arrayref([0]);
```

To fetch the second to last and last column of every row:

```
$tbl_ary_ref = $sth->fetchall_arrayref([-2,-1]);
```

Those two examples both return a reference to an array of array refs.

If $slice is a hash reference, `fetchall_arrayref` fetches each row as a hash reference. If the $slice hash is empty then the keys in the hashes have whatever name lettercase is returned by default. (See "FetchHashKeyName" attribute.) If the $slice hash is *not* empty, then it is used as a slice to select individual columns by name. The values of the hash should be set to 1. The key names of the returned hashes match the letter case of the names in the parameter hash, regardless of the "FetchHashKeyName" attribute.

For example, to fetch all fields of every row as a hash ref:

```
$tbl_ary_ref = $sth->fetchall_arrayref({});
```

To fetch only the fields called "foo" and "bar" of every row as a hash ref (with keys named "foo" and "BAR", regardless of the original capitalization):

```
$tbl_ary_ref = $sth->fetchall_arrayref({ foo=>1, BAR=>1 });
```

Those two examples both return a reference to an array of hash refs.

If $slice is a *reference to a hash reference*, that hash is used to select and rename columns. The keys are 0-based column index numbers and the values are the corresponding keys

for the returned row hashes.

For example, to fetch only the first and second columns of every row as a hash ref (with keys named "k" and "v" regardless of their original names):

```
$tbl_ary_ref = $sth->fetchall_arrayref( \{ 0 => 'k', 1 => 'v' } );
```

If $max_rows is defined and greater than or equal to zero then it is used to limit the number of rows fetched before returning. fetchall_arrayref() can then be called again to fetch more rows. This is especially useful when you need the better performance of fetchall_arrayref() but don't have enough memory to fetch and return all the rows in one go.

Here's an example (assumes RaiseError is enabled):

```
my $rows = []; # cache for batches of rows
while( my $row = ( shift(@$rows) || # get row from cache, or reload␣
 ↪cache:
                    shift(@{$rows=$sth->fetchall_arrayref(undef,10_
 ↪000)||[]}) )
) {
  ...
}
```

That *might* be the fastest way to fetch and process lots of rows using the DBI, but it depends on the relative cost of method calls vs memory allocation.

A standard `while` loop with column binding is often faster because the cost of allocating memory for the batch of rows is greater than the saving by reducing method calls. It's possible that the DBI may provide a way to reuse the memory of a previous batch in future, which would then shift the balance back towards fetchall_arrayref().

fetchall_hashref

```
$hash_ref = $sth->fetchall_hashref($key_field);
```

The `fetchall_hashref` method can be used to fetch all the data to be returned from a prepared and executed statement handle. It returns a reference to a hash containing a key for each distinct value of the $key_field column that was fetched. For each key the corresponding value is a reference to a hash containing all the selected columns and their values, as returned by `fetchrow_hashref()`.

If there are no rows to return, `fetchall_hashref` returns a reference to an empty hash. If an error occurs, `fetchall_hashref` returns the data fetched thus far, which may be none. You should check `$sth->err` afterwards (or use the `RaiseError` attribute) to discover if the data is complete or was truncated due to an error.

The $key_field parameter provides the name of the field that holds the value to be used for the key for the returned hash. For example:

```
$dbh->{FetchHashKeyName} = 'NAME_lc';
$sth = $dbh->prepare("SELECT FOO, BAR, ID, NAME, BAZ FROM TABLE");
$sth->execute;
```

(continues on next page)

(continued from previous page)

```
$hash_ref = $sth->fetchall_hashref('id');
print "Name for id 42 is $hash_ref->{42}->{name}\n";
```

The $key_field parameter can also be specified as an integer column number (counting from 1). If $key_field doesn't match any column in the statement, as a name first then as a number, then an error is returned.

For queries returning more than one 'key' column, you can specify multiple column names by passing $key_field as a reference to an array containing one or more key column names (or index numbers). For example:

```
$sth = $dbh->prepare("SELECT foo, bar, baz FROM table");
$sth->execute;
$hash_ref = $sth->fetchall_hashref( [ qw(foo bar) ] );
print "For foo 42 and bar 38, baz is $hash_ref->{42}->{38}->{baz}\n
 ↪";
```

The fetchall_hashref() method is normally used only where the key fields values for each row are unique. If multiple rows are returned with the same values for the key fields then later rows overwrite earlier ones.

more_results

... not yet documented ...

finish

```
$rc  = $sth->finish;
```

Indicate that no more data will be fetched from this statement handle before it is either executed again or destroyed. You almost certainly do *not* need to call this method.

Adding calls to `finish` after loop that fetches all rows is a common mistake, don't do it, it can mask genuine problems like uncaught fetch errors.

When all the data has been fetched from a `SELECT` statement, the driver will automatically call `finish` for you. So you should *not* call it explicitly *except* when you know that you've not fetched all the data from a statement handle *and* the handle won't be destroyed soon.

The most common example is when you only want to fetch just one row, but in that case the `selectrow_*` methods are usually better anyway.

Consider a query like:

```
SELECT foo FROM table WHERE bar=? ORDER BY baz
```

on a very large table. When executed, the database server will have to use temporary buffer space to store the sorted rows. If, after executing the handle and selecting just a few rows, the handle won't be re-executed for some time and won't be destroyed, the `finish` method can be used to tell the server that the buffer space can be freed.

Calling `finish` resets the "Active" attribute for the statement. It may also make some statement handle attributes (such as `NAME` and `TYPE`) unavailable if they have not already been accessed (and thus cached).

The `finish` method does not affect the transaction status of the database connection. It has nothing to do with transactions. It's mostly an internal "housekeeping" method that is rarely needed. See also "disconnect" and the "Active" attribute.

The `finish` method should have been called `discard_pending_rows`.

rows

```
$rv = $sth->rows;
```

Returns the number of rows affected by the last row affecting command, or -1 if the number of rows is not known or not available.

Generally, you can only rely on a row count after a *non*-`SELECT` `execute` (for some specific operations like `UPDATE` and `DELETE`), or after fetching all the rows of a `SELECT` statement.

For `SELECT` statements, it is generally not possible to know how many rows will be returned except by fetching them all. Some drivers will return the number of rows the application has fetched so far, but others may return -1 until all rows have been fetched. So use of the `rows` method or `$DBI::rows` with `SELECT` statements is not recommended.

One alternative method to get a row count for a `SELECT` is to execute a "SELECT COUNT(*) FROM ..." SQL statement with the same "..." as your query and then fetch the row count from that.

bind_col

```
$rc = $sth->bind_col($column_number, \$var_to_bind);
$rc = $sth->bind_col($column_number, \$var_to_bind, \%attr );
$rc = $sth->bind_col($column_number, \$var_to_bind, $bind_type );
```

Binds a Perl variable and/or some attributes to an output column (field) of a `SELECT` statement. Column numbers count up from 1. You do not need to bind output columns in order to fetch data. For maximum portability between drivers, bind_col() should be called after execute() and not before. See also "bind_columns" for an example.

The binding is performed at a low level using Perl aliasing. Whenever a row is fetched from the database $var_to_bind appears to be automatically updated simply because it now refers to the same memory location as the corresponding column value. This makes using bound variables very efficient. Binding a tied variable doesn't work, currently.

The "bind_param" method performs a similar, but opposite, function for input variables.

Data Types for Column Binding

The `\%attr` parameter can be used to hint at the data type formatting the column should have. For example, you can use:

```
$sth->bind_col(1, undef, { TYPE => SQL_DATETIME });
```

to specify that you'd like the column (which presumably is some kind of datetime type) to be returned in the standard format for SQL_DATETIME, which is 'YYYY-MM-DD HH:MM:SS', rather than the native formatting the database would normally use.

There's no $var_to_bind in that example to emphasize the point that bind_col() works on the underlying column and not just a particular bound variable.

As a short-cut for the common case, the data type can be passed directly, in place of the `\%attr` hash reference. This example is equivalent to the one above:

```
$sth->bind_col(1, undef, SQL_DATETIME);
```

The TYPE value indicates the standard (non-driver-specific) type for this parameter. To specify the driver-specific type, the driver may support a driver-specific attribute, such as `{ ora_type => 97 }`.

The SQL_DATETIME and other related constants can be imported using

```
use DBI qw(:sql_types);
```

See "DBI Constants" for more information.

Few drivers support specifying a data type via a `bind_col` call (most will simply ignore the data type). Fewer still allow the data type to be altered once set. If you do set a column type the type should remain sticky through further calls to bind_col for the same column if the type is not overridden (this is important for instance when you are using a slice in fetchall_arrayref).

The TYPE attribute for bind_col() was first specified in DBI 1.41.

From DBI 1.611, drivers can use the TYPE attribute to attempt to cast the bound scalar to a perl type which more closely matches TYPE. At present DBI supports `SQL_INTEGER`, `SQL_DOUBLE` and `SQL_NUMERIC`. See "sql_type_cast" for details of how types are cast.

Other attributes for Column Binding

The `\%attr` parameter may also contain the following attributes:

- `StrictlyTyped`

  If a TYPE attribute is passed to bind_col, then the driver will attempt to change the bound perl scalar to match the type more closely. If the bound value cannot be cast to the requested TYPE then by default it is left untouched and no error is generated. If you specify `StrictlyTyped` as 1 and the cast fails, this will generate an error.

  This attribute was first added in DBI 1.611. When 1.611 was released few drivers actually supported this attribute but DBD::Oracle and DBD::ODBC should from versions 1.24.

- `DiscardString`

  When the TYPE attribute is passed to "bind_col" and the driver successfully casts the bound perl scalar to a non-string type then if `DiscardString` is set to 1, the string portion of the scalar will be discarded. By default, `DiscardString` is not set.

  This attribute was first added in DBI 1.611. When 1.611 was released few drivers actually supported this attribute but DBD::Oracle and DBD::ODBC should from versions 1.24.

bind_columns

```
$rc = $sth->bind_columns(@list_of_refs_to_vars_to_bind);
```

Calls “bind_col” for each column of the SELECT statement.

The list of references should have the same number of elements as the number of columns in the SELECT statement. If it doesn’t then bind_columns will bind the elements given, up to the number of columns, and then return an error.

For maximum portability between drivers, bind_columns() should be called after execute() and not before.

For example:

```
$dbh->{RaiseError} = 1; # do this, or check every call for errors
$sth = $dbh->prepare(q{ SELECT region, sales FROM sales_by_region }
 ↪);
$sth->execute;
my ($region, $sales);

# Bind Perl variables to columns:
$rv = $sth->bind_columns(\$region, \$sales);

# you can also use Perl's \(...) syntax (see perlref docs):
#     $sth->bind_columns(\($region, $sales));

# Column binding is the most efficient way to fetch data
while ($sth->fetch) {
    print "$region: $sales\n";
}
```

For compatibility with old scripts, the first parameter will be ignored if it is undef or a hash reference.

Here’s a more fancy example that binds columns to the values *inside* a hash (thanks to H.Merijn Brand):

```
$sth->execute;
my %row;
$sth->bind_columns (\( @row{ @{$sth->{NAME_lc} }} ));
while ($sth->fetch) {
    print "$row{region}: $row{sales}\n";
}
```

but has a small drawback: If the data has already been fetched, the call to “bind_columns” will flush current values. If you want to bind_columns after you have fetched, you can use:

```
use feature "refaliasing";
no warnings "experimental::refaliasing";
while (my $row = $sth->fetchrow_arrayref) {
    \(@$data{ $sth->{NAME_lc}->@* }) =  \(@$row);
}
```

or, with older perl versions:

```
use Data::Alias;
alias @$data{ $sth->{NAME_lc}->@* } =  @$row;
```

This is useful in situations when you have many left joins, but want to join your %$data hash to only subset of fetched values.

dump_results

```
$rows = $sth->dump_results($maxlen, $lsep, $fsep, $fh);
```

Fetches all the rows from `$sth`, calls `DBI::neat_list` for each row, and prints the results to `$fh` (defaults to `STDOUT`) separated by `$lsep` (default `"\n"`). `$fsep` defaults to `", "` and `$maxlen` defaults to 35.

This method is designed as a handy utility for prototyping and testing queries. Since it uses "neat_list" to format and edit the string for reading by humans, it is not recommended for data transfer applications.

Statement Handle Attributes

This section describes attributes specific to statement handles. Most of these attributes are read-only.

Changes to these statement handle attributes do not affect any other existing or future statement handles.

Attempting to set or get the value of an unknown attribute generates a warning, except for private driver specific attributes (which all have names starting with a lowercase letter).

Example:

```
... = $h->{NUM_OF_FIELDS};    # get/read
```

Some drivers cannot provide valid values for some or all of these attributes until after `$sth->execute` has been successfully called. Typically the attribute will be `undef` in these situations.

Some attributes, like NAME, are not appropriate to some types of statement, like SELECT. Typically the attribute will be `undef` in these situations.

For drivers which support stored procedures and multiple result sets (see "more_results") these attributes relate to the *current* result set.

See also "finish" to learn more about the effect it may have on some attributes.

NUM_OF_FIELDS

Type: integer, read-only

Number of fields (columns) in the data the prepared statement may return. Statements that don't return rows of data, like `DELETE` and `CREATE` set `NUM_OF_FIELDS` to 0 (though it may be undef in some drivers).

NUM_OF_PARAMS

Type: integer, read-only

The number of parameters (placeholders) in the prepared statement. See SUBSTITUTION VARIABLES below for more details.

NAME

Type: array-ref, read-only

Returns a reference to an array of field names for each column. The names may contain spaces but should not be truncated or have any trailing space. Note that the names have the letter case (upper, lower or mixed) as returned by the driver being used. Portable applications should use "NAME_lc" or "NAME_uc".

```
print "First column name: $sth->{NAME}->[0]\n";
```

Also note that the name returned for (aggregate) functions like `count(*)` or `max(c_foo)` is determined by the database server and not by `DBI` or the `DBD` backend.

NAME_lc

Type: array-ref, read-only

Like `/NAME` but always returns lowercase names.

NAME_uc

Type: array-ref, read-only

Like `/NAME` but always returns uppercase names.

NAME_hash

Type: hash-ref, read-only

NAME_lc_hash

Type: hash-ref, read-only

NAME_uc_hash

Type: hash-ref, read-only

The `NAME_hash`, `NAME_lc_hash`, and `NAME_uc_hash` attributes return column name information as a reference to a hash.

The keys of the hash are the names of the columns. The letter case of the keys corresponds to the letter case returned by the `NAME`, `NAME_lc`, and `NAME_uc` attributes respectively (as described above).

The value of each hash entry is the perl index number of the corresponding column (counting from 0). For example:

```
$sth = $dbh->prepare("select Id, Name from table");
$sth->execute;
@row = $sth->fetchrow_array;
print "Name $row[ $sth->{NAME_lc_hash}{name} ]\n";
```

TYPE

Type: array-ref, read-only

Returns a reference to an array of integer values for each column. The value indicates the data type of the corresponding column.

The values correspond to the international standards (ANSI X3.135 and ISO/IEC 9075) which, in general terms, means ODBC. Driver-specific types that don't exactly match standard types should generally return the same values as an ODBC driver supplied by the makers of the database. That might include private type numbers in ranges the vendor has officially registered with the ISO working group:

```
ftp://sqlstandards.org/SC32/SQL_Registry/
```

Where there's no vendor-supplied ODBC driver to be compatible with, the DBI driver can use type numbers in the range that is now officially reserved for use by the DBI: -9999 to -9000.

All possible values for `TYPE` should have at least one entry in the output of the `type_info_all` method (see "type_info_all").

PRECISION

Type: array-ref, read-only

Returns a reference to an array of integer values for each column.

For numeric columns, the value is the maximum number of digits (without considering a sign character or decimal point). Note that the "display size" for floating point types (REAL, FLOAT, DOUBLE) can be up to 7 characters greater than the precision (for the sign + decimal point + the letter E + a sign + 2 or 3 digits).

For any character type column the value is the OCTET_LENGTH, in other words the number of bytes, not characters.

(More recent standards refer to this as COLUMN_SIZE but we stick with PRECISION for backwards compatibility.)

SCALE

Type: array-ref, read-only

Returns a reference to an array of integer values for each column. NULL (`undef`) values indicate columns where scale is not applicable.

NULLABLE

Type: array-ref, read-only

Returns a reference to an array indicating the possibility of each column returning a null. Possible values are `0` (or an empty string) = no, 1 = yes, 2 = unknown.

```
print "First column may return NULL\n" if $sth->{NULLABLE}->[0];
```

CursorName

Type: string, read-only

Returns the name of the cursor associated with the statement handle, if available. If not available or if the database driver does not support the "`where current of ...`" SQL syntax, then it returns `undef`.

Database

Type: dbh, read-only

Returns the parent $dbh of the statement handle.

Statement

Type: string, read-only

Returns the statement string passed to the “prepare” method.

ParamValues

Type: hash ref, read-only

Returns a reference to a hash containing the values currently bound to placeholders. The keys of the hash are the ‘names’ of the placeholders, typically integers starting at 1. Returns undef if not supported by the driver.

See “ShowErrorStatement” for an example of how this is used.

* Keys:

If the driver supports `ParamValues` but no values have been bound yet then the driver should return a hash with placeholders names in the keys but all the values undef, but some drivers may return a ref to an empty hash because they can’t pre-determine the names.

It is possible that the keys in the hash returned by `ParamValues` are not exactly the same as those implied by the prepared statement. For example, DBD::Oracle translates ‘?’ placeholders into ‘:pN’ where N is a sequence number starting at 1 with a hard limit of 99999.

* Values:

It is possible that the values in the hash returned by `ParamValues` are not *exactly* the same as those passed to bind_param() or execute(). The driver may have slightly modified values in some way based on the TYPE the value was bound with. For example a

floating point value bound as an SQL_INTEGER type may be returned as an integer. The values returned by `ParamValues` can be passed to another bind_param() method with the same TYPE and will be seen by the database as the same value. See also "ParamTypes" below.

The `ParamValues` attribute was added in DBI 1.28.

ParamTypes

Type: hash ref, read-only

Returns a reference to a hash containing the type information currently bound to placeholders. Returns undef if not supported by the driver.

* Keys:

See "ParamValues" above.

* Values:

The hash values are hashrefs of type information in the same form as that passed to the various bind_param() methods (See "bind_param" for the format and values).

It is possible that the values in the hash returned by `ParamTypes` are not exactly the same as those passed to bind_param() or execute(). Param attributes specified using the abbreviated form, like this:

```
$sth->bind_param(1, SQL_INTEGER);
```

are returned in the expanded form, as if called like this:

```
$sth->bind_param(1, { TYPE => SQL_INTEGER });
```

The driver may have modified the type information in some way based on the bound values, other hints provided by the prepare()'d SQL statement, or alternate type mappings required by the driver or target database system. The driver may also add private keys (with names beginning with the drivers reserved prefix, e.g., odbc_xxx).

* Example:

The keys and values in the returned hash can be passed to the various bind_param() methods to effectively reproduce a previous param binding. For example:

```
# assuming $sth1 is a previously prepared statement handle
my $sth2 = $dbh->prepare( $sth1->{Statement} );
my $ParamValues = $sth1->{ParamValues} || {};
my $ParamTypes  = $sth1->{ParamTypes}  || {};
$sth2->bind_param($_, $ParamValues->{$_}, $ParamTypes->{$_})
  for keys %{ {%$ParamValues, %$ParamTypes} };
$sth2->execute();
```

The `ParamTypes` attribute was added in DBI 1.49. Implementation is the responsibility of individual drivers; the DBI layer default implementation simply returns undef.

ParamArrays

Type: hash ref, read-only

Returns a reference to a hash containing the values currently bound to placeholders with "execute_array" or "bind_param_array". The keys of the hash are the 'names' of the placeholders, typically integers starting at 1. Returns undef if not supported by the driver or no arrays of parameters are bound.

Each key value is an array reference containing a list of the bound parameters for that column.

For example:

```
$sth = $dbh->prepare("INSERT INTO staff (id, name) values (?,?)");
$sth->execute_array({},[1,2], ['fred','dave']);
if ($sth->{ParamArrays}) {
    foreach $param (keys %{$sth->{ParamArrays}}) {
        printf "Parameters for %s : %s\n", $param,
        join(",", @{$sth->{ParamArrays}->{$param}});
    }
}
```

It is possible that the values in the hash returned by `ParamArrays` are not *exactly* the same as those passed to "bind_param_array" or "execute_array". The driver may have slightly modified values in some way based on the TYPE the value was bound with. For example a floating point value bound as an SQL_INTEGER type may be returned as an integer.

It is also possible that the keys in the hash returned by `ParamArrays` are not exactly the same as those implied by the prepared statement. For example, DBD::Oracle translates '?' placeholders into ':pN' where N is a sequence number starting at 1 with a hard limit of 99999.

RowsInCache

Type: integer, read-only

If the driver supports a local row cache for `SELECT` statements, then this attribute holds the number of un-fetched rows in the cache. If the driver doesn't, then it returns `undef`. Note that some drivers pre-fetch rows on execute, whereas others wait till the first fetch.

See also the "RowCacheSize" database handle attribute.

FURTHER INFORMATION

Catalog Methods

An application can retrieve metadata information from the DBMS by issuing appropriate queries on the views of the Information Schema. Unfortunately, `INFORMATION_SCHEMA` views are seldom supported by the DBMS. Special methods (catalog methods) are available to return result sets for a small but important portion of that metadata:

```
column_info
foreign_key_info
primary_key_info
table_info
statistics_info
```

All catalog methods accept arguments in order to restrict the result sets. Passing `undef` to an optional argument does not constrain the search for that argument. However, an empty string ('') is treated as a regular search criteria and will only match an empty value.

Note: SQL/CLI and ODBC differ in the handling of empty strings. An empty string will not restrict the result set in SQL/CLI.

Most arguments in the catalog methods accept only *ordinary values*, e.g. the arguments of `primary_key_info()`. Such arguments are treated as a literal string, i.e. the case is significant and quote characters are taken literally.

Some arguments in the catalog methods accept *search patterns* (strings containing '_' and/or '%'), e.g. the `$table` argument of `column_info()`. Passing '%' is equivalent to leaving the argument `undef`.

Caveat: The underscore ('_') is valid and often used in SQL identifiers. Passing such a value to a search pattern argument may return more rows than expected! To include pattern characters as literals, they must be preceded by an escape character which can be achieved with

```
$esc = $dbh->get_info( 14 );  # SQL_SEARCH_PATTERN_ESCAPE
$search_pattern =~ s/([_%])/$esc$1/g;
```

The ODBC and SQL/CLI specifications define a way to change the default behaviour described above: All arguments (except *list value arguments*) are treated as *identifier* if the `SQL_ATTR_METADATA_ID` attribute is set to `SQL_TRUE`. *Quoted identifiers* are very similar to *ordinary values*, i.e. their body (the string within the quotes) is interpreted literally. *Unquoted identifiers* are compared in UPPERCASE.

The DBI (currently) does not support the `SQL_ATTR_METADATA_ID` attribute, i.e. it behaves like an ODBC driver where `SQL_ATTR_METADATA_ID` is set to `SQL_FALSE`.

Transactions

Transactions are a fundamental part of any robust database system. They protect against errors and database corruption by ensuring that sets of related changes to the database take place in atomic (indivisible, all-or-nothing) units.

This section applies to databases that support transactions and where `AutoCommit` is off. See "AutoCommit" for details of using `AutoCommit` with various types of databases.

The recommended way to implement robust transactions in Perl applications is to enable "RaiseError" and catch the error that's 'thrown' as an exception. For example, using Try::Tiny:

```
use Try::Tiny;
$dbh->{AutoCommit} = 0;  # enable transactions, if possible
```

(continues on next page)

(continued from previous page)

```
$dbh->{RaiseError} = 1;
try {
    foo(...)        # do lots of work here
    bar(...)        # including inserts
    baz(...)        # and updates
    $dbh->commit;   # commit the changes if we get this far
} catch {
    warn "Transaction aborted because $_"; # Try::Tiny copies $@ into $_
    # now rollback to undo the incomplete changes
    # but do it in an eval{} as it may also fail
    eval { $dbh->rollback };
    # add other application on-error-clean-up code here
};
```

If the `RaiseError` attribute is not set, then DBI calls would need to be manually checked for errors, typically like this:

```
$h->method(@args) or die $h->errstr;
```

With `RaiseError` set, the DBI will automatically `die` if any DBI method call on that handle (or a child handle) fails, so you don't have to test the return value of each method call. See "RaiseError" for more details.

A major advantage of the `eval` approach is that the transaction will be properly rolled back if *any* code (not just DBI calls) in the inner application dies for any reason. The major advantage of using the `$h->{RaiseError}` attribute is that all DBI calls will be checked automatically. Both techniques are strongly recommended.

After calling `commit` or `rollback` many drivers will not let you fetch from a previously active `SELECT` statement handle that's a child of the same database handle. A typical way round this is to connect the the database twice and use one connection for `SELECT` statements.

See "AutoCommit" and "disconnect" for other important information about transactions.

Handling BLOB / LONG / Memo Fields

Many databases support "blob" (binary large objects), "long", or similar datatypes for holding very long strings or large amounts of binary data in a single field. Some databases support variable length long values over 2,000,000,000 bytes in length.

Since values of that size can't usually be held in memory, and because databases can't usually know in advance the length of the longest long that will be returned from a `SELECT` statement (unlike other data types), some special handling is required.

In this situation, the value of the `$h->{LongReadLen}` attribute is used to determine how much buffer space to allocate when fetching such fields. The `$h->{LongTruncOk}` attribute is used to determine how to behave if a fetched value can't fit into the buffer.

See the description of "LongReadLen" for more information.

When trying to insert long or binary values, placeholders should be used since there are often limits on the maximum size of an `INSERT` statement and the "quote" method

generally can't cope with binary data. See "Placeholders and Bind Values".

Simple Examples

Here's a complete example program to select and fetch some data:

```
my $data_source = "dbi::DriverName:db_name";
my $dbh = DBI->connect($data_source, $user, $password)
    or die "Can't connect to $data_source: $DBI::errstr";

my $sth = $dbh->prepare( q{
        SELECT name, phone
        FROM mytelbook
}) or die "Can't prepare statement: $DBI::errstr";

my $rc = $sth->execute
    or die "Can't execute statement: $DBI::errstr";

print "Query will return $sth->{NUM_OF_FIELDS} fields.\n\n";
print "Field names: @{ $sth->{NAME} }\n";

while (($name, $phone) = $sth->fetchrow_array) {
    print "$name: $phone\n";
}
# check for problems which may have terminated the fetch early
die $sth->errstr if $sth->err;

$dbh->disconnect;
```

Here's a complete example program to insert some data from a file. (This example uses `RaiseError` to avoid needing to check each call).

```
my $dbh = DBI->connect("dbi:DriverName:db_name", $user, $password, {
    RaiseError => 1, AutoCommit => 0
});

my $sth = $dbh->prepare( q{
    INSERT INTO table (name, phone) VALUES (?, ?)
});

open FH, "<phone.csv" or die "Unable to open phone.csv: $!";
while (<FH>) {
    chomp;
    my ($name, $phone) = split /,/;
    $sth->execute($name, $phone);
}
close FH;

$dbh->commit;
$dbh->disconnect;
```

Here's how to convert fetched NULLs (undefined values) into empty strings:

```
while($row = $sth->fetchrow_arrayref) {
  # this is a fast and simple way to deal with nulls:
  $_ //= '' for @$row;
  print "@$row\n";
}
```

The q{...} style quoting used in these examples avoids clashing with quotes that may be used in the SQL statement. Use the double-quote like qq{...} operator if you want to interpolate variables into the string. See *"Quote and Quote-like Operators" in perlop* for more details.

Threads and Thread Safety

Perl 5.7 and later support a new threading model called iThreads. (The old "5.005 style" threads are not supported by the DBI.)

In the iThreads model each thread has its own copy of the perl interpreter. When a new thread is created the original perl interpreter is 'cloned' to create a new copy for the new thread.

If the DBI and drivers are loaded and handles created before the thread is created then it will get a cloned copy of the DBI, the drivers and the handles.

However, the internal pointer data within the handles will refer to the DBI and drivers in the original interpreter. Using those handles in the new interpreter thread is not safe, so the DBI detects this and croaks on any method call using handles that don't belong to the current thread (except for DESTROY).

Because of this (possibly temporary) restriction, newly created threads must make their own connections to the database. Handles can't be shared across threads.

But BEWARE, some underlying database APIs (the code the DBD driver uses to talk to the database, often supplied by the database vendor) are not thread safe. If it's not thread safe, then allowing more than one thread to enter the code at the same time may cause subtle/serious problems. In some cases allowing more than one thread to enter the code, even if *not* at the same time, can cause problems. You have been warned.

Using DBI with perl threads is not yet recommended for production environments. For more information see https://www.perlmonks.org/index.pl?node_id=288022

Signal Handling and Canceling Operations

[The following only applies to systems with unix-like signal handling. I'd welcome additions for other systems, especially Windows.]

Beginning in perl 5.8.0 perl implements 'safe' signal handling if your system has the POSIX sigaction() routine. Now when a signal is delivered perl just makes a note of it but does *not* run the

Although this change made signal handling safe, it also lead to a problem with signals being deferred for longer than you'd like. If a signal arrived while executing a system call, such as waiting for data on a network connection, the signal is noted and then the system call that was executing returns with an EINTR error code to indicate that it was interrupted. All fine so far.

The problem comes when the code that made the system call sees the EINTR code and decides it's going to call it again. Perl doesn't do that, but database code sometimes does. If that happens then the signal handler doesn't get called until later. Maybe much later.

Fortunately there are ways around this which we'll discuss below. Unfortunately they make signals unsafe again.

The two most common uses of signals in relation to the DBI are for canceling operations when the user types Ctrl-C (interrupt), and for implementing a timeout using `alarm()` and `$SIG{ALRM}`.

- Cancel

  The DBI provides a `cancel` method for statement handles. The `cancel` method should abort the current operation and is designed to be called from a signal handler. For example:

```
$SIG{INT} = sub { $sth->cancel };
```

  However, few drivers implement this (the DBI provides a default method that just returns `undef`) and, even if implemented, there is still a possibility that the statement handle, and even the parent database handle, will not be usable afterwards.

  If `cancel` returns true, then it has successfully invoked the database engine's own cancel function. If it returns false, then `cancel` failed. If it returns `undef`, then the database driver does not have cancel implemented - very few do.

- Timeout

  The traditional way to implement a timeout is to set `$SIG{ALRM}` to refer to some code that will be executed when an ALRM signal arrives and then to call alarm($seconds) to schedule an ALRM signal to be delivered $seconds in the future. For example:

```
my $failed;
eval {
  local $SIG{ALRM} = sub { die "TIMEOUT\n" }; # N.B. \n required
  eval {
    alarm($seconds);
    ... code to execute with timeout here (which may die) ...
    1;
  } or $failed = 1;
  # outer eval catches alarm that might fire JUST before this
 ↪alarm(0)
  alarm(0);  # cancel alarm (if code ran fast)
  die "$@" if $failed;
  1;
} or $failed = 1;
if ( $failed ) {
  if ( defined $@ and $@ eq "TIMEOUT\n" ) { ... }
  else { ... } # some other error
}
```

  The first (outer) eval is used to avoid the unlikely but possible chance that the "code to execute" dies and the alarm fires before it is cancelled. Without the outer eval,

if this happened your program will die if you have no ALRM handler or a non-local alarm handler will be called.

Unfortunately, as described above, this won't always work as expected, depending on your perl version and the underlying database code.

With Oracle for instance (DBD::Oracle), if the system which hosts the database is down the DBI->connect() call will hang for several minutes before returning an error.

The solution on these systems is to use the `POSIX::sigaction()` routine to gain low level access to how the signal handler is installed.

The code would look something like this (for the DBD-Oracle connect()):

```
use POSIX qw(:signal_h);

my $mask = POSIX::SigSet->new( SIGALRM ); # signals to mask in the
↪handler
my $action = POSIX::SigAction->new(
    sub { die "connect timeout\n" },        # the handler code ref
    $mask,
    # not using (perl 5.8.2 and later) 'safe' switch or sa_flags
);
my $oldaction = POSIX::SigAction->new();
sigaction( SIGALRM, $action, $oldaction );
my $dbh;
my $failed;
eval {
   eval {
     alarm(5); # seconds before time out
     $dbh = DBI->connect("dbi:Oracle:$dsn" ... );
     1;
   } or $failed = 1;
   alarm(0); # cancel alarm (if connect worked fast)
   die "$@\n" if $failed; # connect died
   1;
} or $failed = 1;
sigaction( SIGALRM, $oldaction );  # restore original signal handler
if ( $failed ) {
  if ( defined $@ and $@ eq "connect timeout\n" ) {...}
  else { # connect died }
}
```

See previous example for the reasoning around the double eval.

Similar techniques can be used for canceling statement execution.

Unfortunately, this solution is somewhat messy.

For a cleaner implementation that works across perl versions, see Lincoln Baxter's Sys::SigAction module at Sys::SigAction. The documentation for Sys::SigAction includes an longer discussion of this problem, and a DBD::Oracle test script.

Be sure to read all the signal handling sections of the perlipc manual.

And finally, two more points to keep firmly in mind. Firstly, remember that what we've done here is essentially revert to old style *unsafe* handling of these signals. So do as little as possible in the handler. Ideally just die(). Secondly, the handles in use at the time the signal is handled may not be safe to use afterwards.

Subclassing the DBI

DBI can be subclassed and extended just like any other object oriented module. Before we talk about how to do that, it's important to be clear about the various DBI classes and how they work together.

By default `$dbh = DBI->connect(...)` returns a $dbh blessed into the `DBI::db` class. And the `$dbh->prepare` method returns an $sth blessed into the `DBI::st` class (actually it simply changes the last four characters of the calling handle class to be `::st`).

The leading 'DBI' is known as the 'root class' and the extra '::db' or '::st' are the 'handle type suffixes'. If you want to subclass the DBI you'll need to put your overriding methods into the appropriate classes. For example, if you want to use a root class of `MySubDBI` and override the do(), prepare() and execute() methods, then your do() and prepare() methods should be in the `MySubDBI::db` class and the execute() method should be in the `MySubDBI::st` class.

To setup the inheritance hierarchy the @ISA variable in `MySubDBI::db` should include `DBI::db` and the @ISA variable in `MySubDBI::st` should include `DBI::st`. The `MySubDBI` root class itself isn't currently used for anything visible and so, apart from setting @ISA to include `DBI`, it can be left empty.

So, having put your overriding methods into the right classes, and setup the inheritance hierarchy, how do you get the DBI to use them? You have two choices, either a static method call using the name of your subclass:

```
$dbh = MySubDBI->connect(...);
```

or specifying a `RootClass` attribute:

```
$dbh = DBI->connect(..., { RootClass => 'MySubDBI' });
```

If both forms are used then the attribute takes precedence.

The only differences between the two are that using an explicit RootClass attribute will a) make the DBI automatically attempt to load a module by that name if the class doesn't exist, and b) won't call your MySubDBI::connect() method, if you have one.

When subclassing is being used then, after a successful new connect, the DBI->connect method automatically calls:

```
$dbh->connected($dsn, $user, $pass, \%attr);
```

The default method does nothing. The call is made just to simplify any post-connection setup that your subclass may want to perform. The parameters are the same as passed to DBI->connect. If your subclass supplies a connected method, it should be part of the MySubDBI::db package.

One more thing to note: you must let the DBI do the handle creation. If you want to override the connect() method in your *::dr class then it must still call SUPER::connect

to get a $dbh to work with. Similarly, an overridden prepare() method in *::db must still call SUPER::prepare to get a $sth. If you try to create your own handles using bless() then you'll find the DBI will reject them with an "is not a DBI handle (has no magic)" error.

Here's a brief example of a DBI subclass. A more thorough example can be found in `t/subclass.t` in the DBI distribution.

```
package MySubDBI;

use strict;

use DBI;
our @ISA = qw(DBI);

package MySubDBI::db;
our @ISA = qw(DBI::db);

sub prepare {
  my ($dbh, @args) = @_;
  my $sth = $dbh->SUPER::prepare(@args)
      or return;
  $sth->{private_mysubdbi_info} = { foo => 'bar' };
  return $sth;
}

package MySubDBI::st;
our @ISA = qw(DBI::st);

sub fetch {
  my ($sth, @args) = @_;
  my $row = $sth->SUPER::fetch(@args)
      or return;
  do_something_magical_with_row_data($row)
      or return $sth->set_err(1234, "The magic failed", undef,
 ↪"fetch");
  return $row;
}
```

When calling a SUPER::method that returns a handle, be careful to check the return value before trying to do other things with it in your overridden method. This is especially important if you want to set a hash attribute on the handle, as Perl's autovivification will bite you by (in)conveniently creating an unblessed hashref, which your method will then return with usually baffling results later on like the error "dbih_getcom handle HASH(0xa4451a8) is not a DBI handle (has no magic)". It's best to check right after the call and return undef immediately on error, just like DBI would and just like the example above.

If your method needs to record an error it should call the set_err() method with the error code and error string, as shown in the example above. The error code and error string will be recorded in the handle and available via `$h->err` and `$DBI::errstr` etc. The set_err() method always returns an undef or empty list as appropriate. Since your method should nearly always return an undef or empty list as soon as an error

is detected it's handy to simply return what set_err() returns, as shown in the example above.

If the handle has `RaiseError`, `PrintError`, or `HandleError` etc. set then the set_err() method will honour them. This means that if `RaiseError` is set then set_err() won't return in the normal way but will 'throw an exception' that can be caught with an `eval` block.

You can stash private data into DBI handles via `$h->{private_...*}`. See the entry under "ATTRIBUTES COMMON TO ALL HANDLES" for info and important caveats.

Memory Leaks

When tracking down memory leaks using tools like Devel::Leak you'll find that some DBI internals are reported as 'leaking' memory. This is very unlikely to be a real leak. The DBI has various caches to improve performance and the apparent leaks are simply the normal operation of these caches.

The most frequent sources of the apparent leaks are "ChildHandles", "prepare_cached" and "connect_cached".

For example http://stackoverflow.com/questions/13338308/perl-dbi-memory-leak

Given how widely the DBI is used, you can rest assured that if a new release of the DBI did have a real leak it would be discovered, reported, and fixed immediately. The leak you're looking for is probably elsewhere. Good luck!

TRACING

The DBI has a powerful tracing mechanism built in. It enables you to see what's going on 'behind the scenes', both within the DBI and the drivers you're using.

Trace Settings

Which details are written to the trace output is controlled by a combination of a *trace level*, an integer from 0 to 15, and a set of *trace flags* that are either on or off. Together these are known as the *trace settings* and are stored together in a single integer. For normal use you only need to set the trace level, and generally only to a value between 1 and 4.

Each handle has its own trace settings, and so does the DBI. When you call a method the DBI merges the handles settings into its own for the duration of the call: the trace flags of the handle are OR'd into the trace flags of the DBI, and if the handle has a higher trace level then the DBI trace level is raised to match it. The previous DBI trace settings are restored when the called method returns.

Trace Levels

Trace *levels* are as follows:

```
0 - Trace disabled.
1 - Trace top-level DBI method calls returning with results or␣
 ↪errors.
```

(continues on next page)

(continued from previous page)

```
2 - As above, adding tracing of top-level method entry with␣
 ↪parameters.
3 - As above, adding some high-level information from the driver
    and some internal information from the DBI.
4 - As above, adding more detailed information from the driver.
    This is the first level to trace all the rows being fetched.
5 to 15 - As above but with more and more internal information.
```

Trace level 1 is best for a simple overview of what's happening. Trace levels 2 thru 4 a good choice for general purpose tracing. Levels 5 and above are best reserved for investigating a specific problem, when you need to see "inside" the driver and DBI.

The trace output is detailed and typically very useful. Much of the trace output is formatted using the "neat" function, so strings in the trace output may be edited and truncated by that function.

Trace Flags

Trace *flags* are used to enable tracing of specific activities within the DBI and drivers. The DBI defines some trace flags and drivers can define others. DBI trace flag names begin with a capital letter and driver specific names begin with a lowercase letter, as usual.

Currently the DBI defines these trace flags:

```
ALL - turn on all DBI and driver flags (not recommended)
SQL - trace SQL statements executed
      (not yet implemented in DBI but implemented in some DBDs)
CON - trace connection process
ENC - trace encoding (unicode translations etc)
      (not yet implemented in DBI but implemented in some DBDs)
DBD - trace only DBD messages
      (not implemented by all DBDs yet)
TXN - trace transactions
      (not implemented in all DBDs yet)
```

The "parse_trace_flags" and "parse_trace_flag" methods are used to convert trace flag names into the corresponding integer bit flags.

Enabling Trace

The `$h->trace` method sets the trace settings for a handle and `DBI->trace` does the same for the DBI.

In addition to the "trace" method, you can enable the same trace information, and direct the output to a file, by setting the `DBI_TRACE` environment variable before starting Perl. See "DBI_TRACE" for more information.

Finally, you can set, or get, the trace settings for a handle using the `TraceLevel` attribute.

All of those methods use parse_trace_flags() and so allow you set both the trace level and multiple trace flags by using a string containing the trace level and/or flag names

separated by vertical bar ("|") or comma (",") characters. For example:

```
local $h->{TraceLevel} = "3|SQL|foo";
```

Trace Output

Initially trace output is written to STDERR. Both the `$h->trace` and `DBI->trace` methods take an optional $trace_file parameter, which may be either the name of a file to be opened by DBI in append mode, or a reference to an existing writable (possibly layered) filehandle. If $trace_file is a filename, and can be opened in append mode, or $trace_file is a writable filehandle, then *all* trace output (currently including that from other handles) is redirected to that file. A warning is generated if $trace_file can't be opened or is not writable.

Further calls to trace() without $trace_file do not alter where the trace output is sent. If $trace_file is undefined, then trace output is sent to STDERR and, if the prior trace was opened with $trace_file as a filename, the previous trace file is closed; if $trace_file was a filehandle, the filehandle is **not** closed.

NOTE: If $trace_file is specified as a filehandle, the filehandle should not be closed until all DBI operations are completed, or the application has reset the trace file via another call to `trace()` that changes the trace file.

Tracing to Layered Filehandles

NOTE:

- Tied filehandles are not currently supported, as tie operations are not available to the PerlIO methods used by the DBI.

Perl provides the ability to layer various "disciplines" on an open filehandle via the *PerlIO* module.

A simple example of using PerlIO layers is to use a scalar as the output:

```
my $scalar = '';
open( my $fh, "+>:scalar", \$scalar );
$dbh->trace( 2, $fh );
```

Now all trace output is simply appended to $scalar.

A more complex application of tracing to a layered filehandle is the use of a custom layer (_Refer to _Perlio::via *for details on creating custom PerlIO layers.*). Consider an application with the following logger module:

```
package MyFancyLogger;

sub new
{
    my $self = {};
    my $fh;
    open $fh, '>', 'fancylog.log';
    $self->{_fh} = $fh;
```

(continues on next page)

(continued from previous page)

```
    $self->{_buf} = '';
    return bless $self, shift;
}

sub log
{
    my $self = shift;
    return unless exists $self->{_fh};
    my $fh = $self->{_fh};
    $self->{_buf} .= shift;
#
# DBI feeds us pieces at a time, so accumulate a complete line
# before outputting
#
    print $fh "At ", scalar localtime(), ':', $self->{_buf}, "\n"␣
 ↪and
    $self->{_buf} = ''
        if $self->{_buf}=~tr/\n//;
}

sub close {
    my $self = shift;
    return unless exists $self->{_fh};
    my $fh = $self->{_fh};
    print $fh "At ", scalar localtime(), ':', $self->{_buf}, "\n"␣
 ↪and
    $self->{_buf} = ''
        if $self->{_buf};
    close $fh;
    delete $self->{_fh};
}

1;
```

To redirect DBI traces to this logger requires creating a package for the layer:

```
package PerlIO::via::MyFancyLogLayer;

sub PUSHED
{
    my ($class,$mode,$fh) = @_;
    my $logger;
    return bless \$logger,$class;
}

sub OPEN {
    my ($self, $path, $mode, $fh) = @_;
    #
    # $path is actually our logger object
```

(continues on next page)

(continued from previous page)

```
    #
    $$self = $path;
    return 1;
}

sub WRITE
{
    my ($self, $buf, $fh) = @_;
    $$self->log($buf);
    return length($buf);
}

sub CLOSE {
    my $self = shift;
    $$self->close();
    return 0;
}

1;
```

The application can then cause DBI traces to be routed to the logger using

```
use PerlIO::via::MyFancyLogLayer;

open my $fh, '>:via(MyFancyLogLayer)', MyFancyLogger->new();

$dbh->trace('SQL', $fh);
```

Now all trace output will be processed by MyFancyLogger's log() method.

Trace Content

Many of the values embedded in trace output are formatted using the neat() utility function. This means they may be quoted, sanitized, and possibly truncated if longer than `$DBI::neat_maxlen`. See "neat" for more details.

Tracing Tips

You can add tracing to your own application code using the "trace_msg" method.

It can sometimes be handy to compare trace files from two different runs of the same script. However using a tool like `diff` on the original log output doesn't work well because the trace file is full of object addresses that may differ on each run.

The DBI includes a handy utility called dbilogstrip that can be used to 'normalize' the log content. It can be used as a filter like this:

```
DBI_TRACE=2 perl yourscript.pl ...args1... 2>&1 | dbilogstrip >␣
↪dbitrace1.log
DBI_TRACE=2 perl yourscript.pl ...args2... 2>&1 | dbilogstrip >␣
```

(continues on next page)

(continued from previous page)

```
 ↪dbitrace2.log
diff -u dbitrace1.log dbitrace2.log
```

See dbilogstrip for more information.

DBI ENVIRONMENT VARIABLES

The DBI module recognizes a number of environment variables, but most of them should not be used most of the time. It is better to be explicit about what you are doing to avoid the need for environment variables, especially in a web serving system where web servers are stingy about which environment variables are available.

DBI_DSN

The DBI_DSN environment variable is used by DBI->connect if you do not specify a data source when you issue the connect. It should have a format such as "dbi:Driver:databasename".

DBI_DRIVER

The DBI_DRIVER environment variable is used to fill in the database driver name in DBI->connect if the data source string starts "dbi::" (thereby omitting the driver). If DBI_DSN omits the driver name, DBI_DRIVER can fill the gap.

DBI_AUTOPROXY

The DBI_AUTOPROXY environment variable takes a string value that starts "dbi:Proxy:" and is typically followed by "hostname=...;port=...". It is used to alter the behaviour of DBI->connect. For full details, see DBI::Proxy documentation.

DBI_USER

The DBI_USER environment variable takes a string value that is used as the user name if the DBI->connect call is given undef (as distinct from an empty string) as the username argument. Be wary of the security implications of using this.

DBI_PASS

The DBI_PASS environment variable takes a string value that is used as the password if the DBI->connect call is given undef (as distinct from an empty string) as the password argument. Be extra wary of the security implications of using this.

DBI_DBNAME (obsolete)

The DBI_DBNAME environment variable takes a string value that is used only when the obsolescent style of DBI->connect (with driver name as fourth parameter) is used, and when no value is provided for the first (database name) argument.

DBI_TRACE

The DBI_TRACE environment variable specifies the global default trace settings for the DBI at startup. Can also be used to direct trace output to a file. When the DBI is loaded it does:

```
DBI->trace(split /=/, $ENV{DBI_TRACE}, 2) if $ENV{DBI_TRACE};
```

So if DBI_TRACE contains an "=" character then what follows it is used as the name of the file to append the trace to.

output appended to that file. If the name begins with a number followed by an equal sign (=), then the number and the equal sign are stripped off from the name, and the number is used to set the trace level. For example:

```
DBI_TRACE=1=dbitrace.log perl your_test_script.pl
```

On Unix-like systems using a Bourne-like shell, you can do this easily on the command line:

```
DBI_TRACE=2 perl your_test_script.pl
```

See "TRACING" for more information.

PERL_DBI_DEBUG (obsolete)

An old variable that should no longer be used; equivalent to DBI_TRACE.

DBI_PROFILE

The DBI_PROFILE environment variable can be used to enable profiling of DBI method calls. See *DBI::Profile* for more information.

DBI_PUREPERL

The DBI_PUREPERL environment variable can be used to enable the use of DBI::PurePerl. See *DBI::PurePerl* for more information.

WARNING AND ERROR MESSAGES

Fatal Errors

- Can't call method "prepare" without a package or object reference

  The `$dbh` handle you're using to call `prepare` is probably undefined because the preceding `connect` failed. You should always check the return status of DBI methods, or use the "RaiseError" attribute.

- Can't call method "execute" without a package or object reference

  The `$sth` handle you're using to call `execute` is probably undefined because the preceding `prepare` failed. You should always check the return status of DBI methods, or use the "RaiseError" attribute.

- DBI/DBD internal version mismatch

  The DBD driver module was built with a different version of DBI than the one currently being used. You should rebuild the DBD module under the current version of DBI.

  (Some rare platforms require "static linking". On those platforms, there may be an old DBI or DBD driver version actually embedded in the Perl executable being used.)

- DBD driver has not implemented the AutoCommit attribute

  The DBD driver implementation is incomplete. Consult the author.

- Can't [sg]et %s->{%s}: unrecognised attribute

  You attempted to set or get an unknown attribute of a handle. Make sure you have spelled the attribute name correctly; case is significant (e.g., "Autocommit" is not the same as "AutoCommit").

Pure-Perl DBI

A pure-perl emulation of the DBI is included in the distribution for people using pure-perl drivers who, for whatever reason, can't install the compiled DBI. See *DBI::PurePerl*.

SEE ALSO

Driver and Database Documentation

Refer to the documentation for the DBD driver that you are using.

Refer to the SQL Language Reference Manual for the database engine that you are using.

ODBC and SQL/CLI Standards Reference Information

More detailed information about the semantics of certain DBI methods that are based on ODBC and SQL/CLI standards is available on-line via microsoft.com, for ODBC, and www.jtc1sc32.org for the SQL/CLI standard:

```
DBI method         ODBC function     SQL/CLI Working Draft
----------         -------------     ---------------------
column_info        SQLColumns        Page 124
foreign_key_info   SQLForeignKeys    Page 163
get_info           SQLGetInfo        Page 214
primary_key_info   SQLPrimaryKeys    Page 254
table_info         SQLTables         Page 294
type_info          SQLGetTypeInfo    Page 239
statistics_info    SQLStatistics
```

To find documentation on the ODBC function you can browse Microsoft's documentation at:

https://learn.microsoft.com/en-us/sql/odbc/reference/odbc-programmer-s-reference

And for SQL/CLI standard information on SQLColumns you'd read page 124 of the (very large) SQL/CLI Working Draft available from:

https://web.archive.org/web/20071107215745/http://jtc1sc32.org/doc/N0701-0750/32N0744T.pdf

Standards Reference Information

A hyperlinked, browsable version of the BNF syntax for SQL92 (plus Oracle 7 SQL and PL/SQL) is available here:

https://web.archive.org/web/20040212094722/http://cui.unige.ch/db-research/Enseignement/analyseinfo/SQL92/BNFindex.html

You can find more information about SQL standards online by searching for the appropriate standard names and numbers. For example, searching for "ANSI/ISO/IEC International Standard (IS) Database Language SQL - Part 1: SQL/Framework" you'll find a copy at:

ftp://ftp.iks-jena.de/mitarb/lutz/standards/sql/ansi-iso-9075-1-1999.pdf

Books and Articles

Programming the Perl DBI, by Alligator Descartes and Tim Bunce. http://books.perl.org/book/154

Programming Perl 3rd Ed. by Larry Wall, Tom Christiansen & Jon Orwant. http://books.perl.org/book/134

Learning Perl by Randal Schwartz. http://books.perl.org/book/101

Details of many other books related to perl can be found at http://books.perl.org

Perl Modules

Index of DBI related modules available from CPAN:

```
L<https://metacpan.org/search?q=DBD%3A%3A>
L<https://metacpan.org/search?q=DBIx%3A%3A>
L<https://metacpan.org/search?q=DBI>
```

For a good comparison of RDBMS-OO mappers and some OO-RDBMS mappers (including Class::DBI, Alzabo, and DBIx::RecordSet in the former category and Tangram and SPOPS in the latter) see the Perl Object-Oriented Persistence project pages at:

```
http://poop.sourceforge.net
```

A similar page for Java toolkits can be found at:

```
http://c2.com/cgi-bin/wiki?ObjectRelationalToolComparison
```

Mailing List

The *dbi-users* mailing list is the primary means of communication among users of the DBI and its related modules. For details send email to:

```
L<dbi-users-help@perl.org>
```

There are typically between 700 and 900 messages per month. You have to subscribe in order to be able to post. However you can opt for a 'post-only' subscription.

Mailing list archives (of variable quality) are held at:

```
http://groups.google.com/groups?group=perl.dbi.users
http://www.xray.mpe.mpg.de/mailing-lists/dbi/
http://www.mail-archive.com/dbi-users%40perl.org/
```

Assorted Related Links

The DBI "Home Page":

```
http://dbi.perl.org/
```

Other DBI related links:

```
http://www.perlmonks.org/?node=DBI%20recipes
http://www.perlmonks.org/?node=Speeding%20up%20the%20DBI
```

Other database related links:

```
http://www.connectionstrings.com/
```

Security, especially the "SQL Injection" attack:

```
http://bobby-tables.com/
http://online.securityfocus.com/infocus/1644
```

FAQ

See http://dbi.perl.org/support/

AUTHORS

DBI by Tim Bunce (1994-2024), The DBI developer group (2024..)

This pod text by Tim Bunce, J. Douglas Dunlop, Jonathan Leffler and others. Perl by Larry Wall and the `perl5-porters`.

COPYRIGHT

The DBI module is Copyright (c) 1994-2024 Tim Bunce. Ireland. The DBI developer group (2024-2024) All rights reserved.

You may distribute under the terms of either the GNU General Public License or the Artistic License, as specified in the Perl 5.12.0 README file.

SUPPORT / WARRANTY

The DBI is free Open Source software. IT COMES WITHOUT WARRANTY OF ANY KIND.

Support

My consulting company, Data Plan Services, offers annual and multi-annual support contracts for the DBI. These provide sustained support for DBI development, and sustained value for you in return. Contact me for details.

Sponsor Enhancements

If your company would benefit from a specific new DBI feature, please consider sponsoring its development. Work is performed rapidly, and usually on a fixed-price payment-on-delivery basis. Contact me for details.

Using such targeted financing allows you to contribute to DBI development, and rapidly get something specific and valuable in return.

ACKNOWLEDGEMENTS

The creator: Tim Bunce!

I would like to acknowledge the valuable contributions of the many people Tim has worked with on the DBI project, especially in the early years (1992-1994). In no particular order: Kevin Stock, Buzz Moschetti, Kurt Andersen, Ted Lemon, William Hails, Garth Kennedy, Michael Peppler, Neil S. Briscoe, Jeff Urlwin, David J. Hughes, Jeff Stander, Forrest D Whitcher, Larry Wall, Jeff Fried, Roy Johnson, Paul Hudson, Georg Rehfeld, Steve Sizemore, Ron Pool, Jon Meek, Tom Christiansen, Steve Baumgarten, Randal Schwartz, and a whole lot more.

Then, of course, there are the poor souls who have struggled through untold and undocumented obstacles to actually implement DBI drivers. Among their ranks are Jochen Wiedmann, Alligator Descartes, Jonathan Leffler, Jeff Urlwin, Michael Peppler, Henrik Tougaard, Edwin Pratomo, Davide Migliavacca, Jan Pazdziora, Peter Haworth, Edmund Mergl, Steve Williams, Thomas Lowery, and Phlip Plumlee. Without them, the DBI would not be the practical reality it is today. I'm also especially grateful to Alligator Descartes for starting work on the first edition of the "Programming the Perl DBI" book and letting me jump on board.

The DBI and DBD::Oracle were originally developed while I was Technical Director (CTO) of the Paul Ingram Group in the UK. So I'd especially like to thank Paul for his generosity and vision in supporting this work for many years.

A couple of specific DBI features have been sponsored by enlightened companies:

The development of the swap_inner_handle() method was sponsored by BizRate.com

The development of DBD::Gofer and related modules was sponsored by Shopzilla.com (https::connexity.com).

After the release of version 1.643, the development was lead by H.Merijn Brand with the help of the DBI team. Thanks to all that added issues, did Pull Requests or helped in any way to keep the DBI healthy.

CONTRIBUTING

As you can see above, many people have contributed to the DBI and drivers in many ways over many years.

If you'd like to help then see `CONTRIBUTING.md`.

If you'd like the DBI to do something new or different then a good way to make that happen is to do it yourself and send me a patch to the source code that shows the changes. (But read "Speak before you patch" below.)

Browsing the source code repository

Use https://github.com/perl5-dbi/dbi

How to create a patch using Git

The DBI source code is maintained using Git. To access the source you'll need to install a Git client. Then, to get the source code, do:

```
git clone https://github.com/perl5-dbi/dbi.git DBI-git
```

The source code will now be available in the new subdirectory `DBI-git`.

When you want to synchronize later, issue the command

```
git pull --all
```

Make your changes, test them, test them again until everything passes. If there are no tests for the new feature you added or a behaviour change, the change should include a new test. Then commit the changes. Either use

```
git gui
```

or

```
git commit -a -m 'Message to my changes'
```

If you get any conflicts reported you'll need to fix them first.

Then generate the patch file to be mailed:

```
git format-patch -1 --attach
```

which will create a file 0001-*.patch (where * relates to the commit message). Read the patch file, as a sanity check, and then email it to dbi-dev@perl.org.

If you have a github account, you can also fork the repository, commit your changes to the forked repository and then do a pull request.

How to create a patch without Git

Unpack a fresh copy of the distribution:

```
wget http://cpan.metacpan.org/authors/id/T/TI/TIMB/DBI-1.627.tar.gz
tar xfz DBI-1.627.tar.gz
```

Rename the newly created top level directory:

```
mv DBI-1.627 DBI-1.627.your_foo
```

Edit the contents of DBI-1.627.your_foo/* till it does what you want.

Test your changes and then remove all temporary files:

```
make test && make distclean
```

Go back to the directory you originally unpacked the distribution:

```
cd ..
```

Unpack *another* copy of the original distribution you started with:

```
tar xfz DBI-1.627.tar.gz
```

Then create a patch file by performing a recursive `diff` on the two top level directories:

```
diff -purd DBI-1.627 DBI-1.627.your_foo > DBI-1.627.your_foo.patch
```

Speak before you patch

For anything non-trivial or possibly controversial it's a good idea to discuss (on dbi-dev@perl.org) the changes you propose before actually spending time working on them. Otherwise you run the risk of them being rejected because they don't fit into some larger plans you may not be aware of.

You can also reach the developers on IRC (chat). If they are on-line, the most likely place to talk to them is the #dbi channel on irc.perl.org

TRANSLATIONS

A German translation of this manual (possibly slightly out of date) is available, thanks to O'Reilly, at:

```
http://www.oreilly.de/catalog/perldbiger/
```

OTHER RELATED WORK AND PERL MODULES

- Apache::DBI

  To be used with the Apache daemon together with an embedded Perl interpreter like `mod_perl`. Establishes a database connection which remains open for the lifetime of the HTTP daemon. This way the CGI connect and disconnect for every database access becomes superfluous.

- SQL Parser

  See also the SQL::Statement module, SQL parser and engine.

TODO

Documentation

These entries are still to be written:

- DBIf_TRACE_CON
- DBIf_TRACE_DBD
- DBIf_TRACE_ENC
- DBIf_TRACE_SQL
- DBIf_TRACE_TXN
- DBIpp_cm_XX
- DBIpp_cm_br
- DBIpp_cm_cs
- DBIpp_cm_dd
- DBIpp_cm_dw
- DBIpp_cm_hs
- DBIpp_ph_XX
- DBIpp_ph_cn
- DBIpp_ph_cs
- DBIpp_ph_qm
- DBIpp_ph_sp
- DBIpp_st_XX
- DBIpp_st_bs
- DBIpp_st_qq
- SQL_ALL_TYPES
- SQL_ARRAY
- SQL_ARRAY_LOCATOR
- SQL_BIGINT
- SQL_BINARY
- SQL_BIT
- SQL_BLOB
- SQL_BLOB_LOCATOR
- SQL_BOOLEAN

- SQL_CHAR
- SQL_CLOB
- SQL_CLOB_LOCATOR
- SQL_CURSOR_DYNAMIC
- SQL_CURSOR_FORWARD_ONLY
- SQL_CURSOR_KEYSET_DRIVEN
- SQL_CURSOR_STATIC
- SQL_CURSOR_TYPE_DEFAULT
- SQL_DATE
- SQL_DATETIME
- SQL_DECIMAL
- SQL_DOUBLE
- SQL_FLOAT
- SQL_GUID
- SQL_INTEGER
- SQL_INTERVAL
- SQL_INTERVAL_DAY
- SQL_INTERVAL_DAY_TO_HOUR
- SQL_INTERVAL_DAY_TO_MINUTE
- SQL_INTERVAL_DAY_TO_SECOND
- SQL_INTERVAL_HOUR
- SQL_INTERVAL_HOUR_TO_MINUTE
- SQL_INTERVAL_HOUR_TO_SECOND
- SQL_INTERVAL_MINUTE
- SQL_INTERVAL_MINUTE_TO_SECOND
- SQL_INTERVAL_MONTH
- SQL_INTERVAL_SECOND
- SQL_INTERVAL_YEAR
- SQL_INTERVAL_YEAR_TO_MONTH
- SQL_LONGVARBINARY
- SQL_LONGVARCHAR
- SQL_MULTISET
- SQL_MULTISET_LOCATOR
- SQL_NUMERIC

- SQL_REAL
- SQL_REF
- SQL_ROW
- SQL_SMALLINT
- SQL_TIME
- SQL_TIMESTAMP
- SQL_TINYINT
- SQL_TYPE_DATE
- SQL_TYPE_TIME
- SQL_TYPE_TIMESTAMP
- SQL_TYPE_TIMESTAMP_WITH_TIMEZONE
- SQL_TYPE_TIME_WITH_TIMEZONE
- SQL_UDT
- SQL_UDT_LOCATOR
- SQL_UNKNOWN_TYPE
- SQL_VARBINARY
- SQL_VARCHAR
- SQL_WCHAR
- SQL_WLONGVARCHAR
- SQL_WVARCHAR
- connect_test_perf
- constant
- dbi_profile
- dbi_profile_merge
- dbi_profile_merge_nodes
- dbi_time
- disconnect_all
- driver_prefix
- dump_dbd_registry
- dump_handle
- init_rootclass
- install_driver
- installed_methods
- setup_driver

Modules

- DBI::Const
- DBI::DBD
- DBI::Gofer
- DBI::Profile - Performance profiling and benchmarking for the DBI
- DBI::ProfileData - manipulate DBI::ProfileDumper data dumps
- DBI::ProfileDumper - profile DBI usage and output data to a file
- DBI::ProfileSubs - Subroutines for dynamic profile Path
- DBI::ProxyServer - a server for the DBD::Proxy driver
- DBI::PurePerl - BEGIN { $ENV{DBI_PUREPERL} = 2 } use DBI;
- DBI::SQL
- DBI::Util

DBI::Const

Namespace containing the following modules:

- DBI::Const::GetInfo
- DBI::Const::GetInfoReturn - Data and functions for describing GetInfo results
- DBI::Const::GetInfoType - Data describing GetInfo type codes

DBI::Const::GetInfo

Namespace containing the following modules:

- DBI::Const::GetInfo::ANSI - ISO/IEC SQL/CLI Constants for GetInfo
- DBI::Const::GetInfo::ODBC - ODBC Constants for GetInfo

DBI::Const::GetInfo::ANSI

ISO/IEC SQL/CLI Constants for GetInfo

SYNOPSIS

```
The API for this module is private and subject to change.
```

DESCRIPTION

Information requested by GetInfo().

See: A.1 C header file SQLCLI.H, Page 316, 317.

The API for this module is private and subject to change.

REFERENCES

```
ISO/IEC FCD 9075-3:200x Information technology - Database Languages␣
↪-
SQL - Part 3: Call-Level Interface (SQL/CLI)

SC32 N00744 = WG3:VIE-005 = H2-2002-007

Date: 2002-01-15
```

%ReturnTypes

See: Codes and data types for implementation information (Table 28), Page 85, 86.

Mapped to ODBC datatype names.

%ReturnValues

See: A.1 C header file SQLCLI.H, Page 317, 318.

TODO

Corrections, e.g.:

```
SQL_TRANSACTION_ISOLATION_OPTION vs. SQL_TRANSACTION_ISOLATION
```

DBI::Const::GetInfo::ODBC

ODBC Constants for GetInfo

SYNOPSIS

```
The API for this module is private and subject to change.
```

DESCRIPTION

Information requested by GetInfo().

The API for this module is private and subject to change.

REFERENCES

```
MDAC SDK 2.6
ODBC version number (0x0351)

sql.h
sqlext.h
```

%ReturnTypes

See: mk:@MSITStore:X:\dm\cli\mdac\sdk26\Docs\odbc.chm::/htm/odbcsqlgetinfo.htm

```
=>     : alias
=> !!! : edited
```

%ReturnValues

See: sql.h, sqlext.h Edited: SQL_TXN_ISOLATION_OPTION

TODO

```
Corrections?
SQL_NULL_COLLATION: ODBC vs ANSI
Unique values for $ReturnValues{...}?, e.g. SQL_FILE_USAGE
```

DBI::Const::GetInfoReturn

Data and functions for describing GetInfo results

SYNOPSIS

The interface to this module is undocumented and liable to change.

DESCRIPTION

Data and functions for describing GetInfo results

DBI::Const::GetInfoType

Data describing GetInfo type codes

SYNOPSIS

```
use DBI::Const::GetInfoType;
```

DESCRIPTION

Imports a %GetInfoType hash which maps names for GetInfo Type Codes into their corresponding numeric values. For example:

```
$database_version = $dbh->get_info( $GetInfoType{SQL_DBMS_VER} );
```

The interface to this module is new and nothing beyond what is written here is guaranteed.

DBI::DBD

Namespace containing the following modules:

- `DBI::DBD::Metadata` - Generate the code and data for some DBI metadata methods
- `DBI::DBD::SqlEngine` - Base class for DBI drivers without their own SQL engine

DBI::DBD::Metadata

Generate the code and data for some DBI metadata methods

SYNOPSIS

The idea is to extract metadata information from a good quality ODBC driver and use it to generate code and data to use in your own DBI driver for the same database.

To generate code to support the get_info method:

```
perl -MDBI::DBD::Metadata -e "write_getinfo_pm('dbi:ODBC:dsn-name',
↪'user','pass','Driver')"

perl -MDBI::DBD::Metadata -e write_getinfo_pm dbi:ODBC:foo_db␣
↪username password Driver
```

To generate code to support the type_info method:

```
perl -MDBI::DBD::Metadata -e "write_typeinfo_pm('dbi:ODBC:dsn-name',
↪'user','pass','Driver')"

perl -MDBI::DBD::Metadata -e write_typeinfo_pm dbi:ODBC:dsn-name␣
↪user pass Driver
```

Where `dbi:ODBC:dsn-name` is the connection to use to extract the data, and `Driver` is the name of the driver you want the code generated for (the driver name gets embedded into the output in numerous places).

Generating a GetInfo package for a driver

The `write_getinfo_pm` in the DBI::DBD::Metadata module generates a DBD::Driver::GetInfo package on standard output.

This method generates a DBD::Driver::GetInfo package from the data source you specified in the parameter list or in the environment variable DBI_DSN. DBD::Driver::GetInfo should help a DBD author implement the DBI get_info() method. Because you are just creating this package, it is very unlikely that DBD::Driver already provides a good implementation for get_info(). Thus you will probably connect via DBD::ODBC.

Once you are sure that it is producing reasonably sane data, you should typically redirect the standard output to lib/DBD/Driver/GetInfo.pm, and then hand edit the result. Do not forget to update your Makefile.PL and MANIFEST to include this as an extra PM file that should be installed.

If you connect via DBD::ODBC, you should use version 0.38 or greater;

Please take a critical look at the data returned! ODBC drivers vary dramatically in their quality.

The generator assumes that most values are static and places these values directly in the %info hash. A few examples show the use of CODE references and the implementation via subroutines. It is very likely that you will have to write additional subroutines for values depending on the session state or server version, e.g. SQL_DBMS_VER.

A possible implementation of DBD::Driver::db::get_info() may look like:

```
sub get_info {
  my($dbh, $info_type) = @_;
  require DBD::Driver::GetInfo;
  my $v = $DBD::Driver::GetInfo::info{int($info_type)};
  $v = $v->($dbh) if ref $v eq 'CODE';
  return $v;
}
```

Please replace Driver (or "<foo>") with the name of your driver. Note that this stub function is generated for you by write_getinfo_pm function, but you must manually transfer the code to Driver.pm.

Generating a TypeInfo package for a driver

The `write_typeinfo_pm` function in the DBI::DBD::Metadata module generates on standard output the data needed for a driver's type_info_all method. It also provides default implementations of the type_info_all method for inclusion in the driver's main implementation file.

The driver parameter is the name of the driver for which the methods will be generated; for the sake of examples, this will be "Driver". Typically, the dsn parameter will be of the form "dbi:ODBC:odbc_dsn", where the odbc_dsn is a DSN for one of the driver's databases. The user and pass parameters are the other optional connection parameters that will be provided to the DBI connect method.

Once you are sure that it is producing reasonably sane data, you should typically redirect the standard output to lib/DBD/Driver/TypeInfo.pm, and then hand edit the result if necessary. Do not forget to update your Makefile.PL and MANIFEST to include this as an extra PM file that should be installed.

Please take a critical look at the data returned! ODBC drivers vary dramatically in their quality.

The generator assumes that all the values are static and places these values directly in the %info hash.

A possible implementation of DBD::Driver::type_info_all() may look like:

```
sub type_info_all {
  my ($dbh) = @_;
  require DBD::Driver::TypeInfo;
  return [ @$DBD::Driver::TypeInfo::type_info_all ];
}
```

Please replace Driver (or "<foo>") with the name of your driver. Note that this stub function is generated for you by the write_typeinfo_pm function, but you must manually transfer the code to Driver.pm.

AUTHORS

Jonathan Leffler jleffler@us.ibm.com (previously jleffler@informix.com), Jochen Wiedmann joe@ispsoft.de, Steffen Goeldner sgoeldner@cpan.org, and Tim Bunce dbi-users@perl.org.

DBI::DBD::SqlEngine

Base class for DBI drivers without their own SQL engine

SYNOPSIS

```
package DBD::myDriver;

use base qw(DBI::DBD::SqlEngine);

sub driver {
    ...
    my $drh = $proto->SUPER::driver ($attr);
    ...
    return $drh->{class};
    }

package DBD::myDriver::dr;

our @ISA = qw(DBI::DBD::SqlEngine::dr);

sub data_sources { ... }
...

package DBD::myDriver::db;

our @ISA = qw(DBI::DBD::SqlEngine::db);

sub init_valid_attributes { ... }
sub init_default_attributes { ... }
sub set_versions { ... }
sub validate_STORE_attr { my ($dbh, $attrib, $value) = @_; ... }
sub validate_FETCH_attr { my ($dbh, $attrib) = @_; ... }
sub get_myd_versions { ... }
sub get_avail_tables { ... }

package DBD::myDriver::st;

our @ISA = qw(DBI::DBD::SqlEngine::st);
```

(continues on next page)

(continued from previous page)

```
sub FETCH { ... }
sub STORE { ... }

package DBD::myDriver::Statement;

our @ISA = qw(DBI::DBD::SqlEngine::Statement);

sub open_table { ... }

package DBD::myDriver::Table;

our @ISA = qw(DBI::DBD::SqlEngine::Table);

sub new { ... }
```

DESCRIPTION

DBI::DBD::SqlEngine abstracts the usage of SQL engines from the DBD. DBD authors can concentrate on the data retrieval they want to provide.

It is strongly recommended that you read DBD::File::Developers and DBD::File::Roadmap, because many of the DBD::File API is provided by DBI::DBD::SqlEngine.

Currently the API of DBI::DBD::SqlEngine is experimental and will likely change in the near future to provide the table meta data basics like DBD::File.

DBI::DBD::SqlEngine expects that any driver in inheritance chain has a *DBI prefix*.

Metadata

The following attributes are handled by DBI itself and not by DBI::DBD::SqlEngine, thus they all work as expected:

```
Active
ActiveKids
CachedKids
CompatMode              (Not used)
InactiveDestroy
AutoInactiveDestroy
Kids
PrintError
RaiseError
Warn                    (Not used)
```

The following DBI attributes are handled by DBI::DBD::SqlEngine:

AutoCommit

Always on.

ChopBlanks

Works.

NUM_OF_FIELDS

Valid after `$sth->execute`.

NUM_OF_PARAMS

Valid after `$sth->prepare`.

NAME

Valid after `$sth->execute`; probably undef for Non-Select statements.

NULLABLE

Not really working, always returns an array ref of ones, as DBD::CSV does not verify input data. Valid after `$sth->execute`; undef for non-select statements.

The following DBI attributes and methods are not supported:

- bind_param_inout
- CursorName
- LongReadLen
- LongTruncOk

DBI::DBD::SqlEngine specific attributes

In addition to the DBI attributes, you can use the following dbh attributes:

sql_engine_version

Contains the module version of this driver (**readonly**)

sql_nano_version

Contains the module version of DBI::SQL::Nano (**readonly**)

sql_statement_version

Contains the module version of SQL::Statement, if available (**readonly**)

sql_handler

Contains the SQL Statement engine, either DBI::SQL::Nano or SQL::Statement (**readonly**).

sql_parser_object

Contains an instantiated instance of SQL::Parser (**readonly**). This is filled when used first time (only when used with SQL::Statement).

sql_sponge_driver

Contains an internally used DBD::Sponge handle (**readonly**).

sql_valid_attrs

Contains the list of valid attributes for each DBI::DBD::SqlEngine based driver (**readonly**).

sql_readonly_attrs

Contains the list of those attributes which are readonly (**readonly**).

sql_identifier_case

Contains how DBI::DBD::SqlEngine deals with non-quoted SQL identifiers:

```
* SQL_IC_UPPER (1) means all identifiers are internally converted
  into upper-cased pendants
* SQL_IC_LOWER (2) means all identifiers are internally converted
  into lower-cased pendants
* SQL_IC_MIXED (4) means all identifiers are taken as they are
```

These conversions happen if (and only if) no existing identifier matches. Once existing identifier is used as known.

The SQL statement execution classes doesn't have to care, so don't expect `sql_identifier_case` affects column names in statements like

```
SELECT * FROM foo
```

sql_quoted_identifier_case

Contains how DBI::DBD::SqlEngine deals with quoted SQL identifiers (**readonly**). It's fixated to SQL_IC_SENSITIVE (3), which is interpreted as SQL_IC_MIXED.

sql_flags

Contains additional flags to instantiate an SQL::Parser. Because an SQL::Parser is instantiated only once, it's recommended to set this flag before any statement is executed.

sql_dialect

Controls the dialect understood by SQL::Parser. Possible values (delivery state of SQL::Statement):

```
* ANSI
* CSV
* AnyData
```

Defaults to "CSV". Because an SQL::Parser is instantiated only once and SQL::Parser doesn't allow one to modify the dialect once instantiated, it's strongly recommended to set this flag before any statement is executed (best place is connect attribute hash).

sql_engine_in_gofer

This value has a true value in case of this driver is operated via *DBD::Gofer*. The impact of being operated via Gofer is a read-only driver (not read-only databases!), so you cannot modify any attributes later - neither any table settings. **But** you won't get an error in cases you modify table attributes, so please carefully watch `sql_engine_in_gofer`.

sql_meta

Private data area which contains information about the tables this module handles. Table meta data might not be available until the table has been accessed for the first time e.g., by issuing a select on it however it is possible to pre-initialize attributes for each table you use.

DBI::DBD::SqlEngine recognizes the (public) attributes `col_names`, `table_name`, `readonly`, `sql_data_source` and `sql_identifier_case`. Be very careful when modifying attributes you do not know, the consequence might be a destroyed or corrupted table.

While `sql_meta` is a private and readonly attribute (which means, you cannot modify it's values), derived drivers might provide restricted write access through another attribute. Well known accessors are `csv_tables` for DBD::CSV, `ad_tables` for DBD::AnyData and `dbm_tables` for *DBD::DBM*.

sql_table_source

Controls the class which will be used for fetching available tables.

See "DBI::DBD::SqlEngine::TableSource" for details.

sql_data_source

Contains the class name to be used for opening tables.

See "DBI::DBD::SqlEngine::DataSource" for details.

Driver private methods

Default DBI methods

data_sources

The `data_sources` method returns a list of subdirectories of the current directory in the form "dbi:CSV:f_dir=$dirname".

If you want to read the subdirectories of another directory, use

```
my ($drh)  = DBI->install_driver ("CSV");
my (@list) = $drh->data_sources (f_dir => "/usr/local/csv_data");
```

list_tables

This method returns a list of file names inside $dbh->{f_dir}. Example:

```
my ($dbh)  = DBI->connect ("dbi:CSV:f_dir=/usr/local/csv_data");
my (@list) = $dbh->func ("list_tables");
```

Note that the list includes all files contained in the directory, even those that have non-valid table names, from the view of SQL.

Additional methods

The following methods are only available via their documented name when DBI::DBD::SqlEngine is used directly. Because this is only reasonable for testing purposes, the real names must be used instead. Those names can be computed by replacing the `sql_` in the method name with the driver prefix.

sql_versions

Signature:

```
sub sql_versions (;$) {
  my ($table_name) = @_;
  $table_name ||= ".";
  ...
  }
```

Returns the versions of the driver, including the DBI version, the Perl version, DBI::PurePerl version (if DBI::PurePerl is active) and the version of the SQL engine in use.

```
my $dbh = DBI->connect ("dbi:File:");
my $sql_versions = $dbh->func( "sql_versions" );
print "$sql_versions\n";
__END__
# DBI::DBD::SqlEngine  0.05 using SQL::Statement 1.402
# DBI                  1.623
```

(continues on next page)

(continued from previous page)

```
# OS                    netbsd (6.99.12)
# Perl                  5.016002 (x86_64-netbsd-thread-multi)
```

Called in list context, sql_versions will return an array containing each line as single entry.

Some drivers might use the optional (table name) argument and modify version information related to the table (e.g. DBD::DBM provides storage backend information for the requested table, when it has a table name).

sql_get_meta

Signature:

```
sub sql_get_meta ($$) {
    my ($table_name, $attrib) = @_;
    ...
    }
```

Returns the value of a meta attribute set for a specific table, if any. See sql_meta for the possible attributes.

A table name of "." (single dot) is interpreted as the default table. This will retrieve the appropriate attribute globally from the dbh. This has the same restrictions as `$dbh->{$attrib}`.

sql_set_meta

Signature:

```
sub sql_set_meta ($$$) {
    my ($table_name, $attrib, $value) = @_;
    ...
    }
```

Sets the value of a meta attribute set for a specific table. See sql_meta for the possible attributes.

A table name of "." (single dot) is interpreted as the default table which will set the specified attribute globally for the dbh. This has the same restrictions as `$dbh->{$attrib} = $value`.

sql_clear_meta

Signature:

```
sub sql_clear_meta ($) {
    my ($table_name) = @_;
    ...
    }
```

Clears the table specific meta information in the private storage of the dbh.

Extensibility

DBI::DBD::SqlEngine::TableSource

Provides data sources and table information on database driver and database handle level.

```
package DBI::DBD::SqlEngine::TableSource;

sub data_sources ($;$) {
  my ($class, $drh, $attrs) = @_;
  ...
  }

sub avail_tables {
  my ( $class, $drh ) = @_;
  ...
  }
```

The `data_sources` method is called when the user invokes any of the following:

```
@ary = DBI->data_sources ($driver);
@ary = DBI->data_sources ($driver, \%attr);

@ary = $dbh->data_sources ();
@ary = $dbh->data_sources (\%attr);
```

The `avail_tables` method is called when the user invokes any of the following:

```
@names = $dbh->tables ($catalog, $schema, $table, $type);

$sth = $dbh->table_info ($catalog, $schema, $table, $type);
$sth = $dbh->table_info ($catalog, $schema, $table, $type, \%attr);

$dbh->func ("list_tables");
```

Every time where an `\%attr` argument can be specified, this `\%attr` object's `sql_table_source` attribute is preferred over the `$dbh` attribute or the driver default, e.g.

```
@ary = DBI->data_sources ("dbi:CSV:", {
  f_dir => "/your/csv/tables",
  # note: this class doesn't comes with DBI
  sql_table_source => "DBD::File::Archive::Tar::TableSource",
  # scan tarballs instead of directories
  });
```

When you're going to implement such a DBD::File::Archive::Tar::TableSource class, remember to add correct attributes (including `sql_table_source` and `sql_data_source`) to the returned DSN's.

DBI::DBD::SqlEngine::DataSource

Provides base functionality for dealing with tables. It is primarily designed for allowing transparent access to files on disk or already opened (file-)streams (e.g. for DBD::CSV).

Derived classes shall be restricted to similar functionality, too (e.g. opening streams from an archive, transparently compress/uncompress log files before parsing them,

```
package DBI::DBD::SqlEngine::DataSource;

sub complete_table_name ($$;$) {
  my ($self, $meta, $table, $respect_case) = @_;
  ...
  }
```

The method `complete_table_name` is called when first setting up the *meta information* for a table:

```
"SELECT user.id, user.name, user.shell FROM user WHERE ..."
```

results in opening the table `user`. First step of the table open process is completing the name. Let's imagine you're having a DBD::CSV handle with following settings:

```
$dbh->{sql_identifier_case} = SQL_IC_LOWER;
$dbh->{f_ext} = '.lst';
$dbh->{f_dir} = '/data/web/adrmgr';
```

Those settings will result in looking for files matching `[Uu][Ss][Ee][Rr](\.lst)?$` in `/data/web/adrmgr/`. The scanning of the directory `/data/web/adrmgr/` and the pattern match check will be done in `DBD::File::DataSource::File` by the `complete_table_name` method.

If you intend to provide other sources of data streams than files, in addition to provide an appropriate `complete_table_name` method, a method to open the resource is required:

```
package DBI::DBD::SqlEngine::DataSource;

sub open_data ($) {
  my ($self, $meta, $attrs, $flags) = @_;
  ...
  }
```

After the method `open_data` has been run successfully, the table's meta information are in a state which allows the table's data accessor methods will be able to fetch/store row information. Implementation details heavily depends on the table implementation, whereby the most famous is surely *DBD::File::Table*.

SQL ENGINES

DBI::DBD::SqlEngine currently supports two SQL engines: SQL::Statement and *DBI::SQL::Nano::Statement_*. DBI::SQL::Nano supports a *very* limited subset of SQL statements, but it might be faster for some very simple tasks. SQL::Statement in contrast supports a much larger subset of ANSI SQL.

To use SQL::Statement, you need at least version 1.401 of SQL::Statement and the environment variable `DBI_SQL_NANO` must not be set to a true value.

SUPPORT

You can find documentation for this module with the perldoc command.

```
perldoc DBI::DBD::SqlEngine
```

You can also look for information at:

- RT: CPAN's request tracker

  http://rt.cpan.org/NoAuth/Bugs.html?Dist=DBI http://rt.cpan.org/NoAuth/Bugs.html?Dist=SQL-Statement

- CPAN Ratings

  http://cpanratings.perl.org/d/DBI

- Search CPAN

  http://search.cpan.org/dist/DBI/

Where can I go for more help?

For questions about installation or usage, please ask on the dbi-dev@perl.org mailing list.

If you have a bug report, patch or suggestion, please open a new report ticket on CPAN, if there is not already one for the issue you want to report. Of course, you can mail any of the module maintainers, but it is less likely to be missed if it is reported on RT.

Report tickets should contain a detailed description of the bug or enhancement request you want to report and at least an easy way to verify/reproduce the issue and any supplied fix. Patches are always welcome, too.

ACKNOWLEDGEMENTS

Thanks to Tim Bunce, Martin Evans and H.Merijn Brand for their continued support while developing DBD::File, DBD::DBM and DBD::AnyData. Their support, hints and feedback helped to design and implement this module.

AUTHOR

This module is currently maintained by

H.Merijn Brand < perl5 at tux.freedom.nl > and Jens Rehsack < rehsack at googlemail.com >

The original authors are Jochen Wiedmann and Jeff Zucker.

COPYRIGHT AND LICENSE


```
Copyright (C) 2009-2026 by H.Merijn Brand & Jens Rehsack
Copyright (C) 2004-2009 by Jeff Zucker
Copyright (C) 1998-2004 by Jochen Wiedmann
```

All rights reserved.

You may freely distribute and/or modify this module under the terms of either the GNU General Public License (GPL) or the Artistic License, as specified in the Perl README file.

SEE ALSO

DBI, *DBD::File*, DBD::AnyData and DBD::Sys.

DBI::Gofer

Namespace containing the following modules:

- `DBI::Gofer::Execute` - Executes Gofer requests and returns Gofer responses
- `DBI::Gofer::Request` - Encapsulate a request from DBD::Gofer to DBI::Gofer::Execute
- `DBI::Gofer::Response` - Encapsulate a response from DBI::Gofer::Execute to DBD::Gofer
- `DBI::Gofer::Serializer`
- `DBI::Gofer::Transport`

DBI::Gofer::Execute

Executes Gofer requests and returns Gofer responses

SYNOPSIS

```
$executor = DBI::Gofer::Execute->new( { ...config... });

$response = $executor->execute_request( $request );
```

DESCRIPTION

Accepts a DBI::Gofer::Request object, executes the requested DBI method calls, and returns a DBI::Gofer::Response object.

Any error, including any internal 'fatal' errors are caught and converted into a DBI::Gofer::Response object.

This module is usually invoked by a 'server-side' Gofer transport module. They usually have names in the "`DBI::Gofer::Transport::*`" namespace. Examples include: *DBI::Gofer::Transport::stream* and DBI::Gofer::Transport::mod_perl.

CONFIGURATION

check_request_sub

If defined, it must be a reference to a subroutine that will 'check' the request. It is passed the request object and the executor as its only arguments.

The subroutine can either return the original request object or die with a suitable error message (which will be turned into a Gofer response).

It can also construct and return a new request that should be executed instead of the original request.

check_response_sub

If defined, it must be a reference to a subroutine that will 'check' the response. It is passed the response object, the executor, and the request object. The sub may alter the response object and return undef, or return a new response object.

This mechanism can be used to, for example, terminate the service if specific database errors are seen.

forced_connect_dsn

If set, this DSN is always used instead of the one in the request.

default_connect_dsn

If set, this DSN is used if `forced_connect_dsn` is not set and the request does not contain a DSN itself.

forced_connect_attributes

A reference to a hash of connect() attributes. Individual attributes in `forced_connect_attributes` will take precedence over corresponding attributes in the request.

default_connect_attributes

A reference to a hash of connect() attributes. Individual attributes in the request take precedence over corresponding attributes in `default_connect_attributes`.

max_cached_dbh_per_drh

If set, the loaded drivers will be checked to ensure they don't have more than this number of cached connections. There is no default value. This limit is not enforced for every request.

max_cached_sth_per_dbh

If set, all the cached statement handles will be cleared once the number of cached statement handles rises above this limit. The default is 1000.

forced_single_resultset

If true, then only the first result set will be fetched and returned in the response.

forced_response_attributes

A reference to a data structure that can specify extra attributes to be returned in responses.

```
forced_response_attributes => {
    DriverName => {
        dbh => [ qw(dbh_attrib_name) ],
        sth => [ qw(sth_attrib_name) ],
    },
},
```

This can be useful in cases where the driver has not implemented the private_attribute_info() method and DBI::Gofer::Execute's own fallback list of private attributes doesn't include the driver or attributes you need.

track_recent

If set, specifies the number of recent requests and responses that should be kept by the update_stats() method for diagnostics. See DBI::Gofer::Transport::mod_perl.

Note that this setting can significantly increase memory use. Use with caution.

forced_gofer_random

Enable forced random failures and/or delays for testing. See "DBI_GOFER_RANDOM" below.

DRIVER-SPECIFIC ISSUES

Gofer needs to know about any driver-private attributes that should have their values sent back to the client.

If the driver doesn't support private_attribute_info() method, and very few do, then the module falls back to using some hard-coded details, if available, for the driver being used. Currently hard-coded details are available for the mysql, Pg, Sybase, and SQLite drivers.

TESTING

DBD::Gofer, DBD::Execute and related packages are well tested by executing the DBI test suite with DBI_AUTOPROXY configured to route all DBI calls via DBD::Gofer.

Because Gofer includes timeout and 'retry on error' mechanisms there is a need for some way to trigger delays and/or errors. This can be done via the `forced_gofer_random` configuration item, or else the DBI_GOFER_RANDOM environment variable.

DBI_GOFER_RANDOM

The value of the `forced_gofer_random` configuration item (or else the DBI_GOFER_RANDOM environment variable) is treated as a series of tokens separated by commas.

The tokens can be one of three types:

- fail=R%

  Set the current failure rate to R where R is a percentage. The value R can be floating point, e.g., `fail=0.05%`. Negative values for R have special meaning, see below.

- err=N

  Sets the current failure err value to N (instead of the DBI's default 'standard err value' of 2000000000). This is useful when you want to simulate a specific error.

- delayN=R%

  Set the current random delay rate to R where R is a percentage, and set the current delay duration to N seconds. The values of R and N can be floating point, e.g., `delay0.5=0.2%`. Negative values for R have special meaning, see below.

  If R is an odd number (R % 2 == 1) then a message is logged via warn() which will be returned to, and echoed at, the client.

- methodname

  Applies the current fail, err, and delay values to the named method. If neither a fail nor delay have been set yet then a warning is generated.

For example:

```
$executor = DBI::Gofer::Execute->new( {
  forced_gofer_random => "fail=0.01%,do,delay60=1%,execute",
});
```

will cause the do() method to fail for 0.01% of calls, and the execute() method to fail 0.01% of calls and be delayed by 60 seconds on 1% of calls.

If the percentage value (R) is negative then instead of the failures being triggered randomly (via the rand() function) they are triggered via a sequence number. In other words "`fail=-20%`" will mean every fifth call will fail. Each method has a distinct sequence number.

AUTHOR

Tim Bunce, http://www.tim.bunce.name

LICENCE AND COPYRIGHT

Copyright (c) 2007, Tim Bunce, Ireland. All rights reserved.

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself. See perlartistic.

DBI::Gofer::Request

Encapsulate a request from DBD::Gofer to DBI::Gofer::Execute

DESCRIPTION

This is an internal class.

AUTHOR

Tim Bunce, http://www.tim.bunce.name

LICENCE AND COPYRIGHT

Copyright (c) 2007, Tim Bunce, Ireland. All rights reserved.

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself. See perlartistic.

DBI::Gofer::Response

Encapsulate a response from DBI::Gofer::Execute to DBD::Gofer

DESCRIPTION

This is an internal class.

AUTHOR

Tim Bunce, http://www.tim.bunce.name

LICENCE AND COPYRIGHT

Copyright (c) 2007, Tim Bunce, Ireland. All rights reserved.

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself. See perlartistic.

DBI::Gofer::Serializer

Namespace containing the following modules:

- `DBI::Gofer::Serializer::Base` - base class for Gofer serialization
- `DBI::Gofer::Serializer::DataDumper` - Gofer serialization using DataDumper
- `DBI::Gofer::Serializer::Storable` - Gofer serialization using Storable

DBI::Gofer::Serializer::Base

base class for Gofer serialization

SYNOPSIS

```
$serializer = $serializer_class->new();

$string = $serializer->serialize( $data );
($string, $deserializer_class) = $serializer->serialize( $data );

$data = $serializer->deserialize( $string );
```

DESCRIPTION

DBI::Gofer::Serializer::* classes implement a very minimal subset of the Data::Serializer API.

Gofer serializers are expected to be very fast and are not required to deal with anything other than non-blessed references to arrays and hashes, and plain scalars.

DBI::Gofer::Serializer::DataDumper

Gofer serialization using DataDumper

SYNOPSIS

```
$serializer = DBI::Gofer::Serializer::DataDumper->new();

$string = $serializer->serialize( $data );
```

DESCRIPTION

Uses DataDumper to serialize. Deserialization is not supported. The output of this class is only meant for human consumption.

See also *DBI::Gofer::Serializer::Base*.

DBI::Gofer::Serializer::Storable

Gofer serialization using Storable

SYNOPSIS

```
$serializer = DBI::Gofer::Serializer::Storable->new();

$string = $serializer->serialize( $data );
($string, $deserializer_class) = $serializer->serialize( $data );

$data = $serializer->deserialize( $string );
```

DESCRIPTION

Uses Storable::nfreeze() to serialize and Storable::thaw() to deserialize.

The serialize() method sets local $Storable::forgive_me = 1; so it doesn't croak if it encounters any data types that can't be serialized, such as code refs.

See also *DBI::Gofer::Serializer::Base*.

DBI::Gofer::Transport

Namespace containing the following modules:

- `DBI::Gofer::Transport::Base` - Base class for Gofer transports
- `DBI::Gofer::Transport::pipeone` - DBD::Gofer server-side transport for pipeone
- `DBI::Gofer::Transport::stream` - DBD::Gofer server-side transport for stream

DBI::Gofer::Transport::Base

Base class for Gofer transports

DESCRIPTION

This is the base class for server-side Gofer transports.

It's also the base class for the client-side base class *DBD::Gofer::Transport::Base*.

This is an internal class.

AUTHOR

Tim Bunce, http://www.tim.bunce.name

LICENCE AND COPYRIGHT

Copyright (c) 2007, Tim Bunce, Ireland. All rights reserved.

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself. See perlartistic.

DBI::Gofer::Transport::pipeone

DBD::Gofer server-side transport for pipeone

SYNOPSIS

See *DBD::Gofer::Transport::pipeone*.

AUTHOR

Tim Bunce, http://www.tim.bunce.name

LICENCE AND COPYRIGHT

Copyright (c) 2007, Tim Bunce, Ireland. All rights reserved.

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself. See perlartistic.

DBI::Gofer::Transport::stream

DBD::Gofer server-side transport for stream

SYNOPSIS

See *DBD::Gofer::Transport::stream*.

AUTHOR

Tim Bunce, http://www.tim.bunce.name

LICENCE AND COPYRIGHT

Copyright (c) 2007, Tim Bunce, Ireland. All rights reserved.

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself. See perlartistic.

DBI::Profile

Performance profiling and benchmarking for the DBI

SYNOPSIS

The easiest way to enable DBI profiling is to set the DBI_PROFILE environment variable to 2 and then run your code as usual:

```
DBI_PROFILE=2 prog.pl
```

This will profile your program and then output a textual summary grouped by query when the program exits. You can also enable profiling by setting the Profile attribute of any DBI handle:

```
$dbh->{Profile} = 2;
```

Then the summary will be printed when the handle is destroyed.

Many other values apart from are possible - see “ENABLING A PROFILE” below.

DESCRIPTION

The DBI::Profile module provides a simple interface to collect and report performance and benchmarking data from the DBI.

For a more elaborate interface, suitable for larger programs, see *DBI::ProfileDumper* and dbiprof. For Apache/mod_perl applications see DBI::ProfileDumper::Apache.

OVERVIEW

Performance data collection for the DBI is built around several concepts which are important to understand clearly.

- Method Dispatch

  Every method call on a DBI handle passes through a single ‘dispatch’ function which manages all the common aspects of DBI method calls, such as handling the RaiseError attribute.

- Data Collection

  If profiling is enabled for a handle then the dispatch code takes a high-resolution timestamp soon after it is entered. Then, after calling the appropriate method and just before returning, it takes another high-resolution timestamp and calls a function to record the information. That function is passed the two timestamps plus the DBI handle and the name of the method that was called. That data about a single DBI method call is called a *profile sample*.

- Data Filtering

  If the method call was invoked by the DBI or by a driver then the call is ignored for profiling because the time spent will be accounted for by the original ‘outermost’ call for your code.

  For example, the calls that the selectrow_arrayref() method makes to prepare() and execute() etc. are not counted individually because the time spent in those methods is going to be allocated to the selectrow_arrayref() method when it returns. If this was not done then it would be very easy to double count time spent inside the DBI.

- Data Storage Tree

  The profile data is accumulated as 'leaves on a tree'. The 'path' through the branches of the tree to a particular leaf is determined dynamically for each sample. This is a key feature of DBI profiling.

  For each profiled method call the DBI walks along the Path and uses each value in the Path to step into and grow the Data tree.

  For example, if the Path is

```
[ 'foo', 'bar', 'baz' ]
```

  then the new profile sample data will be *merged* into the tree at

```
$h->{Profile}->{Data}->{foo}->{bar}->{baz}
```

  But it's not very useful to merge all the call data into one leaf node (except to get an overall 'time spent inside the DBI' total). It's more common to want the Path to include dynamic values such as the current statement text and/or the name of the method called to show what the time spent inside the DBI was for.

  The Path can contain some 'magic cookie' values that are automatically replaced by corresponding dynamic values when they're used. These magic cookies always start with a punctuation character.

  For example a value of '`!MethodName`' in the Path causes the corresponding entry in the Data to be the name of the method that was called. For example, if the Path was:

```
[ 'foo', '!MethodName', 'bar' ]
```

  and the selectall_arrayref() method was called, then the profile sample data for that call will be merged into the tree at:

```
$h->{Profile}->{Data}->{foo}->{selectall_arrayref}->{bar}
```

- Profile Data

  Profile data is stored at the 'leaves' of the tree as references to an array of numeric values. For example:

```
[
  106,                  # 0: count of samples at this node
  0.0312958955764771,   # 1: total duration
  0.000490069389343262, # 2: first duration
  0.000176072120666504, # 3: shortest duration
  0.00140702724456787,  # 4: longest duration
  1023115819.83019,     # 5: time of first sample
  1023115819.86576,     # 6: time of last sample
]
```

  After the first sample, later samples always update elements 0, 1, and 6, and may update 3 or 4 depending on the duration of the sampled call.

ENABLING A PROFILE

Profiling is enabled for a handle by assigning to the Profile attribute. For example:

```
$h->{Profile} = DBI::Profile->new();
```

The Profile attribute holds a blessed reference to a hash object that contains the profile data and attributes relating to it.

The class the Profile object is blessed into is expected to provide at least a DESTROY method which will dump the profile data to the DBI trace file handle (STDERR by default).

All these examples have the same effect as each other:

```
$h->{Profile} = 0;
$h->{Profile} = "/DBI::Profile";
$h->{Profile} = DBI::Profile->new();
$h->{Profile} = {};
$h->{Profile} = { Path => [] };
```

Similarly, these examples have the same effect as each other:

```
$h->{Profile} = 6;
$h->{Profile} = "6/DBI::Profile";
$h->{Profile} = "!Statement:!MethodName/DBI::Profile";
$h->{Profile} = { Path => [ '!Statement', '!MethodName' ] };
```

If a non-blessed hash reference is given then the DBI::Profile module is automatically `require`'d and the reference is blessed into that class.

If a string is given then it is processed like this:

```
($path, $module, $args) = split /\//, $string, 3

@path = split /:/, $path
@args = split /:/, $args

eval "require $module" if $module
$module ||= "DBI::Profile"

$module->new( Path => \@Path, @args )
```

So the first value is used to select the Path to be used (see below). The second value, if present, is used as the name of a module which will be loaded and it's new method called. If not present it defaults to DBI::Profile. Any other values are passed as arguments to the new method. For example: "`2/DBIx::OtherProfile/Foo:42`".

Numbers can be used as a shorthand way to enable common Path values. The simplest way to explain how the values are interpreted is to show the code:

```
push @Path, "DBI"           if $path_elem & 0x01;
push @Path, "!Statement"    if $path_elem & 0x02;
push @Path, "!MethodName"   if $path_elem & 0x04;
```

(continues on next page)

(continued from previous page)

```
push @Path, "!MethodClass"  if $path_elem & 0x08;
push @Path, "!Caller2"      if $path_elem & 0x10;
```

So “2” is the same as “!Statement” and “6” (2+4) is the same as “!Statement:!Method”. Those are the two most commonly used values. Using a negative number will reverse the path. Thus “-6” will group by method name then statement.

The splitting and parsing of string values assigned to the Profile attribute may seem a little odd, but there’s a good reason for it. Remember that attributes can be embedded in the Data Source Name string which can be passed in to a script as a parameter. For example:

```
dbi:DriverName(Profile=>2):dbname
dbi:DriverName(Profile=>{Username}:!Statement/MyProfiler/
 ↪Foo:42):dbname
```

And also, if the `DBI_PROFILE` environment variable is set then The DBI arranges for every driver handle to share the same profile object. When perl exits a single profile summary will be generated that reflects (as nearly as practical) the total use of the DBI by the application.

THE PROFILE OBJECT

The DBI core expects the Profile attribute value to be a hash reference and if the following values don’t exist it will create them as needed:

Data

A reference to a hash containing the collected profile data.

Path

The Path value is a reference to an array. Each element controls the value to use at the corresponding level of the profile Data tree.

If the value of Path is anything other than an array reference, it is treated as if it was:

```
[ '!Statement' ]
```

The elements of Path array can be one of the following types:

Special Constant

!Statement

Use the current Statement text. Typically that’s the value of the Statement attribute for the handle the method was called with. Some methods, like commit() and rollback(), are unrelated to a particular statement. For those methods !Statement records an empty string.

For statement handles this is always simply the string that was given to prepare() when the handle was created. For database handles this is the statement that was last prepared

or executed on that database handle. That can lead to a little 'fuzziness' because, for example, calls to the quote() method to build a new statement will typically be associated with the previous statement. In practice this isn't a significant issue and the dynamic Path mechanism can be used to setup your own rules.

!MethodName

Use the name of the DBI method that the profile sample relates to.

!MethodClass

Use the fully qualified name of the DBI method, including the package, that the profile sample relates to. This shows you where the method was implemented. For example:

```
'DBD::_::db::selectrow_arrayref' =>
    0.022902s
'DBD::mysql::db::selectrow_arrayref' =>
    2.244521s / 99 = 0.022445s avg (first 0.022813s, min 0.022051s,␣
↪max 0.028932s)
```

The "DBD::_::db::selectrow_arrayref" shows that the driver has inherited the selectrow_arrayref method provided by the DBI.

But you'll note that there is only one call to DBD::_::db::selectrow_arrayref but another 99 to DBD::mysql::db::selectrow_arrayref. Currently the first call doesn't record the true location. That may change.

!Caller

Use a string showing the filename and line number of the code calling the method.

!Caller2

Use a string showing the filename and line number of the code calling the method, as for !Caller, but also include filename and line number of the code that called that. Calls from DBI:: and DBD:: packages are skipped.

!File

Same as !Caller above except that only the filename is included, not the line number.

!File2

Same as !Caller2 above except that only the filenames are included, not the line number.

!Time

Use the current value of time(). Rarely used. See the more useful `!Time~N` below.

!Time~N

Where `N` is an integer. Use the current value of time() but with reduced precision. The value used is determined in this way:

```
int( time() / N ) * N
```

This is a useful way to segregate a profile into time slots. For example:

```
[ '!Time~60', '!Statement' ]
```

Code Reference

The subroutine is passed the handle it was called on and the DBI method name. The current Statement is in `$_`. The statement string should not be modified, so most subs start with `local $_ = $_;`.

The list of values it returns is used at that point in the Profile Path. Any undefined values are treated as the string "`undef`".

The sub can 'veto' (reject) a profile sample by including a reference to undef (`\undef`) in the returned list. That can be useful when you want to only profile statements that match a certain pattern, or only profile certain methods.

Subroutine Specifier

A Path element that begins with '&' is treated as the name of a subroutine in the DBI::ProfileSubs namespace and replaced with the corresponding code reference.

Currently this only works when the Path is specified by the `DBI_PROFILE` environment variable.

Also, currently, the only subroutine in the DBI::ProfileSubs namespace is `'&norm_std_n3'`. That's a very handy subroutine when profiling code that doesn't use placeholders. See *DBI::ProfileSubs* for more information.

Attribute Specifier

A string enclosed in braces, such as '`{Username}`', specifies that the current value of the corresponding database handle attribute should be used at that point in the Path.

Reference to a Scalar

Specifies that the current value of the referenced scalar be used at that point in the Path. This provides an efficient way to get 'contextual' values into your profile.

Other Values

Any other values are stringified and used literally.

(References, and values that begin with punctuation characters are reserved.)

REPORTING

Report Format

The current accumulated profile data can be formatted and output using

```
print $h->{Profile}->format;
```

To discard the profile data and start collecting fresh data you can do:

```
$h->{Profile}->{Data} = undef;
```

The default results format looks like this:

```
DBI::Profile: 0.001015s 42.7% (5 calls) programname @ YYYY-MM-DD
↪HH:MM:SS
'' =>
    0.000024s / 2 = 0.000012s avg (first 0.000015s, min 0.000009s,
↪max 0.000015s)
'SELECT mode,size,name FROM table' =>
    0.000991s / 3 = 0.000330s avg (first 0.000678s, min 0.000009s,
↪max 0.000678s)
```

Which shows the total time spent inside the DBI, with a count of the total number of method calls and the name of the script being run, then a formatted version of the profile data tree.

If the results are being formatted when the perl process is exiting (which is usually the case when the DBI_PROFILE environment variable is used) then the percentage of time the process spent inside the DBI is also shown. If the process is not exiting then the percentage is calculated using the time between the first and last call to the DBI.

In the example above the paths in the tree are only one level deep and use the Statement text as the value (that's the default behaviour).

The merged profile data at the 'leaves' of the tree are presented as total time spent, count, average time spent (which is simply total time divided by the count), then the time spent on the first call, the time spent on the fastest call, and finally the time spent on the slowest call.

The 'avg', 'first', 'min' and 'max' times are not particularly useful when the profile data path only contains the statement text. Here's an extract of a more detailed example using both statement text and method name in the path:

```
'SELECT mode,size,name FROM table' =>
    'FETCH' =>
        0.000076s
    'fetchrow_hashref' =>
        0.036203s / 108 = 0.000335s avg (first 0.000490s, min 0.
↪000152s, max 0.002786s)
```

Here you can see the 'avg', 'first', 'min' and 'max' for the 108 calls to fetchrow_hashref() become rather more interesting. Also the data for FETCH just shows a time value because it was only called once.

Currently the profile data is output sorted by branch names. That may change in a later version so the leaf nodes are sorted by total time per leaf node.

Report Destination

The default method of reporting is for the DESTROY method of the Profile object to format the results and write them using:

```
DBI->trace_msg($results, 0);  # see $ON_DESTROY_DUMP below
```

to write them to the DBI trace() filehandle (which defaults to STDERR). To direct the DBI trace filehandle to write to a file without enabling tracing the trace() method can be called with a trace level of 0. For example:

```
DBI->trace(0, $filename);
```

The same effect can be achieved without changing the code by setting the DBI_TRACE environment variable to 0=filename.

The $DBI::Profile::ON_DESTROY_DUMP variable holds a code ref that's called to perform the output of the formatted results. The default value is:

```
$ON_DESTROY_DUMP = sub { DBI->trace_msg($results, 0) };
```

Apart from making it easy to send the dump elsewhere, it can also be useful as a simple way to disable dumping results.

CHILD HANDLES

Child handles inherit a reference to the Profile attribute value of their parent. So if profiling is enabled for a database handle then by default the statement handles created from it all contribute to the same merged profile data tree.

PROFILE OBJECT METHODS

format

See "REPORTING".

as_node_path_list

```
@ary = $dbh->{Profile}->as_node_path_list();
@ary = $dbh->{Profile}->as_node_path_list($node, $path);
```

Returns the collected data ($dbh->{Profile}{Data}) restructured into a list of array refs, one for each leaf node in the Data tree. This 'flat' structure is often much simpler for applications to work with.

The first element of each array ref is a reference to the leaf node. The remaining elements are the 'path' through the data tree to that node.

For example, given a data tree like this:

```
{key1a}{key2a}[node1]
{key1a}{key2b}[node2]
{key1b}{key2a}{key3a}[node3]
```

The as_node_path_list() method will return this list:

```
[ [node1], 'key1a', 'key2a' ]
[ [node2], 'key1a', 'key2b' ]
[ [node3], 'key1b', 'key2a', 'key3a' ]
```

The nodes are ordered by key, depth-first.

The $node argument can be used to focus on a sub-tree. If not specified it defaults to $dbh->{Profile}{Data}.

The $path argument can be used to specify a list of path elements that will be added to each element of the returned list. If not specified it defaults to a ref to an empty array.

as_text

```
@txt = $dbh->{Profile}->as_text();
$txt = $dbh->{Profile}->as_text({
    node      => undef,
    path      => [],
    separator => " > ",
    format    => '%1$s: %11$fs / %10$d = %2$fs avg (first %12$fs,␣
↪min %13$fs, max %14$fs)'."\n";
    sortsub   => sub { ... },
);
```

Returns the collected data ($dbh->{Profile}{Data}) reformatted into a list of formatted strings. In scalar context the list is returned as a single concatenated string.

A hashref can be used to pass in arguments, the default values are shown in the example above.

The node and <path> arguments are passed to as_node_path_list().

The `separator` argument is used to join the elements of the path for each leaf node.

The `sortsub` argument is used to pass in a ref to a sub that will order the list. The subroutine will be passed a reference to the array returned by as_node_path_list() and should sort the contents of the array in place. The return value from the sub is ignored. For example, to sort the nodes by the second level key you could use:

```
sortsub => sub { my $ary=shift; @$ary = sort { $a->[2] cmp $b->[2] }␣
↪@$ary }
```

The `format` argument is a `sprintf` format string that specifies the format to use for each leaf node. It uses the explicit format parameter index mechanism to specify which of the arguments should appear where in the string. The arguments to sprintf are:

```
 1:  path to node, joined with the separator
 2:  average duration (total duration/count)
     (3 thru 9 are currently unused)
10:  count
11:  total duration
12:  first duration
13:  smallest duration
14:  largest duration
15:  time of first call
16:  time of first call
```

CUSTOM DATA MANIPULATION

Recall that `$h->{Profile}->{Data}` is a reference to the collected data. Either to a 'leaf' array (when the Path is empty, i.e., DBI_PROFILE env var is 1), or a reference to hash containing values that are either further hash references or leaf array references.

Sometimes it's useful to be able to summarise some or all of the collected data. The dbi_profile_merge_nodes() function can be used to merge leaf node values.

dbi_profile_merge_nodes

```
use DBI qw(dbi_profile_merge_nodes);

$time_in_dbi = dbi_profile_merge_nodes(my $totals=[], @$leaves);
```

Merges profile data node. Given a reference to a destination array, and zero or more references to profile data, merges the profile data into the destination array. For example:

```
$time_in_dbi = dbi_profile_merge_nodes(
    my $totals=[],
    [ 10, 0.51, 0.11, 0.01, 0.22, 1023110000, 1023110010 ],
    [ 15, 0.42, 0.12, 0.02, 0.23, 1023110005, 1023110009 ],
);
```

$totals will then contain

```
[ 25, 0.93, 0.11, 0.01, 0.23, 1023110000, 1023110010 ]
```

and $time_in_dbi will be 0.93;

The second argument need not be just leaf nodes. If given a reference to a hash then the hash is recursively searched for leaf nodes and all those found are merged.

For example, to get the time spent 'inside' the DBI during an http request, your logging code run at the end of the request (i.e. mod_perl LogHandler) could use:

```
my $time_in_dbi = 0;
if (my $Profile = $dbh->{Profile}) { # if DBI profiling is enabled
    $time_in_dbi = dbi_profile_merge_nodes(my $total=[], $Profile->
↪{Data});
    $Profile->{Data} = {}; # reset the profile data
}
```

If profiling has been enabled then $time_in_dbi will hold the time spent inside the DBI for that handle (and any other handles that share the same profile data) since the last request.

Prior to DBI 1.56 the dbi_profile_merge_nodes() function was called dbi_profile_merge(). That name still exists as an alias.

CUSTOM DATA COLLECTION

Using The Path Attribute

```
XXX example to be added later using a selectall_arrayref call
XXX nested inside a fetch loop where the first column of the
XXX outer loop is bound to the profile Path using
XXX bind_column(1, \${ $dbh->{Profile}->{Path}->[0] })
XXX so you end up with separate profiles for each loop
XXX (patches welcome to add this to the docs :)
```

Adding Your Own Samples

The dbi_profile() function can be used to add extra sample data into the profile data tree. For example:

```
use DBI;
use DBI::Profile (dbi_profile dbi_time);

my $t1 = dbi_time(); # floating point high-resolution time

... execute code you want to profile here ...

my $t2 = dbi_time();
dbi_profile($h, $statement, $method, $t1, $t2);
```

The $h parameter is the handle the extra profile sample should be associated with. The $statement parameter is the string to use where the Path specifies !Statement. If $statement is undef then $h->{Statement} will be used. Similarly $method is the string to use if the Path specifies !MethodName. There is no default value for $method.

The $h->{Profile}{Path} attribute is processed by dbi_profile() in the usual way.

The $h parameter is usually a DBI handle but it can also be a reference to a hash, in which case the dbi_profile() acts on each defined value in the hash. This is an efficient way to update multiple profiles with a single sample, and is used by the DashProfiler module.

SUBCLASSING

Alternate profile modules must subclass DBI::Profile to help ensure they work with future versions of the DBI.

CAVEATS

Applications which generate many different statement strings (typically because they don't use placeholders) and profile with !Statement in the Path (the default) will consume memory in the Profile Data structure for each statement. Use a code ref in the Path to return an edited (simplified) form of the statement.

If a method throws an exception itself (not via RaiseError) then it won't be counted in the profile.

If a HandleError subroutine throws an exception (rather than returning 0 and letting RaiseError do it) then the method call won't be counted in the profile.

Time spent in DESTROY is added to the profile of the parent handle.

Time spent in DBI->*() methods is not counted. The time spent in the driver connect method, $drh->connect(), when it's called by DBI->connect is counted if the DBI_PROFILE environment variable is set.

Time spent fetching tied variables, $DBI::errstr, is counted.

Time spent in FETCH for $h->{Profile} is not counted, so getting the profile data doesn't alter it.

DBI::PurePerl does not support profiling (though it could in theory).

For asynchronous queries, time spent while the query is running on the backend is not counted.

A few platforms don't support the gettimeofday() high resolution time function used by the DBI (and available via the dbi_time() function). In which case you'll get integer resolution time which is mostly useless.

On Windows platforms the dbi_time() function is limited to millisecond resolution. Which isn't sufficiently fine for our needs, but still much better than integer resolution. This limited resolution means that fast method calls will often register as taking 0 time. And timings in general will have much more 'jitter' depending on where within the 'current millisecond' the start and end timing was taken.

This documentation could be more clear. Probably needs to be reordered to start with several examples and build from there. Trying to explain the concepts first seems painful and to lead to just as many forward references. (Patches welcome!)

DBI::ProfileData

manipulate DBI::ProfileDumper data dumps

SYNOPSIS

The easiest way to use this module is through the dbiprof frontend (see dbiprof for details):

```
dbiprof --number 15 --sort count
```

This module can also be used to roll your own profile analysis:

```
local $DBI::ProfileData::MAX_PATH_DEPTH = 256; # override the␣
↪maximum path depth of files

# load data from dbi.prof
$prof = DBI::ProfileData->new(File => "dbi.prof");

# get a count of the records (unique paths) in the data set
$count = $prof->count();
```

(continues on next page)

(continued from previous page)

```
# sort by longest overall time
$prof->sort(field => "longest");

# sort by longest overall time, least to greatest
$prof->sort(field => "longest", reverse => 1);

# exclude records with key2 eq 'disconnect'
$prof->exclude(key2 => 'disconnect');

# exclude records with key1 matching /^UPDATE/i
$prof->exclude(key1 => qr/^UPDATE/i);

# remove all records except those where key1 matches /^SELECT/i
$prof->match(key1 => qr/^SELECT/i);

# produce a formatted report with the given number of items
$report = $prof->report(number => 10);

# clone the profile data set
$clone = $prof->clone();

# get access to hash of header values
$header = $prof->header();

# get access to sorted array of nodes
$nodes = $prof->nodes();

# format a single node in the same style as report()
$text = $prof->format($nodes->[0]);

# get access to Data hash in DBI::Profile format
$Data = $prof->Data();
```

DESCRIPTION

This module offers the ability to read, manipulate and format *DBI::ProfileDumper* profile data.

Conceptually, a profile consists of a series of records, or nodes, each of each has a set of statistics and set of keys. Each record must have a unique set of keys, but there is no requirement that every record have the same number of keys.

METHODS

The following methods are supported by DBI::ProfileData objects.

$prof = DBI::ProfileData->new(File => "dbi.prof")

$prof = DBI::ProfileData->new(File => "dbi.prof", Filter => sub { ... })

$prof = DBI::ProfileData->new(Files => ["dbi.prof.1", "dbi.prof.2"])

Creates a new DBI::ProfileData object. Takes either a single file through the File option or a list of Files in an array ref. If multiple files are specified then the header data from the first file is used.

Files

Reference to an array of file names to read.

File

Name of file to read. Takes precedence over `Files`.

DeleteFiles

If true, the files are deleted after being read.

Actually the files are renamed with a `deleteme` suffix before being read, and then, after reading all the files, they're all deleted together.

The files are locked while being read which, combined with the rename, makes it safe to 'consume' files that are still being generated by *DBI::ProfileDumper*.

Filter

The `Filter` parameter can be used to supply a code reference that can manipulate the profile data as it is being read. This is most useful for editing SQL statements so that slightly different statements in the raw data will be merged and aggregated in the loaded data. For example:

```
Filter => sub {
    my ($path_ref, $data_ref) = @_;
    s/foo = '.*?'/foo = '...'/ for @$path_ref;
}
```

Here's an example that performs some normalization on the SQL. It converts all numbers to `N` and all quoted strings to `S`. It can also convert digits to `N` within names. Finally, it summarizes long "IN (...)" clauses.

It's aggressive and simplistic, but it's often sufficient, and serves as an example that you can tailor to suit your own needs:

```
Filter => sub {
    my ($path_ref, $data_ref) = @_;
    local $_ = $path_ref->[0]; # whichever element contains the SQL␣
↪Statement
    s/\b\d+\b/N/g;             # 42 -> N
    s/\b0x[0-9A-Fa-f]+\b/N/g;  # 0xFE -> N
    s/'.*?'/'S'/g;             # single quoted strings (doesn't␣
↪handle escapes)
    s/".*?"/"S"/g;             # double quoted strings (doesn't␣
↪handle escapes)
    # convert names like log_20001231 into log_NNNNNNNN, controlled␣
↪by $opt{n}
    s/([a-z_]+)(\d{$opt{n},})/$1.('N' x length($2))/ieg if $opt{n};
    # abbreviate massive "in (...)" statements and similar
    s!(([NS],){100,})!sprintf("$2,{repeated %d times}",length($1)/
↪2)!eg;
}
```

It's often better to perform this kinds of normalization in the DBI while the data is being collected, to avoid too much memory being used by storing profile data for many different SQL statement. See *DBI::Profile*.

$copy = $prof->clone();

Clone a profile data set creating a new object.

$header = $prof->header();

Returns a reference to a hash of header values. These are the key value pairs included in the header section of the *DBI::ProfileDumper* data format. For example:

```
$header = {
  Path    => [ '!Statement', '!MethodName' ],
  Program => 't/42profile_data.t',
};
```

Note that modifying this hash will modify the header data stored inside the profile object.

$nodes = $prof->nodes()

Returns a reference the sorted nodes array. Each element in the array is a single record in the data set. The first seven elements are the same as the elements provided by *DBI::Profile*. After that each key is in a separate element. For example:

```
$nodes = [
           [
             2,                     # 0, count
             0.0312958955764771,    # 1, total duration
             0.000490069389343262,  # 2, first duration
             0.000176072120666504,  # 3, shortest duration
```

(continues on next page)

(continued from previous page)

```
            0.00140702724456787,    # 4, longest duration
            1023115819.83019,       # 5, time of first event
            1023115819.86576,       # 6, time of last event
            'SELECT foo FROM bar'   # 7, key1
            'execute'               # 8, key2
                                    # 6+N, keyN
          ],
                                    # ...
        ];
```

Note that modifying this array will modify the node data stored inside the profile object.

$count = $prof->count()

Returns the number of items in the profile data set.

$prof->sort(field => "field")

$prof->sort(field => "field", reverse => 1)

Sorts data by the given field. Available fields are:

```
longest
total
count
shortest
```

The default sort is greatest to smallest, which is the opposite of the normal Perl meaning. This, however, matches the expected behavior of the dbiprof frontend.

$count = $prof->exclude(key2 => "disconnect")

$count = $prof->exclude(key2 => "disconnect", case_sensitive => 1)

$count = $prof->exclude(key1 => qr/^SELECT/i)

Removes records from the data set that match the given string or regular expression. This method modifies the data in a permanent fashion - use clone() first to maintain the original data after exclude(). Returns the number of nodes left in the profile data set.

$count = $prof->match(key2 => "disconnect")

$count = $prof->match(key2 => "disconnect", case_sensitive => 1)

$count = $prof->match(key1 => qr/^SELECT/i)

Removes records from the data set that do not match the given string or regular expression. This method modifies the data in a permanent fashion - use clone() first to maintain the original data after match(). Returns the number of nodes left in the profile data set.

$Data = $prof->Data()

Returns the same Data hash structure as seen in *DBI::Profile*. This structure is not sorted. The nodes() structure probably makes more sense for most analysis.

$text = $prof->format($nodes->[0])

Formats a single node into a human-readable block of text.

$text = $prof->report(number => 10)

Produces a report with the given number of items.

AUTHOR

Sam Tregar sam@tregar.com

COPYRIGHT AND LICENSE

Copyright (C) 2002 Sam Tregar

This program is free software; you can redistribute it and/or modify it under the same terms as Perl 5 itself.

DBI::ProfileDumper

profile DBI usage and output data to a file

SYNOPSIS

To profile an existing program using DBI::ProfileDumper, set the DBI_PROFILE environment variable and run your program as usual. For example, using bash:

```
DBI_PROFILE=2/DBI::ProfileDumper program.pl
```

Then analyze the generated file (`dbi.prof`) with dbiprof:

```
dbiprof
```

You can also activate DBI::ProfileDumper from within your code:

```
use DBI;

# profile with default path (2) and output file (dbi.prof)
$dbh->{Profile} = "!Statement/DBI::ProfileDumper";

# same thing, spelled out
$dbh->{Profile} = "!Statement/DBI::ProfileDumper/File:dbi.prof";

# another way to say it
use DBI::ProfileDumper;
```

(continues on next page)

(continued from previous page)

```
$dbh->{Profile} = DBI::ProfileDumper->new(
                      Path => [ '!Statement' ],
                      File => 'dbi.prof' );

# using a custom path
$dbh->{Profile} = DBI::ProfileDumper->new(
    Path => [ "foo", "bar" ],
    File => 'dbi.prof',
);
```

DESCRIPTION

DBI::ProfileDumper is a subclass of *DBI::Profile* which dumps profile data to disk instead of printing a summary to your screen. You can then use dbiprof to analyze the data in a number of interesting ways, or you can roll your own analysis using *DBI::ProfileData*.

NOTE: For Apache/mod_perl applications, use DBI::ProfileDumper::Apache.

USAGE

One way to use this module is just to enable it in your `$dbh`:

```
$dbh->{Profile} = "1/DBI::ProfileDumper";
```

This will write out profile data by statement into a file called `dbi.prof`. If you want to modify either of these properties, you can construct the DBI::ProfileDumper object yourself:

```
use DBI::ProfileDumper;
$dbh->{Profile} = DBI::ProfileDumper->new(
    Path => [ '!Statement' ],
    File => 'dbi.prof'
);
```

The `Path` option takes the same values as in *DBI::Profile*. The `File` option gives the name of the file where results will be collected. If it already exists it will be overwritten.

You can also activate this module by setting the DBI_PROFILE environment variable:

```
$ENV{DBI_PROFILE} = "!Statement/DBI::ProfileDumper";
```

This will cause all DBI handles to share the same profiling object.

METHODS

The following methods are available to be called using the profile object. You can get access to the profile object from the Profile key in any DBI handle:

```
my $profile = $dbh->{Profile};
```

flush_to_disk

```
$profile->flush_to_disk()
```

Flushes all collected profile data to disk and empties the Data hash. Returns the filename written to. If no profile data has been collected then the file is not written and flush_to_disk() returns undef.

The file is locked while it's being written. A process 'consuming' the files while they're being written to, should rename the file first, then lock it, then read it, then close and delete it. The `DeleteFiles` option to *DBI::ProfileData* does the right thing.

This method may be called multiple times during a program run.

empty

```
$profile->empty()
```

Clears the Data hash without writing to disk.

filename

```
$filename = $profile->filename();
```

Get or set the filename.

The filename can be specified as a CODE reference, in which case the referenced code should return the filename to be used. The code will be called with the profile object as its first argument.

DATA FORMAT

The data format written by DBI::ProfileDumper starts with a header containing the version number of the module used to generate it. Then a block of variable declarations describes the profile. After two newlines, the profile data forms the body of the file. For example:

```
DBI::ProfileDumper 2.003762
Path = [ '!Statement', '!MethodName' ]
Program = t/42profile_data.t

+ 1 SELECT name FROM users WHERE id = ?
+ 2 prepare
= 1 0.0312958955764771 0.000490069389343262 0.000176072120666504 0.
 ↪00140702724456787 1023115819.83019 1023115819.86576
+ 2 execute
1 0.0312958955764771 0.000490069389343262 0.000176072120666504 0.
 ↪00140702724456787 1023115819.83019 1023115819.86576
+ 2 fetchrow_hashref
= 1 0.0312958955764771 0.000490069389343262 0.000176072120666504 0.
 ↪00140702724456787 1023115819.83019 1023115819.86576
```

(continues on next page)

(continued from previous page)

```
+ 1 UPDATE users SET name = ? WHERE id = ?
+ 2 prepare
= 1 0.0312958955764771 0.000490069389343262 0.000176072120666504 0.
 ↪00140702724456787 1023115819.83019 1023115819.86576
+ 2 execute
= 1 0.0312958955764771 0.000490069389343262 0.000176072120666504 0.
 ↪00140702724456787 1023115819.83019 1023115819.86576
```

The lines beginning with + signs signify keys. The number after the + sign shows the nesting level of the key. Lines beginning with = are the actual profile data, in the same order as in DBI::Profile.

Note that the same path may be present multiple times in the data file since `format()` may be called more than once. When read by DBI::ProfileData the data points will be merged to produce a single data set for each distinct path.

The key strings are transformed in three ways. First, all backslashes are doubled. Then all newlines and carriage-returns are transformed into `\n` and `\r` respectively. Finally, any NULL bytes (`\0`) are entirely removed. When DBI::ProfileData reads the file the first two transformations will be reversed, but NULL bytes will not be restored.

AUTHOR

Sam Tregar sam@tregar.com

COPYRIGHT AND LICENSE

Copyright (C) 2002 Sam Tregar

This program is free software; you can redistribute it and/or modify it under the same terms as Perl 5 itself.

DBI::ProfileSubs

Subroutines for dynamic profile Path

SYNOPSIS

```
DBI_PROFILE='&norm_std_n3' prog.pl
```

This is new and still experimental.

TO DO

Define come kind of naming convention for the subs.

DBI::ProxyServer

a server for the DBD::Proxy driver

SYNOPSIS

```
use DBI::ProxyServer;
DBI::ProxyServer::main(@ARGV);
```

DESCRIPTION

DBI::Proxy Server is a module for implementing a proxy for the DBI proxy driver, DBD::Proxy. It allows access to databases over the network if the DBMS does not offer networked operations. But the proxy server might be useful for you, even if you have a DBMS with integrated network functionality: It can be used as a DBI proxy in a firewalled environment.

DBI::ProxyServer runs as a daemon on the machine with the DBMS or on the firewall. The client connects to the agent using the DBI driver DBD::Proxy, thus in the exactly same way than using DBD::mysql, DBD::mSQL or any other DBI driver.

The agent is implemented as a RPC::PlServer application. Thus you have access to all the possibilities of this module, in particular encryption and a similar configuration file. DBI::ProxyServer adds the possibility of query restrictions: You can define a set of queries that a client may execute and restrict access to those. (Requires a DBI driver that supports parameter binding.) See “CONFIGURATION FILE”.

The provided driver script, dbiproxy, may either be used as it is or used as the basis for a local version modified to meet your needs.

OPTIONS

When calling the DBI::ProxyServer::main() function, you supply an array of options. These options are parsed by the Getopt::Long module. The ProxyServer inherits all of RPC::PlServer’s and hence Net::Daemon’s options and option handling, in particular the ability to read options from either the command line or a config file. See RPC::PlServer. See Net::Daemon. Available options include

- *chroot* (**–chroot=dir**)

  (UNIX only) After doing a bind(), change root directory to the given directory by doing a chroot(). This is useful for security, but it restricts the environment a lot. For example, you need to load DBI drivers in the config file or you have to create hard links to Unix sockets, if your drivers are using them. For example, with MySQL, a config file might contain the following lines:

  ```
  my $rootdir = '/var/dbiproxy';
  my $unixsockdir = '/tmp';
  my $unixsockfile = 'mysql.sock';
  foreach $dir ($rootdir, "$rootdir$unixsockdir") {
      mkdir 0755, $dir;
  }
  ```

(continues on next page)

(continued from previous page)

```
link("$unixsockdir/$unixsockfile",
     "$rootdir$unixsockdir/$unixsockfile");
require DBD::mysql;

{
    'chroot' => $rootdir,
    ...
}
```

If you don't know chroot(), think of an FTP server where you can see a certain directory tree only after logging in. See also the –group and –user options.

- *clients*

  An array ref with a list of clients. Clients are hash refs, the attributes *accept* (0 for denying access and 1 for permitting) and *mask*, a Perl regular expression for the clients IP number or its host name.

- *configfile* (**–configfile=file**)

  Config files are assumed to return a single hash ref that overrides the arguments of the new method. However, command line arguments in turn take precedence over the config file. See the "CONFIGURATION FILE" section below for details on the config file.

- *debug* (**–debug**)

  Turn debugging mode on. Mainly this asserts that logging messages of level "debug" are created.

- *facility* (**–facility=mode**)

  (UNIX only) Facility to use for *Sys::Syslog*. The default is **daemon**.

- *group* (**–group=gid**)

  After doing a bind(), change the real and effective GID to the given. This is useful, if you want your server to bind to a privileged port (<1024), but don't want the server to execute as root. See also the –user option.

  GID's can be passed as group names or numeric values.

- *localaddr* (**–localaddr=ip**)

  By default a daemon is listening to any IP number that a machine has. This attribute allows one to restrict the server to the given IP number.

- *localport* (**–localport=port**)

  This attribute sets the port on which the daemon is listening. It must be given somehow, as there's no default.

- *logfile* (**–logfile=file**)

  Be default logging messages will be written to the syslog (Unix) or to the event log (Windows NT). On other operating systems you need to specify a log file. The special value "STDERR" forces logging to stderr. See Net::Daemon::Log for details.

- *mode* (–**mode**=**modename**)

  The server can run in three different modes, depending on the environment.

  If you are running Perl 5.005 and did compile it for threads, then the server will create a new thread for each connection. The thread will execute the server's Run() method and then terminate. This mode is the default, you can force it with "–mode=threads".

  If threads are not available, but you have a working fork(), then the server will behave similar by creating a new process for each connection. This mode will be used automatically in the absence of threads or if you use the "–mode=fork" option.

  Finally there's a single-connection mode: If the server has accepted a connection, he will enter the Run() method. No other connections are accepted until the Run() method returns (if the client disconnects). This operation mode is useful if you have neither threads nor fork(), for example on the Macintosh. For debugging purposes you can force this mode with "–mode=single".

- *pidfile* (–**pidfile**=**file**)

  (UNIX only) If this option is present, a PID file will be created at the given location. Default is to not create a pidfile.

- *user* (–**user**=**uid**)

  After doing a bind(), change the real and effective UID to the given. This is useful, if you want your server to bind to a privileged port (<1024), but don't want the server to execute as root. See also the –group and the –chroot options.

  UID's can be passed as group names or numeric values.

- *version* (–**version**)

  Suppresses startup of the server; instead the version string will be printed and the program exits immediately.

SHUTDOWN

DBI::ProxyServer is built on RPC::PlServer which is, in turn, built on Net::Daemon.

You should refer to Net::Daemon for how to shutdown the server, except that you can't because it's not currently documented there (as of v0.43). The bottom-line is that it seems that there's no support for graceful shutdown.

CONFIGURATION FILE

The configuration file is just that of *RPC::PlServer* or *Net::Daemon* with some additional attributes in the client list.

The config file is a Perl script. At the top of the file you may include arbitrary Perl source, for example load drivers at the start (useful to enhance performance), prepare a chroot environment and so on.

The important thing is that you finally return a hash ref of option name/value pairs. The possible options are listed above.

All possibilities of Net::Daemon and RPC::PlServer apply, in particular

- Host and/or User dependent access control
- Host and/or User dependent encryption
- Changing UID and/or GID after binding to the port
- Running in a chroot() environment

Additionally the server offers you query restrictions. Suggest the following client list:

```
'clients' => [
    { 'mask' => '^admin\.company\.com$',
      'accept' => 1,
      'users' => [ 'root', 'wwwrun' ],
    },
    {
      'mask' => '^admin\.company\.com$',
      'accept' => 1,
      'users' => [ 'root', 'wwwrun' ],
      'sql' => {
          'select' => 'SELECT * FROM foo',
          'insert' => 'INSERT INTO foo VALUES (?, ?, ?)'
          }
    }
```

then only the users root and wwwrun may connect from admin.company.com, executing arbitrary queries, but only wwwrun may connect from other hosts and is restricted to

```
$sth->prepare("select");
```

or

```
$sth->prepare("insert");
```

which in fact are “SELECT * FROM foo” or “INSERT INTO foo VALUES (?, ?, ?)”.

Proxyserver Configuration file (bigger example)

This section tells you how to restrict a DBI-Proxy: Not every user from every workstation shall be able to execute every query.

There is a perl program “dbiproxy” which runs on a machine which is able to connect to all the databases we wish to reach. All Perl-DBD-drivers must be installed on this machine. You can also reach databases for which drivers are not available on the machine where you run the program querying the database, e.g. ask MS-Access-database from Linux.

Create a configuration file “proxy_oracle.cfg” at the dbproxy-server:

```
{
    # This shall run in a shell or a DOS-window
    # facility => 'daemon',
    pidfile => 'your_dbiproxy.pid',
    logfile => 1,
```

(continues on next page)

(continued from previous page)

```
    debug => 0,
    mode => 'single',
    localport => '12400',

    # Access control, the first match in this list wins!
    # So the order is important
    clients => [
            # hint to organize:
            # the most specialized rules for single machines/users
↪are 1st
            # then the denying rules
            # then the rules about whole networks

            # rule: internal_webserver
            # desc: to get statistical information
            {
                    # this IP-address only is meant
                    mask => '^10\.95\.81\.243$',
                    # accept (not defer) connections like this
                    accept => 1,
                    # only users from this list
                    # are allowed to log on
                    users => [ 'informationdesk' ],
                    # only this statistical query is allowed
                    # to get results for a web-query
                    sql => {
                            alive => 'select count(*) from dual',
                            statistic_area => 'select count(*) from
↪e01admin.e01e203 where geb_bezei like ?',
                    }
            },

            # rule: internal_bad_guy_1
            {
                    mask => '^10\.95\.81\.1$',
                    accept => 0,
            },

            # rule: employee_workplace
            # desc: get detailed information
            {
                    # any IP-address is meant here
                    mask => '^10\.95\.81\.(\d+)$',
                    # accept (not defer) connections like this
                    accept => 1,
                    # only users from this list
                    # are allowed to log on
                    users => [ 'informationdesk', 'lippmann' ],
                    # all these queries are allowed:
```

(continues on next page)

(continued from previous page)

```
                    sql => {
                            search_city => 'select ort_nr, plz, ort␣
↪from e01admin.e01e200 where plz like ?',
                            search_area => 'select gebiettyp, geb_
↪bezei from e01admin.e01e203 where geb_bezei like ? or geb_bezei␣
↪like ?',
                    }
            },

            # rule: internal_bad_guy_2
            # This does NOT work, because rule "employee_workplace"␣
↪hits
            # with its ip-address-mask of the whole network
            {
                    # don't accept connection from this ip-address
                    mask => '^10\.95\.81\.5$',
                    accept => 0,
            }
    ]
}
```

Start the proxyserver like this:

```
rem well-set Oracle_home needed for Oracle
set ORACLE_HOME=d:\oracle\ora81
dbiproxy --configfile proxy_oracle.cfg
```

Testing the connection from a remote machine

Call a program “dbish” from your commandline. I take the machine from rule “internal_webserver”

```
dbish "dbi:Proxy:hostname=oracle.zdf;port=12400;dsn=dbi:Oracle:e01"␣
↪informationdesk xxx
```

There will be a shell-prompt:

```
informationdesk@dbi...> alive

Current statement buffer (enter '/'...):
alive

informationdesk@dbi...> /
COUNT(*)
'1'
[1 rows of 1 fields returned]
```

Testing the connection with a perl-script

Create a perl-script like this:

```
# file: oratest.pl
# call me like this: perl oratest.pl user password

use strict;
use DBI;

my $user = shift || die "Usage: $0 user password";
my $pass = shift || die "Usage: $0 user password";
my $config = {
        dsn_at_proxy => "dbi:Oracle:e01",
        proxy => "hostname=oechsle.zdf;port=12400",
};
my $dsn = sprintf "dbi:Proxy:%s;dsn=%s",
        $config->{proxy},
        $config->{dsn_at_proxy};

my $dbh = DBI->connect( $dsn, $user, $pass )
        || die "connect did not work: $DBI::errstr";

my $sql = "search_city";
printf "%s\n%s\n%s\n", "="x40, $sql, "="x40;
my $cur = $dbh->prepare($sql);
$cur->bind_param(1,'905%');
&show_result ($cur);

my $sql = "search_area";
printf "%s\n%s\n%s\n", "="x40, $sql, "="x40;
my $cur = $dbh->prepare($sql);
$cur->bind_param(1,'Pfarr%');
$cur->bind_param(2,'Bronnamberg%');
&show_result ($cur);

my $sql = "statistic_area";
printf "%s\n%s\n%s\n", "="x40, $sql, "="x40;
my $cur = $dbh->prepare($sql);
$cur->bind_param(1,'Pfarr%');
&show_result ($cur);

$dbh->disconnect;
exit;


sub show_result {
        my $cur = shift;
        unless ($cur->execute()) {
                print "Could not execute\n";
                return;
```

(continues on next page)

(continued from previous page)

```
        }

        my $rownum = 0;
        while (my @row = $cur->fetchrow_array()) {
                printf "Row is: %s\n", join(", ",@row);
                if ($rownum++ > 5) {
                        print "... and so on\n";
                        last;
                }
        }
        $cur->finish;
}
```

The result

```
C:\>perl oratest.pl informationdesk xxx
========================================
search_city
========================================
Row is: 3322, 9050, Chemnitz
Row is: 3678, 9051, Chemnitz
Row is: 10447, 9051, Chemnitz
Row is: 12128, 9051, Chemnitz
Row is: 10954, 90513, Zirndorf
Row is: 5808, 90513, Zirndorf
Row is: 5715, 90513, Zirndorf
... and so on
========================================
search_area
========================================
Row is: 101, Bronnamberg
Row is: 400, Pfarramt Zirndorf
Row is: 400, Pfarramt Rosstal
Row is: 400, Pfarramt Oberasbach
Row is: 401, Pfarramt Zirndorf
Row is: 401, Pfarramt Rosstal
========================================
statistic_area
========================================
DBD::Proxy::st execute failed: Server returned error: Failed to␣
↪execute method CallMethod: Unknown SQL query: statistic_area at␣
↪E:/Perl/site/lib/DBI/ProxyServer.pm line 258.
Could not execute
```

How the configuration works

The most important section to control access to your dbi-proxy is “client=>” in the file “proxy_oracle.cfg”:

Controlling which person at which machine is allowed to access

- "mask" is a perl regular expression against the plain ip-address of the machine which wishes to connect _or_ the reverse-lookup from a nameserver.
- "accept" tells the dbiproxy-server whether ip-addresses like in "mask" are allowed to connect or not (0/1)
- "users" is a reference to a list of usernames which must be matched, this is NOT a regular expression.

Controlling which SQL-statements are allowed

You can put every SQL-statement you like in simply omitting "sql => ...", but the more important thing is to restrict the connection so that only allowed queries are possible.

If you include an sql-section in your config-file like this:

```
sql => {
        alive => 'select count(*) from dual',
        statistic_area => 'select count(*) from e01admin.e01e203␣
↪where geb_bezei like ?',
}
```

The user is allowed to put two queries against the dbi-proxy. The queries are _not_ "select count(*)...", the queries are "alive" and "statistic_area"! These keywords are replaced by the real query. So you can run a query for "alive":

```
my $sql = "alive";
my $cur = $dbh->prepare($sql);
...
```

The flexibility is that you can put parameters in the where-part of the query so the query are not static. Simply replace a value in the where-part of the query through a question mark and bind it as a parameter to the query.

```
my $sql = "statistic_area";
my $cur = $dbh->prepare($sql);
$cur->bind_param(1,'905%');
# A second parameter would be called like this:
# $cur->bind_param(2,'98%');
```

The result is this query:

```
select count(*) from e01admin.e01e203
where geb_bezei like '905%'
```

Don't try to put parameters into the sql-query like this:

```
# Does not work like you think.
# Only the first word of the query is parsed,
# so it's changed to "statistic_area", the rest is omitted.
# You _have_ to work with $cur->bind_param.
my $sql = "statistic_area 905%";
my $cur = $dbh->prepare($sql);
...
```

Problems

- I don't know how to restrict users to special databases.
- I don't know how to pass query-parameters via dbish

SECURITY WARNING

RPC::PlServer used underneath is not secure due to serializing and deserializing data with *Storable* module. Use the proxy driver only in trusted environment.

AUTHOR

```
Copyright (c) 1997     Jochen Wiedmann
                       Am Eisteich 9
                       72555 Metzingen
                       Germany

                       Email: joe@ispsoft.de
                       Phone: +49 7123 14881
```

The DBI::ProxyServer module is free software; you can redistribute it and/or modify it under the same terms as Perl itself. In particular permission is granted to Tim Bunce for distributing this as a part of the DBI.

SEE ALSO

dbiproxy, *DBD::Proxy*, *DBI*, RPC::PlServer, RPC::PlClient, Net::Daemon, Net::Daemon::Log, *Sys::Syslog*, Win32::EventLog, syslog

DBI::PurePerl

SYNOPSIS

```
BEGIN { $ENV{DBI_PUREPERL} = 2 }
use DBI;
```

DESCRIPTION

This is a pure perl emulation of the DBI internals. In almost all cases you will be better off using standard DBI since the portions of the standard version written in C make it *much* faster.

However, if you are in a situation where it isn't possible to install a compiled version of standard DBI, and you're using pure-perl DBD drivers, then this module allows you to use most common features of DBI without needing any changes in your scripts.

EXPERIMENTAL STATUS

DBI::PurePerl is new so please treat it as experimental pending more extensive testing. So far it has passed all tests with DBD::CSV, DBD::AnyData, DBD::XBase, DBD::Sprite, DBD::mysqlPP. Please send bug reports to Jeff Zucker at jeff@vpservices.com with a cc to dbi-dev@perl.org.

USAGE

The usage is the same as for standard DBI with the exception that you need to set the environment variable DBI_PUREPERL if you want to use the PurePerl version.

```
DBI_PUREPERL == 0 (the default) Always use compiled DBI, die
                 if it isn't properly compiled & installed

DBI_PUREPERL == 1 Use compiled DBI if it is properly compiled
                 & installed, otherwise use PurePerl

DBI_PUREPERL == 2 Always use PurePerl
```

You may set the environment variable in your shell (e.g. with set or setenv or export, etc) or else set it in your script like this:

```
BEGIN { $ENV{DBI_PUREPERL}=2 }
```

before you `use DBI;`.

INSTALLATION

In most situations simply install DBI (see the DBI pod for details).

In the situation in which you can not install DBI itself, you may manually copy DBI.pm and PurePerl.pm into the appropriate directories.

For example:

```
cp DBI.pm      /usr/jdoe/mylibs/.
cp PurePerl.pm /usr/jdoe/mylibs/DBI/.
```

Then add this to the top of scripts:

```
BEGIN {
  $ENV{DBI_PUREPERL} = 1;      # or =2
  unshift @INC, '/usr/jdoe/mylibs';
}
```

(Or should we perhaps patch Makefile.PL so that if DBI_PUREPERL is set to 2 prior to make, the normal compile process is skipped and the files are installed automatically?)

DIFFERENCES BETWEEN DBI AND DBI::PurePerl

Attributes

Boolean attributes still return boolean values but the actual values used may be different, i.e., 0 or undef instead of an empty string.

Some handle attributes are either not supported or have very limited functionality:

```
ActiveKids
InactiveDestroy
AutoInactiveDestroy
Kids
Taint
TaintIn
TaintOut
```

(and probably others)

Tracing

Trace functionality is more limited and the code to handle tracing is only embedded into DBI:PurePerl if the DBI_TRACE environment variable is defined. To enable total tracing you can set the DBI_TRACE environment variable as usual. But to enable individual handle tracing using the trace() method you also need to set the DBI_TRACE environment variable, but set it to 0.

Parameter Usage Checking

The DBI does some basic parameter count checking on method calls. DBI::PurePerl doesn't.

Speed

DBI::PurePerl is slower. Although, with some drivers in some contexts this may not be very significant for you.

By way of example... the test.pl script in the DBI source distribution has a simple benchmark that just does:

```
my $null_dbh = DBI->connect('dbi:NullP:','','');
my $i = 10_000;
$null_dbh->prepare('') while $i--;
```

In other words just prepares a statement, creating and destroying a statement handle, over and over again. Using the real DBI this runs at ~4550 handles per second whereas DBI::PurePerl manages ~2800 per second on the same machine (not too bad really).

May not fully support hash()

If you want to use type 1 hash, i.e., `hash($string,1)` with DBI::PurePerl, you'll need version 1.56 or higher of Math::BigInt (available on CPAN).

Doesn't support preparse()

The DBI->preparse() method isn't supported in DBI::PurePerl.

Doesn't support DBD::Proxy

There's a subtle problem somewhere I've not been able to identify. DBI::ProxyServer seem to work fine with DBI::PurePerl but DBD::Proxy does not work 100% (which is sad because that would be far more useful :) Try re-enabling t/80proxy.t for DBI::PurePerl to see if the problem that remains will affect you're usage.

Others

```
can() - doesn't have any special behaviour
```

Please let us know if you find any other differences between DBI and DBI::PurePerl.

AUTHORS

Tim Bunce and Jeff Zucker.

Tim provided the direction and basis for the code. The original idea for the module and most of the brute force porting from C to Perl was by Jeff. Tim then reworked some core parts to boost the performance and accuracy of the emulation. Thanks also to Randal Schwartz and John Tobey for patches.

COPYRIGHT

Copyright (c) 2002 Tim Bunce Ireland.

See COPYRIGHT section in DBI.pm for usage and distribution rights.

DBI::SQL

Namespace containing the following modules:

- `DBI::SQL::Nano` - a very tiny SQL engine

DBI::SQL::Nano

a very tiny SQL engine

SYNOPSIS

```
BEGIN { $ENV{DBI_SQL_NANO}=1 } # forces use of Nano rather than␣
↪SQL::Statement
use DBI::SQL::Nano;
use Data::Dumper;
my $stmt = DBI::SQL::Nano::Statement->new(
    "SELECT bar,baz FROM foo WHERE qux = 1"
) or die "Couldn't parse";
print Dumper $stmt;
```

DESCRIPTION

DBI::SQL::Nano is meant as a *very* minimal SQL engine for use in situations where SQL::Statement is not available. In most situations you are better off installing SQL::Statement although DBI::SQL::Nano may be faster for some **very** simple tasks.

DBI::SQL::Nano, like SQL::Statement is primarily intended to provide a SQL engine for use with some pure perl DBDs including *DBD::DBM*, DBD::CSV, DBD::AnyData, and DBD::Excel. It is not of much use in and of itself. You can dump out the structure of a parsed SQL statement, but that is about it.

USAGE

Setting the DBI_SQL_NANO flag

By default, when a DBD uses DBI::SQL::Nano, the module will look to see if SQL::Statement is installed. If it is, SQL::Statement objects are used. If SQL::Statement is not available, DBI::SQL::Nano objects are used.

In some cases, you may wish to use DBI::SQL::Nano objects even if SQL::Statement is available. To force usage of DBI::SQL::Nano objects regardless of the availability of SQL::Statement, set the environment variable DBI_SQL_NANO to 1.

You can set the environment variable in your shell prior to running your script (with SET or EXPORT or whatever), or else you can set it in your script by putting this at the top of the script:

```
BEGIN { $ENV{DBI_SQL_NANO} = 1 }
```

Supported SQL syntax

```
Here's a pseudo-BNF.  Square brackets [] indicate optional items;
Angle brackets <> indicate items defined elsewhere in the BNF.

 statement ::=
     DROP TABLE [IF EXISTS] <table_name>
   | CREATE TABLE <table_name> <col_def_list>
   | INSERT INTO <table_name> [<insert_col_list>] VALUES <val_list>
   | DELETE FROM <table_name> [<where_clause>]
```

(continues on next page)

(continued from previous page)

```
  | UPDATE <table_name> SET <set_clause> <where_clause>
  | SELECT <select_col_list> FROM <table_name> [<where_clause>]
                                               [<order_clause>]

the optional IF EXISTS clause ::=
  * similar to MySQL - prevents errors when trying to drop
    a table that doesn't exist

identifiers ::=
  * table and column names should be valid SQL identifiers
  * especially avoid using spaces and commas in identifiers
  * note: there is no error checking for invalid names, some
    will be accepted, others will cause parse failures

table_name ::=
  * only one table (no multiple table operations)
  * see identifier for valid table names

col_def_list ::=
  * a parens delimited, comma-separated list of column names
  * see identifier for valid column names
  * column types and column constraints may be included but are␣
↪ignored
    e.g. these are all the same:
      (id,phrase)
      (id INT, phrase VARCHAR(40))
      (id INT PRIMARY KEY, phrase VARCHAR(40) NOT NULL)
  * you are *strongly* advised to put in column types even though
    they are ignored ... it increases portability

insert_col_list ::=
  * a parens delimited, comma-separated list of column names
  * as in standard SQL, this is optional

select_col_list ::=
  * a comma-separated list of column names
  * or an asterisk denoting all columns

val_list ::=
  * a parens delimited, comma-separated list of values which can␣
↪be:
     * placeholders (an unquoted question mark)
     * numbers (unquoted numbers)
     * column names (unquoted strings)
     * nulls (unquoted word NULL)
     * strings (delimited with single quote marks);
     * note: leading and trailing percent mark (%) and underscore␣
↪(_)
       can be used as wildcards in quoted strings for use with
```

(continues on next page)

(continued from previous page)

```
        the LIKE and CLIKE operators
      * note: escaped single quotation marks within strings are not
        supported, neither are embedded commas, use placeholders␣
↪instead

 set_clause ::=
   * a comma-separated list of column = value pairs
   * see val_list for acceptable value formats

 where_clause ::=
   * a single "column/value <op> column/value" predicate, optionally
     preceded by "NOT"
   * note: multiple predicates combined with ORs or ANDs are not␣
↪supported
   * see val_list for acceptable value formats
   * op may be one of:
        < > >= <= = <> LIKE CLIKE IS
   * CLIKE is a case insensitive LIKE
   * IS NULL matches against NULL and empty strings

 order_clause ::= column_name [ASC|DESC]
   * a single column optional ORDER BY clause is supported
   * as in standard SQL, if neither ASC (ascending) nor
     DESC (descending) is specified, ASC becomes the default
```

TABLES

DBI::SQL::Nano::Statement operates on exactly one table. This table will be opened by inherit from DBI::SQL::Nano::Statement and implements the open_table method.

```
sub open_table ($$$$$)
{
    ...
    return Your::Table->new( \%attributes );
}
```

DBI::SQL::Nano::Statement_ expects a rudimentary interface is implemented by the table object, as well as SQL::Statement expects.

```
package Your::Table;

our @ISA = qw(DBI::SQL::Nano::Table);

sub drop ($$)        { ... }
sub fetch_row ($$$)  { ... }
sub push_row ($$$)   { ... }
sub push_names ($$$) { ... }
sub truncate ($$)    { ... }
sub seek ($$$$)      { ... }
```

The base class interfaces are provided by DBI::SQL::Nano::Table_ in case of relying on DBI::SQL::Nano or SQL::Eval::Table (see SQL::Eval for details) otherwise.

BUGS AND LIMITATIONS

There are no known bugs in DBI::SQL::Nano::Statement. If you find a one and want to report, please see *DBI* for how to report bugs.

DBI::SQL::Nano::Statement is designed to provide a minimal subset for executing SQL statements.

The most important limitation might be the restriction on one table per statement. This implies, that no JOINs are supported and there cannot be any foreign key relation between tables.

The where clause evaluation of DBI::SQL::Nano::Statement is very slow (SQL::Statement uses a precompiled evaluation).

INSERT can handle only one row per statement. To insert multiple rows, use placeholders as explained in DBI.

The DBI::SQL::Nano parser is very limited and does not support any additional syntax such as brackets, comments, functions, aggregations etc.

In contrast to SQL::Statement, temporary tables are not supported.

ACKNOWLEDGEMENTS

Tim Bunce provided the original idea for this module, helped me out of the tangled trap of namespaces, and provided help and advice all along the way. Although I wrote it from the ground up, it is based on Jochen Wiedmann's original design of SQL::Statement, so much of the credit for the API goes to him.

AUTHOR AND COPYRIGHT

This module is originally written by Jeff Zucker < jzucker AT cpan.org >

This module is currently maintained by Jens Rehsack < jrehsack AT googlemail.com >

Copyright (C) 2010 by Jens Rehsack, all rights reserved. Copyright (C) 2004 by Jeff Zucker, all rights reserved.

You may freely distribute and/or modify this module under the terms of either the GNU General Public License (GPL) or the Artistic License, as specified in the Perl README file.

DBI::Util

Namespace containing the following modules:

- `DBI::Util::CacheMemory` - a very fast but very minimal subset of Cache::Memory

DBI::Util::CacheMemory

a very fast but very minimal subset of Cache::Memory

DESCRIPTION

Like Cache::Memory (part of the Cache distribution) but doesn't support any fancy features.

This module aims to be a very fast compatible strict sub-set for simple cases, such as basic client-side caching for DBD::Gofer.

Like Cache::Memory, and other caches in the Cache and Cache::Cache distributions, the data will remain in the cache until cleared, it expires, or the process dies. The cache object simply going out of scope will *not* destroy the data.

METHODS WITH CHANGES

new

All options except `namespace` are ignored.

set

Doesn't support expiry.

purge

Same as clear() - deletes everything in the namespace.

METHODS WITHOUT CHANGES

- clear
- count
- exists
- remove

UNSUPPORTED METHODS

If it's not listed above, it's not supported.

5.2.25 DBM_Filter

Filter DBM keys/values

SYNOPSIS

```
use DBM_Filter ;
use SDBM_File; # or DB_File, GDBM_File, NDBM_File, or ODBM_File

$db = tie %hash, ...

$db->Filter_Push(Fetch => sub {...},
                 Store => sub {...});

$db->Filter_Push('my_filter1');
$db->Filter_Push('my_filter2', params...);

$db->Filter_Key_Push(...) ;
$db->Filter_Value_Push(...) ;

$db->Filter_Pop();
$db->Filtered();

package DBM_Filter::my_filter1;

sub Store { ... }
sub Fetch { ... }

1;

package DBM_Filter::my_filter2;

sub Filter
{
    my @opts = @_;
    ...
    return (
        sub Store { ... },
        sub Fetch { ... } );
}

1;
```

DESCRIPTION

This module provides an interface that allows filters to be applied to tied Hashes associated with DBM files. It builds on the DBM Filter hooks that are present in all the *DB*_File modules included with the standard Perl source distribution from version 5.6.1 onwards. In addition to the *DB*_File modules distributed with Perl, the BerkeleyDB module, available on CPAN, supports the DBM Filter hooks. See perldbmfilter for more details on the DBM Filter hooks.

What is a DBM Filter?

A DBM Filter allows the keys and/or values in a tied hash to be modified by some user-defined code just before it is written to the DBM file and just after it is read back from the DBM file. For example, this snippet of code

```
$some_hash{"abc"} = 42;
```

could potentially trigger two filters, one for the writing of the key "abc" and another for writing the value 42. Similarly, this snippet

```
my ($key, $value) = each %some_hash
```

will trigger two filters, one for the reading of the key and one for the reading of the value.

Like the existing DBM Filter functionality, this module arranges for the `$_` variable to be populated with the key or value that a filter will check. This usually means that most DBM filters tend to be very short.

So what's new?

The main enhancements over the standard DBM Filter hooks are:

- A cleaner interface.
- The ability to easily apply multiple filters to a single DBM file.
- The ability to create "canned" filters. These allow commonly used filters to be packaged into a stand-alone module.

METHODS

This module will arrange for the following methods to be available via the object returned from the `tie` call.

$db->Filter_Push() / $db->Filter_Key_Push() / $db->Filter_Value_Push()

Add a filter to filter stack for the database, `$db`. The three formats vary only in whether they apply to the DBM key, the DBM value or both.

- Filter_Push

  The filter is applied to *both* keys and values.

- Filter_Key_Push

  The filter is applied to the key *only*.

- Filter_Value_Push

  The filter is applied to the value *only*.

$db->Filter_Pop()

Removes the last filter that was applied to the DBM file associated with $db, if present.

$db->Filtered()

Returns TRUE if there are any filters applied to the DBM associated with $db. Otherwise returns FALSE.

Writing a Filter

Filters can be created in two main ways

Immediate Filters

An immediate filter allows you to specify the filter code to be used at the point where the filter is applied to a dbm. In this mode the Filter_*_Push methods expects to receive exactly two parameters.

```
my $db = tie %hash, 'SDBM_File', ...
$db->Filter_Push( Store => sub { },
                  Fetch => sub { });
```

The code reference associated with `Store` will be called before any key/value is written to the database and the code reference associated with `Fetch` will be called after any key/value is read from the database.

For example, here is a sample filter that adds a trailing NULL character to all strings before they are written to the DBM file, and removes the trailing NULL when they are read from the DBM file

```
my $db = tie %hash, 'SDBM_File', ...
$db->Filter_Push( Store => sub { $_ .= "\x00" ; },
                  Fetch => sub { s/\x00$// ;    });
```

Points to note:

1. Both the Store and Fetch filters manipulate $_.

Canned Filters

Immediate filters are useful for one-off situations. For more generic problems it can be useful to package the filter up in its own module.

The usage is for a canned filter is:

```
$db->Filter_Push("name", params)
```

where

- “name”

  is the name of the module to load. If the string specified does not contain the package separator characters “::”, it is assumed to refer to the full module name

"DBM_Filter::name". This means that the full names for canned filters, "null" and "utf8", included with this module are:

```
DBM_Filter::null
DBM_Filter::utf8
```

- params

  any optional parameters that need to be sent to the filter. See the encode filter for an example of a module that uses parameters.

The module that implements the canned filter can take one of two forms. Here is a template for the first

```
package DBM_Filter::null ;

use strict;
use warnings;

sub Store
{
    # store code here
}

sub Fetch
{
    # fetch code here
}

1;
```

Notes:

1. The package name uses the `DBM_Filter::` prefix.
2. The module *must* have both a Store and a Fetch method. If only one is present, or neither are present, a fatal error will be thrown.

The second form allows the filter to hold state information using a closure, thus:

```
package DBM_Filter::encoding ;

use strict;
use warnings;

sub Filter
{
    my @params = @_ ;

    ...
    return {
        Store   => sub { $_ = $encoding->encode($_) },
        Fetch   => sub { $_ = $encoding->decode($_) }
        } ;
```

(continues on next page)

(continued from previous page)

```
}

1;
```

In this instance the “Store” and “Fetch” methods are encapsulated inside a “Filter” method.

Filters Included

A number of canned filers are provided with this module. They cover a number of the main areas that filters are needed when interfacing with DBM files. They also act as templates for your own filters.

The filter included are:

- utf8

  This module will ensure that all data written to the DBM will be encoded in UTF-8.

  This module needs the Encode module.

- encode

  Allows you to choose the character encoding will be store in the DBM file.

- compress

  This filter will compress all data before it is written to the database and uncompressed it on reading.

  This module needs Compress::Zlib.

- int32

  This module is used when interoperating with a C/C++ application that uses a C int as either the key and/or value in the DBM file.

- null

  This module ensures that all data written to the DBM file is null terminated. This is useful when you have a perl script that needs to interoperate with a DBM file that a C program also uses. A fairly common issue is for the C application to include the terminating null in a string when it writes to the DBM file. This filter will ensure that all data written to the DBM file can be read by the C application.

NOTES

Maintain Round Trip Integrity

When writing a DBM filter it is *very* important to ensure that it is possible to retrieve all data that you have written when the DBM filter is in place. In practice, this means that whatever transformation is applied to the data in the Store method, the *exact* inverse operation should be applied in the Fetch method.

If you don’t provide an exact inverse transformation, you will find that code like this will not behave as you expect.

```
while (my ($k, $v) = each %hash)
{
    ...
}
```

Depending on the transformation, you will find that one or more of the following will happen

1. The loop will never terminate.
2. Too few records will be retrieved.
3. Too many will be retrieved.
4. The loop will do the right thing for a while, but it will unexpectedly fail.

Don't mix filtered & non-filtered data in the same database file.

This is just a restatement of the previous section. Unless you are completely certain you know what you are doing, avoid mixing filtered & non-filtered data.

EXAMPLE

Say you need to interoperate with a legacy C application that stores keys as C ints and the values and null terminated UTF-8 strings. Here is how you would set that up

```
my $db = tie %hash, 'SDBM_File', ...

$db->Filter_Key_Push('int32') ;

$db->Filter_Value_Push('utf8');
$db->Filter_Value_Push('null');
```

SEE ALSO

<DB_File>, *GDBM_File*, *NDBM_File*, *ODBM_File*, *SDBM_File*, perldbmfilter

AUTHOR

Paul Marquess pmqs@cpan.org

Modules

- `DBM_Filter::compress` - filter for DBM_Filter
- `DBM_Filter::encode` - filter for DBM_Filter
- `DBM_Filter::int32` - filter for DBM_Filter
- `DBM_Filter::null` - filter for DBM_Filter
- `DBM_Filter::utf8` - filter for DBM_Filter

DBM_Filter::compress

filter for DBM_Filter

SYNOPSIS

```
use SDBM_File; # or DB_File, GDBM_File, NDBM_File, ODBM_File
use DBM_Filter ;

$db = tie %hash, ...
$db->Filter_Push('compress');
```

DESCRIPTION

This DBM filter will compress all data before it is written to the database and uncompressed it on reading.

A fatal error will be thrown if the Compress::Zlib module is not available.

SEE ALSO

DBM_Filter, perldbmfilter, *Compress::Zlib*

AUTHOR

Paul Marquess pmqs@cpan.org

DBM_Filter::encode

filter for DBM_Filter

SYNOPSIS

```
use SDBM_File; # or DB_File, GDBM_File, NDBM_File, ODBM_File
use DBM_Filter ;

$db = tie %hash, ...
$db->Filter_Push('encode' => 'iso-8859-16');
```

DESCRIPTION

This DBM filter allows you to choose the character encoding will be store in the DBM file. The usage is

```
$db->Filter_Push('encode' => ENCODING);
```

where “ENCODING” must be a valid encoding name that the Encode module recognises.

A fatal error will be thrown if:

1. The Encode module is not available.
2. The encoding requested is not supported by the Encode module.

SEE ALSO

DBM_Filter, perldbmfilter, *Encode*

AUTHOR

Paul Marquess pmqs@cpan.org

DBM_Filter::int32

filter for DBM_Filter

SYNOPSIS

```
use SDBM_File; # or DB_File, GDBM_File, NDBM_File, or ODBM_File
use DBM_Filter ;

$db = tie %hash, ...
$db->Filter_Push('int32');
```

DESCRIPTION

This DBM filter is used when interoperating with a C/C++ application that uses a C int as either the key and/or value in the DBM file.

SEE ALSO

DBM_Filter, perldbmfilter

AUTHOR

Paul Marquess pmqs@cpan.org

DBM_Filter::null

filter for DBM_Filter

SYNOPSIS

```
use SDBM_File; # or DB_File, GDBM_File, NDBM_File, or ODBM_File
use DBM_Filter ;

$db = tie %hash, ...
$db->Filter_Push('null');
```

DESCRIPTION

This filter ensures that all data written to the DBM file is null terminated. This is useful when you have a perl script that needs to interoperate with a DBM file that a C program also uses. A fairly common issue is for the C application to include the terminating null in a string when it writes to the DBM file. This filter will ensure that all data written to the DBM file can be read by the C application.

SEE ALSO

DBM_Filter, perldbmfilter

AUTHOR

Paul Marquess pmqs@cpan.org

DBM_Filter::utf8

filter for DBM_Filter

SYNOPSIS

```
use SDBM_File; # or DB_File, GDBM_File, NDBM_File, or ODBM_File
use DBM_Filter;

$db = tie %hash, ...
$db->Filter_Push('utf8');
```

DESCRIPTION

This Filter will ensure that all data written to the DBM will be encoded in UTF-8.

This module uses the Encode module.

SEE ALSO

DBM_Filter, perldbmfilter, *Encode*

AUTHOR

Paul Marquess pmqs@cpan.org

5.2.26 DB_File

Tie a Perl hash or array to a database file: a hash table, a sorted B-tree, or an array of records kept in a plain text file.

`DB_File` binds a variable to a file with `tie`. From then on ordinary hash or array syntax reads and writes the file, and the object `tie` returns offers more: Berkeley DB's own methods (`get`, `put`, `del`, `seq`, `sync`, `fd`), duplicate keys, range lookups, and array methods. The last argument of `tie` picks one of three kinds of database:

- `$DB_HASH` - a tied hash whose keys come back in no particular order. The default.
- `$DB_BTREE` - a tied hash whose keys are kept sorted, by byte value or by a sort sub of your own. It can also hold several values per key, and find the first key at or after a given one.
- `$DB_RECNO` - a tied array over a text file, one element per line (or per fixed-length record). The file is an ordinary text file that other programs can read and write.

Synopsis

```
use DB_File;

# A sorted hash: walk it from a given key on
my $db = tie my %h, 'DB_File', 'fruit.db', O_RDWR|O_CREAT, 0640,
 ↪$DB_BTREE
    or die "fruit.db: $!";
$h{$_} = length for qw(apple banana cherry);
my ($k, $v) = ('b', '');
for (my $st = $db->seq($k, $v, R_CURSOR); $st == 0; $st = $db->seq(
 ↪$k, $v, R_NEXT)) {
    print "$k=$v\n";                # banana=6 cherry=6
}
undef $db;
untie %h;

# A text file as an array
tie my @lines, 'DB_File', 'notes.txt', O_RDWR|O_CREAT, 0640, $DB_
 ↪RECNO
    or die "notes.txt: $!";
push @lines, 'one more line';
untie @lines;
```

Opening a database

```
tie %h, 'DB_File', $filename, $flags, $mode, $DB_HASH;
tie %h, 'DB_File', $filename, $flags, $mode, $DB_BTREE;
tie @a, 'DB_File', $filename, $flags, $mode, $DB_RECNO;
```

`$flags` are `Fcntl`'s `O_RDONLY`, `O_RDWR`, `O_CREAT`, `O_TRUNC` (DB_File exports them); `$mode` is the permission of a file it creates. The defaults are `O_CREAT|O_RDWR`, `0666` and `$DB_HASH`, so `tie %h, 'DB_File', $filename` opens or creates a hash database. A relative file name is made absolute when the database is opened.

`tie` returns the database object, or undef with `$!` set when the file cannot be opened. Keep the object (`$db` above) to call methods on it, and `undef` it before `untie`: the file is closed only when the tied variable and every copy of the object are gone.

An undef file name opens an in-memory database: it is empty when tied and gone when untied.

`$DB_HASH`, `$DB_BTREE` and `$DB_RECNO` are ready-made settings objects. For other settings, make one and set its keys:

```
my $info = DB_File::BTREEINFO->new;
$info->{compare} = sub { lc $_[0] cmp lc $_[1] };
$info->{flags}   = R_DUP;
tie my %h, 'DB_File', 'names.db', O_RDWR|O_CREAT, 0640, $info;
```

- `DB_File::HASHINFO` takes `hash`, a sub returning a number for a key, and `bsize`, `cachesize`, `ffactor`, `nelem`, `lorder`.
- `DB_File::BTREEINFO` takes `compare`, `flags` (`R_DUP`), `prefix`, `cachesize`, `psize`, `lorder`, `maxkeypage`, `minkeypage`.
- `DB_File::RECNOINFO` takes `bval`, `flags` (`R_FIXEDLEN`), `reclen`, `bfname`, `cachesize`, `psize`, `lorder`.

Setting any other key croaks, and so does a plain value (not a reference) for `hash`, `compare` or `prefix`. The `hash` and `prefix` subs, `bsize`, `cachesize`, `ffactor`, `nelem`, `lorder`, `psize`, `maxkeypage` and `minkeypage` tune Berkeley DB's own file layout, which pperl's files do not have: they are accepted and change nothing. The `hash` sub is still called, once when the database opens and once for every operation on a key, and its result unused.

Sorted keys and duplicates

A `$DB_BTREE` database keeps its keys in byte order unless its settings carry a `compare` sub. That sub gets two keys and returns less than, equal to or greater than zero, as `cmp` does. The file does not remember it: every later `tie` of the file must give the same `compare`, or lookups go wrong.

With `flags` set to `R_DUP`, a key may hold several values, kept in the order they were stored. Fetching the key gives its first value; `keys` and `each` return the key once per value; deleting it deletes them all. These methods reach the values one by one:

- `$db->get_dup($key)` - the key's values, as a list; in scalar context, how many there are. `$db->get_dup($key, 1)` returns a hash counting each distinct value instead.
- `$db->find_dup($key, $value)` - 0 when the key holds that value, with the cursor left on it, so that `$db->del($key, R_CURSOR)` deletes just that one; 1 otherwise.
- `$db->del_dup($key, $value)` - deletes that one value of the key: 0 if it was there, 1 if not.

The file remembers that it holds duplicates: a later `tie` without `R_DUP` still has them, and asking for `R_DUP` of a file made without it fails the `tie` unless `O_TRUNC` empties the file.

Arrays

A `$DB_RECNO` array reads its text file when tied and writes it back at `sync` and `untie`, if anything changed. Each element is one line without its newline. `bval` sets another delimiter (`"-"`, or a character's number). With `flags => R_FIXEDLEN` and `reclen`, each element is that many bytes instead, a shorter one padded with `bval` (a space by default) and a longer one refused.

The file is created if it does not exist, and the array can always be written, whatever `$flags` say. Deleting an element moves the ones after it down. Storing past the end creates the elements in between as empty: they read as `undef`, `exists` is false for them, and they are written to the file as empty lines. A negative index counts from the end; one before the start croaks `Modification of non-creatable array value attempted`.

The array methods `push`, `pop`, `shift`, `unshift`, `length` and `splice` work on the database object as on the tied array. With no file name the array lives in memory only, or, when the settings have a `bfname`, in that text file.

The cursor

Every database has one cursor, which `seq` moves and the methods' `R_CURSOR` flag acts at. The tie interface moves it too: `keys`, `each`, pop and the like leave it somewhere else. A method that fails to move it leaves it where it was. After a delete at the cursor, `R_NEXT` goes on to the element after the deleted one and `R_PREV` to the one before.

Files

pperl keeps a `$DB_HASH` or `$DB_BTREE` database in an LMDB file: the file named, with LMDB's lock file, `NAME-lock`, beside it. The Berkeley DB library is not used, and its files cannot be read: tying a real Berkeley DB file (or a gdbm file) croaks, naming what the file is, and the file is left as it was. A key may be at most 511 bytes, 510 if it starts with a NUL byte and 503 in an `R_DUP` database; a longer one croaks.

Writes reach the file at `sync` and when the database is untied: until then the tie holds the file's one writer slot, and another process that writes the file waits for it (readers never wait). Set `$Peta::FFI::LMDB::DBM_WRITE = 'through'` before `tie` to commit every write as it happens, for files that several processes write; see `Peta::FFI::LMDB`.

Constants

DB_File exports the flags its methods take:

- `R_FIRST`, `R_LAST`, `R_NEXT`, `R_PREV` - where `seq` moves the cursor.
- `R_CURSOR` - `seq`: to the given key, or for a `$DB_BTREE` the first key at or after it; `put` and `del`: at the cursor.
- `R_IAFTER`, `R_IBEFORE` - `put` a new element after or before the given one (arrays, and `R_DUP` databases).
- `R_NOOVERWRITE` - `put` only a new key.
- `R_SETCURSOR` - `put`, then leave the cursor on the key.
- `R_DUP` - `DB_File::BTREEINFO` flag: keys with several values.
- `R_FIXEDLEN` - `DB_File::RECNOINFO` flag: fixed-length records.
- `R_RECNOSYNC` - accepted by `sync`; it is 0.

The other names it exports (`BTREEMAGIC`, `HASHMAGIC`, `R_NOKEY`, `RET_SUCCESS` and the like) are not defined: using one croaks `Your vendor has not defined DB_File macro ...`, as it does with the Berkeley DB versions perl links today.

`$DB_File::db_version` and `$DB_File::db_ver` give the version of the Berkeley DB interface the module follows, and `$DB_File::Error` the text of the last error a method reported.

Differences from upstream

- The files are LMDB files, not Berkeley DB files: real Berkeley DB files are refused, and pperl's files cannot be read by perl5's DB_File. A file that is not a DB_File database is never overwritten. Covered by `t/82-native/Peta-FFI-LMDB/100-db_file.t`.
- A `$DB_HASH` database's keys come in byte order, where Berkeley DB's come in the order of its hash buckets. Neither order is promised; code that prints a hash database without sorting prints it in a different order. Covered by `t/82-native/Peta-FFI-LMDB/100-db_file.t`.
- Keys are limited to 511 bytes (503 with `R_DUP`); Berkeley DB has no limit. Covered by `t/82-native/Peta-FFI-LMDB/100-db_file.t`.
- Writes reach the file at `sync` and `untie` (or each one, with `$Peta::FFI::LMDB::DBM_WRITE = 'through'`), and a writer holds the file until then. Covered by `t/82-native/Peta-FFI-LMDB/100-db_file.t`.
- Two ties of one `$DB_BTREE` file in a process must sort it the same way: a second tie with another order (a `compare` sub against none) fails with `EINVAL`. Covered by `t/82-native/Peta-FFI-LMDB/100-db_file.t`.
- With no file name but a `bfname`, a `$DB_RECNO` array keeps its records in `bfname` as a text file, where Berkeley DB writes a database file of its own format there. Covered by `t/82-native/Peta-FFI-LMDB/100-db_file.t`.
- The tuning settings and the `prefix` sub have no effect, and the `hash` sub's result is unused.
- `$DB_File::Error` carries Berkeley DB's message texts without their `BDB0123` numbers.

Everything else - status codes, the cursor, duplicates, Recno's records and text file - behaves as with Berkeley DB, down to its edge cases. Covered by `t/81-xs-native/DB_File/010-phenotype.t` and DB_File's own test suite, `t/81-xs-native/DB_File/100-upstream-db-btree.t` to `130-upstream-db-threads.t`.

Functions

Berkeley DB methods

`del`

Delete a key - every value of it, in an `R_DUP` database - or, with `R_CURSOR`, the one item the database's cursor stands on.

get

Look up a key and put its value in a variable, returning a status instead of the value - so a missing key and a stored undef can be told apart.

put

Store a value under a key, with flags that say where and whether.

fd

The file descriptor of the database's file - for a `$DB_RECNO` array, its text file - or -1 for an in-memory database.

sync

Write everything stored so far to the file and to disk, so that other processes - and the file after a crash - have it.

seq

Move the database's cursor and read the key and value it lands on: walk the database in order, or find the first key at or after a given one.

Array methods

unshift

Put elements at the start of a `$DB_RECNO` array, in the order given.

pop

Remove the last element of a `$DB_RECNO` array and return it.

shift

Remove the first element of a `$DB_RECNO` array and return it; the others move down one.

push

Add elements to the end of a `$DB_RECNO` array.

length

The number of elements of a `$DB_RECNO` array: up to and including its last element that holds a value, so `scalar(@a)` too. `FETCHSIZE` is the same method under the name the tied array calls. It moves the database's cursor to that last element.

Filters

filter_fetch_key

Install a sub that rewrites every key read from the database - by `keys`, `each` or `seq` - before your code sees it.

filter_store_key

Install a sub that rewrites every key on its way into the database - for storing, fetching, deleting or looking up - so the database holds the rewritten form.

filter_fetch_value

Install a sub that rewrites every value read from the database before your code sees it.

filter_store_value

Install a sub that rewrites every value on its way into the database, so the database holds the rewritten form.

unshift

Put elements at the start of a `$DB_RECNO` array, in the order given.

Synopsis

```
$db->unshift('first', 'second');
unshift @lines, 'header';
```

What you get back

0, or the error of the element that could not be stored - the elements after it are not stored either. `UNSHIFT` is the same method under the name the tied array calls.

Edge cases

- An empty array cannot be unshifted onto: nothing is stored, and the method returns 22 (`EINVAL`). `push` the first element instead.
- The elements go before the first element that holds a value, after any empty ones created by storing past them.
- It moves the database's cursor to the first element stored.

Differences from upstream

Fully compatible with upstream, edge cases included. Covered by `t/81-xs-native/DB_File/010-phenotype.t`.

See also

- `push` - the other end of the array
- `shift` - takes the first element off again

pop

Remove the last element of a `$DB_RECNO` array and return it.

Synopsis

```
my $last = $db->pop;
my $line = pop @lines;
```

What you get back

The element's value, or undef for an empty array. `POP` is the same method under the name the tied array calls.

Edge cases

- The array then ends at the last element that holds a value: empty elements before the removed one no longer count in its length, though the text file keeps them as empty lines.
- It leaves the database's cursor where the element was.

Differences from upstream

Fully compatible with upstream. Covered by `t/81-xs-native/DB_File/010-phenotype.t`.

See also

- `shift` - the same at the start of the array
- `length` - how many elements are left

shift

Remove the first element of a `$DB_RECNO` array and return it; the others move down one.

Synopsis

```
my $first = $db->shift;
my $line  = shift @lines;
```

What you get back

The element's value, or undef for an empty array. `SHIFT` is the same method under the name the tied array calls.

Edge cases

- Empty elements at the start are passed over: it removes the first element that holds a value, and the empty ones stay.

Differences from upstream

Fully compatible with upstream. Covered by `t/81-xs-native/DB_File/010-phenotype.t`.

See also

- pop - the same at the end of the array
- `unshift` - puts elements back at the start

push

Add elements to the end of a `$DB_RECNO` array.

Synopsis

```
$db->push('one', 'two');
push @lines, 'three';
```

What you get back

0, or the error of the element that could not be stored - the elements after it are not stored either. `PUSH` is the same method under the name the tied array calls.

Edge cases

- With no elements to add it returns 0 for an array that has elements, and `-30987` (Berkeley DB's `DB_NOTFOUND`) for an empty one.
- The new elements follow the last element that holds a value.

Differences from upstream

Fully compatible with upstream. Covered by `t/81-xs-native/DB_File/010-phenotype.t`.

See also

- pop - takes the last element off again
- `unshift` - adds at the start

del

Delete a key - every value of it, in an `R_DUP` database - or, with `R_CURSOR`, the one item the database's cursor stands on.

Synopsis

```
my $status = $db->del($key);
$status = $db->del($key, R_CURSOR);    # $key unused
```

What you get back

0 when something was deleted, 1 when the key was not there, -1 for an error. On an array, deleting an element created empty returns `-30995` (Berkeley DB's `DB_KEYEMPTY`).

Examples

```
$h{apple} = 'red';
print $db->del('apple');        # 0
print $db->del('apple');        # 1
```

Delete one of a key's duplicates, found with `seq`:

```
my ($k, $v) = ('fruit', '');
$db->seq($k, $v, R_CURSOR);     # the key's first value
$db->del($k, R_CURSOR);         # just that one
```

Edge cases

- With `R_CURSOR` and no cursor set yet, it returns -1.
- After a delete at the cursor, `seq` with `R_NEXT` goes on to the item after the deleted one and `R_PREV` to the one before. Deleting at the cursor again deletes nothing; on an array it returns `-30995`.
- Deleting an array element moves the ones after it down.
- On a database opened read-only it returns -1, and `$DB_File::Error` says so.

Differences from upstream

Fully compatible with upstream. Covered by `t/81-xs-native/DB_File/010-phenotype.t`.

See also

- `seq` - sets the cursor `R_CURSOR` deletes at
- `del_dup` - deletes one given value of a key

get

Look up a key and put its value in a variable, returning a status instead of the value - so a missing key and a stored undef can be told apart.

Synopsis

```
my $value;
my $status = $db->get($key, $value);
```

What you get back

0 with `$value` set when the key is stored, 1 when it is not (leaving `$value` alone), -1 for an error. In an `R_DUP` database `$value` is the key's first value. For an array, `$key` is the element's index.

Examples

```
$h{colour} = 'blue';
my $v;
print $db->get('colour', $v), " $v";   # 0 blue
print $db->get('size', $v);            # 1
```

Edge cases

- An array element created empty returns `-30995` (Berkeley DB's `DB_KEYEMPTY`) and leaves `$value` alone.
- The value passes through a `filter_fetch_value` filter, and the key through `filter_store_key`, as with the tied hash.
- A flags argument other than 0 returns -1.

Differences from upstream

Fully compatible with upstream. Covered by `t/81-xs-native/DB_File/010-phenotype.t`.

See also

- `put` - stores a value
- `seq` - looks up the first key at or after one, and walks on
- `get_dup` - all the values of a key

put

Store a value under a key, with flags that say where and whether.

Synopsis

```
my $status = $db->put($key, $value);
$status = $db->put($key, $value, R_NOOVERWRITE);
```

What you get back

0 when the value was stored, 1 when `R_NOOVERWRITE` found the key already there, -1 for an error.

Flags

- none - store the value, replacing the key's; in an `R_DUP` database, add it after the key's other values.
- `R_NOOVERWRITE` - store only if the key is not there yet.
- `R_CURSOR` - replace the value the database's cursor stands on; the key given is ignored.
- `R_SETCURSOR` - store, then set the cursor on the key (its first value, in an `R_DUP` database).
- `R_IAFTER`, `R_IBEFORE` - put a new array element after or before element `$key`, the ones after it moving up; in an `R_DUP` database, a new value after or before the key's first. `$key` must be a variable: the method writes it back.

Examples

```
print $db->put('a', 1);                    # 0
print $db->put('a', 2, R_NOOVERWRITE);     # 1, and 'a' is still 1
```

Insert into an array:

```
@lines = qw(one three);
my $i = 0;
$db->put($i, 'two', R_IAFTER);             # one two three
```

Edge cases

- `R_IAFTER` and `R_IBEFORE` leave `$key` as it was; they return -1 for an element past the end or created empty, and in a hash database without `R_DUP` (`$DB_File::Error` then says the flag is illegal).
- `R_CURSOR` with no cursor set yet returns -1. After a delete at the cursor, a `$DB_HASH` or `$DB_BTREE` database returns `-30987` (Berkeley DB's `DB_NOTFOUND`) and stores nothing, while an array gets the deleted element back with the new value.

- A fixed-length array refuses a value longer than its records: -1, with `$DB_File::Error` giving both lengths.
- On a database opened read-only it returns -1.

Differences from upstream

Fully compatible with upstream. Covered by `t/81-xs-native/DB_File/010-phenotype.t`.

See also

- `get` - reads a value back
- `del` - removes a key, or the item at the cursor
- `seq` - sets the cursor `R_CURSOR` writes at

fd

The file descriptor of the database's file - for a `$DB_RECNO` array, its text file - or -1 for an in-memory database.

Synopsis

```
my $fd = $db->fd;
```

What you get back

A descriptor number, or -1.

Edge cases

- To keep processes from working on a database at the same time, lock a separate lock file before `tie` and unlock it after `untie`, rather than locking this descriptor: only then is everything a process wrote in the file before the next one reads it.

Differences from upstream

Fully compatible with upstream. Covered by `t/81-xs-native/DB_File/010-phenotype.t`.

See also

- `sync` - writes the database to its file

sync

Write everything stored so far to the file and to disk, so that other processes - and the file after a crash - have it.

Synopsis

```
$db->sync;
```

What you get back

0, or -1 for an error. The only flag it takes is `R_RECNOSYNC`, which is 0; any other returns -1.

Edge cases

- A `$DB_RECNO` array writes its text file only when an element changed since the last write.
- Without `sync`, writes reach the file at `untie`, or as they happen with `$Peta::FFI::LMDB::DBM_WRITE = 'through'`.

Differences from upstream

Fully compatible with upstream. Covered by `t/82-native/Peta-FFI-LMDB/100-db_file.t`.

See also

- `fd` - the file's descriptor

seq

Move the database's cursor and read the key and value it lands on: walk the database in order, or find the first key at or after a given one.

Synopsis

```
my ($key, $value) = ('', '');
my $status = $db->seq($key, $value, R_FIRST);
```

What you get back

0 with `$key` and `$value` set to where the cursor landed, 1 when there is nothing there (past either end, or no such key), -1 for an error. On 1 the cursor stays where it was. For an array, `$key` is an index.

Flags

- `R_FIRST`, `R_LAST` - the first or last key.
- `R_NEXT`, `R_PREV` - the key after or before the cursor; from no cursor yet, the first or the last.
- `R_CURSOR` - in a `$DB_BTREE` database the first key at or after `$key`; in a `$DB_HASH` database `$key` itself; in an array element `$key`.

Examples

Walk a database in order:

```
my ($k, $v) = ('', '');
for (my $st = $db->seq($k, $v, R_FIRST); $st == 0; $st = $db->seq(
 ↪$k, $v, R_NEXT)) {
    print "$k=$v\n";
}
```

Find the first key starting with `Wa` in a `$DB_BTREE` database:

```
@h{qw(Smith Wall Walls)} = (1, 2, 3);
my ($k, $v) = ('Wa', '');
print "$k\n" if $db->seq($k, $v, R_CURSOR) == 0 and $k =~ /^Wa/;   #␣
 ↪Wall
```

Edge cases

- `$key` and `$value` must be variables: the method writes them.
- Array elements created empty are passed over; `R_CURSOR` on one returns `-30995` (Berkeley DB's `DB_KEYEMPTY`).
- In an `R_DUP` database `R_NEXT` and `R_PREV` step through a key's values one by one, in the order they were stored.
- The tie interface (`keys`, `each`) moves the same cursor.

Differences from upstream

Fully compatible with upstream. Covered by `t/81-xs-native/DB_File/010-phenotype.t`.

See also

- `get` - looks up one key without moving the cursor
- `del` and `put` - act at the cursor with `R_CURSOR`
- `get_dup` - all the values of a key

filter_fetch_key

Install a sub that rewrites every key read from the database - by `keys`, `each` or `seq` - before your code sees it.

Synopsis

```
$db->filter_fetch_key(sub { s/\0$// });   # drop a C string's NUL
$db->filter_fetch_key(undef);            # remove the filter
```

What you get back

The filter it replaces, or undef if there was none.

Edge cases

- The sub works on `$_`, a copy of the key, and changes it in place; its return value is ignored.
- A filter that uses the same database croaks `recursion detected in filter_fetch_key`.
- The `DBM_Filter` module offers ready-made filters (`utf8`, `null`, `int32` and others) installed through these four methods.

Differences from upstream

Fully compatible with upstream. Covered by `t/81-xs-native/DB_File/110-upstream-db-hash.t`.

See also

- `filter_store_key` - the other direction, for keys written
- `filter_fetch_value` - the same for values

filter_store_key

Install a sub that rewrites every key on its way into the database - for storing, fetching, deleting or looking up - so the database holds the rewritten form.

Synopsis

```
$db->filter_store_key(sub { $_ .= "\0" });   # keys as C strings
$db->filter_store_key(undef);               # remove the filter
```

What you get back

The filter it replaces, or undef if there was none.

Edge cases

- The sub works on `$_`, a copy: your variable keeps its value.
- It runs for `seq` and `del` too, so a key looked up matches the keys stored.
- A filter that uses the same database croaks `recursion detected in filter_store_key`.

Differences from upstream

Fully compatible with upstream. Covered by `t/81-xs-native/DB_File/110-upstream-db-hash.t`.

See also

- `filter_fetch_key` - undoes the rewrite on the way out
- `filter_store_value` - the same for values

filter_fetch_value

Install a sub that rewrites every value read from the database before your code sees it.

Synopsis

```
$db->filter_fetch_value(sub { $_ = decode('UTF-8', $_) });
$db->filter_fetch_value(undef);    # remove the filter
```

What you get back

The filter it replaces, or undef if there was none.

Edge cases

- The sub works on `$_` and changes it in place. It runs only for a value that was found: a missing key's undef passes unfiltered.
- A filter that uses the same database croaks `recursion detected in filter_fetch_value`.

Differences from upstream

Fully compatible with upstream. Covered by `t/81-xs-native/DB_File/110-upstream-db-hash.t`.

See also

- `filter_store_value` - the rewrite on the way in
- `filter_fetch_key` - the same for keys

filter_store_value

Install a sub that rewrites every value on its way into the database, so the database holds the rewritten form.

Synopsis

```
$db->filter_store_value(sub { $_ = encode('UTF-8', $_) });
$db->filter_store_value(undef);    # remove the filter
```

What you get back

The filter it replaces, or undef if there was none.

Edge cases

- The sub works on `$_`, a copy: your variable keeps its value.
- A filter that uses the same database croaks `recursion detected in filter_store_value`.

Differences from upstream

Fully compatible with upstream. Covered by `t/81-xs-native/DB_File/110-upstream-db-hash.t`.

See also

- `filter_fetch_value` - undoes the rewrite on the way out
- `filter_store_key` - the same for keys

5.2.27 Data

Namespace containing the following modules:

- `Data::Dumper` - Stringify Perl data structures into valid Perl code - feed the result back through `eval` and you rebuild the original structure.
- `Data::UUID` - Generate UUIDs (GUIDs): time-based ones that differ from every other, and name-based ones that are the same for the same namespace and name.

Data::Dumper

Stringify Perl data structures into valid Perl code - feed the result back through `eval` and you rebuild the original structure.

Two calling styles cover almost everything:

- **Functional**: `Dumper(\%h, \@a)` - one call, one string, uses the package-global configuration variables (`$Data::Dumper::Indent` etc.).
- **Object-oriented**: `Data::Dumper->new([\%h, \@a], ['hash', 'array'])` returns a configurable dumper object. Chain accessors to tune output, then call `->Dump` to produce the string.

The main knobs:

- `$Indent` - `0` compact, 1 line-wrapped, 2 line-wrapped with alignment (the default and the most readable).
- `$Purity` - when true, round-trip fidelity for self-referential and cyclic structures at the cost of extra fixup statements.
- `$Sortkeys` - 1 for alphabetical, or a coderef returning the key order per hash.
- `$Deepcopy` - break shared references and emit each one as a copy.
- `$Terse` - omit the `$VAR1 =` prefix; emits a bare expression.
- `$Useqq` - quote strings with `""` and visible escapes (`\n`, `\t`, `\x{...}`) instead of `'...'`.
- `$Sparseseen` - skip populating the internal "seen" hash for values that the dumper can prove are never revisited. Serialization gets faster; the price is that the contents of the seen hash, as read through `Seen`, can no longer be relied on.

Cycles and shared references are tracked through an internal *seen* table keyed by referent address; a second visit prints as a backref like `$VAR1->[0]` instead of recursing forever.

The full surface is the exported `Dumper`, the constructor `new`, the serializers `Dump` and `Dumpxs`, the seen-table controls `Reset` and `Seen`, the input accessors `Values` and `Names`, and the configuration accessors listed above. `Dump` and `Dumpxs` take the same arguments and produce the same output; `Dumpxs` is the faster of the two, and `Dump` delegates to it unless `$Data::Dumper::Useperl` is true.

```
use Data::Dumper;
$Data::Dumper::Sortkeys = 1;

print Dumper({ name => 'pperl', list => [1, 2, 'three'] });
# $VAR1 = {
#           'list' => [
#                       1,
#                       2,
#                       'three'
#                     ],
#           'name' => 'pperl'
#         };
```

(continues on next page)

(continued from previous page)

```
my $d = Data::Dumper->new([[1, 2]], ['pair']);
print $d->Indent(0)->Dump, "\n";                 # $pair = [1,2];

{
    local $Data::Dumper::Indent = 0;
    local $Data::Dumper::Terse  = 1;
    print Dumper([1, { a => undef }]), "\n";    # [1,{'a' => undef}]
}
```

This is a native implementation of the XS half of Data::Dumper 2.192; its `Data/Dumper.pm` is used unchanged, and `use Data::Dumper` exports `Dumper`.

Differences from upstream

None.

Functions

Core API

Dumpxs

Serialize one or more values into valid Perl code, fast.

Dumpxs

Serialize one or more values into valid Perl code, fast.

`Dumpxs` and `Dump` take the same arguments and produce the same output; `Dumpxs` is the faster of the two, and `Dump` delegates here unless `$Data::Dumper::Useperl` is true.

Synopsis

```
use Data::Dumper;
print Data::Dumper->new([\%h, \@a], ['hash', 'array'])->Dumpxs;
print Data::Dumper::Dumpxs('Data::Dumper', [\%h], ['hash']);
```

What you get back

In scalar context, a single string containing one `$VAR = ...;` statement per input value, separated by `$self->{sep}` (default `"\n"`). In list context, one string per input - useful when you want to post-process each dump separately.

With `Terse(1)`, the `$VAR =` prefix and trailing `;` are dropped and you get a bare Perl expression suitable for embedding.

Examples

Named output, default formatting:

```
my %h = (a => 1, b => [2, 3]);
print Data::Dumper->new([\%h], ['h'])->Dumpxs;
# $h = {
#   'a' => 1,
#   'b' => [2, 3]
# };
```

Compact, sorted, double-quoted:

```
print Data::Dumper->new([\%h])
                  ->Indent(0)->Sortkeys(1)->Useqq(1)->Dumpxs;
# $VAR1 = {"a" => 1,"b" => [2,3]};
```

Circular structure - the cycle is emitted as a backref, not an infinite expansion:

```
my @a = (1, 2);
push @a, \@a;
print Data::Dumper->new([\@a])->Dumpxs;
# $VAR1 = [1, 2, $VAR1];
```

List context gives you one string per argument:

```
my ($s1, $s2) = Data::Dumper->new([\%h, \@a])->Dumpxs;
```

Edge cases

- First argument is not a blessed hashref: `Dumpxs` will call `new()` internally using the remaining arguments, so `Data::Dumper::Dumpxs('Data::Dumper', \@vals, \@names)` works.
- `maxrecurse` exceeded: croaks with `"Recursion limit of N exceeded"` after finishing the current top-level value so partial output is not lost from the caller's perspective.
- `maxdepth` exceeded: prints the quoted stringification of the ref (e.g. `'ARRAY(0x...)'`) at the cut-off point rather than croaking.
- CODE references: emitted as `sub { "DUMMY" }` unless `Deparse(1)` is set, in which case the deparsed source is emitted.
- Regexp references: emitted as `qr/.../flags`.
- `Sortkeys` as a coderef: called once per hash to obtain the key order; a truthy non-coderef sorts alphabetically.

See also

- `Dump` - the pure-Perl fallback; identical output, slower.
- `Dumper` - the procedural one-liner wrapping `new(...)->Dumpxs`.
- `Seen` - pre-seed the seen-table to name pre-existing references.
- `Reset` - clear the seen-table between unrelated `Dumpxs` calls.

Data::UUID

Generate UUIDs (GUIDs): time-based ones that differ from every other, and name-based ones that are the same for the same namespace and name.

A generator object (`Data::UUID->new`) makes version 1 UUIDs from the clock and a node id of its own, and version 3 UUIDs from an MD5 of a namespace UUID and a name. Every UUID comes in four forms: 16 raw bytes, the 36-character string, `0x` and 32 hex digits, or Base64.

Synopsis

```
use Data::UUID;    # exports NameSpace_DNS, NameSpace_URL,␣
↪NameSpace_OID, NameSpace_X500

my $ug   = Data::UUID->new;
my $uuid = $ug->create;                          # 16 bytes, version␣
↪1
print $ug->to_string($uuid), "\n";               # e.g. 4162F712-
↪1DD2-11B2-B17E-C09EFE1DC403
print $ug->create_from_name_str(NameSpace_DNS, 'www.example.com'),
↪"\n";
                                                  # 5DF41881-3AED-
↪3515-88A7-2F4A814CF09E
my $back = $ug->from_string('6ba7b810-9dad-11d1-80b4-00c04fd430c8');
print $ug->compare($uuid, $back), "\n";          # -1, 0 or 1
```

The binary form

The 16 bytes are Data::UUID's own: the time and version fields in the machine's byte order, as its C structure holds them, not the network order of RFC 4122. `to_string` and `from_string` convert correctly between them and the standard text form; the binary form itself is for Data::UUID's own functions.

This is a native implementation of Data::UUID 1.227's `UUID.xs`; its `Data/UUID.pm` is used unchanged.

Differences from upstream

Five, each where perl 5.44 with UUID.xs reads or writes memory it does not own (t/81-xs-native/Data-UUID/150-or-better.t):

- A binary UUID argument shorter than 16 bytes is taken with zero bytes after its end; perl5 reads past the string.
- `from_b64string` of fewer than 16 bytes fills the rest with zero bytes; perl5 returns uninitialised memory there.
- `from_b64string` of more than 16 bytes keeps the first 16; perl5 writes the rest past its buffer.
- A byte above 0x7F in `from_b64string`'s input is skipped like any other character outside Base64; perl5 looks it up before its table.
- `from_string` of a string that ends after a single hex digit of a byte before the last dies "from_string(...) failed..."; perl5 takes the next byte from past the end of the string.

Functions

Generator

new

Create a generator, with a node id of its own.

create

Make a new time-based (version 1) UUID.

Also available as `Data::UUID::create_b64`, `Data::UUID::create_bin`, `Data::UUID::create_hex`, `Data::UUID::create_str`.

create_from_name

Make the name-based (version 3) UUID of a name in a namespace: the same namespace and name give the same UUID, everywhere and always.

Also available as `Data::UUID::create_from_name_b64`, `Data::UUID::create_from_name_bin`, `Data::UUID::create_from_name_hex`, `Data::UUID::create_from_name_str`.

Conversion

compare

Compare two binary UUIDs: -1, 0 or 1.

to_b64string

Convert a binary UUID to text.

Also available as `Data::UUID::to_hexstring`, `Data::UUID::to_string`.

from_b64string

Convert text back to a binary UUID.

Also available as `Data::UUID::from_hexstring`, `Data::UUID::from_string`.

new

Create a generator, with a node id of its own.

The node id is drawn from an MD5 of the host id, the time and the host name, with the multicast bit set so that it can never be a network card's address. The object is always a `Data::UUID`, also when `new` is called through a subclass. `$!` is 0 afterwards.

Synopsis

```
my $ug = Data::UUID->new;
```

create

Make a new time-based (version 1) UUID.

Also available as `Data::UUID::create_b64`, `Data::UUID::create_bin`, `Data::UUID::create_hex`, `Data::UUID::create_str`.

`create` and `create_bin` return the 16 binary bytes, `create_str` the 36-character string, `create_hex` `0x` and 32 hex digits, `create_b64` 24 Base64 characters. Successive UUIDs of one generator compare in the order they were made.

Synopsis

```
my $id  = $ug->create_str;    # e.g. 4162F712-1DD2-11B2-B17E-
↪C09EFE1DC403
my $bin = $ug->create;
```

create_from_name

Make the name-based (version 3) UUID of a name in a namespace: the same namespace and name give the same UUID, everywhere and always.

Also available as `Data::UUID::create_from_name_b64`, `Data::UUID::create_from_name_bin`, `Data::UUID::create_from_name_hex`, `Data::UUID::create_from_name_str`.

The namespace is a binary UUID, usually one of the exported `NameSpace_DNS`, `NameSpace_URL`, `NameSpace_OID` and `NameSpace_X500`. The name is taken as bytes; a character above U+00FF dies "Wide character in subroutine entry". The forms are those of `create`.

Synopsis

```
print $ug->create_from_name_str(NameSpace_DNS, 'www.example.com');
# 5DF41881-3AED-3515-88A7-2F4A814CF09E
```

compare

Compare two binary UUIDs: -1, 0 or 1.

As in Data::UUID, the fields are compared in turn and a later field that differs decides, the node's last differing byte above all; two UUIDs of one generator compare in the order they were made.

Synopsis

```
my @sorted = sort { $ug->compare($a, $b) } @uuids;
```

to_b64string

Convert a binary UUID to text.

Also available as `Data::UUID::to_hexstring`, `Data::UUID::to_string`.

`to_string` gives the 36-character form in upper case, `to_hexstring` `0x` and 32 upper-case hex digits, `to_b64string` 24 Base64 characters.

Synopsis

```
print $ug->to_string(NameSpace_DNS);   # 6BA7B810-9DAD-11D1-80B4-
↪00C04FD430C8
```

from_b64string

Convert text back to a binary UUID.

Also available as `Data::UUID::from_hexstring`, `Data::UUID::from_string`.

`from_string` and `from_hexstring` take 32 hex digits in either case, with an optional `0x` in front and a `-` allowed before any byte; anything after the 16th byte is ignored, and a string that runs out before it dies "from_string(...) failed...". `from_b64string` takes Base64, skipping characters outside it.

Synopsis

```
my $bin = $ug->from_string('6ba7b810-9dad-11d1-80b4-00c04fd430c8');
```

5.2.28 Devel

Namespace containing the following modules:

- `Devel::Cover` - Measure which statements, branches, conditions and subroutines a program ran, and how long each statement took: the collector under Devel::Cover.
- `Devel::Peek` - Inspect the internal representation of any Perl scalar - a debugging tool for working with SVs.
- `Devel::SelfStubber` - generate stubs for a SelfLoading module
- `Devel::Size` - how much memory a Perl variable uses

Devel::Cover

Measure which statements, branches, conditions and subroutines a program ran, and how long each statement took: the collector under Devel::Cover.

`perl -MDevel::Cover program` runs the program and writes what it covered to `cover_db`; the `cover` command turns that into a report. `cover -test` does both for a distribution's test suite.

Synopsis

```
pperl --cpan Devel::Cover
pperl -MDevel::Cover yourprog args
cover
cover -test
```

This is a native implementation of the XS half of Devel::Cover 1.52 (`Cover.xs`); its modules and the `cover` command are upstream's, installed with `pperl --cpan Devel::Cover`.

Differences from upstream

- While Devel::Cover is loaded, pperl's JIT compiles no loops and gives back those it had compiled: every op runs in the interpreter, so the counts are perl's.

Devel::Peek

Inspect the internal representation of any Perl scalar - a debugging tool for working with SVs.

`Devel::Peek` is the tool you reach for when the question is *"what does Perl actually think this value is?"*. Typical uses:

- chasing a refcount leak - something holds a reference longer than you expect, and you want to see `REFCNT` climb;
- confirming that a value came back as the type you meant (`IV` vs `PV` vs `PVMG`, a blessed object, a tied scalar);
- looking at the flags on an `SV` (`POK`, `IOK`, `ROK`, `UTF8`, `TEMP`, `OBJECT`) to diagnose dual-var, stringification, or tainting surprises;

- peeking inside arrays, hashes, and references to see how Perl has laid them out and which slots contain what.

Synopsis

```
use Devel::Peek;

my $x = 42;
Dump($x);
```

A minimal `Dump($x)` for a plain integer looks like this on `STDERR`:

```
SV = IV(0xbc818) at 0xbe9a8
  REFCNT = 1
  FLAGS = (IOK,pIOK)
  IV = 42
```

Every line is meaningful. `SV = IV(...)` names the body type. `REFCNT` is what you stare at when tracking leaks. `FLAGS` tells you which of the `IV` / `NV` / `PV` slots below it are live; `IOK` means the `IV` field is current, `POK` means the `PV` field is, `ROK` means the scalar is a reference and there is another `SV = ...` block nested below it. Arrays dump with `FILL` (last index) and `MAX` (capacity) and then their elements as `Elt No. N` sub-dumps. Hashes dump with `KEYS`, bucket occupancy, and one sub-dump per entry.

The entry points:

- `Dump($sv, $depth=4)` - dump one value. `@array` and `%hash` dump the aggregate itself rather than the first element; other arguments are evaluated in scalar rvalue context.
- `DumpArray($depth, @values)` - dump several values in one call, each tagged with its position. Useful when inspecting a function's return list.
- `SvREFCNT(\$sv)` - return the refcount of the referent of a reference (the outer reference itself is subtracted). Pairs well with `Dump` when you want the number without scraping it out of a dump.
- `DumpWithOP($sv, $depth=4)` - same as `Dump` but also walks into compiled ops when the scalar carries one (a `PVCV`, for example). Equivalent to setting `$Devel::Peek::dump_ops` around a `Dump` call.
- `DumpProg()` - dump the program's main op tree. Aimed at people debugging the compiler, not at regular scripts.
- `DeadCode()` - walk the SV arena for scalars frozen into inactive `CV`s. A deep-internals tool; most users never touch it.
- `CvGV($cv)` - return the `GV` associated with a code reference, so you can recover a sub's package-qualified name.

The `mstat` family

`mstat`, `fill_mstats`, `mstats_fillhash`, and `mstats2hash` report on Perl's malloc arena - bucket counts, free/used chunks, sbrk activity. They only produce real numbers when Perl was built with its bundled `malloc` (rare in modern builds). Under a standard build, `mstat` prints `perl not compiled with MYMALLOC` and the hash-filling variants croak with the same message. This is the upstream behaviour, not a pperl limitation; scripts that guard on it already handle the non-`MYMALLOC` case.

The `runops` and `:opd` knobs

`use Devel::Peek ':opd=st'` switches the interpreter to its debugging dispatch loop and turns on per-op trace flags (`s` stack, `t` trace, `P` profile). `runops_debug()` is the run-time toggle for the same switch; `debug_flags()` reads and writes the `$^D` letter mask directly. Reach for these when you want to see which ops execute and in what order - they are the programmable equivalent of `perl -Dst`.

All output from `Dump`, `DumpArray`, `DumpProg`, and `mstat` goes to `STDERR`. The global `$Devel::Peek::pv_limit` caps how many characters of each PV are printed; `0` means no limit.

This is a native implementation of the XS half of Devel::Peek 1.36; its `Devel/Peek.pm` is used unchanged, and `use Devel::Peek` exports `Dump`, `DumpArray`, `DeadCode`, `mstat` and the other functions.

Differences from upstream

None in the format. The addresses, reference counts and flags `Dump` prints are pperl's own, as perl5's are its own: two runs of perl5 differ there as well.

Functions

Scalar inspection

`Dump`

Dump one scalar's internal representation to `STDERR`. The usual entry point for `Devel::Peek`, and the one to reach for first when you need to see what Perl has under the hood.

`DumpArray`

Dump several values in one call, each tagged with its index. Useful when inspecting a function's return list or an argument list without wrapping every element in its own `Dump`.

`SvREFCNT`

Return the reference count of a referent. `SvREFCNT(\$x)` answers *"how many live references to* `$x` *exist right now?"* - the usual tool for tracking a suspected leak without pulling the number out of a full `Dump`.

DeadCode

Report scalars "frozen" into inactive code references - a deep-internals probe for leaks inside closed-over CVs.

DumpProg

Dump the program's main op tree. A compiler-debugging tool, not a value-inspection tool.

CvGV

Recover the GV (typeglob) associated with a code reference. `CvGV($cv)` is how you go from an anonymous-looking sub ref back to the name Perl filed it under - useful when inspecting caller frames, dispatch tables, or closures assigned to named slots.

Memory stats

fill_mstats

Fill a scalar with a machine-readable snapshot of Perl's malloc state. `fill_mstats($buf)` is the cheap member of the `mstat` family: pair it with `mstats2hash` later when you actually want to read the numbers.

mstats_fillhash

Populate a hash with Perl's malloc statistics. `mstats_fillhash(%hash)` fills `%hash` with fields like `nbuckets`, `total`, `totfree`, `sbrks`, plus the per-bucket `free` / `used` / `mem_size` / `available_size` array references. See `perldebguts` for the full field list.

mstats2hash

Decode a `fill_mstats` buffer into a hash. `mstats2hash($buf, %hash)` is the readout step for the buffered form: cheap `fill_mstats` inside a hot loop, one `mstats2hash` pass at the end to turn the buffer into inspectable fields.

mstat

Print a one-line memory snapshot to STDERR. The classic marker-in-a-log tool - sprinkle `mstat "Point 5"` across a program to see how arena size moves between points.

Runops hooks

runops_debug

Switch the interpreter between its standard and debugging op-dispatch loops, and report which one was active before the call.

Dump

Dump one scalar's internal representation to STDERR. The usual entry point for Devel::Peek, and the one to reach for first when you need to see what Perl has under the hood.

Dump($sv, $depth=4). $depth caps both recursion into nested references and the number of array/hash elements shown at each level; pass a larger value to see past the default cutoff. When called on @array or %hash directly, the aggregate itself is dumped (not its first element). Any other argument is evaluated in scalar rvalue context.

The dump layout follows the shape shown in the module summary: an SV = <type>(body) at <head> line, then REFCNT, FLAGS, and whichever of the IV / NV / PV / RV / array / hash slots are populated. Reference dumps nest a child SV = ... block inside the parent.

Two package variables tune the output:

- $Devel::Peek::pv_limit - cap on characters printed per PV; 0 lifts the cap entirely.
- $Devel::Peek::dump_ops - when true, CV-like scalars also print their op tree (see DumpWithOP for the scoped form).

Returns nothing; all output is on STDERR.

DumpArray

Dump several values in one call, each tagged with its index. Useful when inspecting a function's return list or an argument list without wrapping every element in its own Dump.

DumpArray($depth, @values). The first argument is the recursion depth (same meaning as Dump's second argument); the remaining arguments are the values to dump. Each value is preceded by a header line of the form Elt No. N 0x<address>, where N counts from zero. After the header the element is rendered exactly as Dump would render it.

Respects $Devel::Peek::pv_limit and $Devel::Peek::dump_ops the same way Dump does. Returns nothing; output goes to STDERR.

SvREFCNT

Return the reference count of a referent. SvREFCNT(\$x) answers *"how many live references to* $x *exist right now?"* - the usual tool for tracking a suspected leak without pulling the number out of a full Dump.

The argument must be a reference (prototype \[$@%&*]). The referent's refcount is returned, minus one so that the temporary reference created by the \ operator does not skew the answer. Passing anything that is not a reference croaks with Usage: Devel::Peek::SvREFCNT(SCALAR); calling it (with &) on other than exactly one argument croaks with Usage: Devel::Peek::SvREFCNT(sv).

Typical pattern: snapshot the count before and after a suspected leak site and look for growth that does not drop back.

DeadCode

Report scalars "frozen" into inactive code references - a deep-internals probe for leaks inside closed-over CVs.

`DeadCode()` walks Perl's SV arena looking for PVCV bodies whose pads still hold live scalars even though the sub is no longer reachable, and returns the findings as an array reference. Most scripts never need this; it is a tool for people chasing exotic leaks that `Dump` and `SvREFCNT` cannot locate.

Returns an array reference. An empty result means "nothing matched the arena-walk heuristic" - which is the expected outcome under pperl, whose allocator layout differs from the arena the upstream walk was written against.

mstat

Print a one-line memory snapshot to STDERR. The classic marker-in-a-log tool - sprinkle `mstat "Point 5"` across a program to see how arena size moves between points.

`mstat($marker="Devel::Peek::mstat: ")`. The marker prefixes the output so you can match lines to call sites when the log gets noisy. Output format is documented under `perldebguts/Using $ENV{PERL_DEBUG_MSTATS}`.

Only produces real numbers when Perl was built with its bundled `malloc`. On a standard build the line reads `<marker>: perl not compiled with MYMALLOC`, which is useful as a pass-through breadcrumb even when the numbers are unavailable.

Devel::SelfStubber

generate stubs for a SelfLoading module

SYNOPSIS

To generate just the stubs:

```
use Devel::SelfStubber;
Devel::SelfStubber->stub('MODULENAME','MY_LIB_DIR');
```

or to generate the whole module with stubs inserted correctly

```
use Devel::SelfStubber;
$Devel::SelfStubber::JUST_STUBS=0;
Devel::SelfStubber->stub('MODULENAME','MY_LIB_DIR');
```

MODULENAME is the Perl module name, e.g. Devel::SelfStubber, NOT 'Devel/SelfStubber' or 'Devel/SelfStubber.pm'.

MY_LIB_DIR defaults to '.' if not present.

DESCRIPTION

Devel::SelfStubber prints the stubs you need to put in the module before the __DATA__ token (or you can get it to print the entire module with stubs correctly placed). The stubs ensure that if a method is called, it will get loaded. They are needed specifically for inherited autoloaded methods.

This is best explained using the following example:

Assume four classes, A,B,C & D.

A is the root class, B is a subclass of A, C is a subclass of B, and D is another subclass of A.

```
    A
   / \
  B   D
 /
C
```

If D calls an autoloaded method 'foo' which is defined in class A, then the method is loaded into class A, then executed. If C then calls method 'foo', and that method was reimplemented in class B, but set to be autoloaded, then the lookup mechanism never gets to the AUTOLOAD mechanism in B because it first finds the method already loaded in A, and so erroneously uses that. If the method foo had been stubbed in B, then the lookup mechanism would have found the stub, and correctly loaded and used the sub from B.

So, for classes and subclasses to have inheritance correctly work with autoloading, you need to ensure stubs are loaded.

The SelfLoader can load stubs automatically at module initialization with the statement 'SelfLoader->load_stubs()';, but you may wish to avoid having the stub loading overhead associated with your initialization (though note that the SelfLoader::load_stubs method will be called sooner or later - at latest when the first sub is being autoloaded). In this case, you can put the sub stubs before the __DATA__ token. This can be done manually, but this module allows automatic generation of the stubs.

By default it just prints the stubs, but you can set the global $Devel::SelfStubber::JUST_STUBS to 0 and it will print out the entire module with the stubs positioned correctly.

At the very least, this is useful to see what the SelfLoader thinks are stubs - in order to ensure future versions of the SelfStubber remain in step with the SelfLoader, the SelfStubber actually uses the SelfLoader to determine which stubs are needed.

Devel::Size

`Devel::Size` - how much memory a Perl variable uses.

```
use Devel::Size qw(size total_size);

my $bytes = size("A string");         # the scalar itself
my @list  = (1 .. 1000);
my $flat  = size(\@list);             # the array, not its elements
my $deep  = total_size(\@list);       # the array and all it holds
```

`size` counts a variable and the memory it owns directly: a scalar's string buffer, an array's slot vector, a hash's buckets and keys. `total_size` also follows what an array or hash contains, and what references point to, counting each thing reachable once. Pass a reference to measure an array, a hash or a code reference.

The numbers are the sizes of the interpreter's own structures, laid out as perl 5.44 lays them out, so they agree with perl 5.44 running the XS Devel::Size. `$Devel::Size::warn` (default 1) turns the module's warnings about sizes it cannot work out completely on or off; `$Devel::Size::dangle` (default 0) enables warnings about invalid pointers met on the way.

This is a native implementation of the CPAN distribution Devel-Size 0.87; its `Devel/Size.pm` is used unchanged.

Functions

Other Functions

size

`size($thing)` (`ix` 0) / `total_size($thing)` (`ix` 2, its ALIAS): bytes used by `$thing`, or by what a reference points to.

Also available as `Devel::Size::total_size`.

5.2.29 Digest

Modules that calculate message digests

SYNOPSIS

```
$md5   = Digest->new("MD5");
$sha1  = Digest->new("SHA-1");
$sha256 = Digest->new("SHA-256");
$sha384 = Digest->new("SHA-384");
$sha512 = Digest->new("SHA-512");

$hmac = Digest->HMAC_MD5($key);
```

DESCRIPTION

The `Digest::` modules calculate digests, also called "fingerprints" or "hashes", of some data, called a message. The digest is (usually) some small/fixed size string. The actual size of the digest depend of the algorithm used. The message is simply a sequence of arbitrary bytes or bits.

An important property of the digest algorithms is that the digest is *likely* to change if the message change in some way. Another property is that digest functions are one-way functions, that is it should be *hard* to find a message that correspond to some given digest. Algorithms differ in how "likely" and how "hard", as well as how efficient they are to compute.

Note that the properties of the algorithms change over time, as the algorithms are analyzed and machines grow faster. If your application for instance depends on it being "impossible" to generate the same digest for a different message it is wise to make it easy to plug in stronger algorithms as the one used grow weaker. Using the interface documented here should make it easy to change algorithms later.

All `Digest::` modules provide the same programming interface. A functional interface for simple use, as well as an object oriented interface that can handle messages of arbitrary length and which can read files directly.

The digest can be delivered in three formats:

- *binary*

  This is the most compact form, but it is not well suited for printing or embedding in places that can't handle arbitrary data.

- *hex*

  A twice as long string of lowercase hexadecimal digits.

- *base64*

  A string of portable printable characters. This is the base64 encoded representation of the digest with any trailing padding removed. The string will be about 30% longer than the binary version. *MIME::Base64* tells you more about this encoding.

The functional interface is simply importable functions with the same name as the algorithm. The functions take the message as argument and return the digest. Example:

```
use Digest::MD5 qw(md5);
$digest = md5($message);
```

There are also versions of the functions with "_hex" or "_base64" appended to the name, which returns the digest in the indicated form.

OO INTERFACE

The following methods are available for all `Digest::` modules:

- $ctx = Digest->XXX($arg,...)
- $ctx = Digest->new(XXX => $arg,...)
- $ctx = Digest::XXX->new($arg,...)

  The constructor returns some object that encapsulate the state of the message-digest algorithm. You can add data to the object and finally ask for the digest. The "XXX" should of course be replaced by the proper name of the digest algorithm you want to use.

  The two first forms are simply syntactic sugar which automatically load the right module on first use. The second form allow you to use algorithm names which contains letters which are not legal perl identifiers, e.g. "SHA-1". If no implementation for the given algorithm can be found, then an exception is raised.

  To know what arguments (if any) the constructor takes (the `$args,...` above) consult the docs for the specific digest implementation.

If new() is called as an instance method (i.e. $ctx->new) it will just reset the state the object to the state of a newly created object. No new object is created in this case, and the return value is the reference to the object (i.e. $ctx).

- $other_ctx = $ctx->clone

  The clone method creates a copy of the digest state object and returns a reference to the copy.

- $ctx->reset

  This is just an alias for $ctx->new.

- $ctx->add($data)
- $ctx->add($chunk1, $chunk2, ...)

  The string value of the $data provided as argument is appended to the message we calculate the digest for. The return value is the $ctx object itself.

  If more arguments are provided then they are all appended to the message, thus all these lines will have the same effect on the state of the $ctx object:

```
$ctx->add("a"); $ctx->add("b"); $ctx->add("c");
$ctx->add("a")->add("b")->add("c");
$ctx->add("a", "b", "c");
$ctx->add("abc");
```

  Most algorithms are only defined for strings of bytes and this method might therefore croak if the provided arguments contain chars with ordinal number above 255.

- $ctx->addfile($io_handle)

  The $io_handle is read until EOF and the content is appended to the message we calculate the digest for. The return value is the $ctx object itself.

  The addfile() method will croak() if it fails reading data for some reason. If it croaks it is unpredictable what the state of the $ctx object will be in. The addfile() method might have been able to read the file partially before it failed. It is probably wise to discard or reset the $ctx object if this occurs.

  In most cases you want to make sure that the $io_handle is in "binmode" before you pass it as argument to the addfile() method.

- $ctx->add_bits($data, $nbits)
- $ctx->add_bits($bitstring)

  The add_bits() method is an alternative to add() that allow partial bytes to be appended to the message. Most users can just ignore this method since typical applications involve only whole-byte data.

  The two argument form of add_bits() will add the first $nbits bits from $data. For the last potentially partial byte only the high order `$nbits % 8` bits are used. If $nbits is greater than `length($data) * 8`, then this method would do the same as `$ctx->add($data)`.

The one argument form of add_bits() takes a $bitstring of "1" and "0" chars as argument. It's a shorthand for `$ctx->add_bits(pack("B*", $bitstring), length($bitstring))`.

The return value is the $ctx object itself.

This example shows two calls that should have the same effect:

```
$ctx->add_bits("111100001010");
$ctx->add_bits("\xF0\xA0", 12);
```

Most digest algorithms are byte based and for these it is not possible to add bits that are not a multiple of 8, and the add_bits() method will croak if you try.

- $ctx->digest

  Return the binary digest for the message.

  Note that the `digest` operation is effectively a destructive, read-once operation. Once it has been performed, the $ctx object is automatically `reset` and can be used to calculate another digest value. Call $ctx->clone->digest if you want to calculate the digest without resetting the digest state.

- $ctx->hexdigest

  Same as $ctx->digest, but will return the digest in hexadecimal form.

- $ctx->b64digest

  Same as $ctx->digest, but will return the digest as a base64 encoded string without padding.

- $ctx->base64_padded_digest

  Same as $ctx->digest, but will return the digest as a base64 encoded string.

Digest speed

This table should give some indication on the relative speed of different algorithms. It is sorted by throughput based on a benchmark done with of some implementations of this API:

```
Algorithm      Size    Implementation                  MB/s

MD4            128     Digest::MD4 v1.3              165.0
MD5            128     Digest::MD5 v2.33              98.8
SHA-256        256     Digest::SHA2 v1.1.0            66.7
SHA-1          160     Digest::SHA v4.3.1             58.9
SHA-1          160     Digest::SHA1 v2.10             48.8
SHA-256        256     Digest::SHA v4.3.1             41.3
Haval-256      256     Digest::Haval256 v1.0.4        39.8
SHA-384        384     Digest::SHA2 v1.1.0            19.6
SHA-512        512     Digest::SHA2 v1.1.0            19.3
SHA-384        384     Digest::SHA v4.3.1             19.2
SHA-512        512     Digest::SHA v4.3.1             19.2
Whirlpool      512     Digest::Whirlpool v1.0.2       13.0
```

(continues on next page)

(continued from previous page)

```
MD2              128     Digest::MD2 v2.03                9.5

Adler-32          32     Digest::Adler32 v0.03            1.3
CRC-16            16     Digest::CRC v0.05                1.1
CRC-32            32     Digest::CRC v0.05                1.1
MD5              128     Digest::Perl::MD5 v1.5           1.0
CRC-CCITT         16     Digest::CRC v0.05                0.8
```

These numbers was achieved Apr 2004 with ActivePerl-5.8.3 running under Linux on a P4 2.8 GHz CPU. The last 5 entries differ by being pure perl implementations of the algorithms, which explains why they are so slow.

SEE ALSO

Digest::Adler32, Digest::CRC, Digest::Haval256, Digest::HMAC, Digest::MD2, Digest::MD4, *Digest::MD5*, *Digest::SHA*, *Digest::SHA1*, Digest::SHA2, Digest::Whirlpool

New digest implementations should consider subclassing from *Digest::base*.

MIME::Base64

http://en.wikipedia.org/wiki/Cryptographic_hash_function

AUTHOR

Gisle Aas gisle@aas.no

The `Digest::` interface is based on the interface originally developed by Neil Winton for his `MD5` module.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

```
Copyright 1998-2006 Gisle Aas.
Copyright 1995,1996 Neil Winton.
```

Modules

- `Digest::MD5` - Compute MD5-128 message digests - fixed 128-bit fingerprints of arbitrary byte strings per RFC 1321.
- `Digest::SHA` - Compute SHA-1 and SHA-2 family message digests: SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, and SHA-512/256, with optional HMAC keying.
- `Digest::SHA1` - Compute SHA-1 message digests through the interface of Digest::SHA1, the CPAN module older code uses where newer code uses Digest::SHA.
- `Digest::base` - Digest base class
- `Digest::file` - Calculate digests of files

Digest::MD5

Compute MD5-128 message digests - fixed 128-bit fingerprints of arbitrary byte strings per RFC 1321.

`Digest::MD5` offers two idioms. Pick whichever fits the shape of the data you have at hand.

Functional one-shot

When the whole message is already in memory as a single string (or a few pieces you can list as arguments), the functional form is the shortest path:

```
use Digest::MD5 qw(md5 md5_hex md5_base64);

my $bin = md5($data);           # 16 raw bytes
my $hex = md5_hex($data);       # 32 lowercase hex chars
my $b64 = md5_base64($data);    # 22 base64 chars, no padding
```

All three take any number of arguments and concatenate them before hashing, so `md5("a", "b", "c")` equals `md5("abc")`.

OO incremental

When the message arrives in pieces - streaming bytes, a file read chunk by chunk, or data assembled across multiple code paths - build a context object, feed it, and ask for the digest at the end:

```
use Digest::MD5;

my $ctx = Digest::MD5->new;
$ctx->add($chunk1);
$ctx->add($chunk2, $chunk3);
$ctx->addfile($fh);
my $hex = $ctx->hexdigest;      # also resets the context
```

`add` returns the object, so calls chain: `$ctx->add("a")->add("b")`. Calling any of the `digest` / `hexdigest` / `b64digest` methods is destructive - the context is reset to an empty state afterwards, so the same object is immediately reusable. To peek at the digest without resetting, clone first: `$ctx->clone->hexdigest`.

Output encodings

Every function and method comes in three encodings of the same 128-bit digest:

- **Raw binary** (`md5`, `digest`) - 16 bytes. Suitable for packing into fixed-width fields or feeding into another binary protocol.
- **Hex** (`md5_hex`, `hexdigest`) - 32 lowercase characters from `0-9a-f`. This is the form you usually see in ETag headers and checksum files.
- **Base64** (`md5_base64`, `b64digest`) - 22 characters from the standard alphabet `A-Z a-z 0-9 + /`. The result is *not* padded to a multiple of 4; append `"=="` yourself if an interop partner expects padding.

File-handle ingestion

`addfile` reads an open file handle until EOF and feeds every byte into the context without buffering the whole thing in memory, so it scales to files of arbitrary size. Make sure the handle is in `binmode` first - MD5 is defined over bytes, not decoded text.

Security context

MD5 is **cryptographically broken for collision resistance**: attackers can construct two different messages with the same digest. Do not use it for new digital signatures, certificate fingerprints, password hashing, or any setting where an adversary chooses the input. For those, reach for `Digest::SHA` (SHA-256 or better) or a purpose-built password hash.

MD5 remains perfectly adequate - and widely used - for non-adversarial integrity checks: ETags, cache keys, change detection in build systems, deduplication of trusted data, and checksums over storage where the threat model is accidental corruption rather than forgery.

Bytes only - no wide characters

MD5 hashes bytes. Feeding a string containing code points above `U+00FF` croaks with `Wide character in subroutine entry`. To hash a Unicode string, decide on an encoding first and pass the encoded bytes: `md5_hex(encode_utf8($str))`.

Differences from upstream

This is Digest::MD5 2.59; `Digest/MD5.pm` is upstream's. Two differences, each where perl 5.44 with MD5.xs fails (t/81-xs-native/Digest-MD5/230-or-better.t):

- `Digest::MD5::new("No::Such::Class")` blesses into a new class of that name; perl5 crashes for a class that does not exist yet.
- `$ctx->context($blocks, $state)` with a state shorter than 16 bytes reads the missing bytes as zero; perl5 reads past the end of the string.

Functions

Functional digests

`md5`

One-shot MD5 digest, returned as 16 raw bytes.

`md5_hex`

One-shot MD5 digest, returned as a 32-character lowercase hex string.

`md5_base64`

One-shot MD5 digest, returned as a 22-character base64 string.

OO lifecycle

`new`

Create a fresh MD5 context object, or reset an existing one to empty.

`clone`

Make an independent copy of the current digest state.

`reset`

Clear an existing context so it behaves like a freshly created one.

OO input

`add`

Feed one or more pieces of data into the MD5 context.

`addfile`

Read an open file handle to EOF and feed every byte into the context.

`add_bits`

Add a bit-oriented quantum of data - but only in multiples of 8 bits.

OO output

`digest`

Finalise the context and return the 16-byte raw binary digest.

`hexdigest`

Finalise the context and return the digest as 32 lowercase hex chars.

`b64digest`

Finalise the context and return the digest as a 22-char base64 string.

Utility

context

Save or restore the raw internal MD5 state - for advanced use only.

new

Create a fresh MD5 context object, or reset an existing one to empty.

Synopsis

```
my $ctx = Digest::MD5->new;   # class method - new empty context
$ctx->new;                    # instance method - reset in place
```

What you get back

A blessed `Digest::MD5` reference whose internal state is the MD5 initial vector - i.e. the digest of the empty string is `md5("")`. Called on an existing instance, `new` resets that instance and returns it; no second object is allocated. The common idiom `Digest::MD5->new->add($data)->hexdigest` relies on this chaining.

Examples

Fresh context, then feed and finalise:

```
my $ctx = Digest::MD5->new;
$ctx->add("hello");
print $ctx->hexdigest;   # 5d41402abc4b2a76b9719d911017c592
```

Reset an existing context instead of allocating a new one:

```
$ctx->add("round 1")->hexdigest;   # digest + auto-reset
$ctx->new;                          # explicit reset (idiomatic no-
 ↪op here)
$ctx->add("round 2");
```

Chain straight through in one expression:

```
my $hex = Digest::MD5->new->add("foo", "bar")->hexdigest;
```

Edge cases

- Called on a subclass name, `new` blesses into that subclass, so `MyDigest->new` returns a `MyDigest` object whose internals are a `Digest::MD5` context.
- Called as an instance method, the same object is returned - the caller's variable is unchanged.

Differences from upstream

Fully compatible with upstream `Digest::MD5` 2.59.

See also

- `clone` - copy the current context instead of resetting it
- `reset` - explicit alias for the instance-method form of new
- `add` - feed data into the context you just created
- `hexdigest` - common companion call for one-shot chained use

clone

Make an independent copy of the current digest state.

Synopsis

```
my $copy = $ctx->clone;
```

What you get back

A new `Digest::MD5` object whose internal state is a byte-for-byte copy of the original - same accumulated length, same four state words, same partial-block buffer. The two contexts are independent afterwards: further `add` calls on one do not affect the other. The class of the returned object matches the class of `$ctx`, so cloning a subclass instance yields a subclass instance.

The primary use is peeking at an intermediate digest without destroying the ongoing computation. Because `digest` / `hexdigest` / `b64digest` reset the context, the pattern `$ctx->clone->hexdigest` is how you extract a snapshot and keep hashing.

Examples

Per-line running digest over a stream:

```
my $ctx = Digest::MD5->new;
while (my $line = <$fh>) {
    $ctx->add($line);
    print "after line $.: ", $ctx->clone->hexdigest, "\n";
}
my $final = $ctx->hexdigest;    # digest of the whole stream
```

Fork a digest to feed two suffixes:

```
my $base = Digest::MD5->new;
$base->add("common prefix ");

my $a = $base->clone->add("suffix A")->hexdigest;
my $b = $base->clone->add("suffix B")->hexdigest;
```

Edge cases

- Cloning immediately after `new` yields a context equivalent to a fresh `Digest::MD5->new`.
- Cloning an object whose state was restored via `context` copies that restored state, not a fresh one.

Differences from upstream

Fully compatible with upstream `Digest::MD5` 2.59.

See also

- `new` - create a fresh empty context from scratch
- `digest` - the destructive read that makes `clone` worth having
- `context` - save/restore raw state if you need persistence
- `add` - continue feeding data into the clone or the original

add

Feed one or more pieces of data into the MD5 context.

Synopsis

```
$ctx->add($data);
$ctx->add($a, $b, $c);                # same as add($a.$b.$c)
$ctx->add("a")->add("b")->add("c");   # chained equivalent
```

What you get back

The same context object, so `add` calls chain and compose with `hexdigest` / `digest` / `b64digest`. Every argument is flattened into a byte stream in order; the digest depends only on the concatenation, not on how you split it across calls or arguments.

Examples

One-shot digest of a concatenated message:

```
my $hex = Digest::MD5->new->add("hello world")->hexdigest;
# 5eb63bbbe01eeed093cb22bb8f5acdc3
```

Stream a file line by line:

```
open(my $fh, '<:raw', $path) or die $!;
my $ctx = Digest::MD5->new;
while (my $line = <$fh>) {
    $ctx->add($line);
```

(continues on next page)

(continued from previous page)

```
}
print $ctx->hexdigest, "\n";
```

Multiple arguments are equivalent to concatenation - all four of these produce the same digest:

```
Digest::MD5->new->add("a")->add("b")->add("c")->hexdigest;
Digest::MD5->new->add("a", "b", "c")->hexdigest;
Digest::MD5->new->add("abc")->hexdigest;
md5_hex("abc");
```

Edge cases

- **Wide characters.** Arguments containing code points above `U+00FF` croak with `Wide character in subroutine entry`. Encode first: `$ctx->add(encode_utf8($str))`.
- **Empty strings** are accepted and change nothing.
- **No arguments** (`$ctx->add` with no extra args) is a no-op and returns the context.
- `undef` arguments stringify to the empty string; warnings depend on the caller's `use warnings` state.

Differences from upstream

Fully compatible with upstream `Digest::MD5` 2.59.

See also

- `add_bits` - interface-compatibility wrapper for bit-oriented input
- `addfile` - feed an entire file handle without buffering
- `new` - typical predecessor
- `hexdigest` - typical successor; auto-resets the context
- `clone` - snapshot the intermediate state without stopping `add`

addfile

Read an open file handle to EOF and feed every byte into the context.

Synopsis

```
open(my $fh, '<:raw', $path) or die $!;
$ctx->addfile($fh);
```

What you get back

The same context object, for chaining. `addfile` reads the handle in 4 KiB blocks and feeds them into MD5 without loading the whole file into memory, so it handles files of any size. Once the call returns, the handle is positioned at EOF; the caller is responsible for closing it.

Put the handle in `binmode` (or open with `:raw`) before calling - MD5 hashes bytes, and a text-mode handle on Windows or one with a decoding layer will change what bytes reach the digest.

Examples

Digest a file in one chain:

```
open(my $fh, '<:raw', "/etc/passwd") or die $!;
print Digest::MD5->new->addfile($fh)->hexdigest, "\n";
close $fh;
```

Combine file contents with extra data:

```
my $ctx = Digest::MD5->new;
$ctx->add($salt);
$ctx->addfile($fh);
$ctx->add($pepper);
my $hex = $ctx->hexdigest;
```

Skip the object entirely for the one-file common case:

```
# Same result, shorter:
my $hex = Digest::MD5->new->addfile($fh)->hexdigest;
```

Edge cases

- The handle is resolved as perl resolves any filehandle argument: a glob, a glob or IO reference, or a bareword name such as `"STDIN"`. A name that is no handle croaks `Bad filehandle: NAME`, `undef` croaks `Can't use an undefined value as filehandle reference`, and an unopened handle croaks `No filehandle passed`.
- Croaks with `"Reading from filehandle failed"` if `read` returns an I/O error; the context's state after such a failure is unspecified - discard or `reset` it.
- Reading an empty handle (EOF immediately) is a no-op and returns the context unchanged.
- Handles decoded as UTF-8 still work, but you are hashing the raw *decoded* bytes the PerlIO layer returns, which is rarely what you want. Prefer `binmode`.

Differences from upstream

Fully compatible with upstream `Digest::MD5` 2.59.

See also

- `add` - feed in-memory data instead of a file
- `add_bits` - bit-oriented input for cross-digest compatibility
- `new` - typical predecessor when digesting one file
- `hexdigest` - typical successor for a single-file chain

digest

Finalise the context and return the 16-byte raw binary digest.

Synopsis

```
my $bin = $ctx->digest;        # exactly 16 bytes
```

What you get back

A 16-byte Perl string holding the raw digest. This is the most compact representation - handy for binary protocols or packing into fixed-width records - but it typically contains non-printable bytes, so do not paste it into log lines without encoding.

`digest` is destructive. After it returns, the context is automatically reset to the empty state and is immediately ready to digest a new message. To read the digest without destroying the state, use `$ctx->clone->digest` instead.

Examples

Raw bytes, then convert to hex by hand:

```
my $bin = Digest::MD5->new->add("abc")->digest;
printf "%v02x\n", $bin;   # 90.01.50.98.3c.d2.4f.b0.d6.96.3f.7d.28.
 ↪e1.7f.72
```

Pack into a fixed-width binary record:

```
my $record = pack("A16 a16", $filename, $ctx->digest);
```

Peek at an intermediate digest without resetting:

```
my $snapshot = $ctx->clone->digest;   # $ctx keeps going
$ctx->add($more);
```

Edge cases

- Called on a context with no data added, returns the MD5 of the empty string (`d41d8cd98f00b204e9800998ecf8427e` as hex).
- Resets *before* returning, so calling `digest` twice in a row without adding data between the calls gives the empty-string digest the second time.

Differences from upstream

Fully compatible with upstream `Digest::MD5` 2.59.

See also

- `hexdigest` - same digest, formatted as 32 hex characters
- `b64digest` - same digest, base64-encoded
- `md5` - one-shot functional form
- `clone` - take a snapshot digest without resetting the context

hexdigest

Finalise the context and return the digest as 32 lowercase hex chars.

Synopsis

```
my $hex = $ctx->hexdigest;    # 32 chars, [0-9a-f]
```

What you get back

A 32-character ASCII string containing only `0-9` and `a-f`. This is the form you want for ETags, `md5sum`-style checksum files, log output, and anywhere humans will read the digest. No `0x` prefix, no separators, lowercase.

`hexdigest` is destructive: the context is reset to empty after the digest is produced. Clone first if you need to keep going.

Examples

Common one-liner:

```
my $hex = Digest::MD5->new->add("hello")->hexdigest;
# 5d41402abc4b2a76b9719d911017c592
```

Digest the same context twice by cloning:

```
$ctx->add("stream so far");
my $checkpoint = $ctx->clone->hexdigest;
$ctx->add("more data");
my $final      = $ctx->hexdigest;
```

Build an `md5sum`-compatible line:

```
open(my $fh, '<:raw', $path) or die $!;
printf "%s  %s\n", Digest::MD5->new->addfile($fh)->hexdigest, $path;
```

Edge cases

- Empty context gives `d41d8cd98f00b204e9800998ecf8427e` (the MD5 of the empty string).
- The result is always lowercase; if an external format demands uppercase, call `uc` on it.
- Resets the context, so `$ctx->hexdigest` twice without intervening `add` calls gives the empty-string digest the second time.

Differences from upstream

Fully compatible with upstream `Digest::MD5` 2.59.

See also

- `digest` - same content as 16 raw bytes
- `b64digest` - shorter base64 encoding of the same digest
- `md5_hex` - one-shot functional form
- `clone` - preserve the context before finalising

b64digest

Finalise the context and return the digest as a 22-char base64 string.

Synopsis

```
my $b64 = $ctx->b64digest;    # 22 chars, no padding
```

What you get back

A 22-character string using the standard base64 alphabet `A-Z a-z 0-9 + /`. The result is **not** padded with `=`; the full base64 encoding of 16 bytes would be 24 characters, but the tail is fixed so the padding is redundant. Append `"=="` yourself if an interop partner strictly requires multiples of 4.

Base64 gives you a compact, one-line-friendly identifier that is roughly 30% shorter than hex while still printable. Common in cache keys, URL query parameters, and condensed log formats.

`b64digest` is destructive: the context is reset after use.

Examples

Compact digest for a cache key:

```
my $key = Digest::MD5->new->add($url)->b64digest;
```

Pad to a multiple of 4 for strict base64 consumers:

```
my $padded = Digest::MD5->new->add($data)->b64digest . "==";
```

Peek without consuming:

```
my $snapshot = $ctx->clone->b64digest;
```

Edge cases

- Empty context gives `1B2M2Y8AsgTpgAmY7PhCfg` (the MD5 of the empty string in base64).
- Contains `/` and `+`, so the result is **not** safe to drop into a URL path without further encoding - use URL-safe base64 for that (substitute `-` for `+` and `_` for `/`).
- Resets the context after returning.

Differences from upstream

Fully compatible with upstream `Digest::MD5` 2.59.

See also

- `digest` - raw bytes
- `hexdigest` - hex-encoded digest (longer but URL-safe)
- `md5_base64` - one-shot functional form
- `clone` - preserve the context before finalising

context

Save or restore the raw internal MD5 state - for advanced use only.

Synopsis

```
my @state = $ctx->context;                          # get
$ctx->context($blocks, $state_bytes, $partial);  # restore
```

What you get back

Called with no extra arguments, returns a list that fully captures the current digest state:

1. `$blocks` - number of complete 64-byte blocks processed so far, as an unsigned integer.

2. `$state_bytes` - 16 raw bytes holding the four MD5 state words in little-endian order.
3. `$partial` - present only when some bytes have been added that do not yet fill a full block. Up to 63 bytes of unprocessed data.

Called with those same three values, restores the context to the captured state and returns the object itself. This is the only way to persist a digest across process boundaries - serialise the three values, hand them off, then restore on the other side and continue adding.

For almost every use case, `clone` is a cleaner tool. Reach for `context` only when the state needs to survive something `clone` cannot cross, such as a database round-trip or an out-of-band storage step.

Examples

Round-trip a partial digest through storage:

```
my $ctx = Digest::MD5->new;
$ctx->add($first_half);
my @saved = $ctx->context;            # freeze state

# ... later, possibly in another process ...

my $ctx = Digest::MD5->new;
$ctx->context(@saved);                # thaw state
$ctx->add($second_half);
print $ctx->hexdigest;                # same as hashing whole thing
```

Serialise for storage:

```
use Storable qw(freeze thaw);
my $blob = freeze([$ctx->context]);   # three values in, one blob␣
↪out
# ...
my $ctx = Digest::MD5->new;
$ctx->context(@{ thaw($blob) });
```

Edge cases

- The third (`$partial`) value is only returned when `length($partial) > 0`. Be prepared for a two-element list when the input fits exactly into whole blocks.
- Passing a `$state_bytes` argument shorter than 16 bytes silently leaves the state words untouched. Do not rely on this - pass exactly what `context` gave you.
- `context` is the only method whose interface is genuinely raw. Its output is sensitive to the algorithm, not to a documented wire format; do not pretend values produced by one digest class are interchangeable with another.

Differences from upstream

Fully compatible with upstream `Digest::MD5` 2.59.

See also

- `clone` - the usual way to snapshot state in-process
- `digest` - finalise rather than persist
- `new` - fresh context for use with `context` as a restore target
- `add` - continue hashing after a restore

md5

One-shot MD5 digest, returned as 16 raw bytes.

Synopsis

```
use Digest::MD5 qw(md5);
my $bin = md5($data);               # 16 bytes
my $bin = md5($a, $b, $c);          # md5($a.$b.$c)
```

What you get back

A 16-byte Perl string holding the raw MD5 digest. All arguments are concatenated (in order) and the result is hashed, so `md5("a", "b", "c")` is the same as `md5("abc")`. The function does not allocate an object; it runs through a stack-local context, which makes it the cheapest way to hash something you already hold in memory.

Use `md5_hex` or `md5_base64` when you need a printable form.

Examples

Digest a string to raw bytes:

```
use Digest::MD5 qw(md5);
my $bin = md5("hello world");       # 16 bytes
printf "%v02x\n", $bin;
# 5e.b6.3b.bb.e0.1e.ee.d0.93.cb.22.bb.8f.5a.cd.c3
```

Concatenation is implicit:

```
my $a = md5("foo", "bar");
my $b = md5("foobar");
# $a eq $b
```

Pack into a binary record:

```
my $blob = pack("N a16", length($payload), md5($payload));
```

Edge cases

- `md5()` with no arguments returns the MD5 of the empty string (`d41d8cd98f00b204e9800998ecf8427e` in hex).
- Wide-character arguments (code points above `U+00FF`) croak with `Wide character in subroutine entry`. Encode first: `md5(encode_utf8($str))`.
- Called as a class or instance method (e.g. `Digest::MD5->md5(...)`), it warns about probable misuse but still returns a digest based on all arguments, including the invocant. Prefer the plain function call.

Differences from upstream

Fully compatible with upstream `Digest::MD5` 2.59.

See also

- `md5_hex` - same digest, as 32 lowercase hex characters
- `md5_base64` - same digest, base64-encoded
- `digest` - OO form; reuse one context across many inputs
- `addfile` - for data too large to hold in memory at once

md5_hex

One-shot MD5 digest, returned as a 32-character lowercase hex string.

Synopsis

```
use Digest::MD5 qw(md5_hex);
my $hex = md5_hex($data);              # 32 chars
my $hex = md5_hex($a, $b, $c);         # md5_hex($a.$b.$c)
```

What you get back

A 32-character ASCII string containing only `0-9` and `a-f`. This is the encoding used by `md5sum`, HTTP ETag headers, and virtually every human-readable checksum format. All arguments are concatenated before hashing.

Examples

The canonical one-liner:

```
use Digest::MD5 qw(md5_hex);
print md5_hex("foobarbaz"), "\n";
# 6df23dc03f9b54cc38a0fc1483df6e21
```

Compose a cache key from structured data:

```
my $key = md5_hex($user_id, "|", $query, "|", $locale);
```

Digest bytes from a multi-byte string via UTF-8 encoding:

```
use Encode qw(encode_utf8);
my $hex = md5_hex(encode_utf8("abc\x{300}"));
# 8c2d46911f3f5a326455f0ed7a8ed3b3
```

Edge cases

- `md5_hex()` with no arguments returns `d41d8cd98f00b204e9800998ecf8427e`.
- Wide-character arguments croak; see `md5` for the fix.
- Result is always lowercase. Call `uc` if a tool requires uppercase hex.

Differences from upstream

Fully compatible with upstream `Digest::MD5` 2.59.

See also

- `md5` - same digest as raw bytes
- `md5_base64` - same digest, shorter base64 form
- `hexdigest` - OO form over a reusable context
- `add` - feed data incrementally when you cannot fit it in memory

md5_base64

One-shot MD5 digest, returned as a 22-character base64 string.

Synopsis

```
use Digest::MD5 qw(md5_base64);
my $b64 = md5_base64($data);       # 22 chars, no padding
```

What you get back

A 22-character string from the standard base64 alphabet `A-Z a-z 0-9 + /`. The tail is **not** padded to a multiple of 4 - the redundancy would encode nothing - so if an interop partner demands strict base64, append `"=="` yourself.

Base64 is about 30% shorter than hex and still fits on one line in a log or URL query parameter. The price is that it contains `/` and `+`, which are not safe in URL paths without further escaping.

Examples

Compact digest for a cache key or tag:

```
use Digest::MD5 qw(md5_base64);
my $tag = md5_base64($payload);
# 22-character identifier, e.g. "rL0Y20zC+Fzt72VPzMSk2A"
```

Pad for strict base64 parsers:

```
my $padded = md5_base64($payload) . "==";
```

Store alongside the content in a manifest:

```
printf "%s  %s\n", md5_base64($bytes), $name;
```

Edge cases

- `md5_base64()` with no arguments returns `1B2M2Y8AsgTpgAmY7PhCfg`.
- Wide-character arguments croak; encode first.
- The `+` and `/` characters require percent-encoding if the result appears in a URL path. Consider a URL-safe base64 transform (`tr|+/|-_|`) when serving digests in URLs.

Differences from upstream

Fully compatible with upstream `Digest::MD5` 2.59.

See also

- `md5` - raw-byte form
- `md5_hex` - hex form (longer, URL-safer)
- `b64digest` - OO form over a reusable context
- `add` - feed data incrementally for larger inputs

reset

Clear an existing context so it behaves like a freshly created one.

Synopsis

```
$ctx->reset;                    # discard accumulated data
$ctx->reset->add($data);        # reset and keep going in one line
```

What you get back

The same object, returned so calls chain. After `reset`, the internal state is identical to `Digest::MD5->new`: accumulated length is zero, the four state words hold the MD5 initial vector, and the partial-block buffer is empty. No new allocation happens - the existing `MD5_CTX` is overwritten in place.

`reset` is an alias for the instance-method form of `new`, exposed as a separate method for readability. Use whichever reads better at the call site.

Examples

Reuse a context across independent digests:

```
my $ctx = Digest::MD5->new;
my $h1  = $ctx->add("first")->hexdigest;    # digest + implicit
 ↪reset
my $h2  = $ctx->add("second")->hexdigest;   # starts fresh
```

Explicit reset after a partial computation you want to throw away:

```
$ctx->add("oops, wrong data");
$ctx->reset;
$ctx->add($correct_data);
print $ctx->hexdigest;
```

Edge cases

- Calling `reset` on a fresh context is a harmless no-op.
- Reset does not change the class of the object, so subclasses remain blessed into the subclass after a reset.

Differences from upstream

Fully compatible with upstream `Digest::MD5` 2.59.

See also

- `new` - same effect when called as an instance method
- `digest` - implicitly resets after returning the digest
- `add` - the usual next step after `reset`

add_bits

Add a bit-oriented quantum of data - but only in multiples of 8 bits.

Synopsis

```
$ctx->add_bits($bitstring);         # "01101000..." form
$ctx->add_bits($data, $nbits);      # byte-string + bit count
```

What you get back

The same context object, for chaining. MD5 is a byte-oriented algorithm, so `add_bits` exists purely for interface compatibility with other `Digest::*` modules that *do* hash at bit granularity. It accepts exactly the same two calling forms they do, but requires the bit count to be a whole multiple of 8 and croaks otherwise.

Unless you are writing code that must work across multiple digest backends, use `add` directly - it is clearer and has no restrictions.

Examples

Bit-string form - a string of `'0'` and `'1'` characters:

```
$ctx->add_bits("0110100001101001");   # feeds bytes "hi"
```

Byte-string + bit count - takes the first `$nbits / 8` bytes:

```
$ctx->add_bits("hello world", 40);    # feeds "hello"
```

Chain alongside `add`:

```
my $hex = Digest::MD5->new
                     ->add_bits("0110100001101001")   # "hi"
                     ->add(" there")
                     ->hexdigest;
```

Edge cases

- Croaks with `"Number of bits must be multiple of 8 for this algorithm"` when the bit string's length, or the explicit `$nbits` argument, is not a multiple of 8.
- In the two-argument form, `$nbits` larger than `8 * length($data)` feeds all of `$data`.

Differences from upstream

Fully compatible with upstream `Digest::MD5` 2.59.

See also

- `add` - the direct byte-level input method
- `addfile` - stream bytes from a file handle
- `new` - typical predecessor in a chain

- `hexdigest` - typical successor in a chain

Digest::SHA

Compute SHA-1 and SHA-2 family message digests: SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, and SHA-512/256, with optional HMAC keying.

Two call styles are available. The **functional interface** (`sha256_hex`, `hmac_sha512_base64`, and friends) is a one-shot: hand it the data, get the digest back. The **object interface** (`Digest::SHA->new(256)`, `->add`, `->digest`) lets you feed data incrementally - from streams, files, or a mix of sources - and finalize when you are ready.

Every digest can be rendered in three encodings: raw binary (fixed-length bytes), lowercase hex, or unpadded Base64. Choose the encoding by the function suffix (`_hex`, `_base64`, or none for raw). HMAC variants use the same suffixes and live under the `hmac_*` prefix.

File input is supported through `addfile`, which reads a filehandle or filename and feeds its contents straight into the context. A `bits` helper (`add_bits`) lets you hash arbitrary-length bit strings, not just whole bytes. Context state is serialisable with `getstate` / `putstate` (text) and `_getstate` / `_putstate` (packed binary), so a long-running hash can be suspended and resumed across processes.

Pick **SHA-256** by default for new work: it is fast on 32-bit and 64-bit CPUs alike and has the widest interoperability. Prefer **SHA-512** (or its truncated `SHA-512/256` cousin) on 64-bit hardware when you want more performance headroom or a larger internal state; SHA-512 often outperforms SHA-256 there despite the larger output. Avoid **SHA-1** for anything collision-sensitive - it survives only for legacy interoperability.

This is Digest::SHA 6.04; `Digest/SHA.pm` is upstream's.

Synopsis

```
use Digest::SHA qw(sha256_hex sha1_base64 hmac_sha256_hex);

print sha256_hex('abc'), "\n";                  # ba7816bf8f01...
↪f20015ad
print sha1_base64('abc'), "\n";                 #␣
↪qZk+NkcGgWq6PiVxeFDCbJzQ2J0
print hmac_sha256_hex('data', 'key'), "\n";     # 5031fe3d989c...
↪ce221bd0

my $sha = Digest::SHA->new(256);
$sha->add('a', 'b');
$sha->add('c');
print $sha->hexdigest, "\n";                    # the same as␣
↪sha256_hex('abc')

open my $fh, '<', $file or die;
print Digest::SHA->new(512)->addfile($fh)->b64digest, "\n";
```

The functional digests

One function per algorithm and encoding, each taking any number of strings, hashed as their concatenation:

algorithm	raw bytes	hex	base64 (unpadded)
SHA-1	`sha1`	`sha1_hex`	`sha1_base64`
SHA-224	`sha224`	`sha224_hex`	`sha224_base64`
SHA-256	`sha256`	`sha256_hex`	`sha256_base64`
SHA-384	`sha384`	`sha384_hex`	`sha384_base64`
SHA-512	`sha512`	`sha512_hex`	`sha512_base64`
SHA-512/224	`sha512224`	`sha512224_hex`	`sha512224_base64`
SHA-512/256	`sha512256`	`sha512256_hex`	`sha512256_base64`

`hmac_` before each name (`hmac_sha256_hex($data, $key)`) gives the keyed HMAC; the key is the last argument. Strings must be bytes: a character above 0xFF dies "Wide character".

Differences from upstream

Two, each where perl 5.44 with SHA.xs fails (t/81-xs-native/Digest-SHA/495-or-better.t):

- The low-level entry points Digest/SHA.pm builds on (`shainit`, `shawrite`, `sharewind`), given an object that is not a Digest::SHA, refuse it: `shainit` and `shawrite` return 0. perl5 crashes.
- `shawrite($bits, $count, $sha)` with a count past the end of `$bits` writes the bits the string has and returns their number; perl5 reads past the end of the string. (`add_bits`, the documented way in, never passes such a count.)

Functions

Functional digests

sha1

Compute a SHA digest of the concatenated arguments in one call.

Also available as `Digest::SHA::sha1_base64`, `Digest::SHA::sha1_hex`, `Digest::SHA::sha224`, `Digest::SHA::sha224_base64`, `Digest::SHA::sha224_hex`, `Digest::SHA::sha256`, `Digest::SHA::sha256_base64`, `Digest::SHA::sha256_hex`, `Digest::SHA::sha384`, `Digest::SHA::sha384_base64`, `Digest::SHA::sha384_hex`, `Digest::SHA::sha512`, `Digest::SHA::sha512224`, `Digest::SHA::sha512224_base64`, `Digest::SHA::sha512224_hex`, `Digest::SHA::sha512256`, `Digest::SHA::sha512256_base64`, `Digest::SHA::sha512256_hex`, `Digest::SHA::sha512_base64`, `Digest::SHA::sha512_hex`.

OO lifecycle

shainit

(Re)initialise a `Digest::SHA` object for an algorithm; the low-level entry point behind `$sha->new($alg)` on an existing object.

sharewind

Rewind a `Digest::SHA` object to the empty message of its algorithm; the low-level entry point behind `$sha->reset`.

newSHA

Construct a `Digest::SHA` object of class `$classname` for an algorithm number: 1, 224, 256, 384, 512, 512224 or 512256. Any other number returns `undef`.

clone

Copy a `Digest::SHA` object, current state included, into a new object of the same class.

Synopsis

```
my $partial = $sha->clone;
print $partial->hexdigest;   # digest so far; $sha keeps going
```

new

Construct or reinitialise a `Digest::SHA` object.

reset

Rewind a `Digest::SHA` object, optionally switching algorithm.

OO input

shawrite

Feed the first `$bitcnt` bits of `$bitstr` into a `Digest::SHA` object; the low-level entry point behind `add_bits`.

add

Append data to the message being hashed.

add_bits

Feed an arbitrary number of bits into a `Digest::SHA` object.

addfile

Feed a filehandle or a file by name into a `Digest::SHA` object.

OO output

digest

Finish the message and return its digest as raw bytes, then rewind the object to the empty message.

hexdigest

Finish the message and return its digest as lowercase hex, then rewind the object to the empty message.

b64digest

Finish the message and return its digest as unpadded Base64, then rewind the object to the empty message.

algorithm

Return the algorithm code of a `Digest::SHA` object.

hashsize

Return the digest length of a `Digest::SHA` object in bits.

HMAC

hmac_sha1

Compute a keyed HMAC digest of the data arguments in one call.

Also available as `Digest::SHA::hmac_sha1_base64`, `Digest::SHA::hmac_sha1_hex`, `Digest::SHA::hmac_sha224`, `Digest::SHA::hmac_sha224_base64`, `Digest::SHA::hmac_sha224_hex`, `Digest::SHA::hmac_sha256`, `Digest::SHA::hmac_sha256_base64`, `Digest::SHA::hmac_sha256_hex`, `Digest::SHA::hmac_sha384`, `Digest::SHA::hmac_sha384_base64`, `Digest::SHA::hmac_sha384_hex`, `Digest::SHA::hmac_sha512`, `Digest::SHA::hmac_sha512224`, `Digest::SHA::hmac_sha512224_base64`, `Digest::SHA::hmac_sha512224_hex`, `Digest::SHA::hmac_sha512256`, `Digest::SHA::hmac_sha512256_base64`, `Digest::SHA::hmac_sha512256_hex`, `Digest::SHA::hmac_sha512_base64`, `Digest::SHA::hmac_sha512_hex`.

Other Functions

getstate

Export a `Digest::SHA` object's state as a human-readable text string.

putstate

Construct or restore a `Digest::SHA` object from a text state string.

dump

Write the current `Digest::SHA` state to a file or STDOUT in text form.

load

Construct a `Digest::SHA` object from a text state file written by `dump`.

sha1

Compute a SHA digest of the concatenated arguments in one call.

Also available as `Digest::SHA::sha1_base64`, `Digest::SHA::sha1_hex`, `Digest::SHA::sha224`, `Digest::SHA::sha224_base64`, `Digest::SHA::sha224_hex`, `Digest::SHA::sha256`, `Digest::SHA::sha256_base64`, `Digest::SHA::sha256_hex`, `Digest::SHA::sha384`, `Digest::SHA::sha384_base64`, `Digest::SHA::sha384_hex`, `Digest::SHA::sha512`, `Digest::SHA::sha512224`, `Digest::SHA::sha512224_base64`, `Digest::SHA::sha512224_hex`, `Digest::SHA::sha512256`, `Digest::SHA::sha512256_base64`, `Digest::SHA::sha512256_hex`, `Digest::SHA::sha512_base64`, `Digest::SHA::sha512_hex`.

There are 21 functional entry points: 7 algorithms crossed with 3 output encodings. The name you call selects both the algorithm and the encoding: the plain name returns the **raw binary** digest (a fixed-length byte string), `_hex` the **lowercase hex** digest, `_base64` the **unpadded Base64** digest.

Algorithm	Raw binary	Hex	Base64
SHA-1	`sha1`	`sha1_hex`	`sha1_base64`
SHA-224	`sha224`	`sha224_hex`	`sha224_base64`
SHA-256	`sha256`	`sha256_hex`	`sha256_base64`
SHA-384	`sha384`	`sha384_hex`	`sha384_base64`
SHA-512	`sha512`	`sha512_hex`	`sha512_base64`
SHA-512/224	`sha512224`	`sha512224_hex`	`sha512224_base64`
SHA-512/256	`sha512256`	`sha512256_hex`	`sha512256_base64`

All arguments are concatenated byte-wise before hashing, so `sha256_hex("foo", "bar")` is identical to `sha256_hex("foobar")`.

Synopsis

```
use Digest::SHA qw(sha256_hex sha512_base64);
my $hex = sha256_hex($data);
my $b64 = sha512_base64(@chunks);
```

What you get back

A plain scalar. For the raw-binary variants it is a byte string of fixed length (20 bytes for SHA-1, 28 for SHA-224, 32 for SHA-256, 48 for SHA-384, 64 for SHA-512). For _hex variants it is twice that length. For _base64 variants it is the unpadded Base64 encoding (no trailing = characters).

Examples

```
use Digest::SHA qw(sha256_hex);
print sha256_hex("abc");
# e.g.␣
↪ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad
```

```
use Digest::SHA qw(sha1_hex);
my $digest = sha1_hex("The quick brown fox ", "jumps over the lazy␣
↪dog");
# identical to sha1_hex("The quick brown fox jumps over the lazy dog
↪")
```

```
use Digest::SHA qw(sha512_base64);
my $tag = sha512_base64($payload);
# 86-character unpadded Base64 string
```

```
use Digest::SHA qw(sha256);
my $raw = sha256("");     # 32 raw bytes
print length $raw;        # 32
```

Edge cases

- No arguments: hashes the empty string. `sha256_hex()` returns the well-known `e3b0c442...` empty-input digest.
- Wide characters (codepoint above 255) croak with `Wide character in subroutine entry` - encode to UTF-8 first if you want to hash text above Latin-1.

Differences from upstream

Fully compatible with upstream `Digest::SHA` 6.04.

See also

- `hmac_sha256_hex` - keyed variant for message authentication.
- `Digest::SHA->new(256)` - incremental interface for large or streamed input.
- `sha256` - raw-binary counterpart of `sha256_hex`.

hmac_sha1

Compute a keyed HMAC digest of the data arguments in one call.

Also available as `Digest::SHA::hmac_sha1_base64`, `Digest::SHA::hmac_sha1_hex`, `Digest::SHA::hmac_sha224`, `Digest::SHA::hmac_sha224_base64`, `Digest::SHA::hmac_sha224_hex`, `Digest::SHA::hmac_sha256`, `Digest::SHA::hmac_sha256_base64`, `Digest::SHA::hmac_sha256_hex`, `Digest::SHA::hmac_sha384`, `Digest::SHA::hmac_sha384_base64`, `Digest::SHA::hmac_sha384_hex`, `Digest::SHA::hmac_sha512`, `Digest::SHA::hmac_sha512224`, `Digest::SHA::hmac_sha512224_base64`, `Digest::SHA::hmac_sha512224_hex`, `Digest::SHA::hmac_sha512256`, `Digest::SHA::hmac_sha512256_base64`, `Digest::SHA::hmac_sha512256_hex`, `Digest::SHA::hmac_sha512_base64`, `Digest::SHA::hmac_sha512_hex`.

There are 21 `hmac_sha*` entry points: 7 algorithms crossed with 3 output encodings. The name you call selects both the underlying SHA algorithm and the encoding: the plain name returns the **raw binary** MAC, `_hex` the **lowercase hex** MAC, `_base64` the **unpadded Base64** MAC.

Algorithm	Raw binary	Hex	Base64
SHA-1	`hmac_sha1`	`hmac_sha1_hex`	`hmac_sha1_base64`
SHA-224	`hmac_sha224`	`hmac_sha224_hex`	`hmac_sha224_base64`
SHA-256	`hmac_sha256`	`hmac_sha256_hex`	`hmac_sha256_base64`
SHA-384	`hmac_sha384`	`hmac_sha384_hex`	`hmac_sha384_base64`
SHA-512	`hmac_sha512`	`hmac_sha512_hex`	`hmac_sha512_base64`
SHA-512/224	`hmac_sha512224`	`hmac_sha512224_hex`	`hmac_sha512224_base64`
SHA-512/256	`hmac_sha512256`	`hmac_sha512256_hex`	`hmac_sha512256_base64`

The **last** argument is the key, every preceding argument is message data (concatenated byte-wise). With a single argument the key is that argument and the message is empty. With no arguments both key and message are empty.

Synopsis

```
use Digest::SHA qw(hmac_sha256_hex hmac_sha1_base64);
my $tag = hmac_sha256_hex($message, $key);
my $b64 = hmac_sha1_base64(@chunks, $key);
```

What you get back

A plain scalar. The raw-binary form has the fixed length of the chosen algorithm's digest (20 bytes for SHA-1, 32 for SHA-256, 64 for SHA-512, and so on). The `_hex` form is twice that length; the `_base64` form is the unpadded Base64 encoding.

Examples

```
use Digest::SHA qw(hmac_sha256_hex);
my $mac = hmac_sha256_hex("Hi There", "\x0b" x 20);
# b0344c61d8db38535ca8afceaf0bf12b881dc200c9833da726e9376c2e32cff7
```

```
use Digest::SHA qw(hmac_sha1_hex);
# Concatenated message: "Hello, " + "world!"
my $mac = hmac_sha1_hex("Hello, ", "world!", $shared_secret);
```

```
use Digest::SHA qw(hmac_sha512_base64);
my $cookie_tag = hmac_sha512_base64($cookie_body, $server_secret);
```

```
use Digest::SHA qw(hmac_sha256_hex);
my $tag = hmac_sha256_hex($key);   # one arg: key only, empty␣
↪message
```

Edge cases

- Single-argument calls treat the sole argument as the key and hash an empty message. That is rarely what you want - pass message and key explicitly.
- Keys longer than the algorithm's block size are hashed down to digest size before use (standard HMAC key reduction).
- Wide characters in either key or data croak with `Wide character in subroutine entry`.

Differences from upstream

Fully compatible with upstream `Digest::SHA` 6.04.

See also

- `sha256_hex` - unkeyed one-shot digest for integrity-only needs.
- There is no OO HMAC: a `Digest::SHA` object has no `hmac_*` methods. If you need incremental HMAC use `Digest::HMAC_SHA1` or feed a keyed construction manually.
- `hmac_sha1_hex` - SHA-1 variant; prefer SHA-256 or larger for new code.

newSHA

Construct a `Digest::SHA` object of class `$classname` for an algorithm number: 1, 224, 256, 384, 512, 512224 or 512256. Any other number returns `undef`.

`Digest::SHA->new` is the usual way in; it strips a name such as `"SHA-256"` down to its digits and calls `$class->newSHA($alg)`.

Synopsis

```
my $sha = Digest::SHA->newSHA(256);
my $sub = Digest::SHA::newSHA("My::Digest", 1);   # blessed into
 ↪My::Digest
```

add

Append data to the message being hashed.

Every argument is taken as bytes (`SvPVbyte`); a string holding a character above U+00FF croaks `Wide character in subroutine entry`. Returns the object, so calls chain; `undef` when called on anything that is not a `Digest::SHA`.

Synopsis

```
$sha->add($chunk1, $chunk2)->add($chunk3);
print $sha->hexdigest;
```

algorithm

Return the algorithm code of a `Digest::SHA` object.

The code is the same integer you can pass to `new`: 1, 224, 256, 384, 512, 512224, or 512256. Useful when a function receives a `Digest::SHA` object from elsewhere and needs to act differently based on which variant it is.

Synopsis

```
my $code = $sha->algorithm;
```

What you get back

An integer in the set `{1, 224, 256, 384, 512, 512224, 512256}`.

Examples

```
my $sha = Digest::SHA->new("SHA-256");
print $sha->algorithm;   # 256
```

```
my $sha = Digest::SHA->new("SHA-512/224");
print $sha->algorithm;   # 512224
```

```
# Dispatch on algorithm:
my $code = $sha->algorithm;
my $name = $code == 1 ? "SHA-1" : "SHA-$code";
```

Edge cases

- Called on a non-`Digest::SHA` object, returns `undef`.

Differences from upstream

Fully compatible with upstream `Digest::SHA` 6.04.

See also

- `hashsize` - digest length in bits for this algorithm.
- `new` - accepts the values returned here as its algorithm argument.

hashsize

Return the digest length of a `Digest::SHA` object in bits.

For a given algorithm the hashsize is a constant: 160 for SHA-1, 224 for SHA-224, 256 for SHA-256 and SHA-512/256, 384 for SHA-384, 512 for SHA-512. Divide by 8 to get the byte length of the output from `digest`.

Synopsis

```
my $bits = $sha->hashsize;
```

What you get back

An integer, always a multiple of 8.

Examples

```
my $sha = Digest::SHA->new(256);
print $sha->hashsize;         # 256
print $sha->hashsize / 8;     # 32  (bytes per digest)
```

```
my $sha = Digest::SHA->new("SHA-512/224");
print $sha->hashsize;         # 224
```

```
# Preallocate output buffer for $sha->digest:
my $buf = "\0" x ($sha->hashsize / 8);
```

Edge cases

- Called on a non-`Digest::SHA` object, returns `undef`.

Differences from upstream

Fully compatible with upstream `Digest::SHA` 6.04.

See also

- `algorithm` - the integer algorithm code for this object.
- `digest` - output whose length is `hashsize / 8` bytes.

new

Construct or reinitialise a `Digest::SHA` object.

The primary object constructor. Works as both a class method (`Digest::SHA->new(256)`) and an instance method (`$sha->new(512)`), and accepts the algorithm in any of the shapes `Digest::SHA` recognises: an integer like 256, a string like `"SHA-256"`, `"sha256"`, or `"512/224"`. When called without an algorithm argument it defaults to SHA-1.

As an instance method, new reuses the existing object: supplying a new algorithm resets state and switches to that algorithm; omitting it simply rewinds the state (same as `reset`).

Synopsis

```
my $sha = Digest::SHA->new(256);
my $sha = Digest::SHA->new("SHA-512/256");
$sha->new;                      # rewind, keep algorithm
$sha->new(384);                 # rewind, switch to SHA-384
```

What you get back

A blessed `Digest::SHA` object, or `undef` when the algorithm argument cannot be parsed as a supported algorithm.

Examples

```
my $sha = Digest::SHA->new("sha256");
$sha->add("The quick brown fox ", "jumps over the lazy dog");
print $sha->hexdigest;
# d7a8fbb307d7809469ca9abcb0082e4f8d5651e46d3cdb762d02d0bf37c9e592
```

```
my $sha = Digest::SHA->new(512);
$sha->add_bits("01011010");       # hash exactly 8 bits
```

```
my $sha = Digest::SHA->new(256);
$sha->add("first");
$sha->new(512);                  # rewind and switch
$sha->add("second");             # now hashing with SHA-512
```

Edge cases

- No algorithm argument defaults to SHA-1.
- An unparseable algorithm returns undef; check the return value before chaining ->add.
- Called on an existing object, new mutates that object in place and returns it - it does not allocate a second context.

Differences from upstream

Fully compatible with upstream Digest::SHA 6.04.

See also

- reset - explicit rewind, same semantics as the no-argument instance form.
- clone - copy the current state into a fresh object.
- newSHA - lower-level constructor taking an integer algorithm code.

reset

Rewind a Digest::SHA object, optionally switching algorithm.

Clears buffered data and restores the initial hash state so the same object can be reused for a fresh digest. With no argument the algorithm is preserved; with an algorithm argument the object is rewound **and** retargeted at that algorithm. Accepts the same algorithm shapes as new (integer code or string like "SHA-256").

Synopsis

```
$sha->reset;              # rewind, keep algorithm
$sha->reset(512);         # rewind and switch to SHA-512
$sha->reset("SHA-384");   # string form
```

What you get back

The receiver on success (chainable), or undef if an algorithm argument was supplied that could not be parsed.

Examples

```
my $sha = Digest::SHA->new(256);
$sha->add("first");
my $a = $sha->hexdigest;       # finalises and auto-resets already
$sha->reset;                   # defensive rewind before reuse
$sha->add("second");
my $b = $sha->hexdigest;
```

```
my $sha = Digest::SHA->new(256);
$sha->reset(512)->add("hello");
print $sha->algorithm;         # 512
```

```
my $sha = Digest::SHA->new(256);
my $ok = $sha->reset("bogus"); # undef
```

Edge cases

- Called without an algorithm on a reference that is not a `Digest::SHA` object, returns that reference unchanged.
- After the built-in auto-reset inside `digest`, calling `reset` explicitly is harmless.

Differences from upstream

- Called without an algorithm on a reference that is not a `Digest::SHA` object, upstream crashes with a segmentation fault (its XS `sharewind` is handed a null context); pperl leaves the reference alone and returns it.

See also

- `new` - when called on an instance, functionally equivalent to `reset`.
- `digest` - auto-rewinds after finalisation, so explicit `reset` is only needed when you abandon a computation partway.
- `clone` - if you want a copy of the pre-reset state, clone first.

add_bits

Feed an arbitrary number of bits into a `Digest::SHA` object.

Unlike add, this method is not restricted to whole bytes. It accepts two call forms:

- **One-argument form** `add_bits($bitstr)` hashes a string of `'0'` and `'1'` characters - the hash takes one bit per character, so `"01011010"` adds exactly 8 bits of input.
- **Two-argument form** `add_bits($data, $numbits)` hashes the first `$numbits` bits of the packed binary string `$data`. If `$numbits` exceeds `8 * length($data)` it is capped at that value.

Most of the SHA standard is defined on bit-strings rather than byte-strings; this is the entry point for the bit-exact test vectors in FIPS 180-4.

Synopsis

```
$sha->add_bits("11001010");          # 8 bits from ASCII 0/1 string
$sha->add_bits($packed, 13);         # first 13 bits of $packed
```

What you get back

The receiver, for chaining.

Examples

```
my $sha = Digest::SHA->new(256);
$sha->add_bits("01011010" x 8);   # 64 bits
print $sha->hexdigest;
```

```
# FIPS 180-4 1-bit test vector for SHA-1
my $sha = Digest::SHA->new(1);
$sha->add_bits("0");
print $sha->hexdigest;
# bb6b3e18f0115b57925241676f5b1ae88747b08a
```

```
my $sha = Digest::SHA->new(256);
$sha->add_bits("\xab\xcd", 12);  # first 12 bits of 0xabcd
```

Edge cases

- The one-argument form packs the string with `pack("B*", ...)`, so every character counts as one bit: a character other than `'0'` or `'1'` is not skipped but read by its lowest bit. An empty string is a no-op.
- In the two-argument form, a zero bit count leaves the hash state untouched, and a negative one hashes all of `$data`.
- Called on a reference that is not a `Digest::SHA` object, returns that reference unchanged.

Differences from upstream

- A negative bit count in the two-argument form hashes all of `$data`; upstream hands it to XS as a huge unsigned count, reads past the end of the string and crashes.
- Called on a reference that is not a `Digest::SHA` object, upstream crashes with a segmentation fault (its XS `shawrite` is handed a null context); pperl leaves the reference alone and returns it.

See also

- `add` - standard byte-aligned input path; use it unless you truly have sub-byte data.
- `digest` - finalises the running hash once input is complete.
- `addfile` - byte-oriented file ingestion.

addfile

Feed a filehandle or a file by name into a `Digest::SHA` object.

The public file-ingestion entry point. Accepts either an open filehandle or a filename string, plus an optional mode character that controls how bytes are read:

- `"b"` (default): raw binary read. Use this for anything that is not human-readable text, or when you need bit-exact file digests.
- `"U"`: universal newlines. When the file looks like text (`-T`), CR/LF and lone CR line endings are read as LF, so a text file gives the same digest whatever its line endings; any other file is read as in `"b"`.
- `"0"`: BITS mode. Interpret every `'0'` and `'1'` character in the file as a single bit and feed those bits into the hash. Anything else in the file is ignored.

Synopsis

```
$sha->addfile("data.bin");                  # filename, binary
$sha->addfile($fh);                         # handle, binary
$sha->addfile("notes.txt", "U");            # universal newlines
$sha->addfile("bits.txt", "0");             # ASCII bit stream
```

What you get back

The receiver, for chaining.

Examples

```
my $sha = Digest::SHA->new(256);
$sha->addfile("archive.tar.gz");
print $sha->hexdigest;
```

```
# Cross-platform digest of a source file
my $sha = Digest::SHA->new(256);
$sha->addfile("src/main.c", "U");
```

```
# BITS mode: hash the bit pattern "01011010" repeated
my $sha = Digest::SHA->new(1);
$sha->addfile("bitstream.txt", "0");
```

```
my $sha = Digest::SHA->new(256);
open my $fh, "<:raw", "photo.jpg" or die $!;
$sha->addfile($fh);
```

Edge cases

- A missing or unreadable filename croaks with `Open failed: ...`; a read error, such as on a closed filehandle, croaks with `Read failed: ...`.
- The mode applies only to a filename; a filehandle is read as it is, whatever mode is given.
- Unknown mode characters fall back to binary.
- Empty files leave the hash state untouched.
- Called on something that is not a `Digest::SHA` object, dies with `Can't locate object method` or `Can't call method`.

Differences from upstream

Fully compatible with upstream `Digest::SHA` 6.04.

See also

- `add` - use when the data is already in a Perl scalar.
- `add_bits` - for bit-aligned input strings.
- `digest` - finalise and return the digest after the file has been consumed.

getstate

Export a `Digest::SHA` object's state as a human-readable text string.

Produces a multi-line plain-text dump of the current hashing state, including algorithm number, hash registers, pending block, and message length counters. The text form is portable across machines of different endianness and lends itself to inspection, diffing, and manual editing.

Pair with `putstate` to reconstruct the state later.

Synopsis

```
my $text = $sha->getstate;
```

What you get back

A newline-terminated ASCII string.

Examples

```
my $sha = Digest::SHA->new(256);
$sha->add("hello");
print $sha->getstate;
# alg:256
# H:...
# block:...
# blockcnt:40
# lenhh:0
# lenhl:0
# lenlh:0
# lenll:40
```

```
# Resume on another machine:
my $text = $sha->getstate;
...
my $resumed = Digest::SHA->putstate($text);
$resumed->add($more_data);
```

Edge cases

- The text format is stable across `Digest::SHA` versions and endianness.
- Called on something that is not a `Digest::SHA` object, dies with `Can't locate object method` or `Can't call method`.

Differences from upstream

Fully compatible with upstream `Digest::SHA` 6.04.

See also

- `putstate` - inverse: parses text and constructs or updates an object.
- dump - write the text state to a file or STDOUT.
- `_getstate` - binary counterpart for same-machine use.

putstate

Construct or restore a `Digest::SHA` object from a text state string.

Accepts the text produced by `getstate` or dump and either:

- As a **class method** (`Digest::SHA->putstate($text)`), returns a new object with algorithm and state taken from the string.
- As an **instance method** (`$sha->putstate($text)`), rewrites the receiver in place, switching algorithm if the string asks for a different one.

The algorithm is read from the text, so - unlike `_putstate` - you do not need to preconfigure the object with the correct algorithm.

Synopsis

```
my $sha = Digest::SHA->putstate($text);
$sha->putstate($text);
```

What you get back

A `Digest::SHA` object on success, `undef` if the string could not be parsed.

Examples

```
my $text = $other->getstate;
my $sha  = Digest::SHA->putstate($text);
$sha->add($remaining_data);
```

```
# Update an existing object from a state string:
my $sha = Digest::SHA->new(256);
$sha->putstate($text);
```

```
# Invalid input returns undef:
my $sha = Digest::SHA->putstate("garbage");
print defined $sha ? "ok" : "undef"; # undef
```

Edge cases

- Unrecognised or truncated text returns `undef` rather than croaking.
- When used as an instance method, the receiver's algorithm is overwritten by whatever the text specifies.

Differences from upstream

Fully compatible with upstream `Digest::SHA` 6.04.

See also

- `getstate` - produces the strings this method consumes.
- `load` - convenience wrapper that reads the text from a file.
- `_putstate` - binary-state counterpart for same-machine use.

dump

Write the current `Digest::SHA` state to a file or STDOUT in text form.

With a filename argument, writes the text produced by `getstate` to that file, truncating it if it exists. Without an argument, prints the same text to STDOUT. The companion `load` method reads it back.

Synopsis

```
$sha->dump("hash.state");
$sha->dump;                    # to STDOUT
```

What you get back

The receiver on success (truthy), `undef` on I/O failure.

Examples

```
my $sha = Digest::SHA->new(256);
$sha->add("partial");
$sha->dump("checkpoint.state");
# ... later, perhaps in another process ...
my $resumed = Digest::SHA->load("checkpoint.state");
$resumed->add("tail");
print $resumed->hexdigest;
```

```
# Debug-dump to the terminal:
$sha->dump;
```

```
# Check the return value for I/O errors:
defined $sha->dump("/nonexistent/path/file")
    or warn "could not write state";
```

Edge cases

- An unwritable path returns `undef` rather than croaking.
- Called on something that is not a `Digest::SHA` object, dies with `Can't locate object method` or `Can't call method`.

Differences from upstream

Fully compatible with upstream `Digest::SHA` 6.04.

See also

- `load` - counterpart that reads the file `dump` produces.
- `getstate` - underlying text-state producer.
- `putstate` - text-state ingester.

load

Construct a `Digest::SHA` object from a text state file written by `dump`.

Reads the file, parses its contents as a text state string, and returns a new object with the algorithm and internal state recorded in the file. Typically called as a class method, but works as an instance method too (the receiver is overwritten in place and returned, as with `putstate`).

Synopsis

```
my $sha = Digest::SHA->load("hash.state");
```

What you get back

A `Digest::SHA` object on success, `undef` if the file could not be read or the contents could not be parsed.

Examples

```
my $sha = Digest::SHA->new(256);
$sha->add("one");
$sha->dump("hash.state");

my $resumed = Digest::SHA->load("hash.state");
$resumed->add("two");
print $resumed->hexdigest;
# same as Digest::SHA->new(256)->add("one")->add("two")->hexdigest
```

```
# Missing file returns undef rather than croaking:
my $sha = Digest::SHA->load("/does/not/exist");
print defined $sha ? "ok" : "undef"; # undef
```

```
# Works as an instance method too (and overwrites the receiver):
my $dummy = Digest::SHA->new(1);
my $sha = $dummy->load("hash.state");
```

Edge cases

- I/O failure or a parse error returns `undef`.
- The returned object's algorithm comes from the file, not from any receiver.

Differences from upstream

Fully compatible with upstream `Digest::SHA` 6.04.

See also

- dump - writes the file load reads.
- putstate - parses the text state directly from a string.
- _putstate - binary-state counterpart.

Digest::SHA1

Compute SHA-1 message digests through the interface of Digest::SHA1, the CPAN module older code uses where newer code uses Digest::SHA.

The **functional interface** (sha1, sha1_hex, sha1_base64) hashes its arguments in one call. The **object interface** (Digest::SHA1->new, add, addfile, digest) takes the data in pieces. Every digest comes as raw bytes (20), lowercase hex (40) or unpadded Base64 (27). sha1_transform is the bare SHA-1 compression of one block, for the FIPS 186 random-number construction.

The digests are computed by the engine of the built-in Digest::SHA, which uses the processor's SHA instructions where it has them, so the two modules always agree.

This is a native implementation of Digest::SHA1 2.13's SHA1.xs; its Digest/SHA1.pm is used unchanged.

Synopsis

```
use Digest::SHA1 qw(sha1_hex sha1_base64);

print sha1_hex('abc'), "\n";      #␣
↪a9993e364706816aba3e25717850c26c9cd0d89d
print sha1_base64('abc'), "\n";   # qZk+NkcGgWq6PiVxeFDCbJzQ2J0

my $ctx = Digest::SHA1->new;
$ctx->add('a', 'b');
$ctx->add('c');
print $ctx->hexdigest, "\n";      # the same as sha1_hex('abc')
```

Differences from upstream

Two, each where perl 5.44 with SHA1.xs fails (t/81-xs-native/Digest-SHA1/150-or-better.t):

- sha1_transform given more than 64 bytes uses the first 64. perl5 copies the whole string into a 64-byte buffer and aborts with "buffer overflow detected".
- A method called on the class name instead of an object (Digest::SHA1->add($data)) dies "Not a reference to a Digest::SHA1 object". perl5 crashes.

Functions

Functional digests

sha1

Compute the SHA-1 digest of the concatenated arguments in one call.

Also available as `Digest::SHA1::sha1_base64`, `Digest::SHA1::sha1_hex`.

sha1_transform

The SHA-1 compression function over one 64-byte block, from the standard initial state: the 20-byte result, with no padding and no length.

OO lifecycle

new

Create a Digest::SHA1 object, or reset an existing one.

clone

Copy a Digest::SHA1 object, the data added so far included, into a new object of the same class.

Synopsis

```
my $so_far = $ctx->clone->hexdigest;   # $ctx keeps going
```

OO input

add

Append data to the message being hashed.

addfile

Append everything that can be read from a filehandle to the message.

OO output

b64digest

Finish the message and return its digest, then reset the object to the empty message.

Also available as `Digest::SHA1::digest`, `Digest::SHA1::hexdigest`.

new

Create a Digest::SHA1 object, or reset an existing one.

Called on a class it returns a new object of that class, ready for the empty message. Called on an object it forgets what was added and returns the same object.

Synopsis

```
my $ctx = Digest::SHA1->new;
$ctx->add($data);
$ctx->new;                  # back to the empty message
```

add

Append data to the message being hashed.

Every argument is taken as bytes; a string holding a character above U+00FF dies “Wide character in subroutine entry”. Returns the object, so calls chain.

Synopsis

```
$ctx->add($chunk1, $chunk2)->add($chunk3);
```

addfile

Append everything that can be read from a filehandle to the message.

The handle is read to its end as it is, bytes as they come, so a text file should be opened with `binmode` to hash its exact contents. Dies “No filehandle passed” for a handle that is not open for reading and “Reading from filehandle failed” when a read fails. Returns the object.

Synopsis

```
open my $fh, '<:raw', $file or die "$file: $!";
print Digest::SHA1->new->addfile($fh)->hexdigest, "  $file\n";
```

b64digest

Finish the message and return its digest, then reset the object to the empty message.

Also available as `Digest::SHA1::digest`, `Digest::SHA1::hexdigest`.

`digest` returns the 20 raw bytes, `hexdigest` 40 lowercase hex digits, `b64digest` 27 Base64 characters without the = padding.

Synopsis

```
my $ctx = Digest::SHA1->new;
$ctx->add('abc');
print $ctx->hexdigest, "\n";   # a9993e36...
print $ctx->hexdigest, "\n";   # da39a3ee..., the empty message
```

sha1

Compute the SHA-1 digest of the concatenated arguments in one call.

Also available as `Digest::SHA1::sha1_base64`, `Digest::SHA1::sha1_hex`.

`sha1` returns the 20 raw bytes, `sha1_hex` 40 lowercase hex digits, `sha1_base64` 27 Base64 characters without the = padding. No argument hashes the empty string.

Synopsis

```
use Digest::SHA1 qw(sha1 sha1_hex sha1_base64);
my $hex = sha1_hex($data);
my $raw = sha1(@chunks);       # the same as sha1(join '', @chunks)
```

Edge cases

- Under `-w` (`$^W`), a call that looks like a method call warns: `sha1_hex($ctx)` "probably called as method", `Digest::SHA1->sha1_hex($data)` "probably called as class method", and any other single reference "called with reference argument".
- A character above U+00FF dies "Wide character in subroutine entry".

sha1_transform

The SHA-1 compression function over one 64-byte block, from the standard initial state: the 20-byte result, with no padding and no length.

A block shorter than 64 bytes is padded with zero bytes; bytes past the 64th are not used. This is the function G of FIPS 186-2's random-number generator, not a digest of the data.

Synopsis

```
use Digest::SHA1 qw(sha1_transform);
my $g = sha1_transform($xkey);
```

Digest::base

Digest base class

SYNOPSIS

```
package Digest::Foo;
use base 'Digest::base';
```

DESCRIPTION

The `Digest::base` class provide implementations of the methods `addfile` and `add_bits` in terms of add, and of the methods `hexdigest` and `b64digest` in terms of `digest`.

Digest implementations might want to inherit from this class to get this implementations of the alternative *add* and *digest* methods. A minimal subclass needs to implement the following methods by itself:

```
new
clone
add
digest
```

The arguments and expected behaviour of these methods are described in *Digest*.

SEE ALSO

Digest

Digest::file

Calculate digests of files

SYNOPSIS

```
# Poor mans "md5sum" command
use Digest::file qw(digest_file_hex);
for (@ARGV) {
    print digest_file_hex($_, "MD5"), "  $_\n";
}
```

DESCRIPTION

This module provide 3 convenience functions to calculate the digest of files. The following functions are provided:

- digest_file($file, $algorithm, [$arg,...])

  This function will calculate and return the binary digest of the bytes of the given file. The function will croak if it fails to open or read the file.

  The $algorithm is a string like “MD2”, “MD5”, “SHA-1”, “SHA-512”. Additional arguments are passed to the constructor for the implementation of the given algorithm.

- digest_file_hex($file, $algorithm, [$arg,...])

  Same as digest_file(), but return the digest in hex form.

- digest_file_base64($file, $algorithm, [$arg,...])

  Same as digest_file(), but return the digest as a base64 encoded string.

SEE ALSO

Digest

5.2.30 DirHandle

(obsolete) supply object methods for directory handles

SYNOPSIS

```
# recommended approach since Perl 5.6: do not use DirHandle
if (opendir my $d, '.') {
    while (readdir $d) { something($_); }
    rewind $d;
    while (readdir $d) { something_else($_); }
}

# how you would use this module if you were going to
use DirHandle;
if (my $d = DirHandle->new(".")) {
    while (defined($_ = $d->read)) { something($_); }
    $d->rewind;
    while (defined($_ = $d->read)) { something_else($_); }
}
```

DESCRIPTION

There is no reason to use this module nowadays.

The `DirHandle` method provide an alternative interface to the opendir(), closedir(), readdir(), and rewinddir() functions.

Up to Perl 5.5, opendir() could not autovivify a directory handle from `undef`, so using a lexical handle required using a function from *Symbol* to create an anonymous glob, which took a separate step. `DirHandle` encapsulates this, which allowed cleaner code than opendir(). Since Perl 5.6, opendir() alone has been all you need for lexical handles.

5.2.31 Dumpvalue

provides screen dump of Perl data.

SYNOPSIS

```
use Dumpvalue;
my $dumper = Dumpvalue->new;
$dumper->set(globPrint => 1);
$dumper->dumpValue(\*::);
$dumper->dumpvars('main');
my $dump = $dumper->stringify($some_value);
```

DESCRIPTION

Creation

A new dumper is created by a call

```
$d = Dumpvalue->new(option1 => value1, option2 => value2)
```

Recognized options:

- `arrayDepth`, `hashDepth`

  Print only first N elements of arrays and hashes. If false, prints all the elements.

- `compactDump`, `veryCompact`

  Change style of array and hash dump. If true, short array may be printed on one line.

- `globPrint`

  Whether to print contents of globs.

- `dumpDBFiles`

  Dump arrays holding contents of debugged files.

- `dumpPackages`

  Dump symbol tables of packages.

- `dumpReused`

  Dump contents of “reused” addresses.

- `tick`, `quoteHighBit`, `printUndef`

  Change style of string dump. Default value of `tick` is `auto`, one can enable either double-quotish dump, or single-quotish by setting it to `"` or `'`. By default, characters with high bit set are printed *as is*. If `quoteHighBit` is set, they will be quoted.

- `usageOnly`

  rudimentary per-package memory usage dump. If set, `dumpvars` calculates total size of strings in variables in the package.

- unctrl

  Changes the style of printout of strings. Possible values are `unctrl` and `quote`.

- subdump

  Whether to try to find the subroutine name given the reference.

- bareStringify

  Whether to write the non-overloaded form of the stringify-overloaded objects.

- quoteHighBit

  Whether to print chars with high bit set in binary or “as is”.

- stopDbSignal

  Whether to abort printing if debugger signal flag is raised.

Later in the life of the object the methods may be queries with get() method and set() method (which accept multiple arguments).

Methods

- dumpValue

```
$dumper->dumpValue($value);
$dumper->dumpValue([$value1, $value2]);
```

  Prints a dump to the currently selected filehandle.

- dumpValues

```
$dumper->dumpValues($value1, $value2);
```

  Same as `$dumper->dumpValue([$value1, $value2]);`.

- stringify

```
my $dump = $dumper->stringify($value [,$noticks] );
```

  Returns the dump of a single scalar without printing. If the second argument is true, the return value does not contain enclosing ticks. Does not handle data structures.

- dumpvars

```
$dumper->dumpvars('my_package');
$dumper->dumpvars('my_package', 'foo', '~bar$', '!......');
```

  The optional arguments are considered as literal strings unless they start with `~` or `!`, in which case they are interpreted as regular expressions (possibly negated).

  The second example prints entries with names `foo`, and also entries with names which ends on `bar`, or are shorter than 5 chars.

- set_quote

```
$d->set_quote('"');
```

  Sets `tick` and `unctrl` options to suitable values for printout with the given quote char. Possible values are `auto`, `'` and `"`.

- set_unctrl

```
$d->set_unctrl('unctrl');
```

  Sets `unctrl` option with checking for an invalid argument. Possible values are `unctrl` and `quote`.

- compactDump

```
$d->compactDump(1);
```

  Sets `compactDump` option. If the value is 1, sets to a reasonable big number.

- veryCompact

```
$d->veryCompact(1);
```

  Sets `compactDump` and `veryCompact` options simultaneously.

- set

```
$d->set(option1 => value1, option2 => value2);
```

- get

```
@values = $d->get('option1', 'option2');
```

5.2.32 DynaLoader

Dynamically load C libraries into Perl code

> **Note**
>
> Under pperl the `dl_*` functions work as described here, on any shared library: `dl_load_file`, `dl_find_symbol`, `dl_findfile` and the rest. What pperl never does is load a *module's* compiled part from a shared object: see `bootstrap()` below.

SYNOPSIS

```
package YourPackage;
require DynaLoader;
@ISA = qw(... DynaLoader ...);
__PACKAGE__->bootstrap;

# optional method for 'global' loading
sub dl_load_flags { 0x01 }
```

DESCRIPTION

This document defines a standard generic interface to the dynamic linking mechanisms available on many platforms. Its primary purpose is to implement automatic dynamic loading of Perl modules.

This document serves as both a specification for anyone wishing to implement the DynaLoader for a new platform and as a guide for anyone wishing to use the DynaLoader directly in an application.

The DynaLoader is designed to be a very simple high-level interface that is sufficiently general to cover the requirements of SunOS, HP-UX, Linux, VMS and other platforms.

It is also hoped that the interface will cover the needs of OS/2, NT etc and also allow pseudo-dynamic linking (using `ld -A` at runtime).

It must be stressed that the DynaLoader, by itself, is practically useless for accessing non-Perl libraries because it provides almost no Perl-to-C 'glue'. There is, for example, no mechanism for calling a C library function or supplying arguments. A C::DynaLib module is available from CPAN sites which performs that function for some common system types. And since the year 2000, there's also Inline::C, a module that allows you to write Perl subroutines in C. Also available from your local CPAN site.

DynaLoader Interface Summary

```
@dl_library_path
@dl_resolve_using
@dl_require_symbols
$dl_debug
$dl_dlext
@dl_librefs
@dl_modules
@dl_shared_objects
                                                Implemented in:
bootstrap($modulename)                               Perl
@filepaths = dl_findfile(@names)                     Perl
$flags = $modulename->dl_load_flags                  Perl
$symref  = dl_find_symbol_anywhere($symbol)          Perl

$libref  = dl_load_file($filename, $flags)           C
$status  = dl_unload_file($libref)                   C
$symref  = dl_find_symbol($libref, $symbol)          C
@symbols = dl_undef_symbols()                        C
dl_install_xsub($name, $symref [, $filename])        C
$message = dl_error                                  C
```

- @dl_library_path

  The standard/default list of directories in which dl_findfile() will search for libraries etc. Directories are searched in order: $dl_library_path[0], [1], ... etc

  @dl_library_path is initialised to hold the list of 'normal' directories (`/usr/lib`, etc) determined by **Configure** (`$Config{'libpth'}`). This should ensure portability across a wide range of platforms.

  @dl_library_path should also be initialised with any other directories that can

be determined from the environment at runtime (such as LD_LIBRARY_PATH for SunOS).

After initialisation @dl_library_path can be manipulated by an application using push and unshift before calling dl_findfile(). Unshift can be used to add directories to the front of the search order either to save search time or to override libraries with the same name in the 'normal' directories.

The load function that dl_load_file() calls may require an absolute pathname. The dl_findfile() function and @dl_library_path can be used to search for and return the absolute pathname for the library/object that you wish to load.

- @dl_resolve_using

  A list of additional libraries or other shared objects which can be used to resolve any undefined symbols that might be generated by a later call to load_file().

  This is only required on some platforms which do not handle dependent libraries automatically. For example the Socket Perl extension library (`auto/Socket/Socket.so`) contains references to many socket functions which need to be resolved when it's loaded. Most platforms will automatically know where to find the 'dependent' library (e.g., `/usr/lib/libsocket.so`). A few platforms need to be told the location of the dependent library explicitly. Use @dl_resolve_using for this.

  Example usage:

```
@dl_resolve_using = dl_findfile('-lsocket');
```

- @dl_require_symbols

  A list of one or more symbol names that are in the library/object file to be dynamically loaded. This is only required on some platforms.

- @dl_librefs

  An array of the handles returned by successful calls to dl_load_file(), made by bootstrap, in the order in which they were loaded. Can be used with dl_find_symbol() to look for a symbol in any of the loaded files.

- @dl_modules

  An array of module (package) names that have been bootstrap'ed.

- @dl_shared_objects

  An array of file names for the shared objects that were loaded.

- dl_error()

  Syntax:

```
$message = dl_error();
```

  Error message text from the last failed DynaLoader function. Note that, similar to errno in unix, a successful function call does not reset this message.

  Implementations should detect the error as soon as it occurs in any of the other functions and save the corresponding message for later retrieval. This will avoid problems on some platforms (such as SunOS) where the error message is very temporary (e.g., dlerror()).

- $dl_debug

  Internal debugging messages are enabled when $dl_debug is set true. Currently setting $dl_debug only affects the Perl side of the DynaLoader. These messages should help an application developer to resolve any DynaLoader usage problems.

  $dl_debug is set to `$ENV{'PERL_DL_DEBUG'}` if defined.

  For the DynaLoader developer/porter there is a similar debugging variable added to the C code (see dlutils.c) and enabled if Perl was built with the **-DDEBUGGING** flag. This can also be set via the PERL_DL_DEBUG environment variable. Set to 1 for minimal information or higher for more.

- $dl_dlext

  When specified (localised) in a module's `.pm` file, indicates the extension which the module's loadable object will have. For example:

```
local $DynaLoader::dl_dlext = 'unusual_ext';
```

  would indicate that the module's loadable object has an extension of `unusual_ext` instead of the more usual `$Config{dlext}`. NOTE: This also requires that the module's `Makefile.PL` specify (in `WriteMakefile()`):

```
DLEXT => 'unusual_ext',
```

- dl_findfile()

  Syntax:

```
@filepaths = dl_findfile(@names)
```

  Determine the full paths (including file suffix) of one or more loadable files given their generic names and optionally one or more directories. Searches directories in @dl_library_path by default and returns an empty list if no files were found.

  Names can be specified in a variety of platform independent forms. Any names in the form **-lname** are converted into `libname.*`, where `.*` is an appropriate suffix for the platform.

  If a name does not already have a suitable prefix and/or suffix then the corresponding file will be searched for by trying combinations of prefix and suffix appropriate to the platform: "$name.o", "lib$name.*" and "$name".

  If any directories are included in @names they are searched before @dl_library_path. Directories may be specified as **-Ldir**. Any other names are treated as filenames to be searched for.

  Using arguments of the form `-Ldir` and `-lname` is recommended.

  Example:

```
@dl_resolve_using = dl_findfile(qw(-L/usr/5lib -lposix));
```

- dl_expandspec()

  Syntax:

```
$filepath = dl_expandspec($spec)
```

Some unusual systems, such as VMS, require special filename handling in order to deal with symbolic names for files (i.e., VMS's Logical Names).

To support these systems a dl_expandspec() function can be implemented either in the `dl_*.xs` file or code can be added to the dl_expandspec() function in `DynaLoader.pm`. See `DynaLoader_pm.PL` for more information.

- dl_load_file()

  Syntax:

```
$libref = dl_load_file($filename, $flags)
```

  Dynamically load $filename, which must be the path to a shared object or library. An opaque 'library reference' is returned as a handle for the loaded object. Returns undef on error.

  The $flags argument to alters dl_load_file behaviour.
  Assigned bits:

```
0x01  make symbols available for linking later dl_load_file's.
      (only known to work on Solaris 2 using dlopen(RTLD_
↪GLOBAL))
      (ignored under VMS; this is a normal part of image␣
↪linking)
```

  (On systems that provide a handle for the loaded object such as SunOS and HPUX, $libref will be that handle. On other systems $libref will typically be $filename or a pointer to a buffer containing $filename. The application should not examine or alter $libref in any way.)

  This is the function that does the real work. It should use the current values of @dl_require_symbols and @dl_resolve_using if required.

```
SunOS: dlopen($filename)
HP-UX: shl_load($filename)
Linux: dld_create_reference(@dl_require_symbols); dld_link(
↪$filename)
VMS:   lib$find_image_symbol($filename,$dl_require_symbols[0])
```

  (The dlopen() function is also used by Solaris and some versions of Linux, and is a common choice when providing a "wrapper" on other mechanisms as is done in the OS/2 port.)

- dl_unload_file()

  Syntax:

```
$status = dl_unload_file($libref)
```

  Dynamically unload $libref, which must be an opaque 'library reference' as returned from dl_load_file. Returns one on success and zero on failure. This function is optional and may not necessarily be provided on all platforms.

If it is defined and perl is compiled with the C macro DL_UNLOAD_ALL_AT_EXIT defined, then it is called automatically when the interpreter exits for every shared object or library loaded by DynaLoader::bootstrap. All such library references are stored in @dl_librefs by DynaLoader::Bootstrap as it loads the libraries. The files are unloaded in last-in, first-out order.

This unloading is usually necessary when embedding a shared-object perl (e.g. one configured with -Duseshrplib) within a larger application, and the perl interpreter is created and destroyed several times within the lifetime of the application. In this case it is possible that the system dynamic linker will unload and then subsequently reload the shared libperl without relocating any references to it from any files DynaLoaded by the previous incarnation of the interpreter. As a result, any shared objects opened by DynaLoader may point to a now invalid 'ghost' of the libperl shared object, causing apparently random memory corruption and crashes. This behaviour is most commonly seen when using Apache and mod_perl built with the APXS mechanism.

```
SunOS: dlclose($libref)
HP-UX: ???
Linux: ???
VMS:   ???
```

(The dlclose() function is also used by Solaris and some versions of Linux, and is a common choice when providing a "wrapper" on other mechanisms as is done in the OS/2 port.)

- dl_load_flags()

  Syntax:

```
$flags = dl_load_flags $modulename;
```

  Designed to be a method call, and to be overridden by a derived class (i.e. a class which has DynaLoader in its @ISA). The definition in DynaLoader itself returns 0, which produces standard behavior from dl_load_file().

- dl_find_symbol()

  Syntax:

```
$symref = dl_find_symbol($libref, $symbol)
```

  Return the address of the symbol $symbol or undef if not found. If the target system has separate functions to search for symbols of different types then dl_find_symbol() should search for function symbols first and then other types.

  The exact manner in which the address is returned in $symref is not currently defined. The only initial requirement is that $symref can be passed to, and understood by, dl_install_xsub().

```
SunOS: dlsym($libref, $symbol)
HP-UX: shl_findsym($libref, $symbol)
Linux: dld_get_func($symbol) and/or dld_get_symbol($symbol)
VMS:   lib$find_image_symbol($libref,$symbol)
```

- dl_find_symbol_anywhere()

  Syntax:

  ```
  $symref = dl_find_symbol_anywhere($symbol)
  ```

  Applies dl_find_symbol() to the members of @dl_librefs and returns the first match found.

- dl_undef_symbols()

  Example

  ```
  @symbols = dl_undef_symbols()
  ```

  Return a list of symbol names which remain undefined after load_file(). Returns `()` if not known. Don't worry if your platform does not provide a mechanism for this. Most do not need it and hence do not provide it, they just return an empty list.

- dl_install_xsub()

  Syntax:

  ```
  dl_install_xsub($perl_name, $symref [, $filename])
  ```

  Create a new Perl external subroutine named $perl_name using $symref as a pointer to the function which implements the routine. This is simply a direct call to newXS()/newXS_flags(). Returns a reference to the installed function.

  The $filename parameter is used by Perl to identify the source file for the function if required by die(), caller() or the debugger. If $filename is not defined then "DynaLoader" will be used.

- bootstrap()

  Syntax:

  bootstrap($module [...])

  This is the normal entry point for automatic dynamic loading in Perl.

  Under pperl, `bootstrap()` never loads a module's shared object. It starts the module's compiled part when that part is built into pperl, and otherwise dies with `Can't locate loadable object for module $module in @INC`, the message modules with a pure-Perl fallback test for. The steps below are what perl does.

  It performs the following actions:

  - locates an auto/$module directory by searching @INC
  - uses dl_findfile() to determine the filename to load
  - sets @dl_require_symbols to `("boot_$module")`
  - executes an `auto/$module/$module.bs` file if it exists (typically used to add to @dl_resolve_using any files which are required to load the module on the current platform)
  - calls dl_load_flags() to determine how to load the file.

- calls dl_load_file() to load the file
- calls dl_undef_symbols() and warns if any symbols are undefined
- calls dl_find_symbol() for "boot_$module"
- calls dl_install_xsub() to install it as "${module}::bootstrap"
- calls &{"${module}::bootstrap"} to bootstrap the module (actually it uses the function reference returned by dl_install_xsub for speed)

All arguments to bootstrap() are passed to the module's bootstrap function. The default code generated by `xsubpp` expects $module [, $version] If the optional $version argument is not given, it defaults to `$XS_VERSION // $VERSION` in the module's symbol table.

AUTHOR

Tim Bunce, 11 August 1994.

This interface is based on the work and comments of (in no particular order): Larry Wall, Robert Sanders, Dean Roehrich, Jeff Okamoto, Anno Siegel, Thomas Neumann, Paul Marquess, Charles Bailey, myself and others.

Larry Wall designed the elegant inherited bootstrap mechanism and implemented the first Perl 5 dynamic loader using it.

Solaris global loading added by Nick Ing-Simmons with design/coding assistance from Tim Bunce, January 1996.

5.2.33 EV

A fast, full-featured event loop: call your code when a file handle is readable or writable, a timer expires, a signal arrives, a child process changes state or a file changes.

You create watchers - objects recording your interest in an event - and then run the loop, which waits for events and calls each watcher's callback. `EV::io`, `EV::timer`, `EV::periodic`, `EV::signal`, `EV::child`, `EV::stat`, `EV::idle`, `EV::prepare`, `EV::check`, `EV::fork`, `EV::embed` and `EV::async` create watchers on the default loop; `EV::run` runs it. A program needing more than one loop makes more with `EV::Loop->new`, which has the same functions as methods.

Synopsis

```
use EV;

my $once  = EV::timer 2, 0, sub { warn "2 seconds passed\n" };
my $every = EV::timer 0, 3, sub { warn "now, then every 3 seconds\n"␣
 ↪};
my $input = EV::io *STDIN, EV::READ, sub {
    my ($w, $revents) = @_;            # every callback gets these␣
 ↪two
    warn "you typed: ", scalar <STDIN>;
};
```

(continues on next page)

(continued from previous page)

```
my $quit  = EV::signal 'QUIT', sub { EV::break };

EV::run;    # until EV::break, or no active watcher is left
```

Watchers

A watcher stays alive as long as the object its constructor returned does: keep it in a variable, and `undef` it to stop and free the watcher. Every callback is called with the watcher and a bitmask of the events received (`EV::READ`, `EV::TIMER` and the like), so one sub can serve several watchers.

Every constructor starts the watcher; its `_ns` twin (`EV::io_ns`, `EV::timer_ns` ...) creates it stopped, for a later `start`. The methods all watchers share - `start`, `stop`, `is_active`, `data`, `cb`, `priority`, `keepalive` and more - are `EV::Watcher`'s; each watcher class (`EV::IO`, `EV::Timer`, `EV::Periodic`, `EV::Signal`, `EV::Child`, `EV::Stat`, `EV::Embed`, `EV::Async`) adds its own. Changing a running watcher (`set`, `priority`, `fh` ...) stops and restarts it, losing events pending for it. `EV::idle`, `EV::prepare`, `EV::check` and `EV::fork` watchers have only the shared methods.

An exception thrown by a callback does not end the loop: EV calls `$EV::DIED` with the error in `$@`, which by default warns `EV: error in callback (ignoring): ...` and goes on.

Priorities

A watcher's priority, from `EV::MINPRI` (-2) to `EV::MAXPRI` (2), decides which of several pending watchers runs first; new watchers have priority 0. An idle watcher runs only when nothing of its priority or higher is pending.

Constants

Called as functions (`EV::READ`), as EV exports nothing:

- Events: `EV::READ`, `EV::WRITE`, `EV::TIMER`, `EV::PERIODIC`, `EV::SIGNAL`, `EV::CHILD`, `EV::STAT`, `EV::IDLE`, `EV::PREPARE`, `EV::CHECK`, `EV::EMBED`, `EV::FORK`, `EV::CLEANUP`, `EV::ASYNC`, `EV::CUSTOM`, `EV::ERROR`, `EV::IO`, `EV::NONE`, `EV::UNDEF`.
- `EV::run` flags `EV::RUN_NOWAIT`, `EV::RUN_ONCE`; `EV::break` modes `EV::BREAK_ONE`, `EV::BREAK_ALL`, `EV::BREAK_CANCEL`.
- Backends `EV::BACKEND_SELECT`, `EV::BACKEND_POLL`, `EV::BACKEND_EPOLL`, `EV::BACKEND_KQUEUE`, `EV::BACKEND_DEVPOLL`, `EV::BACKEND_PORT`, `EV::BACKEND_ALL`, `EV::BACKEND_MASK`, and loop flags `EV::FLAG_AUTO`, `EV::FLAG_FORKCHECK`, `EV::FLAG_SIGNALFD`, `EV::FLAG_NOSIGMASK`, `EV::FLAG_NOENV`, `EV::FLAG_NOINOTIFY`.
- `EV::MINPRI`, `EV::MAXPRI`; `EV::VERSION_MAJOR`, `EV::VERSION_MINOR` (of libev).
- The names of EV 3 kept for old programs: `EV::TIMEOUT`, `EV::LOOP_NONBLOCK`, `EV::LOOP_ONESHOT`, `EV::UNLOOP_CANCEL`, `EV::UNLOOP_ONE`, `EV::UNLOOP_ALL`, `EV::FLAG_NOSIGFD` (0).

`EV::CHECK` shares its name with Perl's `CHECK` blocks: write it after `use EV`, where it is compiled as the constant.

Backends

The default loop waits with epoll. select, poll, Linux AIO and io_uring are there too, to choose with `EV::Loop->new` flags or the `LIBEV_FLAGS` environment variable (the backend bits added together). `EV::supported_backends` lists them. File status watchers use inotify where it works.

Signals and fork

Perl's own `%SIG` handlers run only when Perl runs, which in an event loop may be late: use `EV::signal` watchers instead, or keep an `EV::check` watcher (`my $c = EV::check sub {}`) so that Perl gets control on every loop iteration.

After `fork`, the child's default loop is reset for it automatically the next time it is used. Loops made with `EV::Loop->new` need `$loop->loop_fork` in the child, or the flag `EV::FLAG_FORKCHECK`.

`$EV::API` holds the C function table EV gives modules such as `Coro::EV` that drive the loop from compiled code.

Differences from upstream

- `EV::signal` (and `EV::Loop->signal`, `$w->signal`) take the second names of signals, `IOT`, `CLD` and `POLL`, as the signals they name; upstream takes them for nonexistent signal numbers 65 to 67 and corrupts its memory. A number that is no signal is refused with `illegal signal number or name`. Covered by `t/81-xs-native/EV/310-signal-alias.t`.
- `EV::Periodic->reschedule_cb` keeps its callback and the watcher alive; upstream frees them early. See `EV::Periodic`.
- A signal watched by EV and by Event reaches both. Under perl5 their handlers compete and the module that installed its handler last takes the signal; the other's watchers never fire. Between EV's own loops a signal still belongs to one, as upstream.

Everything else behaves as with EV 4.37 and the libev 4.33 it bundles. Covered by EV's own test suite, `t/81-xs-native/EV/100-upstream-00_load.t` to `200-upstream-11_signal.t`.

Modules

- `EV::Async` - An async watcher, as `EV::async` makes: anything may trigger it with `send`, and its callback then runs in its loop - the way compiled code or another loop wakes a loop that is waiting.
- `EV::Child` - A child process watcher, as `EV::child` makes: its callback runs when a child ends (or, traced, stops or continues).
- `EV::Embed` - An embed watcher, as `EV::embed` makes: it lets running one loop also serve another loop's I/O.

- EV::IO - An I/O watcher, as EV::io makes: its callback runs when a file handle is readable or writable.
- EV::Loop - An event loop of your own, beside the default one: EV::Loop->new makes one, and EV::default_loop returns the default loop as an EV::Loop too.
- EV::Periodic - A wall-clock timer, as EV::periodic makes: its callback runs at a given time, at every multiple of an interval, or whenever a reschedule sub says - see EV::periodic for the three modes.
- EV::Signal - A signal watcher, as EV::signal makes: its callback runs when the process receives a signal.
- EV::Stat - A file status watcher, as EV::stat makes: its callback runs when a path's status changes - created, deleted, written, its permissions or times changed.
- EV::Timer - A relative timer, as EV::timer makes: its callback runs after a number of seconds, and optionally again at a repeat interval.
- EV::Watcher - The methods every EV watcher has: start and stop it, ask whether it is active or pending, change its callback, data and priority, and decide whether it keeps the loop running.

Functions

Running the loop

default_loop

The default loop, as an EV::Loop object: the one the EV:: functions and watchers use, and the only one that can watch signals and child processes. EV creates it when loaded, so the optional flags (EV::FLAG_*, EV::BACKEND_* bits) matter only to a call after EV::default_destroy, which creates it again. Undef when the loop cannot be created.

default_destroy

Destroy the default loop, with every watcher on it. Nothing may use it until EV::default_loop creates a new one; watcher objects made on the old one must not be used again. Rarely wanted: a program ending need not call it.

iteration

How many times the loop has waited for events - a generation counter. EV::loop_count is its old name.

Also available as EV::loop_count.

depth

How many EV::run calls are in progress: 0 outside the loop, 1 in a callback, more when a callback runs the loop itself. EV::loop_depth is its old name.

Also available as EV::loop_depth.

loop

Run the event loop: wait for events and call the watchers' callbacks.

Also available as `EV::run`.

break

Make `EV::run` return once the current callbacks are done: the innermost call with `EV::break` or `EV::break EV::BREAK_ONE`, all of them with `EV::BREAK_ALL`; `EV::BREAK_CANCEL` withdraws a break not yet acted on. `EV::unloop` is the old name.

Also available as `EV::unloop`.

Watchers

io

Watch a file handle: call the callback whenever it is readable or writable without blocking.

Also available as `EV::io_ns`.

timer

Call the callback after a number of seconds, and optionally again and again after that.

Also available as `EV::timer_ns`.

periodic

Call the callback at wall-clock times: once at a given time, at every multiple of an interval, or whenever a sub of yours says.

Also available as `EV::periodic_ns`.

signal

Call the callback when a signal arrives: `my $w = EV::signal 'HUP', sub { ... }`. The signal is a name or number, as for `kill` and `%SIG`. EV takes the signal from Perl while the watcher is active, and gives it back when stopped; any number of watchers may watch one signal, but only one loop - starting one in a second loop croaks `unable to start signal watcher, signal N already registered in another loop`. A name or number that is no signal croaks `illegal signal number or name`. `EV::signal_ns` creates it stopped. The second names `IOT`, `CLD` and `POLL` work, as `ABRT`, `CHLD` and `IO`, which upstream does not do (covered by `t/81-xs-native/EV/310-signal-alias.t`).

Also available as `EV::signal_ns`.

idle

Call the callback whenever the loop has nothing more urgent to do: `my $w = EV::idle sub { ... }`. It runs when no watcher of its priority or higher is pending, and while it is active the loop does not block - it is called again and again until stopped. At priority `EV::MINPRI` it runs only when nothing else is pending. `EV::idle_ns` creates it stopped.

Also available as `EV::idle_ns`.

prepare

Call the callback just before the loop blocks to wait: `my $w = EV::prepare sub { ... }`. Watchers can still be created and changed in it - a prepare watcher can set up the I/O and timer watchers another event library needs, and an `EV::check` watcher clean them up. `EV::prepare_ns` creates it stopped.

Also available as `EV::prepare_ns`.

check

Call the callback just after the loop wakes up, before any other callback: `my $w = EV::check sub { ... }`. The partner of `EV::prepare`; an empty one also lets Perl's own `%SIG` handlers run on every iteration. `EV::check_ns` creates it stopped. Write the event constant `EV::CHECK` after `use EV`.

Also available as `EV::check_ns`.

fork

Call the callback in a child process after `fork`, before its event loop next blocks: `my $w = EV::fork sub { ... }`. `EV::fork_ns` creates it stopped.

Also available as `EV::fork_ns`.

child

Call the callback when a child process ends: `my $w = EV::child $pid, 0, sub { my ($w) = @_; my $status = $w->rstatus; ... }`. A `$pid` of 0 watches every child; a true `$trace` also reports a child stopping and continuing. EV reaps the child, so its exit status is `$w->rstatus` (as `$?` would be) and its pid `$w->rpid`. The watcher may be made after the child has already exited, as long as the loop has not run since. Only the default loop watches children. `EV::child_ns` creates it stopped.

Also available as `EV::child_ns`.

stat

Call the callback when a file's status changes - its size, times, permissions, or whether it exists: `my $w = EV::stat '/etc/passwd', 0, sub { ... }`. The path need not exist. Changes come from inotify where it works, else from polling every `$interval` seconds (0: about five seconds, recommended). The watcher's `attr` and `prev` give the status after and before the change. Not meant for thousands of files. `EV::stat_ns` creates it stopped.

Also available as `EV::stat_ns`.

embed

Embed another loop in the default loop: running the default loop then also handles the other loop's I/O. `my $w = EV::embed $other_loop`. Without a callback the embedded loop is serviced automatically (the recommended way); with one, the callback is called when the other loop has I/O pending and must call `$w->sweep`. Only I/O watchers of the embedded loop are served reliably, and only a loop using one of `EV::embeddable_backends` can be embedded - another croaks `passed loop is not embeddable via EV::embed`. `EV::embed_ns` creates it stopped.

Also available as `EV::embed_ns`.

async

Create a watcher another thread of control wakes the loop with: `my $w = EV::async sub { ... }`, triggered by `$w->send`. Of little use from Perl itself; compiled code (and `Coro`) uses it to wake a waiting loop. `EV::async_ns` creates it stopped. See `EV::Async`.

Also available as `EV::async_ns`.

once

Wait once for a handle to become ready, a timeout, or the first of the two, without keeping a watcher object.

Time

sleep

Block the whole process for a number of seconds, which may be fractional: `EV::sleep 0.25`. No watcher runs meanwhile.

time

The current time, in fractional seconds since the epoch, asked of the system now. `EV::now` is cheaper where the loop's time will do.

now

The time the current loop iteration began, in fractional seconds since the epoch: the time relative timers count from, and cheaper than `EV::time`.

now_update

Set the loop's idea of the time (`EV::now`) to the time now. The loop does this every iteration; a callback that runs for a long time before starting timers should do it first, or the timers start late.

suspend

Pause the loop's clock, for a program that stops for a while (a game put in the background with ^Z): between `EV::suspend` and `EV::resume`, timers do not run down, and periodic watchers skip what they would have fired. After `EV::suspend` call nothing on the loop but `EV::resume`.

resume

Restart the loop's clock after `EV::suspend`; only after it.

Events

feed_signal

Act as if a signal had arrived, whichever loop watches it: `EV::feed_signal 'USR1'`. The signal is a name or number, as for `kill`. For handing EV signals caught some other way.

feed_fd_event

Act as if a file descriptor had become ready: `EV::feed_fd_event $fd, EV::READ` makes the I/O watchers on `$fd` that wait for those events pending.

feed_signal_event

Act as if a signal had arrived at the default loop: `EV::feed_signal_event 'HUP'` makes its watchers of that signal pending. A name or number, as for `kill`; one that is no signal croaks `illegal signal number or name`.

pending_count

How many watchers have received events whose callbacks have not run yet.

invoke_pending

Call the callbacks of all pending watchers now.

Backends and tuning

supported_backends

The backends this EV can use, as `EV::BACKEND_*` bits: select, poll, epoll, Linux AIO and io_uring.

recommended_backends

The backends recommended on this system, as `EV::BACKEND_*` bits.

embeddable_backends

The backends a loop can use and still be embedded in another with `EV::embed`, as `EV::BACKEND_*` bits.

backend

The backend the default loop waits with, one `EV::BACKEND_*` bit - `EV::BACKEND_EPOLL` unless chosen otherwise.

loop_verify

Check the loop's internal data for consistency, aborting the program if it is corrupt - for debugging. `EV::loop_verify` is its old name.

Also available as `EV::verify`.

set_io_collect_interval

Make the loop wait at least this many seconds between polls for I/O, collecting more events per wakeup - fewer, larger batches for a busy server, at the cost of latency. 0, the default, waits no longer than needed.

set_timeout_collect_interval

Let timers that expire within this many seconds of each other fire in one wakeup, trading timer precision for fewer wakeups. 0, the default, fires each on time.

EV::Async

An async watcher, as `EV::async` makes: anything may trigger it with `send`, and its callback then runs in its loop - the way compiled code or another loop wakes a loop that is waiting. Several sends before the callback runs give one callback. The methods all watchers share are `EV::Watcher`'s.

Synopsis

```
my $wake = EV::async sub { print "woken\n"; EV::break };
my $t    = EV::timer 1, 0, sub { $wake->send };
EV::run;
```

Functions

Waking

send

Trigger the watcher: its callback runs in the loop's next round.

async_pending

True when send was called and the callback has not run yet.

EV::Child

A child process watcher, as `EV::child` makes: its callback runs when a child ends (or, traced, stops or continues). These methods give the child's pid and status; the methods all watchers share are `EV::Watcher`'s. Only the default loop watches children.

EV reaps the children it reports: their status is `rstatus`, not `$?` from a `waitpid` of your own.

Synopsis

```
my $pid = fork // die "fork: $!";
exec 'sleep', 1 unless $pid;
my $w = EV::child $pid, 0, sub {
    my ($w) = @_;
    printf "%d exited with %d\n", $w->rpid, $w->rstatus >> 8;
};
EV::run;
```

Functions

Status

reinit

Re-install EV's own SIGCHLD handler: `EV::Child::reinit`, after code such as `local $SIG{CHLD}` left another one behind and child watchers stopped firing. Experimental upstream; its form may change.

set

Watch another child: `$w->set($pid, $trace)`, as `EV::child` takes them - `$pid` 0 for any child.

pid

The pid the watcher watches (0 for any child). The same method gives, under its other names, what the last status change reported: `$w->rpid`, the pid of the child that changed - useful when watching any child - and `$w->rstatus`, its wait status as `waitpid` sets `$?` (`>> 8` for the exit code).

Also available as `EV::Child::rpid`, `EV::Child::rstatus`.

EV::Embed

An embed watcher, as `EV::embed` makes: it lets running one loop also serve another loop's I/O. These methods name the embedded loop and serve it by hand; the methods all watchers share are `EV::Watcher`'s.

Synopsis

```
my $other = EV::Loop->new(EV::BACKEND_EPOLL);
my $embed = EV::embed $other;    # served automatically
EV::run;                         # runs both
```

Functions

Embedding

set

Embed another loop instead: `$w->set($loop)`.

other

The embedded loop, an `EV::Loop`.

sweep

Run the embedded loop once, without waiting, handling its pending events - what a watcher made with a callback must do in it.

EV::IO

An I/O watcher, as `EV::io` makes: its callback runs when a file handle is readable or writable. These methods change what it watches; the methods all watchers share are `EV::Watcher`'s.

Changing an active watcher stops and restarts it, losing its pending events.

Synopsis

```
my $w = EV::io $sock, EV::READ, \&on_read;
$w->events(EV::READ | EV::WRITE);    # also wait until writable
$w->set($other_sock, EV::READ);      # watch another handle
```

Functions

Configuration

set

Watch another handle, for other events: `$w->set($fh, EV::READ)`. The handle may be a file handle or a descriptor number; one without a descriptor croaks `illegal file descriptor or filehandle`.

fh

The handle the watcher watches, as it was given; with an argument, watch that handle instead and return the old one.

events

The events the watcher waits for, `EV::READ`, `EV::WRITE` or both; with an argument, wait for those instead and return the old mask.

EV::Loop

An event loop of your own, beside the default one: `EV::Loop->new` makes one, and `EV::default_loop` returns the default loop as an `EV::Loop` too.

An `EV::Loop` has, as methods, the `EV::` functions that work on a loop: `$loop->run`, `$loop->now`, `$loop->timer(...)`, `$loop->io(...)` and so on, each doing for this loop what the `EV::` function does for the default loop - see those for the details. A loop lives as long as the object, or a watcher on it, does.

A loop of your own may use another backend than the default loop (`EV::BACKEND_*` flags), or serve part of a program separately. Child process watchers work only on the default loop, and a signal can be watched by one loop at a time. After `fork`, the child must call `$loop->loop_fork` before using such a loop, unless it was made with `EV::FLAG_FORKCHECK`.

Synopsis

```
my $loop = EV::Loop->new;
my $t = $loop->timer(1, 0, sub { print "tick\n"; $loop->break });
$loop->run;
```

Functions

Loops

new

Make a new event loop: `EV::Loop->new`, or `EV::Loop->new($flags)` with `EV::BACKEND_*` bits to choose its backends and `EV::FLAG_*` bits (`EV::FLAG_FORKCHECK` to recover from `fork` by itself, `EV::FLAG_NOENV` to ignore `LIBEV_FLAGS`). Undef when no requested backend works.

loop_fork

Prepare the loop for use in a child process after `fork`: call it in the child before running the loop again. Loops made with `EV::FLAG_FORKCHECK`, and the default loop, do it by themselves.

Running the loop

iteration

How many times this loop has waited for events, as `EV::iteration`; `loop_count` is its old name.

Also available as `EV::Loop::loop_count`.

depth

How many `run` calls of this loop are in progress, as `EV::depth`; `loop_depth` is its old name.

Also available as `EV::Loop::loop_depth`.

loop

Run this loop: `$loop->run`, `$loop->run(EV::RUN_ONCE)`, `$loop->run(EV::RUN_NOWAIT)`, as `EV::run` runs the default loop. True when it returned with work left. `loop` is its old name.

Also available as `EV::Loop::run`.

break

Make this loop's `run` return, as `EV::break`: `$loop->break`, `$loop->break(EV::BREAK_ALL)`. `unloop` is its old name.

Also available as `EV::Loop::unloop`.

Watchers

io

An I/O watcher on this loop: `$loop->io($fh, EV::READ, sub { ... })`, as `EV::io`; `io_ns` makes it stopped.

Also available as `EV::Loop::io_ns`.

timer

A timer on this loop: `$loop->timer($after, $repeat, sub { ... })`, as `EV::timer`; `timer_ns` makes it stopped.

Also available as `EV::Loop::timer_ns`.

periodic

A wall-clock timer on this loop: `$loop->periodic($at, $interval, $reschedule_cb, sub { ... })`, as `EV::periodic`; `periodic_ns` makes it stopped.

Also available as `EV::Loop::periodic_ns`.

signal

A signal watcher on this loop: `$loop->signal('HUP', sub { ... })`, as `EV::signal`; `signal_ns` makes it stopped. A signal can be watched by one loop at a time.

Also available as `EV::Loop::signal_ns`.

idle

An idle watcher on this loop: `$loop->idle(sub { ... })`, as `EV::idle`; `idle_ns` makes it stopped.

Also available as `EV::Loop::idle_ns`.

prepare

A prepare watcher on this loop, as `EV::prepare`; `prepare_ns` makes it stopped.

Also available as `EV::Loop::prepare_ns`.

check

A check watcher on this loop, as `EV::check`; `check_ns` makes it stopped.

Also available as `EV::Loop::check_ns`.

fork

A fork watcher on this loop, as `EV::fork`; `fork_ns` makes it stopped.

Also available as `EV::Loop::fork_ns`.

child

A child process watcher: `$loop->child($pid, $trace, sub { ... })`, as `EV::child`; `child_ns` makes it stopped. Only the default loop can watch children: call it on `EV::default_loop`.

Also available as `EV::Loop::child_ns`.

stat

A file status watcher on this loop: `$loop->stat($path, $interval, sub { ... })`, as `EV::stat`; `stat_ns` makes it stopped.

Also available as `EV::Loop::stat_ns`.

embed

Embed another loop in this one: `$loop->embed($other)`, as `EV::embed`; `embed_ns` makes the watcher stopped.

Also available as `EV::Loop::embed_ns`.

async

An async watcher on this loop, as `EV::async`; `async_ns` makes it stopped.

Also available as `EV::Loop::async_ns`.

once

Wait once on this loop for a handle, a timeout, or the first of the two: `$loop->once($fh, EV::READ, 10, sub { my ($revents) = @_; ... })`, as `EV::once`.

Time

now

The time this loop's current iteration began, as `EV::now`.

now_update

Set this loop's time to the time now, as `EV::now_update`.

suspend

Pause this loop's clock, as `EV::suspend`; only `resume` may follow.

resume

Restart this loop's clock after `suspend`, as `EV::resume`.

Events

feed_fd_event

Act as if a file descriptor had become ready, for this loop's watchers, as `EV::feed_fd_event`.

pending_count

How many of this loop's watchers are pending, as `EV::pending_count`.

invoke_pending

Call the callbacks of this loop's pending watchers now, as `EV::invoke_pending`.

Tuning

set_io_collect_interval

Make this loop wait at least this long between polls for I/O, as `EV::set_io_collect_interval`.

set_timeout_collect_interval

Let this loop's timers that expire close together fire in one wakeup, as `EV::set_timeout_collect_interval`.

backend

The backend this loop waits with, one `EV::BACKEND_*` bit.

loop_verify

Check this loop's internal data, aborting if it is corrupt, as `EV::verify`; `loop_verify` is its old name.

Also available as `EV::Loop::verify`.

EV::Periodic

A wall-clock timer, as `EV::periodic` makes: its callback runs at a given time, at every multiple of an interval, or whenever a reschedule sub says - see `EV::periodic` for the three modes. These methods read and change its schedule; the methods all watchers share are `EV::Watcher`'s.

Changing the offset or interval takes effect at the next firing; `set`, `again` and a new reschedule sub take effect at once.

Synopsis

```
my $w = EV::periodic 0, 3600, 0, sub { print "on the hour\n" };
printf "next at %s\n", scalar localtime $w->at;
$w->interval(1800);    # on the half hour, from the next firing
```

Differences from upstream

- `reschedule_cb` returns the sub without letting it go: upstream frees the watcher's sub when the returned copy goes away, so every later firing fails with `EV: error in callback (ignoring): Not a CODE reference`, and setting one frees the old sub twice (`Attempt to free unreferenced scalar`).

- A reschedule sub set with `reschedule_cb` on a watcher made without one is used; upstream never calls it, and clearing one makes every later firing fail.
- The watcher and the time a reschedule sub is called with are freed after the call; upstream leaks both, and the watcher with them.

Covered by `t/81-xs-native/EV/300-periodic-reschedule-cb.t`.

Functions

Schedule

again

Stop the watcher and start it again, computing its next firing time anew - after the clock or the reschedule sub's idea of time changed.

set

Set the whole schedule anew: `$w->set($at, $interval, $reschedule_cb)`, the interval and sub optional, as `EV::periodic` takes them. A negative interval croaks `interval value must be >= 0`.

at

When the watcher fires next, in seconds since the epoch.

offset

The `$at` the schedule counts from - the offset of the interval's multiples; with an argument, set a new one (from the next firing) and return the old.

interval

The interval, 0 for none; with an argument, set a new one (from the next firing) and return the old. A negative one croaks `interval value must be >= 0`.

reschedule_cb

The reschedule sub, undef for none; with an argument, use that sub (or none, given a false value) from now on and return the old one. The sub is called with the watcher and the current time and returns the next time to fire; it must not stop or change any periodic watcher or call into the loop. Unlike upstream, the sub returned stays the watcher's, and one set later is used (see *Differences from upstream* above; covered by `t/81-xs-native/EV/300-periodic-reschedule-cb.t`).

EV::Signal

A signal watcher, as `EV::signal` makes: its callback runs when the process receives a signal. These methods change which signal; the methods all watchers share are `EV::Watcher`'s.

A signal can be watched by any number of watchers, but in one loop only: `$w->start` in a second loop croaks `unable to start signal watcher, signal N already registered in another loop`. While a watcher of a signal is active, EV holds the signal; Perl's `%SIG` handler for it is not called.

Synopsis

```
my $w = EV::signal 'TERM', sub { EV::break };
print $w->signal;    # 15
$w->set('INT');
```

Functions

Configuration

set

Watch another signal: `$w->set('USR1')`, a name or number as for `kill`. One that is no signal croaks `illegal signal number or name`.

signal

The signal watched, always as a number; with an argument (name or number), watch that one instead and return the old number. The second names `IOT`, `CLD` and `POLL` are the signals `ABRT`, `CHLD` and `IO` (upstream mistakes them for nonexistent signals; covered by `t/81-xs-native/EV/310-signal-alias.t`).

EV::Stat

A file status watcher, as `EV::stat` makes: its callback runs when a path's status changes - created, deleted, written, its permissions or times changed. These methods read the status before and after and change what is watched; the methods all watchers share are `EV::Watcher`'s.

Synopsis

```
my $w = EV::stat '/var/log/app.log', 0, sub {
    my ($w) = @_;
    my $before = ($w->prev)[7];    # size before the change
    my $after  = ($w->attr)[7];    # size after
    print "grew by ", $after - $before, "\n" if $after > $before;
};
```

Functions

Status

attr

The path's status before the change that triggered the callback, as `stat` returns it. The same method has two other names:

Also available as `EV::Stat::prev`, `EV::Stat::stat`.

Configuration

set

Watch another path, with another polling interval: `$w->set($path, $interval)`, as `EV::stat` takes them.

path

The path watched; with an argument, watch that path instead and return the old one.

interval

The polling interval in use, in seconds - the default chosen when 0 was given; with an argument, set a new one and return the old.

attr

The path's status before the change that triggered the callback, as `stat` returns it. The same method has two other names:

Also available as `EV::Stat::prev`, `EV::Stat::stat`.

- `$w->attr` - the status that triggered the change, as EV last saw it; the difference from `prev` is what changed.
- `$w->stat` - stat the path now (with `lstat`), store that as what EV last saw (so it does not report a change you made yourself), and return it.

In list context each returns the 13 values `stat` does (the block size and block count are estimates), or an empty list when the path does not exist; in scalar context, whether it exists. All three also fill Perl's stat cache, for `-X _` tests, and a missing path sets `$!` to `ENOENT`.

EV::Timer

A relative timer, as `EV::timer` makes: its callback runs after a number of seconds, and optionally again at a repeat interval. These methods re-arm it and read or change its times; the methods all watchers share are `EV::Watcher`'s.

A timer counts on a clock that changes to the system time do not move. Starting one with a negative repeat interval croaks `repeat value must be >= 0`.

Synopsis

```
# an inactivity timeout: re-armed on every read
my $timeout = EV::timer 30, 30, sub { warn "idle for 30 s\n" };
my $reader  = EV::io $sock, EV::READ, sub { ...; $timeout->again };
printf "%.1f s left\n", $timeout->remaining;
```

Functions

Timing

again

Re-arm a repeating timer from now - the way to keep a timeout from firing while there is activity.

remaining

The seconds left until the timer fires, fractional; for an inactive timer, the time it was set to wait.

set

Set both times anew: `$w->set($after, $repeat)` (repeat default 0). An active timer restarts, counting `$after` from now.

repeat

The repeat interval; with an argument, set a new one and return the old. The timer is not restarted: an active timer uses the new interval from its next expiry.

again

Re-arm a repeating timer from now - the way to keep a timeout from firing while there is activity.

Synopsis

```
$w->again;          # restart the repeat interval from now
$w->again(10);      # with a new repeat interval
```

What it does

- An active repeating timer: fires the repeat interval after now.
- An inactive repeating timer: starts, firing the repeat interval after now.
- An active timer without repeat: stops.
- Otherwise nothing.

Make the timer with the same first and repeat time (`EV::timer 30, 30, ...`) and call `again` on each sign of activity.

Edge cases

- A negative new repeat croaks `repeat value must be >= 0`.

Differences from upstream

Fully compatible with upstream. Covered by `t/81-xs-native/EV/110-upstream-01_timer.t`.

See also

- `remaining` - how long until it fires
- `repeat` - the repeat interval alone

EV::Watcher

The methods every EV watcher has: start and stop it, ask whether it is active or pending, change its callback, data and priority, and decide whether it keeps the loop running.

Every watcher class - `EV::IO`, `EV::Timer`, `EV::Periodic`, `EV::Signal`, `EV::Child`, `EV::Stat`, `EV::Idle`, `EV::Prepare`, `EV::Check`, `EV::Fork`, `EV::Embed`, `EV::Async` - inherits these. Each class also has these two, with the same meaning everywhere:

- `$w->start` - start the watcher if it is not active; an active one is left as it is. Watchers start active, except those an `_ns` constructor made.
- `$w->stop` - stop the watcher if it is active, and in any case drop the events received for it whose callback has not run yet.

A watcher is stopped and freed when its last reference goes. A callback is called with the watcher and the events it received; the watcher can be stopped, changed or dropped from inside its own callback.

Synopsis

```
my $w = EV::timer 60, 0, sub { my ($w) = @_; warn $w->data };
$w->data('time is up');
$w->priority(1);                # before other pending watchers
$w->keepalive(0);               # EV::run may return while it waits
$w->stop;
$w->start if !$w->is_active;
```

Functions

State

`is_active`

True while the watcher is active - started and not stopped.

is_pending

True while the watcher has received events whose callback has not run yet.

keepalive

Whether the active watcher keeps `EV::run` from returning, and with an argument set it; returns the setting before, 1 or 0.

loop

The `EV::Loop` the watcher belongs to.

priority

The watcher's priority, from `EV::MINPRI` (-2) to `EV::MAXPRI` (2); with an argument, set a new one and return the old. Of the watchers pending at once, those of higher priority are called first. New watchers have priority 0; a value out of range is brought into it when the watcher starts. Setting it on an active watcher stops and restarts it, losing its pending events.

Callback and data

cb

The watcher's callback, as a code reference; with an argument, set a new one and return the old. The watcher is not restarted: the change can be made at any time, even from inside the callback. Something that is no code reference croaks.

data

A scalar of your own stored in the watcher, undef at first; with an argument, store a new one and return the old. The callback finds it as `$_[0]->data`.

Events

invoke

Call the watcher's callback now, with the events given (default `EV::NONE`): `$w->invoke(EV::TIMER)`.

clear_pending

Drop the events the watcher received whose callback has not run, and return them as a bitmask - as if the callback had run; 0 when nothing was pending.

feed_event

Act as if the watcher had received events: `$w->feed_event(EV::READ)` makes it pending, and its callback runs in the loop's next round.

keepalive

Whether the active watcher keeps `EV::run` from returning, and with an argument set it; returns the setting before, 1 or 0.

Synopsis

```
my $was = $w->keepalive;
$w->keepalive(0);
```

Edge cases

- `EV::run` returns when no active watcher with keepalive on is left. Turn it off for a watcher that should serve while there is other work but not keep the program waiting - a module's long-lived socket watcher, for one.
- New watchers have it on; it can change at any time.

Differences from upstream

Fully compatible with upstream. Covered by `t/81-xs-native/EV/130-upstream-03_keepalive.t`.

See also

- `EV::run` - the loop it concerns

loop

Run the event loop: wait for events and call the watchers' callbacks.

Also available as `EV::run`.

Synopsis

```
EV::run;                  # until EV::break, or no active watcher is left
EV::run EV::RUN_ONCE;     # wait for events once, handle them, return
EV::run EV::RUN_NOWAIT;   # handle what is ready now, without waiting
```

What you get back

True when it returned with work left - after `EV::break`, or with `EV::RUN_ONCE` or `EV::RUN_NOWAIT` - and false when no active watcher that keeps the loop alive was left.

Examples

Wait for input, at most five seconds:

```
my $in  = EV::io *STDIN, EV::READ, sub { print "input\n"; EV::break };
my $max = EV::timer 5, 0, sub { print "timeout\n"; EV::break };
EV::run;
```

Edge cases

- A watcher whose `keepalive` is off does not keep `EV::run` from returning.
- A callback may call `EV::run` again; the inner call returns first.
- `EV::loop` is the old name, from EV 3.

Differences from upstream

Fully compatible with upstream. Covered by `t/81-xs-native/EV/130-upstream-03_keepalive.t`.

See also

- `EV::break` - makes `EV::run` return
- `EV::once` - waits for one event without a watcher object

io

Watch a file handle: call the callback whenever it is readable or writable without blocking.

Also available as `EV::io_ns`.

Synopsis

```
my $w = EV::io $fh_or_fileno, EV::READ, sub { my ($w, $revents) = @_; ... };
my $s = EV::io_ns $sock, EV::READ | EV::WRITE, sub { ... };   # not started
```

What you get back

An `EV::IO` watcher, active (except from `EV::io_ns`) as long as the object lives. The callback gets the watcher and the events that occurred, `EV::READ`, `EV::WRITE` or both.

Examples

Read lines as they arrive:

```
my $w = EV::io *STDIN, EV::READ, sub {
    defined(my $line = <STDIN>) or return EV::break;
    print "got: $line";
};
EV::run;
```

Edge cases

- The handle may be a Perl file handle or a file descriptor number. One without a descriptor croaks `illegal file descriptor or filehandle`.
- EV tells when the descriptor is ready, not whether Perl's buffered handle has data: read with `sysread` (or one line per event, as above) so that buffered data is not left waiting.
- Several watchers may watch one descriptor.

Differences from upstream

Fully compatible with upstream. Covered by `t/81-xs-native/EV/190-upstream-09_brandon.t`.

See also

- `EV::IO` - change the handle or the events of a watcher
- `EV::once` - wait for one event without keeping a watcher

timer

Call the callback after a number of seconds, and optionally again and again after that.

Also available as `EV::timer_ns`.

Synopsis

```
my $once  = EV::timer 2.5, 0, sub { ... };    # in 2.5 seconds, once
my $every = EV::timer 0, 60, sub { ... };     # now, then every␣
 ↪minute
my $later = EV::timer_ns 10, 0, sub { ... };  # created stopped
```

What you get back

An `EV::Timer` watcher. The callback gets the watcher and `EV::TIMER`.

Edge cases

- Times count from the loop's time, `EV::now`, on a clock that system clock changes do not move; a timer for "at this wall-clock time" is `EV::periodic`.
- A repeating timer is restarted with the repeat interval after its callback; it fires at most once per loop iteration and may drift.
- A negative repeat croaks `repeat value must be >= 0`.

Differences from upstream

Fully compatible with upstream. Covered by `t/81-xs-native/EV/110-upstream-01_timer.t`.

See also

- `EV::Timer` - `again`, `remaining` and the repeat interval
- `EV::periodic` - timers on wall-clock time

periodic

Call the callback at wall-clock times: once at a given time, at every multiple of an interval, or whenever a sub of yours says.

Also available as `EV::periodic_ns`.

Synopsis

```
my $at     = EV::periodic $epoch_seconds, 0, 0, sub { ... };   #␣
↪once
my $hourly = EV::periodic 0, 3600, 0, sub { ... };             #␣
↪each full hour
my $custom = EV::periodic 0, 0, \&next_time, sub { ... };      # you␣
↪decide
```

What you get back

An `EV::Periodic` watcher. The callback gets the watcher and `EV::PERIODIC`.

The three modes

- `$interval` and `$reschedule_cb` 0: fire once when the system clock reaches `$at`, however the clock is set meanwhile.
- `$interval` above 0: fire at every `$at + N * $interval`, the next such time first (which may be before `$at`) - every full hour UTC above, without drift.

- `$reschedule_cb` a sub: it is called with the watcher and the current time and returns the next time to fire, at or after that time. It must not stop or change any periodic watcher or call into the loop; to stop, return `1e30` and stop the watcher afterwards.

Examples

Every midnight, local time:

```
use Time::Local ();
my $daily = EV::periodic 0, 0, sub {
    my (undef, $now) = @_;
    my (undef, undef, undef, $d, $m, $y) = localtime $now;
    Time::Local::timelocal_nocheck(0, 0, 0, $d + 1, $m, $y);
}, sub { print "midnight\n" };
```

Differences from upstream

`reschedule_cb` keeps its sub and the watcher alive; see `EV::Periodic`. Covered by `t/81-xs-native/EV/300-periodic-reschedule-cb.t`.

See also

- `EV::timer` - relative timeouts
- `EV::Periodic` - read and change a periodic watcher

once

Wait once for a handle to become ready, a timeout, or the first of the two, without keeping a watcher object.

Synopsis

```
EV::once $fh, EV::READ, 10, sub { my ($revents) = @_; ... };
EV::once undef, 0, 2.5, sub { ... };    # a plain timeout
```

What you get back

Nothing. When the handle is ready or the timeout passes, the callback is called once with the events received: a combination of `EV::READ`, `EV::WRITE`, `EV::TIMER` and `EV::ERROR`. Then both watchers are gone.

Edge cases

- `undef` as the handle (with 0 events) waits for the timeout only; an undef or negative timeout waits for the handle only.
- The callback receives only the events, not a watcher.

Differences from upstream

Fully compatible with upstream. Covered by `t/81-xs-native/EV/120-upstream-02_once.t`.

See also

- `EV::io` - a watcher for a handle, kept as long as needed
- `EV::timer` - a watcher for a timeout

5.2.34 Encode

Convert between Perl character strings and byte encodings.

```
use Encode qw(decode encode from_to find_encoding FB_CROAK);

my $chars = decode('UTF-8', "caf\xc3\xa9");          # bytes in,␣
↪characters out
printf "%d characters, U+%04X last\n", length $chars, ord substr
↪$chars, -1;
# 4 characters, U+00E9 last
printf "latin-1: %vX\n", encode('iso-8859-1', $chars);   # latin-1:␣
↪63.61.66.E9
printf "shiftjis: %vX\n", encode('shiftjis', "\x{65e5}\x{672c}");
↪# shiftjis: 93.FA.96.7B

my $data = "\xe9t\xe9";
from_to($data, 'iso-8859-1', 'utf8');                 # converts in␣
↪place
printf "from_to: %vX\n", $data;                       # from_to: C3.
↪A9.74.C3.A9
print eval { decode('UTF-8', "\xff", FB_CROAK); 1 } ? "decoded\n" :
↪"malformed: dies\n";
print find_encoding('latin1')->name, "\n";            # iso-8859-1
```

A string read from a file or socket is bytes; `decode` turns it into characters, and `encode` turns characters back into bytes for output. The third argument of either, CHECK, says what to do with something that cannot be converted: substitute (the default), die (`FB_CROAK`), warn, or stop and leave the rest (`FB_QUIET`). `encode_utf8` and `decode_utf8` are the UTF-8 shortcuts; `find_encoding` returns the encoding object, and `Encode->encodings(':all')` names every encoding available - 124 here, as in perl 5.44.

The encodings are perl 5.44's, from the same tables: `ascii`, `iso-8859-1`, `cp1252` and UTF-8 are built in; the others are in `Encode::Byte`, `CN`, `JP`, `KR`, `TW`, `Symbol`, `EBCDIC` and `Encode::Unicode`, which Encode loads the first time one of their encodings is named.

This is a native implementation of the XS half of Encode 3.24; `Encode.pm` and its Perl modules (`Encode::Alias`, `Encode::Encoding`, the MIME, Guess, JIS7, HZ and UTF-7 encodings) are used unchanged. `use Encode` exports `decode`, `encode`, `decode_utf8`, `encode_utf8`, `str2bytes`, `bytes2str`, `encodings`, `find_encoding`, `find_mime_encoding` and `clone_encoding`; `from_to`, `is_utf8` and the `FB_*` constants are imported

by name, the constants also with `:fallbacks`.

Differences from upstream

None.

Modules

- `Encode::Alias` - alias definitions to encodings
- `Encode::Byte` - the single-byte encodings beyond Latin-1: the ISO
- `Encode::CJKConstants` - Internally used by Encode::??::ISO_2022_*
- `Encode::CN` - the Simplified Chinese encodings: `euc-cn`, `cp936`
- `Encode::Config` - internally used by Encode
- `Encode::EBCDIC` - the EBCDIC encodings: `cp37`, `cp500`, `cp875`,
- `Encode::Encoder` - Object Oriented Encoder
- `Encode::Encoding` - Encode Implementation Base Class
- `Encode::GSM0338` - ETSI GSM 03.38 Encoding
- `Encode::Guess` - Guesses encoding from data
- `Encode::JP` - the Japanese encodings: `euc-jp`, `shiftjis`, `cp932`,
- `Encode::KR` - the Korean encodings: `euc-kr`, `cp949`, `johab`,
- `Encode::MIME`
- `Encode::Symbol` - the symbol-font encodings: `symbol`, `dingbats`,
- `Encode::TW` - the Traditional Chinese encodings: `big5-eten`,
- `Encode::Unicode` - The XS half of `Encode::Unicode`: the UTF-16, UTF-32 and UCS-2 codecs.
- `Encode::XS`
- `Encode::utf8`

Functions

Other Functions

`is_utf8`

`is_utf8($string [, $check])` - whether the string's UTF-8 flag is on; with a true `$check`, also whether its bytes are well-formed.

`_utf8_on`

`_utf8_on($string)` - turn the UTF-8 flag on without changing the bytes; returns the previous state, or undef for a tainted value or a non-string.

_utf8_off

`_utf8_off($string)` - turn the UTF-8 flag off without changing the bytes; returns the previous state, or undef for a tainted value or a non-string.

decode

`decode($encoding, $octets [, CHECK])` (also `bytes2str`) - decode bytes in the named encoding into a character string.

Also available as `Encode::bytes2str`.

encode

`encode($encoding, $string [, CHECK])` (also `str2bytes`) - encode a character string into bytes in the named encoding.

Also available as `Encode::str2bytes`.

decode_utf8

`decode_utf8($octets [, CHECK])` - decode UTF-8 octets (lax `utf8`).

encode_utf8

`encode_utf8($string)` - the string's characters as UTF-8 octets.

from_to

`from_to($octets, $from, $to [, CHECK])` - convert `$octets` in place from one encoding to another; returns the length of the result in bytes, or undef on failure.

onBOOT

`Encode::onBOOT` - register the built-in table encodings (`ascii`, `iso-8859-1`, `cp1252`, `null`, `ascii-ctrl`); Encode.pm calls it once, after `define_encoding` exists.

encodings

Return the list of encoding names the registry knows about.

find_encoding

Look up an encoding by name and return an object you can call methods on.

resolve_alias

Return the canonical encoding name for an alias, or `undef` if unknown.

Encode::Alias

alias definitions to encodings

SYNOPSIS

```
use Encode;
use Encode::Alias;
define_alias( "newName" => ENCODING);
define_alias( qr/.../ => ENCODING);
define_alias( sub { return ENCODING if ...; } );
```

DESCRIPTION

Allows newName to be used as an alias for ENCODING. ENCODING may be either the name of an encoding or an encoding object (as described in *Encode*).

Currently the first argument to define_alias() can be specified in the following ways:

- As a simple string.
- As a qr// compiled regular expression, e.g.:

  ```
  define_alias( qr/^iso8859-(\d+)$/i => '"iso-8859-$1"' );
  ```

  In this case, if *ENCODING* is not a reference, it is `eval`-ed in order to allow `$1` etc. to be substituted. The example is one way to alias names as used in X11 fonts to the MIME names for the iso-8859-* family. Note the double quotes inside the single quotes.

  (or, you don't have to do this yourself because this example is predefined)

  If you are using a regex here, you have to use the quotes as shown or it won't work. Also note that regex handling is tricky even for the experienced. Use this feature with caution.

- As a code reference, e.g.:

  ```
  define_alias( sub {shift =~ /^iso8859-(\d+)$/i ? "iso-8859-$1"␣
  ↪: undef } );
  ```

  The same effect as the example above in a different way. The coderef takes the alias name as an argument and returns a canonical name on success or undef if not. Note the second argument is ignored if provided. Use this with even more caution than the regex version.

Changes in code reference aliasing

As of Encode 1.87, the older form

```
define_alias( sub { return  /^iso8859-(\d+)$/i ? "iso-8859-$1" :␣
↪undef } );
```

no longer works.

Encode up to 1.86 internally used “local $_” to implement this older form. But consider the code below;

```
use Encode;
$_ = "eeeee" ;
while (/(e)/g) {
  my $utf = decode('aliased-encoding-name', $1);
  print "position:",pos,"\n";
}
```

Prior to Encode 1.86 this fails because of “local $_”.

Alias overloading

You can override predefined aliases by simply applying define_alias(). The new alias is always evaluated first, and when necessary, define_alias() flushes the internal cache to make the new definition available.

```
# redirect SHIFT_JIS to MS/IBM Code Page 932, which is a
# superset of SHIFT_JIS

define_alias( qr/shift.*jis$/i  => '"cp932"' );
define_alias( qr/sjis$/i        => '"cp932"' );
```

If you want to zap all predefined aliases, you can use

```
Encode::Alias->undef_aliases;
```

to do so. And

```
Encode::Alias->init_aliases;
```

gets the factory settings back.

Note that define_alias() will not be able to override the canonical name of encodings. Encodings are first looked up by canonical name before potential aliases are tried.

SEE ALSO

Encode, Encode::Supported

Encode::Byte

`Encode::Byte` - the single-byte encodings beyond Latin-1: the ISO 8859 family (`iso-8859-2` to `iso-8859-16`), the Windows code pages (`cp1250` to `cp1258`), the DOS code pages (`cp437`, `cp850`, `cp866`, ...), the KOI8 family (`koi8-r`, `koi8-u`, `koi8-f`), the Mac encodings (`MacRoman`, `MacCyrillic`, `MacGreek`, ...), `viscii`, `nextstep`, `hp-roman8` and `AdobeStandardEncoding` - 65 in all.

Nothing needs to be loaded: `Encode` loads this module the first time one of its encodings is named, as in `encode('koi8-r', $text)`.

This is a native implementation of Encode::Byte 2.04, from the same tables as upstream; `Encode/Byte.pm` is used unchanged.

Differences from upstream

None.

Encode::CJKConstants

Internally used by Encode::??::ISO_2022_*

Encode::CN

`Encode::CN` - the Simplified Chinese encodings: `euc-cn`, `cp936` (GBK), `hz`, `iso-ir-165`, `MacChineseSimp`, and the raw `gb2312-raw` and `gb12345-raw` tables.

Nothing needs to be loaded: `Encode` loads this module the first time one of its encodings is named, as in `decode('euc-cn', $bytes)`.

This is a native implementation of Encode::CN 2.03, from the same tables as upstream; `Encode/CN.pm` is used unchanged.

Differences from upstream

None.

Modules

- `Encode::CN::HZ` - internally used by Encode::CN

Encode::CN::HZ

internally used by Encode::CN

Encode::Config

internally used by Encode

Encode::EBCDIC

`Encode::EBCDIC` - the EBCDIC encodings: `cp37`, `cp500`, `cp875`, `cp1026`, `cp1047` and `posix-bc`.

Nothing needs to be loaded: `Encode` loads this module the first time one of its encodings is named, as in `decode('cp1047', $bytes)`.

This is a native implementation of Encode::EBCDIC 2.02, from the same tables as upstream; `Encode/EBCDIC.pm` is used unchanged.

Differences from upstream

None.

Encode::Encoder

Object Oriented Encoder

SYNOPSIS

```
use Encode::Encoder;
# Encode::encode("ISO-8859-1", $data);
Encode::Encoder->new($data)->iso_8859_1; # OOP way
# shortcut
use Encode::Encoder qw(encoder);
encoder($data)->iso_8859_1;
# you can stack them!
encoder($data)->iso_8859_1->base64;  # provided base64() is defined
# you can use it as a decoder as well
encoder($base64)->bytes('base64')->latin1;
# stringified
print encoder($data)->utf8->latin1;  # prints the string in latin1
# numified
encoder("\x{abcd}\x{ef}g")->utf8 == 6; # true. bytes::length($data)
```

ABSTRACT

Encode::Encoder allows you to use Encode in an object-oriented style. This is not only more intuitive than a functional approach, but also handier when you want to stack encodings. Suppose you want your UTF-8 string converted to Latin1 then Base64: you can simply say

```
my $base64 = encoder($utf8)->latin1->base64;
```

instead of

```
my $latin1 = encode("latin1", $utf8);
my $base64 = encode_base64($utf8);
```

or the lazier and more convoluted

```
my $base64 = encode_base64(encode("latin1", $utf8));
```

Description

Here is how to use this module.

- There are at least two instance variables stored in a hash reference, {data} and {encoding}.

- When there is no method, it takes the method name as the name of the encoding and encodes the instance *data* with *encoding*. If successful, the instance *encoding* is set accordingly.
- You can retrieve the result via ->data but usually you don't have to because the stringify operator ("") is overridden to do exactly that.

Predefined Methods

This module predefines the methods below:

- $e = Encode::Encoder->new([$data, $encoding]);

  returns an encoder object. Its data is initialized with $data if present, and its encoding is set to $encoding if present.

  When $encoding is omitted, it defaults to utf8 if $data is already in utf8 or "" (empty string) otherwise.

- encoder()

  is an alias of Encode::Encoder->new(). This one is exported on demand.

- $e->data([$data])

  When $data is present, sets the instance data to $data and returns the object itself. Otherwise, the current instance data is returned.

- $e->encoding([$encoding])

  When $encoding is present, sets the instance encoding to $encoding and returns the object itself. Otherwise, the current instance encoding is returned.

- $e->bytes([$encoding])

  decodes instance data from $encoding, or the instance encoding if omitted. If the conversion is successful, the instance encoding will be set to "".

  The name *bytes* was deliberately picked to avoid namespace tainting – this module may be used as a base class so method names that appear in Encode::Encoding are avoided.

Example: base64 transcoder

This module is designed to work with *Encode::Encoding*. To make the Base64 transcoder example above really work, you could write a module like this:

```
package Encode::Base64;
use parent 'Encode::Encoding';
__PACKAGE__->Define('base64');
use MIME::Base64;
sub encode{
    my ($obj, $data) = @_;
    return encode_base64($data);
}
sub decode{
    my ($obj, $data) = @_;
```

(continues on next page)

(continued from previous page)

```
    return decode_base64($data);
}
1;
__END__
```

And your caller module would be something like this:

```
use Encode::Encoder;
use Encode::Base64;

# now you can really do the following

encoder($data)->iso_8859_1->base64;
encoder($base64)->bytes('base64')->latin1;
```

Operator Overloading

This module overloads two operators, stringify ("") and numify (0+).

Stringify dumps the data inside the object.

Numify returns the number of bytes in the instance data.

They come in handy when you want to print or find the size of data.

SEE ALSO

Encode, *Encode::Encoding*

Encode::Encoding

Encode Implementation Base Class

SYNOPSIS

```
package Encode::MyEncoding;
use parent qw(Encode::Encoding);

__PACKAGE__->Define(qw(myCanonical myAlias));
```

DESCRIPTION

As mentioned in *Encode*, encodings are (in the current implementation at least) defined as objects. The mapping of encoding name to object is via the %Encode::Encoding hash. Though you can directly manipulate this hash, it is strongly encouraged to use this base class module and add encode() and decode() methods.

Methods you should implement

You are strongly encouraged to implement methods below, at least either encode() or decode().

- ->encode($string [,$check])

  MUST return the octet sequence representing *$string*.

  - If *$check* is true, it SHOULD modify *$string* in place to remove the converted part (i.e. the whole string unless there is an error). If perlio_ok() is true, SHOULD becomes MUST.
  - If an error occurs, it SHOULD return the octet sequence for the fragment of string that has been converted and modify $string in-place to remove the converted part leaving it starting with the problem fragment. If perlio_ok() is true, SHOULD becomes MUST.
  - If *$check* is false then encode MUST make a "best effort" to convert the string - for example, by using a replacement character.

- ->decode($octets [,$check])

  MUST return the string that *$octets* represents.

  - If *$check* is true, it SHOULD modify *$octets* in place to remove the converted part (i.e. the whole sequence unless there is an error). If perlio_ok() is true, SHOULD becomes MUST.
  - If an error occurs, it SHOULD return the fragment of string that has been converted and modify $octets in-place to remove the converted part leaving it starting with the problem fragment. If perlio_ok() is true, SHOULD becomes MUST.
  - If *$check* is false then decode should make a "best effort" to convert the string - for example by using Unicode's "\x{FFFD}" as a replacement character.

If you want your encoding to work with *encoding* pragma, you should also implement the method below.

- ->cat_decode($destination, $octets, $offset, $terminator [,$check])

  MUST decode *$octets* with *$offset* and concatenate it to *$destination*. Decoding will terminate when $terminator (a string) appears in output. *$offset* will be modified to the last $octets position at end of decode. Returns true if $terminator appears output, else returns false.

Other methods defined in Encode::Encodings

You do not have to override methods shown below unless you have to.

- ->name

  Predefined As:

  ```
  sub name  { return shift->{'Name'} }
  ```

  MUST return the string representing the canonical name of the encoding.

- ->mime_name

  Predefined As:

```
sub mime_name{
  return Encode::MIME::Name::get_mime_name(shift->name);
}
```

  MUST return the string representing the IANA charset name of the encoding.

- ->renew

  Predefined As:

```
sub renew {
  my $self = shift;
  my $clone = bless { %$self } => ref($self);
  $clone->{renewed}++;
  return $clone;
}
```

  This method reconstructs the encoding object if necessary. If you need to store the state during encoding, this is where you clone your object.

  PerlIO ALWAYS calls this method to make sure it has its own private encoding object.

- ->renewed

  Predefined As:

```
sub renewed { $_[0]->{renewed} || 0 }
```

  Tells whether the object is renewed (and how many times). Some modules emit `Use of uninitialized value in null operation` warning unless the value is numeric so return 0 for false.

- ->perlio_ok()

  Predefined As:

```
sub perlio_ok {
  return eval { require PerlIO::encoding } ? 1 : 0;
}
```

  If your encoding does not support PerlIO for some reasons, just;

```
sub perlio_ok { 0 }
```

- ->needs_lines()

  Predefined As:

```
sub needs_lines { 0 };
```

If your encoding can work with PerlIO but needs line buffering, you MUST define this method so it returns true. 7bit ISO-2022 encodings are one example that needs this. When this method is missing, false is assumed.

Example: Encode::ROT13

```
package Encode::ROT13;
use strict;
use parent qw(Encode::Encoding);

__PACKAGE__->Define('rot13');

sub encode($$;$){
    my ($obj, $str, $chk) = @_;
    $str =~ tr/A-Za-z/N-ZA-Mn-za-m/;
    $_[1] = '' if $chk; # this is what in-place edit means
    return $str;
}

# Jr pna or ynml yvxr guvf;
*decode = \&encode;

1;
```

Why the heck Encode API is different?

It should be noted that the *$check* behaviour is different from the outer public API. The logic is that the "unchecked" case is useful when the encoding is part of a stream which may be reporting errors (e.g. STDERR). In such cases, it is desirable to get everything through somehow without causing additional errors which obscure the original one. Also, the encoding is best placed to know what the correct replacement character is, so if that is the desired behaviour then letting low level code do it is the most efficient.

By contrast, if *$check* is true, the scheme above allows the encoding to do as much as it can and tell the layer above how much that was. What is lacking at present is a mechanism to report what went wrong. The most likely interface will be an additional method call to the object, or perhaps (to avoid forcing per-stream objects on otherwise stateless encodings) an additional parameter.

It is also highly desirable that encoding classes inherit from `Encode::Encoding` as a base class. This allows that class to define additional behaviour for all encoding objects.

```
package Encode::MyEncoding;
use parent qw(Encode::Encoding);

__PACKAGE__->Define(qw(myCanonical myAlias));
```

to create an object with `bless {Name => ...}, $class`, and call define_encoding. They inherit their `name` method from `Encode::Encoding`.

Compiled Encodings

Encode provides the enc2xs tool to achieve that. Please see enc2xs for more details.

SEE ALSO

perlmod, enc2xs

Encode::GSM0338

ETSI GSM 03.38 Encoding

SYNOPSIS

```
use Encode qw/encode decode/;
$gsm0338 = encode("gsm0338", $unicode); # loads Encode::GSM0338␣
↪implicitly
$unicode = decode("gsm0338", $gsm0338); # ditto
```

DESCRIPTION

GSM0338 is for GSM handsets. Though it shares alphanumerals with ASCII, control character ranges and other parts are mapped very differently, mainly to store Greek characters. There are also escape sequences (starting with 0x1B) to cover e.g. the Euro sign.

This was once handled by Encode::Bytes but because of all those unusual specifications, Encode 2.20 has relocated the support to this module.

This module implements only *GSM 7 bit Default Alphabet* and *GSM 7 bit default alphabet extension table* according to standard 3GPP TS 23.038 version 16. Therefore *National Language Single Shift* and *National Language Locking Shift* are not implemented nor supported.

Septets

This modules operates with octets (like any other Encode module) and not with packed septets (unlike other GSM standards). Therefore for processing binary SMS or parts of GSM TPDU payload (3GPP TS 23.040) it is needed to do conversion between octets and packed septets. For this purpose perl's `pack` and `unpack` functions may be useful:

```
$bytes = substr(pack('(b*)*', unpack '(A7)*', unpack 'b*',
↪$septets), 0, $num_of_septets);
$unicode = decode('GSM0338', $bytes);

$bytes = encode('GSM0338', $unicode);
$septets = pack 'b*', join '', map { substr $_, 0, 7 } unpack '(A8)*
↪', unpack 'b*', $bytes;
$num_of_septets = length $bytes;
```

Please note that for correct decoding of packed septets it is required to know number of septets packed in binary buffer as binary buffer is always padded with zero bits and 7

zero bits represents character @. Number of septets is also stored in TPDU payload when dealing with 3GPP TS 23.040.

BUGS

Encode::GSM0338 2.7 and older versions (part of Encode 3.06) incorrectly handled zero bytes (character @). This was fixed in Encode::GSM0338 version 2.8 (part of Encode 3.07).

SEE ALSO

3GPP TS 23.038

ETSI TS 123 038 V16.0.0 (2020-07)

Encode

Encode::Guess

Guesses encoding from data

SYNOPSIS

```
# if you are sure $data won't contain anything bogus

use Encode;
use Encode::Guess qw/euc-jp shiftjis 7bit-jis/;
my $utf8 = decode("Guess", $data);
my $data = encode("Guess", $utf8);   # this doesn't work!

# more elaborate way
use Encode::Guess;
my $enc = guess_encoding($data, qw/euc-jp shiftjis 7bit-jis/);
ref($enc) or die "Can't guess: $enc"; # trap error this way
$utf8 = $enc->decode($data);
# or
$utf8 = decode($enc->name, $data)
```

ABSTRACT

Encode::Guess enables you to guess in what encoding a given data is encoded, or at least tries to.

DESCRIPTION

By default, it checks only ascii, utf8 and UTF-16/32 with BOM.

```
use Encode::Guess; # ascii/utf8/BOMed UTF
```

To use it more practically, you have to give the names of encodings to check (*suspects* as follows). The name of suspects can either be canonical names or aliases.

CAVEAT: Unlike UTF-(16|32), BOM in utf8 is NOT AUTOMATICALLY STRIPPED.

```
# tries all major Japanese Encodings as well
 use Encode::Guess qw/euc-jp shiftjis 7bit-jis/;
```

If the `$Encode::Guess::NoUTFAutoGuess` variable is set to a true value, no heuristics will be applied to UTF8/16/32, and the result will be limited to the suspects and `ascii`.

- Encode::Guess->set_suspects

  You can also change the internal suspects list via `set_suspects` method.

  ```
  use Encode::Guess;
  Encode::Guess->set_suspects(qw/euc-jp shiftjis 7bit-jis/);
  ```

- Encode::Guess->add_suspects

  Or you can use `add_suspects` method. The difference is that `set_suspects` flushes the current suspects list while `add_suspects` adds.

  ```
  use Encode::Guess;
  Encode::Guess->add_suspects(qw/euc-jp shiftjis 7bit-jis/);
  # now the suspects are euc-jp,shiftjis,7bit-jis, AND
  # euc-kr,euc-cn, and big5-eten
  Encode::Guess->add_suspects(qw/euc-kr euc-cn big5-eten/);
  ```

- Encode::decode("Guess" ...)

  When you are content with suspects list, you can now

  ```
  my $utf8 = Encode::decode("Guess", $data);
  ```

- Encode::Guess->guess($data)

  But it will croak if:

  - Two or more suspects remain
  - No suspects left

  So you should instead try this;

  ```
  my $decoder = Encode::Guess->guess($data);
  ```

  On success, $decoder is an object that is documented in *Encode::Encoding*. So you can now do this;

  ```
  my $utf8 = $decoder->decode($data);
  ```

  On failure, $decoder now contains an error message so the whole thing would be as follows;

  ```
  my $decoder = Encode::Guess->guess($data);
  die $decoder unless ref($decoder);
  my $utf8 = $decoder->decode($data);
  ```

- guess_encoding($data, [, *list of suspects*])

  You can also try `guess_encoding` function which is exported by default. It takes $data to check and it also takes the list of suspects by option. The optional suspect list is *not reflected* to the internal suspects list.

```
my $decoder = guess_encoding($data, qw/euc-jp euc-kr euc-cn/);
die $decoder unless ref($decoder);
my $utf8 = $decoder->decode($data);
# check only ascii, utf8 and UTF-(16|32) with BOM
my $decoder = guess_encoding($data);
```

CAVEATS

- Because of the algorithm used, ISO-8859 series and other single-byte encodings do not work well unless either one of ISO-8859 is the only one suspect (besides ascii and utf8).

```
use Encode::Guess;
# perhaps ok
my $decoder = guess_encoding($data, 'latin1');
# definitely NOT ok
my $decoder = guess_encoding($data, qw/latin1 greek/);
```

  The reason is that Encode::Guess guesses encoding by trial and error. It first splits $data into lines and tries to decode the line for each suspect. It keeps it going until all but one encoding is eliminated out of suspects list. ISO-8859 series is just too successful for most cases (because it fills almost all code points in \x00-\xff).

- Do not mix national standard encodings and the corresponding vendor encodings.

```
# a very bad idea
my $decoder
   = guess_encoding($data, qw/shiftjis MacJapanese cp932/);
```

  The reason is that vendor encoding is usually a superset of national standard so it becomes too ambiguous for most cases.

- On the other hand, mixing various national standard encodings automagically works unless $data is too short to allow for guessing.

```
# This is ok if $data is long enough
my $decoder =
 guess_encoding($data, qw/euc-cn
                          euc-jp shiftjis 7bit-jis
                          euc-kr
                          big5-eten/);
```

- DO NOT PUT TOO MANY SUSPECTS! Don't you try something like this!

```
my $decoder = guess_encoding($data,
                             Encode->encodings(":all"));
```

It is, after all, just a guess. You should alway be explicit when it comes to encodings. But there are some, especially Japanese, environment that guess-coding is a must. Use this module with care.

TO DO

Encode::Guess does not work on EBCDIC platforms.

SEE ALSO

Encode, *Encode::Encoding*

Encode::JP

`Encode::JP` - the Japanese encodings: `euc-jp`, `shiftjis`, `cp932`, `iso-2022-jp`, `iso-2022-jp-1`, `7bit-jis`, `MacJapanese`, and the raw `jis0201-raw`, `jis0208-raw` and `jis0212-raw` tables.

Nothing needs to be loaded: `Encode` loads this module the first time one of its encodings is named, as in `encode('shiftjis', $text)`.

This is a native implementation of Encode::JP 2.05, from the same tables as upstream; `Encode/JP.pm` is used unchanged.

Differences from upstream

None.

Modules

- `Encode::JP::H2Z` - internally used by Encode::JP::2022_JP*
- `Encode::JP::JIS7` - internally used by Encode::JP

Encode::JP::H2Z

internally used by Encode::JP::2022_JP*

Encode::JP::JIS7

internally used by Encode::JP

Encode::KR

`Encode::KR` - the Korean encodings: `euc-kr`, `cp949`, `johab`, `iso-2022-kr`, `MacKorean` and the raw `ksc5601-raw` table.

Nothing needs to be loaded: `Encode` loads this module the first time one of its encodings is named, as in `decode('euc-kr', $bytes)`.

This is a native implementation of Encode::KR 2.03, from the same tables as upstream; `Encode/KR.pm` is used unchanged.

Differences from upstream

None.

Modules

- `Encode::KR::2022_KR` - internally used by Encode::KR

Encode::KR::2022_KR

internally used by Encode::KR

Encode::MIME

Namespace containing the following modules:

- `Encode::MIME::Header` - MIME encoding for an unstructured email header
- `Encode::MIME::Name` - internally used by Encode

Encode::MIME::Header

MIME encoding for an unstructured email header

SYNOPSIS

```
use Encode qw(encode decode);

my $mime_str = encode("MIME-Header", "Sample:Text \N{U+263A}");
# $mime_str is "=?UTF-8?B?U2FtcGxlOlRleHQg4pi6?="

my $mime_q_str = encode("MIME-Q", "Sample:Text \N{U+263A}");
# $mime_q_str is "=?UTF-8?Q?Sample=3AText_=E2=98=BA?="

my $str = decode("MIME-Header",
    "=?ISO-8859-1?B?SWYgeW91IGNhbiByZWFkIHRoaXMgeW8=?=\r\n " .
    "=?ISO-8859-2?B?dSB1bmRlcnN0YW5kIHRoZSBleGFtcGxlLg==?="
);
# $str is "If you can read this you understand the example."

use Encode qw(decode :fallbacks);
use Encode::MIME::Header;
local $Encode::MIME::Header::STRICT_DECODE = 1;
my $strict_string = decode("MIME-Header", $mime_string, FB_CROAK);
# use strict decoding and croak on errors
```

ABSTRACT

This module implements RFC 2047 MIME encoding for an unstructured field body of the email header. It can also be used for RFC 822 'text' token. However, it cannot be used directly for the whole header with the field name or for the structured header fields like From, To, Cc, Message-Id, etc... There are 3 encoding names supported by this module: `MIME-Header`, `MIME-B` and `MIME-Q`.

DESCRIPTION

Decode method takes an unstructured field body of the email header (or RFC 822 'text' token) as its input and decodes each MIME encoded-word from input string to a sequence of bytes according to RFC 2047 and RFC 2231. Subsequently, each sequence of bytes with the corresponding MIME charset is decoded with *the Encode module* and finally, one output string is returned. Text parts of the input string which do not contain MIME encoded-word stay unmodified in the output string. Folded newlines between two consecutive MIME encoded-words are discarded, others are preserved in the output string. `MIME-B` can decode Base64 variant, `MIME-Q` can decode Quoted-Printable variant and `MIME-Header` can decode both of them. If *Encode module* does not support particular MIME charset or chosen variant then an action based on *CHECK flags* is performed (by default, the MIME encoded-word is not decoded).

Encode method takes a scalar string as its input and uses *strict UTF-8* encoder for encoding it to UTF-8 bytes. Then a sequence of UTF-8 bytes is encoded into MIME encoded-words (`MIME-Header` and `MIME-B` use a Base64 variant while `MIME-Q` uses a Quoted-Printable variant) where each MIME encoded-word is limited to 75 characters. MIME encoded-words are separated by `CRLF SPACE` and joined to one output string. Output string is suitable for unstructured field body of the email header.

Both encode and decode methods propagate *CHECK flags* when encoding and decoding the MIME charset.

BUGS

Versions prior to 2.22 (part of Encode 2.83) have a malfunctioning decoder and encoder. The MIME encoder infamously inserted additional spaces or discarded white spaces between consecutive MIME encoded-words, which led to invalid MIME headers produced by this module. The MIME decoder had a tendency to discard white spaces, incorrectly interpret data or attempt to decode Base64 MIME encoded-words as Quoted-Printable. These problems were fixed in version 2.22. It is highly recommended not to use any version prior 2.22!

Versions prior to 2.24 (part of Encode 2.87) ignored *CHECK flags*. The MIME encoder used *not strict utf8* encoder for input Unicode strings which could lead to invalid UTF-8 sequences. MIME decoder used also *not strict utf8* decoder and additionally called the decode method with a `Encode::FB_PERLQQ` flag (thus user-specified *CHECK flags* were ignored). Moreover, it automatically croaked when a MIME encoded-word contained unknown encoding. Since version 2.24, this module uses *strict UTF-8* encoder and decoder. And *CHECK flags* are correctly propagated.

Since version 2.22 (part of Encode 2.83), the MIME encoder should be fully compliant to RFC 2047 and RFC 2231. Due to the aforementioned bugs in previous versions of the MIME encoder, there is a *less strict* compatible mode for the MIME decoder which is

used by default. It should be able to decode MIME encoded-words encoded by pre 2.22 versions of this module. However, note that this is not correct according to RFC 2047.

In default *not strict* mode the MIME decoder attempts to decode every substring which looks like a MIME encoded-word. Therefore, the MIME encoded-words do not need to be separated by white space. To enforce a correct *strict* mode, set variable `$Encode::MIME::Header::STRICT_DECODE` to 1 e.g. by localizing:

```
use Encode::MIME::Header;
local $Encode::MIME::Header::STRICT_DECODE = 1;
```

AUTHORS

Pali pali@cpan.org

SEE ALSO

Encode, RFC 822, RFC 2047, RFC 2231

Encode::MIME::Name

internally used by Encode

SEE ALSO

I18N::Charset

Encode::Symbol

`Encode::Symbol` - the symbol-font encodings: `symbol`, `dingbats`, `AdobeSymbol`, `AdobeZdingbat`, `MacSymbol` and `MacDingbats`.

Nothing needs to be loaded: `Encode` loads this module the first time one of its encodings is named, as in `decode('symbol', $bytes)`.

This is a native implementation of Encode::Symbol 2.02, from the same tables as upstream; `Encode/Symbol.pm` is used unchanged.

Differences from upstream

None.

Encode::TW

`Encode::TW` - the Traditional Chinese encodings: `big5-eten`, `big5-hkscs`, `cp950` and `MacChineseTrad`.

Nothing needs to be loaded: `Encode` loads this module the first time one of its encodings is named, as in `decode('big5-eten', $bytes)`.

This is a native implementation of Encode::TW 2.03, from the same tables as upstream; `Encode/TW.pm` is used unchanged.

Differences from upstream

None.

Encode::Unicode

The XS half of `Encode::Unicode`: the UTF-16, UTF-32 and UCS-2 codecs.

`Encode/Unicode.pm` is upstream's. It creates one blessed hash per encoding - `Name`, `size`, `endian`, `ucs2` - and registers it with `Encode::define_encoding`. The two methods below read those attributes and convert, exactly as `Unicode.xs` does: BOM detection and big-endian fallback for the BOM-less `UTF-16`/`UTF-32`, surrogate pairs, the U+FFFD substitution, and the CHECK handling (croak, warn, leave or consume the source).

```
use Encode;
printf "%vX\n", encode('UTF-16LE', "hi");           # 68.0.69.0
print decode('UTF-16', "\xFE\xFF\0A\0B"), "\n";     # AB (the BOM␣
↪says big-endian)
printf "%vX\n", encode('UTF-16', "\x{1F600}");      # FE.FF.D8.3D.DE.
↪0: BOM, surrogate pair
```

The encodings are `UTF-16`, `UTF-16BE`, `UTF-16LE`, `UTF-32`, `UTF-32BE`, `UTF-32LE`, `UCS-2BE` and `UCS-2LE`; `Encode` loads this module the first time one of them is named. This is a native implementation of the XS half of Encode::Unicode 2.20.

Differences from upstream

None.

Modules

- `Encode::Unicode::UTF7` - UTF-7 encoding

Functions

Other Functions

decode

`$obj->decode($octets [, $check])` - UTF-16/32 or UCS-2 octets to a character string.

encode

`$obj->encode($string [, $check])` - a character string to UTF-16/32 or UCS-2 octets.

Encode::Unicode::UTF7

UTF-7 encoding

SYNOPSIS

```
use Encode qw/encode decode/;
$utf7 = encode("UTF-7", $utf8);
$utf8 = decode("UTF-7", $ucs2);
```

ABSTRACT

This module implements UTF-7 encoding documented in RFC 2152. UTF-7, as its name suggests, is a 7-bit re-encoded version of UTF-16BE. It is designed to be MTA-safe and expected to be a standard way to exchange Unicoded mails via mails. But with the advent of UTF-8 and 8-bit compliant MTAs, UTF-7 is hardly ever used.

UTF-7 was not supported by Encode until version 1.95 because of that. But Unicode::String, a module by Gisle Aas which adds Unicode supports to non-utf8-savvy perl did support UTF-7, the UTF-7 support was added so Encode can supersede Unicode::String 100%.

In Practice

When you want to encode Unicode for mails and web pages, however, do not use UTF-7 unless you are sure your recipients and readers can handle it. Very few MUAs and WWW Browsers support these days (only Mozilla seems to support one). For general cases, use UTF-8 for message body and MIME-Header for header instead.

SEE ALSO

Encode, *Encode::Unicode*, Unicode::String

RFC 2781 http://www.ietf.org/rfc/rfc2152.txt

decode

`$obj->decode($octets [, $check])` - UTF-16/32 or UCS-2 octets to a character string.

The byte order comes from the object's `endian`; with none (`UTF-16`, `UTF-32`) a leading BOM decides it and a missing one means big-endian. Surrogate pairs are combined; a lone or malformed surrogate, a surrogate in UCS-2 or UTF-32, and a non-character become U+FFFD - or croak with `FB_CROAK`, or warn with `FB_WARN`. A true CHECK without `LEAVE_SRC` leaves the undecoded tail in `$octets`.

encode

`$obj->encode($string [, $check])` - a character string to UTF-16/32 or UCS-2 octets.

With no fixed byte order (`UTF-16`, `UTF-32`) the output starts with a big-endian BOM. Characters above U+FFFF become surrogate pairs in UTF-16; in UCS-2, and anything above U+10FFFF, they become U+FFFD - or croak with `FB_CROAK`, or warn with `FB_WARN`. A partial UTF-8 character at the end croaks when CHECK has `DIE_ON_ERR` or `LEAVE_SRC`.

Encode::XS

Functions

Other Functions

`renew`

`$enc->renew` - a new reference to the same encoding object (Encode::XS encodings are not cloned).

`renewed`

`$enc->renewed` - always 0 for a table-driven encoding.

`name`

`$enc->name` - the encoding's canonical name.

`cat_decode`

`$enc->cat_decode($dst, $src, $off, $term [, CHECK])` - decode `$src` from byte offset `$off` onto the end of `$dst`, stopping after the terminator `$term`; updates `$off` and returns true if the terminator was found.

`decode`

`$enc->decode($octets [, CHECK])` - decode bytes of this encoding to a character string.

`encode`

`$enc->encode($string [, CHECK])` - encode a character string into bytes of this encoding.

`needs_lines`

`$enc->needs_lines` - false: a table-driven encoding carries no state across lines.

`perlio_ok`

`$enc->perlio_ok` - true if the `:encoding()` layer can use this encoding, which is whether PerlIO::encoding loads.

mime_name

`$enc->mime_name` - the encoding's MIME charset name, from Encode::MIME::Name.

Encode::utf8

Functions

Other Functions

decode

`$utf8->decode($octets [, CHECK])` - decode UTF-8 octets, lax for `utf8` and strict for `UTF-8`, applying CHECK to malformations.

encode

`$utf8->encode($string [, CHECK])` - encode characters as UTF-8. The strict `UTF-8` validates the string first and applies CHECK to what is not interchangeable UTF-8; lax `utf8` copies its octets.

encodings

Return the list of encoding names the registry knows about.

Synopsis

```
my @all = Encode->encodings();      # class method - the documented
↪form
my @all = Encode->encodings(':all');
```

Names are sorted case-insensitively and deduplicated; each canonical name appears exactly once regardless of how many aliases it has, and `Internal`, `Unicode` and `Guess` are excluded as upstream does. Use `resolve_alias` or `find_encoding` to turn any of these into a canonical name or an encoding object.

Only the encodings currently LOADED are reported. An encoding becomes loaded the first time any of its names is resolved, and it arrives with the rest of its module: `find_encoding("cp1251")` takes the list from seven names to seventy-one, because it requires `Encode::Byte`. `:all` merges in everything still dormant; a module name merges in that module's share without loading it.

Differences from upstream

Fully compatible with upstream.

find_encoding

Look up an encoding by name and return an object you can call methods on.

Synopsis

```
my $enc = find_encoding($name);
my $bytes  = $enc->encode($string);
my $string = $enc->decode($bytes);
my $canon  = $enc->name;
```

What you get back

A blessed encoding object: `Encode::utf8` for the UTF-8 family, `Encode::Unicode` for UTF-16 and UTF-32, `Encode::XS` for the table-driven encodings, or the class of whichever module defines the encoding. If the name cannot be resolved, `find_encoding` returns `undef`.

The object provides the `Encode::Encoding` methods, among them `encode`, `decode`, `name` (returns the canonical string), and `perlio_ok` (true when the `:encoding()` layer can use the encoding). `renew` returns the same object for an `Encode::XS` encoding and a copy for the others.

Name resolution is case-insensitive and treats `-` and `_` interchangeably. “utf-8”, “UTF-8”, and “utf_8” all resolve to the same object; “UTF8” without the dash is the lax `utf8` encoding, a different one.

Examples

Cache an encoding object and reuse it in a loop:

```
my $enc = find_encoding('UTF-8');
while (my $line = <$fh>) {
    my $string = $enc->decode($line);
    # ... work with $string ...
}
```

Query by alias:

```
my $enc = find_encoding('latin1');
print $enc->name;   # iso-8859-1
```

Edge cases

- Unknown names return `undef` after falling back to `Encode::Alias::find_alias` for aliases registered with `define_alias`.
- An `Encode::XS` object is a blessed scalar reference with no fields; the others are blessed hashes with a `Name` key. Code that pokes inside is not portable across encodings.

Differences from upstream

Fully compatible with upstream.

See also

- `resolve_alias` - get just the canonical name string.
- `encodings` - list every name the registry knows.
- `encode`, `decode` - the function forms, if you don't want to hold an object.

5.2.35 English

use nice English (or awk) names for ugly punctuation variables

SYNOPSIS

```
use English;
use English qw( -no_match_vars ) ;  # Avoids regex performance
                                    # penalty in perl 5.18 and
                                    # earlier
...
if ($ERRNO =~ /denied/) { ... }
```

DESCRIPTION

This module provides aliases for the built-in variables whose names no one seems to like to read. Variables with side-effects which get triggered just by accessing them (like $0) will still be affected.

For those variables that have an **awk** version, both long and short English alternatives are provided. For example, the `$/` variable can be referred to either $RS or $INPUT_RECORD_SEPARATOR if you are using the English module.

See *perlvar* for a complete list of these.

PERFORMANCE

NOTE: This was fixed in perl 5.20. Mentioning these three variables no longer makes a speed difference. This section still applies if your code is to run on perl 5.18 or earlier.

This module can provoke sizeable inefficiencies for regular expressions, due to unfortunate implementation details. If performance matters in your application and you don't need $PREMATCH, $MATCH, or $POSTMATCH, try doing

```
use English qw( -no_match_vars ) ;
```

. **It is especially important to do this in modules to avoid penalizing all applications which use them.**

5.2.36 Env

perl module that imports environment variables as scalars or arrays

SYNOPSIS

```
use Env;
use Env qw(PATH HOME TERM);
use Env qw($SHELL @LD_LIBRARY_PATH);
```

DESCRIPTION

Perl maintains environment variables in a special hash named `%ENV`. For when this access method is inconvenient, the Perl module `Env` allows environment variables to be treated as scalar or array variables.

The `Env::import()` function ties environment variables with suitable names to global Perl variables with the same names. By default it ties all existing environment variables (`keys %ENV`) to scalars. If the `import` function receives arguments, it takes them to be a list of variables to tie; it's okay if they don't yet exist. The scalar type prefix '$' is inferred for any element of this list not prefixed by '$' or '@'. Arrays are implemented in terms of `split` and `join`, using `$Config::Config{path_sep}` as the delimiter.

After an environment variable is tied, merely use it like a normal variable. You may access its value

```
@path = split(/:/, $PATH);
print join("\n", @LD_LIBRARY_PATH), "\n";
```

or modify it

```
$PATH .= ":/any/path";
push @LD_LIBRARY_PATH, $dir;
```

however you'd like. Bear in mind, however, that each access to a tied array variable requires splitting the environment variable's string anew.

The code:

```
use Env qw(@PATH);
push @PATH, '/any/path';
```

is almost equivalent to:

```
use Env qw(PATH);
$PATH .= ":/any/path";
```

except that if `$ENV{PATH}` started out empty, the second approach leaves it with the (odd) value ":`/any/path`", but the first approach leaves it with "`/any/path`".

To remove a tied environment variable from the environment, assign it the undefined value

```
undef $PATH;
undef @LD_LIBRARY_PATH;
```

LIMITATIONS

On VMS systems, arrays tied to environment variables are read-only. Attempting to change anything will cause a warning.

AUTHOR

Chip Salzenberg <chip@fin.uucp> and Gregor N. Purdy <gregor@focusresearch.com>

5.2.37 Errno

Symbolic names for the `errno(3)` values your system uses - `EINTR`, `EAGAIN`, `ENOENT`, `EACCES`, `EEXIST`, `EIO`, `ENOTDIR`, `EISDIR`, `EBUSY`, `EPIPE`, `ECONNRESET`, `ECONNREFUSED`, `ETIMEDOUT`, and every other errno name the host libc defines.

Most scripts reach for `Errno` to compare `$!` against a specific error after a failed system call, or to probe `%!` for the same purpose without touching `$!` by name.

Synopsis

```
use Errno qw(EINTR EAGAIN);

if ($! == EINTR) { retry() }     # function form
if ($!{EAGAIN})  { backoff() }   # hash form via %!
```

The names and their numbers come from the errno.h of the host pperl was built on, and from the headers it includes, so scripts see exactly the errnos the kernel will actually raise. The `:POSIX` tag groups the POSIX-standard names for bulk import; nothing is exported by default.

Upstream's `Errno.pm` is generated when perl is built, from those headers (ext/Errno/Errno_pm.PL); pperl's build reads them the same way (src/build_errno.rs), so this native implementation of Errno 1.39 has the same names with the same values, the ones the libc crate lacks included.

Differences from upstream

None.

5.2.38 Event

`Event` - an event loop: watchers for timers, file descriptors, signals, variables, idle time, groups and custom event sources, each calling back into Perl when its event happens.

```
use Event qw(loop unloop);

Event->timer(after => 1, interval => 1, cb => sub { print "tick\n" }
↪);
Event->io(fd => \*STDIN, poll => 'r', cb => sub {
    my $line = <STDIN>;
    unloop() unless defined $line;
});
loop();
```

Each `Event->TYPE(%attributes)` creates a watcher and starts it (unless `parked => 1`). When the watched condition occurs, an event is queued at the watcher's priority and `loop` runs its callback with the event as argument; `$event->w` is the watcher, `$event->hits` the number of occurrences it stands for. Watchers are stopped, restarted, suspended or cancelled with the methods of the same names.

The watcher types are `Event::idle`, `Event::timer`, `Event::io`, `Event::var` and `Event::signal`, which `use Event` loads, and `Event::group` and `Event::generic`, which are loaded with their own `use`. Their shared methods are `Event::Watcher`'s; events are `Event::Event` objects; `Event::Watcher::Tied` is the base for watcher types written in Perl.

Timers take fractional seconds when Time::HiRes is available, which `use Event` loads by default; `use Event qw(NO_TIME_HIRES)` prevents that.

Other XS modules can drive the loop from C through `$Event::API` (EventAPI.h, version 22), as Coro::Event does.

This is a native implementation of the CPAN module Event 1.28; its `Event.pm` and every `Event::*` module are used unchanged.

Exports

Nothing by default. On request: `time`, `all_watchers`, `all_running`, `all_idle`, `one_event`, `sweep`, `loop`, `unloop`, `unloop_all`, `sleep`, `queue`, `queue_pending`, `QUEUES`, `PRIO_NORMAL`, `PRIO_HIGH`. (`all_events` and `all_queued` are listed too, but defined nowhere.)

Functions written in Perl

From `Event.pm`:

- `loop()` / `loop($timeout)` - run callbacks until `unloop` or `unloop_all`, and return the value given to it; with `$timeout`, stop after that many seconds, returning `$timeout`. Loops nest. A callback that dies does not end the loop: the error goes to `$Event::DIED` and the loop goes on. With no active watcher left it warns "Event: loop without active watchers" and returns.
- `sweep()` / `sweep($max_prio)` - queue what has happened (queue_pending) and run every queued event of priority better than `$max_prio`, without waiting; by default all events but those of idle watchers. Errors are trapped as in `loop`.
- `Event->add_hooks(name => sub { ... }, ...)` - Perl code run at fixed points of each round: `prepare` (before waiting; returns the longest time to wait),

`check` (after waiting), `asynccheck` (after collecting signals), `callback` (before each callback). Hooks slow the loop down considerably.

- `Event::verbose_exception_handler` - a `$Event::DIED` handler that also lists the attributes of the failing watcher.

Priorities

Events are run in priority order, and in order of arrival within a priority. Priorities go from -1 (highest) to 6 (lowest); `PRIO_HIGH` is 2, `PRIO_NORMAL` 4, and `QUEUES` (7) is the number of queued priorities. An event of priority -1 is not queued: its callback runs at once, as soon as the event occurs. A watcher starts at its type's `$Event::TYPE::DefaultPriority`, or `PRIO_NORMAL` where that is not set; only `Event::signal` sets it, to `PRIO_HIGH`. The constructor attributes `nice => $offset` (added to the default), `async => 1` (priority -1) and `prio => $level` override it, `prio` winning over `async` over `nice`.

Customization

- `$Event::DIED` - called as `$Event::DIED->($event, $error)` when a callback dies inside `loop` or `sweep`; the default warns "Event: trapped error in", the watcher's `desc` and the error. The loop goes on afterwards; a handler that wants to end it can call `Event::unloop_all()`.
- `$Event::DebugLevel` - diagnostics on STDERR: from 1, why a watcher stopped when a change left it unable to generate events; from 2, each wait for events; from 3, each watcher created and the value `loop` returns; from 4, the errors `loop` traps. The finer per-watcher levels exist only when Event is built with debugging (see `Event::Watcher`'s debug).

Differences from upstream

- `Event::queue` checks its argument count. Called past its prototype with none, or with three or more, XS Event reads memory at random; here it dies with its usage (t/81-xs-native/Event/340-queue-arguments.t).
- Destroying an `Event::generic::Source` stops the watchers still watching it; with XS Event it corrupts memory (t/81-xs-native/Event/350-source-destroyed.t; see `Event::generic::Source`).

Modules

- `Event::Event` - An event, as a callback receives it: one occurrence (or several, merged) of what a watcher watches.
- `Event::Watcher` - The base class of every watcher: the methods and attributes all watcher types share.
- `Event::generic` - A generic watcher: an event whenever Perl code says so.
- `Event::group` - A group watcher: an event when none of its member watchers has run its callback for `timeout` seconds - a watchdog over a set of watchers.
- `Event::idle` - An idle watcher: an event when the loop has nothing else to do - no event queued, nothing arriving.

- `Event::io` - An io watcher: an event when a file descriptor is ready to read or to write, or has an exception pending - and, optionally, when it has been quiet for a given time.
- `Event::signal` - A signal watcher: an event when the process receives a signal.
- `Event::timer` - A timer watcher: an event at a given time, and optionally at every interval after it.
- `Event::var` - A var watcher: an event whenever a scalar variable is written to, or read.

Functions

Running the loop

unloop

`unloop($result)`: make the innermost running `loop` return `$result` (undef when omitted), once the current callback returns.

unloop_all

`unloop_all($result)`: make every running `loop`, however deeply nested, return `$result`, as if `unloop` were called for each. It does not `die`.

null_loops_per_second

`null_loops_per_second($seconds)`: a benchmark of the system's polling. For `$seconds` seconds it polls a pipe that has nothing to report, as an idle loop would, and returns how many polls one second allows.

one_event

`one_event($timeout)`: run one callback, waiting for an event if none is queued - at most `$timeout` seconds, 60 by default. Returns true if a callback ran. `loop` is built on it, but unlike `loop` it does not trap exceptions: a callback's `die` propagates out of `one_event`.

Events

queue

`queue($watcher)`, `queue($watcher, $hits)` or `queue($watcher, $event)`: queue an event for the watcher - a new one of one hit, a new one of `$hits` hits, or `$event`, an event of this watcher (one of another dies "queue: event doesn't match watcher"). The watcher need not be active. Returns nothing. Other argument counts (possible only by calling it as `&Event::queue(...)`, past its prototype) die "Usage: Event::queue(watcher, [hits | event])".

queue_pending

Turn what has happened into queued events without running any callback: poll descriptors without waiting, expire due timers, and turn counted signals into events. `sweep` and `one_event` do this themselves; call it directly before `pending` or `start` to take signals that already arrived into account.

queue_time

`queue_time($prio)`: the latest time a callback was started for an event of priority `$prio` or better (0 to 6), or undef if none has been. Other priorities die "queue_time(N) out of domain [0..6]".

Watchers

all_watchers

Every watcher that is not cancelled, stopped ones included, newest first.

all_idle

The idle watchers waiting for the loop to go idle, oldest first. While the queue is busy they stay here; one with `max` set fires anyway when that time has passed (see `Event::idle`).

all_running

The watchers whose callbacks are running, the most recently entered first; in scalar context only that one (undef when none is).

Time

cache_time_api

Switch Event's clock to Time::HiRes's, for fractional seconds; true on success, false when Time::HiRes is not loaded. `use Event` calls it after loading Time::HiRes, unless imported with `NO_TIME_HIRES`.

time

The current time as Event sees it, in seconds since the epoch: fractional with Time::HiRes (see `cache_time_api`), whole seconds without. Timer times are in these units.

sleep

`sleep($seconds)`: block for that long, fractions included, without running the loop. Returns nothing.

Event::Event

An event, as a callback receives it: one occurrence (or several, merged) of what a watcher watches. `loop` calls the watcher's callback with the event as its only argument.

Events of io and var watchers are `Event::Event::Io` objects, which add `got`; events of generic watchers are `Event::Event::Dataful` objects, which add `data`. Both are subclasses of `Event::Event`.

Synopsis

```
use Event;

Event->io(fd => \*STDIN, poll => 'r', cb => sub {
    my $e = shift;
    printf "%s: %s, %d hit(s) at priority %d\n",
        $e->w->desc, $e->got, $e->hits, $e->prio;
    Event::unloop() if eof STDIN;
});
Event::loop();
```

Modules

- `Event::Event::Dataful`
- `Event::Event::Io`

Functions

Attributes

mom

The old name of `w`. Returns the same; the first 16 calls in a program also warn "'mom' renamed to 'w'".

w

The watcher that generated the event. Read-only.

hits

How many occurrences this event stands for. Signals arriving in quick succession are merged into one event; for the other watcher types it is usually 1.

prio

The priority the event was queued at. It can differ from the watcher's current `prio`: the watcher's may have changed since, and `Event::queue` and the C API can queue at any priority.

Event::Event::Dataful

Functions

Attributes

data

The data, on an `Event::Event::Dataful`: the scalar passed to the event source's `event` method (`Event::generic::Source`), or undef. Read-only.

Event::Event::Io

Functions

Attributes

got

What happened, on an `Event::Event::Io`: a string of the letters `r`, `w`, `e` and `t` (read, write, exception, timeout) that is also the number made of the matching bits `R`, `W`, `E` and `T` of `Event::Watcher`. The watcher's `poll` says what it waits for; `got` says which of those occurred.

Event::Watcher

The base class of every watcher: the methods and attributes all watcher types share. A watcher is made by its type's constructor (`Event->timer(...)`, `Event::io->new(...)`, ...; see `Event`), lives on without a reference being kept to it, and is switched between active and inactive with `start` and `stop`. When the condition it watches occurs, an active watcher queues an event at its priority, and the loop passes that event to its callback.

Every attribute is a method: called with no argument it returns the value, called with one it sets the value and returns it. Constructors take the same attributes as `name => value` pairs.

Methods written in Perl

`Event::Watcher` (from `Event/Watcher.pm`) also has:

- `$w->attributes` - the names of the watcher's attributes, its own type's followed by those of its base classes.
- `$w->configure` - with no arguments, a list of `name => value` pairs of all attributes; with pairs, sets them.
- `$w->data` / `$w->data($anything)` - one slot for the user's own data.
- `$w->private` / `$w->private($anything)` - like `data`, but each calling package has its own slot; meant for the implementers of watcher types and subclasses.
- `Event::Watcher->register` - called by a watcher type's module to make `Event->TYPE(...)` construct it.
- `$w->init(\%attributes)` - the common half of every constructor: sets the priority (from the type's `$DefaultPriority` and the `nice`, `async` and `prio` attributes),

every attribute the watcher has a method for, a `desc` naming the caller's file and line if none was given, and starts the watcher unless `parked` is true.

Constants

Exported on request (`use Event::Watcher qw(R W E T)`):

- `R` (1), `W` (2), `E` (4), `T` (8) - the read, write, exception and timeout bits of `poll` and `got`.
- `ACTIVE` (1), `SUSPEND` (4) - bits of `Event::Watcher::Tied`'s `flags`.

Modules

- `Event::Watcher::Tied` - The base class for watcher types written in Perl.

Functions

Activity

again

Like `start`, except for watchers that repeat in a special way: a repeating timer computes its next expiry from `interval`.

start

Activate the watcher; an active watcher does nothing. A watcher that lacks what it needs to generate events refuses with "Event: can't start 'DESC' REASON" - for instance a watcher without a callback ("without callback"), or a repeating timer without `interval`. Constructors call `start` unless given `parked => 1`.

suspend

`$w->suspend(1)` suspends the watcher, `$w->suspend(0)` lifts the suspension. A suspended watcher generates no events and its queued events are cancelled, but it stays active; suspension is a debugging aid. Called without an argument it suspends, warns "Ambiguous use of suspend" and returns true.

resume

Deprecated: the same as `suspend(0)`, with a warning saying so.

stop

Deactivate the watcher. A running callback completes; queued events are cancelled. `start` or `again` activates it again.

cancel

Stop the watcher and destroy it. A running callback completes; queued events are cancelled. A cancelled watcher answers its read-only methods (`prio`, `desc`, ...) but can no longer be started.

now

Generate an event now, even if the watcher is stopped (but not if it is suspended). The event is queued at the watcher's priority like any other, so the callback runs when the loop reaches it. Dies if the watcher has no callback.

State

pending

The watcher's events waiting in the queue. In list context, the events themselves; in scalar context, true if there is any.

is_running

0 when the callback is not running, otherwise how many times it has been entered - more than 1 when a `reentrant` callback runs `loop` or `sweep` itself.

is_active

True if the watcher is started. Suspension does not change it.

is_suspended

True if the watcher is suspended (see `suspend`).

is_cancelled

True if the watcher is cancelled.

Attributes

use_keys

Deprecated; does nothing but warn "use_keys is deprecated".

cb

The callback: a code reference, or `[$class_or_object, $method_name]` to call a method. It is called with the event as its only argument. Anything else dies "Callback must be a code ref or [$object, $method_name]"; a method that does not exist yet only warns. Setting undef removes the callback and stops the watcher.

cbtime

When the callback was last called, in the units of `Event::time`. Read-only: setting it dies.

desc

A name for the watcher, used in diagnostics. Constructors set it to `?? FILE:LINE` of their caller when none is given.

debug

Debugging for this watcher. Stored and returned, but without effect: its per-watcher diagnostics exist only when Event is built with debugging, which a standard build of the module, like this one, is not.

prio

The priority its events are queued at, -1 (highest) to 6 (lowest); see `Event`.

reentrant

Whether the callback may be entered again while it runs, which happens when it calls `loop` or `sweep`. True by default; when false the watcher is suspended during such nested calls. It cannot be turned off while the callback is already entered more than once.

repeat

Whether the watcher stays active after an event; when false it stops after one. The default depends on the type: io, signal, var, group and generic watchers repeat; timers and idle watchers repeat when given `interval`, or `min` or `max`.

max_cb_tm

The most seconds a callback may run, an integer, 1 by default. It takes effect only with the statistics `Event::Stats` installs; otherwise it is recorded and not enforced. A value Perl does not hold as an integer - the string `"5"`, the number 2.5 - sets 0.

Event::Watcher::Tied

The base class for watcher types written in Perl. A subclass makes its watchers with `allocate`, and the loop calls back into it:

- `_start($w, $repeat)` - required; called by `start` (`$repeat` false) and `again` (true). It must arrange for events, by queueing them with `Event::queue` or by setting `at`.
- `_alarm($w)` - called when the time set with `at` has come; required once `at` is used.
- `_stop($w)` - called by `stop`, if the subclass has it.

The shared watcher methods and attributes are `Event::Watcher`'s.

Synopsis

```
package My::Ticker;
use base 'Event::Watcher::Tied';

sub new {
    my ($class, %arg) = @_;
    my $w = $class->allocate({});
    $w->init(\%arg);
    $w;
}
sub _start { my ($w) = @_; $w->at(Event::time() + 1) }
sub _alarm { my ($w) = @_; Event::queue($w); $w->at(Event::time() +
↪1) }
```

Functions

Construction

allocate

Make a new, inactive watcher blessed into `$class`, a subclass; `$template`, a hash or array reference, becomes the object. Dies "Bad template" when it is no reference. The subclass's constructor then usually calls `init` (see `Event::Watcher`).

Attributes

hard

The flag timers use to choose how to reckon their next time (see `Event::timer`). A tied watcher's own `_start` and `_alarm` decide its times, so here it is only stored, for them to read.

at

The time `_alarm` is to be called, in the units of `Event::time`. Setting a time schedules the alarm, replacing a scheduled one; setting undef only cancels it.

flags

The watcher's state bits `ACTIVE` (1) and `SUSPEND` (4) (see `Event::Watcher`). Setting it can change only the bit 0x4000, which is on by default: a repeating watcher with it on stops generating events while its callback runs. Trying to change any other bit warns "Other flags (0xN) cannot be changed".

Event::generic

A generic watcher: an event whenever Perl code says so. Events come from an event source, an `Event::generic::Source` object, whose `event` method gives every active watcher of the source an event; data passed to `event` reaches the callbacks as the event's `data`. The events are `Event::Event::Dataful` objects.

`use Event` does not load this watcher type; `use Event::generic` does, and with it `Event::generic::Source`.

`Event->generic(%attributes)` (or `Event::generic->new(...)`, or the source's `watch` method) takes the shared attributes of `Event::Watcher` and `source`. A generic watcher repeats. It refuses to start without a callback or without a source ("without source").

Synopsis

```
use Event;
use Event::generic;

my $source = Event::generic::Source->new;
my $w = $source->watch(cb => sub { print "got ", $_[0]->data, "\n" }
 ↪);
$source->event("hello");
Event::sweep();
```

Modules

- `Event::generic::Source` - An event source for generic watchers (see `Event::generic`): calling its `event` method gives an event to each of its active watchers.

Functions

Attributes

source

The event source watched: an `Event::generic::Source`, or undef for none. Anything else dies. Changing it moves an active watcher to the new source.

Event::generic::Source

An event source for generic watchers (see `Event::generic`): calling its `event` method gives an event to each of its active watchers.

Written in Perl (`Event/generic.pm`), beside the methods below:

- `Event::generic::Source->new` - a new source.
- `$source->watch(%attributes)` - a new `Event::generic` watcher of this source, as `Event->generic(source => $source, %attributes)`.

A source lives as long as a reference to it does, and each of its watchers holds one, so it normally outlives them.

Differences from upstream

- A source destroyed while watchers still watch it - by calling its DESTROY directly - stops those watchers, as if their `source` were set to undef. Afterwards `event` on it queues nothing, its watchers cannot be started ("without source"), and perl's own later DESTROY of it is harmless. XS Event frees the source but leaves its watchers linked to the freed memory, and frees it a second time; perl aborts with "double free or corruption" (t/81-xs-native/Event/350-source-destroyed.t).

Synopsis

```
use Event;
use Event::generic;

my $jobs = Event::generic::Source->new;
$jobs->watch(cb => sub { run_job($_[0]->data) });
$jobs->event({ id => 1 });
Event::loop();
```

Functions

Sources

`allocate`

The constructor's primitive: a new source blessed into `$class`, with `$template` (a hash or array reference) as the object; `new` calls it with an empty hash. Dies "Bad template" when `$template` is no reference.

`event`

`$source->event` or `$source->event($data)`: queue one event on every active watcher of the source. A copy of `$data` (undef when omitted) becomes each event's `data`. Returns nothing.

Event::group

A group watcher: an event when none of its member watchers has run its callback for `timeout` seconds - a watchdog over a set of watchers.

`use Event` does not load this watcher type; `use Event::group` does (`Event->group` without it still works, with a warning that autoloading is deprecated).

`Event->group(%attributes)` (or `Event::group->new(...)`) takes the shared attributes of `Event::Watcher`, `timeout`, and any number of `add => $watcher` pairs; `del` there is ignored with a warning. A group repeats. It refuses to start without a callback, or without a `timeout` ("repeating group has no timeout").

Synopsis

```
use Event;
use Event::group;

my $in  = Event->io(fd => \*STDIN, cb => \&read_input);
my $tick = Event->timer(interval => 5, cb => \&poll_server);
Event->group(add => $in, add => $tick, timeout => 60,
             cb => sub { warn "nothing happened for a minute\n" });
Event::loop();
```

Functions

Attributes

timeout

Seconds of silence after which the group fires: a number, or a reference to a number. A negative value warns and counts as 0; anything else that is not a number dies.

add

`$group->add($watcher)` makes `$watcher` a member; returns nothing. Adding the group to itself dies "can't add group 'DESC' to itself".

del

`$group->del($watcher)` removes a member; returns nothing. A watcher that is not a member is ignored.

Event::idle

An idle watcher: an event when the loop has nothing else to do - no event queued, nothing arriving. `min` keeps it from firing more often than every so many seconds, and `max` makes it fire after that many seconds even if the loop never goes idle.

`Event->idle(%attributes)` (or `Event::idle->new(...)`) takes the shared attributes of `Event::Watcher` and the ones below. Given `min` or `max`, the watcher repeats; otherwise it fires once. `repeat` overrides either.

`Event::all_idle` lists the idle watchers waiting for the loop to go idle.

Synopsis

```
use Event;

Event->idle(min => 1, max => 10, cb => sub { flush_caches() });
Event::loop();
```

Functions

Attributes

hard

How the waiting time is reckoned when the watcher starts again after its callback: when true from the time the callback was called, when false (the default) from the time it returned.

max

The most seconds to wait for the loop to go idle; after that the watcher fires anyway, as an ordinary event at its priority. A number, a reference to a number, or undef (the default) for no limit.

min

The fewest seconds between two events of this watcher: it starts waiting for idleness only that long after it was started or its callback ran. A number, a reference to a number, or undef for no minimum; 0.01 by default.

Event::io

An io watcher: an event when a file descriptor is ready to read or to write, or has an exception pending - and, optionally, when it has been quiet for a given time. Its events are `Event::Event::Io` objects, whose `got` says what happened.

`Event->io(%attributes)` (or `Event::io->new(...)`) takes the shared attributes of `Event::Watcher` and the ones below. An io watcher repeats, and stays registered while its callback runs. Its default priority is `PRIO_NORMAL`. It refuses to start without a callback ("without io callback"), and with neither a descriptor to poll nor a timeout ("because there is nothing to watch").

One watcher can watch for several conditions, or several watchers can share a descriptor; both work.

The bit constants `R`, `W`, `E` and `T` can be imported from `Event::io` as well as from `Event::Watcher`.

Synopsis

```
use Event;

Event->io(fd => \*STDIN, poll => 'r', timeout => 30,
    cb => sub {
        my $e = shift;
        if ($e->got =~ /t/) { print "30 seconds without input\n";␣
↪return }
        defined(my $line = <STDIN>) or return Event::unloop();
        print "read: $line";
```

(continues on next page)

(continued from previous page)

```
    });
Event::loop();
```

Functions

Attributes

poll

What to wait for: a string of the letters `r` (read), `w` (write) and `e` (exception), or the number made of the bits `R`, `W` and `E`; `"r"` by default. Another letter warns "Ignored 'x' in poll mask" and other bits warn too. The value returned is both the string and the number. It includes `t` exactly when a `timeout` is set, whatever was asked.

fd

The descriptor to watch: a glob or glob reference (`\*STDIN`, an `IO::Handle` object), or a file descriptor number. Its descriptor is taken when the watcher starts; a glob without an open handle dies "isn't a valid IO", anything else "can't find fileno".

timeout

Seconds without a callback after which an event comes anyway, with `t` in its `got`; 0 or undef (the default) for none. Each callback restarts the count. A watcher that does not repeat fires the timeout once and then clears it.

timeout_cb

The callback for timeouts, in the forms `cb` takes; while it is undef (the default), timeouts go to `cb` too. Unlike `cb`, a `[$object, $method_name]` naming a method that does not exist is taken without a warning.

Event::signal

A signal watcher: an event when the process receives a signal.

`Event->signal(%attributes)` (or `Event::signal->new(...)`) takes the shared attributes of `Event::Watcher` and `signal`. A signal watcher repeats; its default priority is `PRIO_HIGH`. It refuses to start without a callback or a signal.

The signal handler only counts arrivals. The counts become events when the loop next looks (`Event::queue_pending`, which `loop`, `sweep` and `one_event` call), so signals can be handled later than, and out of order with, other events; several arrivals of a signal in between make one event whose `hits` is their number. The watchers that receive the event are the ones active when the counts are collected, not when the signal arrived.

A signal is handled either through `%SIG` or by Event, not both: while at least one watcher for it is active, Event's handler is installed directly, without `%SIG`; when the last one stops, the signal is set back to its default action.

Synopsis

```
use Event;

Event->signal(signal => 'HUP', cb => sub { reload_config() });
Event->signal(signal => 'INT', cb => sub { Event::unloop_all() });
Event::loop();
```

Functions

Attributes

signal

The signal, by its name without SIG (`'INT'`, `'HUP'`), as the keys of `%SIG`; it is returned under its main name. An unknown name dies "Unrecognized signal 'NAME'", and a signal that cannot be caught (`KILL`, `STOP`) dies "Signal 'NAME' cannot be caught". Changing it moves an active watcher to the new signal.

Event::timer

A timer watcher: an event at a given time, and optionally at every interval after it. Times are in the units of `Event::time` - seconds, fractional when Time::HiRes is loaded, which `use Event` does.

`Event->timer(%attributes)` (or `Event::timer->new(...)`) takes the shared attributes of `Event::Watcher`, the ones below, and `after`:

- `at => $time` - the first expiry.
- `after => $seconds` - the first expiry, that many seconds from now; also the `interval` when none is given. `at` and `after` together die "'after' and 'at' are mutually exclusive".
- `interval => $seconds` - repeat this often. Given alone, the first expiry is one interval from now.

A timer repeats when it was given `interval` (or `after`, which sets it), unless `repeat => 0` is given; otherwise it fires once. Its default priority is `PRIO_NORMAL`.

Synopsis

```
use Event;

Event->timer(after => 0.5, cb => sub { print "once\n" }, repeat =>␣
↪0);
Event->timer(interval => 1, cb => sub { print "every second\n" });
Event::loop();
```

Functions

Attributes

at

The time of the next expiry. Setting it moves an active timer to the new time.

hard

How a repeating timer reckons its next expiry: when true, one `interval` after the expiry just handled, so the timer keeps to its schedule however long the callback takes; when false (the default), one `interval` after the callback returned.

interval

Seconds between expiries: a number, or a reference to a number, which is then read at every expiry. A negative interval warns and counts as 0; anything else that is not a number dies.

Event::var

A var watcher: an event whenever a scalar variable is written to, or read. Its events are `Event::Event::Io` objects, whose `got` is `w` for a write and `r` for a read.

`Event->var(%attributes)` (or `Event::var->new(...)`) takes the shared attributes of `Event::Watcher` and the ones below. A var watcher repeats; its default priority is `PRIO_NORMAL`. It refuses to start without a callback, without a variable ("watching what?"), with an empty `poll`, or on a read-only variable.

Perl reads variables in more places than are apparent from the source, so watching reads is rarely what is wanted; the default is to watch writes only.

Synopsis

```
use Event;

my $status = 'idle';
Event->var(var => \$status, cb => sub { print "status: $status\n" }
 ↪);
Event->timer(after => 1, cb => sub { $status = 'busy' }, repeat =>␣
 ↪0);
Event::loop(2);
```

Functions

Attributes

var

The variable, as a reference to a plain scalar. A reference to an array, hash or anything else beyond a scalar dies "Var watchers can only watch plain vanilla scalars", and a value

that is no reference dies “Expecting a reference”. The watcher keeps the reference it was given.

poll

What to watch for: `"w"` (writes, the default), `"r"` (reads) or `"rw"`, or the number made of the bits `R` and `W` (see `Event::Watcher`). The value returned is both the string and the number.

one_event

`one_event($timeout)`: run one callback, waiting for an event if none is queued - at most `$timeout` seconds, 60 by default. Returns true if a callback ran. `loop` is built on it, but unlike `loop` it does not trap exceptions: a callback's `die` propagates out of `one_event`.

One round: turn counted signals into events; if an event of priority better than 5 is queued, run it and return; otherwise wait for descriptors and timers (no longer than the next timer and `$timeout`), queue what happened, turn signals into events again, run the best queued event if there is one, and otherwise run an idle watcher.

5.2.39 Exporter

Implements default import method for modules

SYNOPSIS

In module `YourModule.pm`:

```
package YourModule;
use Exporter 'import';
our @EXPORT_OK = qw(munge frobnicate);  # symbols to export on␣
↪request
```

or

```
package YourModule;
require Exporter;
our @ISA = qw(Exporter);  # inherit all of Exporter's methods
our @EXPORT_OK = qw(munge frobnicate);  # symbols to export on␣
↪request
```

or

```
package YourModule;
use parent 'Exporter';  # inherit all of Exporter's methods
our @EXPORT_OK = qw(munge frobnicate);  # symbols to export on␣
↪request
```

In other files which wish to use `YourModule`:

```
use YourModule qw(frobnicate);      # import listed symbols
frobnicate ($left, $right)          # calls YourModule::frobnicate
```

Take a look at “Good Practices” for some variants you will like to use in modern Perl code.

DESCRIPTION

The Exporter module implements an `import` method which allows a module to export functions and variables to its users' namespaces. Many modules use Exporter rather than implementing their own `import` method because Exporter provides a highly flexible interface, with an implementation optimised for the common case.

Perl automatically calls the `import` method when processing a `use` statement for a module. Modules and `use` are documented in *perlfunc* and perlmod. Understanding the concept of modules and how the `use` statement operates is important to understanding the Exporter.

How to Export

The arrays `@EXPORT` and `@EXPORT_OK` in a module hold lists of symbols that are going to be exported into the users name space by default, or which they can request to be exported, respectively. The symbols can represent functions, scalars, arrays, hashes, or typeglobs. The symbols must be given by full name with the exception that the ampersand in front of a function is optional, e.g.

```
our @EXPORT    = qw(afunc $scalar @array);   # afunc is a function
our @EXPORT_OK = qw(&bfunc %hash *typeglob); # explicit prefix on &
 ↪bfunc
```

If you are only exporting function names it is recommended to omit the ampersand, as the implementation is faster this way.

Selecting What to Export

Do **not** export method names!

Do **not** export anything else by default without a good reason!

Exports pollute the namespace of the module user. If you must export try to use `@EXPORT_OK` in preference to `@EXPORT` and avoid short or common symbol names to reduce the risk of name clashes.

Generally anything not exported is still accessible from outside the module using the `YourModule::item_name` (or `$blessed_ref->method`) syntax. By convention you can use a leading underscore on names to informally indicate that they are ‘internal’ and not for public use.

(It is actually possible to get private functions by saying:

```
my $subref = sub { ... };
$subref->(@args);             # Call it as a function
$obj->$subref(@args);         # Use it as a method
```

However if you use them for methods it is up to you to figure out how to make inheritance work.)

As a general rule, if the module is trying to be object oriented then export nothing. If it's just a collection of functions then `@EXPORT_OK` anything but use `@EXPORT` with caution. For function and method names use barewords in preference to names prefixed with ampersands for the export lists.

Other module design guidelines can be found in perlmod.

How to Import

In other files which wish to use your module there are three basic ways for them to load your module and import its symbols:

- `use YourModule;`

  This imports all the symbols from YourModule's `@EXPORT` into the namespace of the `use` statement.

- `use YourModule ();`

  This causes perl to load your module but does not import any symbols.

- `use YourModule qw(...);`

  This imports only the symbols listed by the caller into their namespace. All listed symbols must be in your `@EXPORT` or `@EXPORT_OK`, else an error occurs. The advanced export features of Exporter are accessed like this, but with list entries that are syntactically distinct from symbol names.

Unless you want to use its advanced features, this is probably all you need to know to use Exporter.

Advanced Features

Specialised Import Lists

If any of the entries in an import list begins with !, : or / then the list is treated as a series of specifications which either add to or delete from the list of names to import. They are processed left to right. Specifications are in the form:

```
[!]name         This name only
[!]:DEFAULT     All names in @EXPORT
[!]:tag         All names in $EXPORT_TAGS{tag} anonymous array
[!]/pattern/    All names in @EXPORT and @EXPORT_OK which match
```

A leading ! indicates that matching names should be deleted from the list of names to import. If the first specification is a deletion it is treated as though preceded by :DEFAULT. If you just want to import extra names in addition to the default set you will still need to include :DEFAULT explicitly.

e.g., `Module.pm` defines:

```
our @EXPORT      = qw(A1 A2 A3 A4 A5);
our @EXPORT_OK   = qw(B1 B2 B3 B4 B5);
```

(continues on next page)

(continued from previous page)

```
our %EXPORT_TAGS = (T1 => [qw(A1 A2 B1 B2)], T2 => [qw(A1 A2 B3␣
↪B4)]);
```

Note that you cannot use tags in @EXPORT or @EXPORT_OK.

Names in EXPORT_TAGS must also appear in @EXPORT or @EXPORT_OK.

An application using Module can say something like:

```
use Module qw(:DEFAULT :T2 !B3 A3);
```

Other examples include:

```
use Socket qw(!/^[AP]F_/ !SOMAXCONN !SOL_SOCKET);
use POSIX  qw(:errno_h :termios_h !TCSADRAIN !/^EXIT/);
```

Remember that most patterns (using //) will need to be anchored with a leading ^, e.g., `/^EXIT/` rather than `/EXIT/`.

You can say `BEGIN { $Exporter::Verbose=1 }` to see how the specifications are being processed and what is actually being imported into modules.

Exporting Without Using Exporter's import Method

Exporter has a special method, 'export_to_level' which is used in situations where you can't directly call Exporter's import method. The export_to_level method looks like:

```
MyPackage->export_to_level(
    $where_to_export, $package, @what_to_export
);
```

where `$where_to_export` is an integer telling how far up the calling stack to export your symbols, and `@what_to_export` is an array telling what symbols *to* export (usually this is `@_`). The `$package` argument is currently unused.

For example, suppose that you have a module, A, which already has an import function:

```
package A;

our @ISA = qw(Exporter);
our @EXPORT_OK = qw($b);

sub import
{
    $A::b = 1;     # not a very useful import method
}
```

and you want to Export symbol `$A::b` back to the module that called package A. Since Exporter relies on the import method to work, via inheritance, as it stands Exporter::import() will never get called. Instead, say the following:

```
package A;
our @ISA = qw(Exporter);
our @EXPORT_OK = qw($b);

sub import
{
    $A::b = 1;
    A->export_to_level(1, @_);
}
```

This will export the symbols one level 'above' the current package - ie: to the program or module that used package A.

Note: Be careful not to modify `@_` at all before you call export_to_level - or people using your package will get very unexplained results!

Exporting Without Inheriting from Exporter

By including Exporter in your `@ISA` you inherit an Exporter's import() method but you also inherit several other helper methods which you probably don't want and complicate the inheritance tree. To avoid this you can do:

```
package YourModule;
use Exporter qw(import);
```

which will export Exporter's own import() method into YourModule. Everything will work as before but you won't need to include Exporter in `@YourModule::ISA`.

Note: This feature was introduced in version 5.57 of Exporter, released with perl 5.8.3.

Module Version Checking

The Exporter module will convert an attempt to import a number from a module into a call to `$module_name->VERSION($value)`. This can be used to validate that the version of the module being used is greater than or equal to the required version.

For historical reasons, Exporter supplies a `require_version` method that simply delegates to `VERSION`. Originally, before `UNIVERSAL::VERSION` existed, Exporter would call `require_version`.

Since the `UNIVERSAL::VERSION` method treats the `$VERSION` number as a simple numeric value it will regard version 1.10 as lower than 1.9. For this reason it is strongly recommended that you use numbers with at least two decimal places, e.g., 1.09.

Managing Unknown Symbols

In some situations you may want to prevent certain symbols from being exported. Typically this applies to extensions which have functions or constants that may not exist on some systems.

The names of any symbols that cannot be exported should be listed in the `@EXPORT_FAIL` array.

If a module attempts to import any of these symbols the Exporter will give the module an opportunity to handle the situation before generating an error. The Exporter will call an export_fail method with a list of the failed symbols:

```
@failed_symbols = $module_name->export_fail(@failed_symbols);
```

If the `export_fail` method returns an empty list then no error is recorded and all the requested symbols are exported. If the returned list is not empty then an error is generated for each symbol and the export fails. The Exporter provides a default `export_fail` method which simply returns the list unchanged.

Uses for the `export_fail` method include giving better error messages for some symbols and performing lazy architectural checks (put more symbols into `@EXPORT_FAIL` by default and then take them out if someone actually tries to use them and an expensive check shows that they are usable on that platform).

Tag Handling Utility Functions

Since the symbols listed within `%EXPORT_TAGS` must also appear in either `@EXPORT` or `@EXPORT_OK`, two utility functions are provided which allow you to easily add tagged sets of symbols to `@EXPORT` or `@EXPORT_OK`:

```
our %EXPORT_TAGS = (foo => [qw(aa bb cc)], bar => [qw(aa cc dd)]);

Exporter::export_tags('foo');     # add aa, bb and cc to @EXPORT
Exporter::export_ok_tags('bar');  # add aa, cc and dd to @EXPORT_OK
```

Any names which are not tags are added to `@EXPORT` or `@EXPORT_OK` unchanged but will trigger a warning (with `-w`) to avoid misspelt tags names being silently added to `@EXPORT` or `@EXPORT_OK`. Future versions may make this a fatal error.

Generating Combined Tags

If several symbol categories exist in `%EXPORT_TAGS`, it's usually useful to create the utility ":all" to simplify "use" statements.

The simplest way to do this is:

```
our  %EXPORT_TAGS = (foo => [qw(aa bb cc)], bar => [qw(aa cc dd)]);

 # add all the other ":class" tags to the ":all" class,
 # deleting duplicates
 {
   my %seen;

   push @{$EXPORT_TAGS{all}},
     grep {!$seen{$_}++} @{$EXPORT_TAGS{$_}} foreach keys %EXPORT_
↪TAGS;
 }
```

`CGI.pm` creates an ":all" tag which contains some (but not really all) of its categories. That could be done with one small change:

```
# add some of the other ":class" tags to the ":all" class,
# deleting duplicates
{
  my %seen;

  push @{$EXPORT_TAGS{all}},
    grep {!$seen{$_}++} @{$EXPORT_TAGS{$_}}
      foreach qw/html2 html3 netscape form cgi internal/;
}
```

Note that the tag names in %EXPORT_TAGS don't have the leading ':'.

AUTOLOADed Constants

Many modules make use of AUTOLOADing for constant subroutines to avoid having to compile and waste memory on rarely used values (see *perlsub* for details on constant subroutines). Calls to such constant subroutines are not optimized away at compile time because they can't be checked at compile time for constancy.

Even if a prototype is available at compile time, the body of the subroutine is not (it hasn't been AUTOLOADed yet). perl needs to examine both the () prototype and the body of a subroutine at compile time to detect that it can safely replace calls to that subroutine with the constant value.

A workaround for this is to call the constants once in a BEGIN block:

```
package My ;

use Socket ;

foo( SO_LINGER );  ## SO_LINGER NOT optimized away; called at␣
 ↪runtime
BEGIN { SO_LINGER }
foo( SO_LINGER );  ## SO_LINGER optimized away at compile time.
```

This forces the AUTOLOAD for SO_LINGER to take place before SO_LINGER is encountered later in My package.

If you are writing a package that AUTOLOADs, consider forcing an AUTOLOAD for any constants explicitly imported by other packages or which are usually used when your package is used.

Good Practices

Declaring @EXPORT_OK and Friends

When using Exporter with the standard strict and warnings pragmas, the our keyword is needed to declare the package variables @EXPORT_OK, @EXPORT, @ISA, etc.

```
our @ISA = qw(Exporter);
our @EXPORT_OK = qw(munge frobnicate);
```

If backward compatibility for Perls **under** 5.6 is important, one must write instead a `use vars` statement.

```
use vars qw(@ISA @EXPORT_OK);
@ISA = qw(Exporter);
@EXPORT_OK = qw(munge frobnicate);
```

Playing Safe

There are some caveats with the use of runtime statements like `require Exporter` and the assignment to package variables, which can be very subtle for the unaware programmer. This may happen for instance with mutually recursive modules, which are affected by the time the relevant constructions are executed.

The ideal way to never have to think about that is to use `BEGIN` blocks and the simple import method. So the first part of the "SYNOPSIS" code could be rewritten as:

```
package YourModule;

use strict;
use warnings;

use Exporter 'import';
BEGIN {
  our @EXPORT_OK = qw(munge frobnicate);  # symbols to export on
 ↪request
}
```

Or if you need to inherit from Exporter:

```
package YourModule;

use strict;
use warnings;

BEGIN {
  require Exporter;
  our @ISA = qw(Exporter);  # inherit all of Exporter's methods
  our @EXPORT_OK = qw(munge frobnicate);  # symbols to export on
 ↪request
}
```

The `BEGIN` will assure that the loading of `Exporter.pm` and the assignments to `@ISA` and `@EXPORT_OK` happen immediately like `use`, leaving no room for something to get awry or just plain wrong.

With respect to loading `Exporter` and inheriting, there are alternatives with the use of modules like `base` and `parent`.

```
use base qw(Exporter);
# or
use parent qw(Exporter);
```

Any of these statements are nice replacements for `BEGIN { require Exporter; our @ISA = qw(Exporter); }` with the same compile-time effect. The basic difference is that `base` code interacts with declared `fields` while `parent` is a streamlined version of the older `base` code to just establish the IS-A relationship.

For more details, see the documentation and code of *base* and *parent*.

Another thorough remedy to that runtime vs. compile-time trap is to use Exporter::Easy, which is a wrapper of Exporter that allows all boilerplate code at a single gulp in the use statement.

```
use Exporter::Easy (
    OK => [ qw(munge frobnicate) ],
);
# @ISA setup is automatic
# all assignments happen at compile time
```

What Not to Export

You have been warned already in "Selecting What to Export" to not export:

- method names (because you don't need to and that's likely to not do what you want),
- anything by default (because you don't want to surprise your users... badly)
- anything you don't need to (because less is more)

There's one more item to add to this list. Do **not** export variable names. Just because `Exporter` lets you do that, it does not mean you should.

```
@EXPORT_OK = qw($svar @avar %hvar); # DON'T!
```

Exporting variables is not a good idea. They can change under the hood, provoking horrible effects at-a-distance that are too hard to track and to fix. Trust me: they are not worth it.

To provide the capability to set/get class-wide settings, it is best instead to provide accessors as subroutines or class methods instead.

SEE ALSO

`Exporter` is definitely not the only module with symbol exporter capabilities. At CPAN, you may find a bunch of them. Some are lighter. Some provide improved APIs and features. Pick the one that fits your needs. The following is a sample list of such modules.

```
Exporter::Easy
Exporter::Lite
Exporter::Renaming
Exporter::Tidy
Sub::Exporter / Sub::Installer
Perl6::Export / Perl6::Export::Attrs
```

LICENSE

This library is free software. You can redistribute it and/or modify it under the same terms as Perl itself.

Modules

- `Exporter::Heavy` - Exporter guts

Exporter::Heavy

Exporter guts

SYNOPSIS

(internal use only)

DESCRIPTION

No user-serviceable parts inside.

5.2.40 ExtUtils

Namespace containing the following modules:

- `ExtUtils::CBuilder` - Compile and link C code for Perl modules
- `ExtUtils::Command` - utilities to replace common UNIX commands in Makefiles etc.
- `ExtUtils::Embed` - Utilities for embedding Perl in C/C++ applications
- `ExtUtils::HasCompiler` - whether a distribution's compiled part will work (pperl)
- `ExtUtils::Install` - install files from here to there
- `ExtUtils::Installed` - Inventory management of installed modules
- `ExtUtils::Liblist` - determine libraries to use and how to use them
- `ExtUtils::MM` - OS adjusted ExtUtils::MakeMaker subclass
- `ExtUtils::MM_Any` - Platform-agnostic MM methods
- `ExtUtils::MM_Unix` - methods used by ExtUtils::MakeMaker
- `ExtUtils::MY` - ExtUtils::MakeMaker subclass for customization
- `ExtUtils::MakeMaker` - Create a module Makefile
- `ExtUtils::Manifest` - Utilities to write and check a MANIFEST file
- `ExtUtils::Miniperl` - write the C code for miniperlmain.c and perlmain.c
- `ExtUtils::PL2Bat` - Batch file creation to run perl scripts on Windows
- `ExtUtils::Packlist` - manage .packlist files
- `ExtUtils::testlib` - add blib/* directories to @INC

ExtUtils::CBuilder

Compile and link C code for Perl modules

SYNOPSIS

```
use ExtUtils::CBuilder;

my $b = ExtUtils::CBuilder->new(%options);
$obj_file = $b->compile(source => 'MyModule.c');
$lib_file = $b->link(objects => $obj_file);
```

DESCRIPTION

This module can build the C portions of Perl modules by invoking the appropriate compilers and linkers in a cross-platform manner. It was motivated by the `Module::Build` project, but may be useful for other purposes as well. However, it is *not* intended as a general cross-platform interface to all your C building needs. That would have been a much more ambitious goal!

METHODS

- new

  Returns a new `ExtUtils::CBuilder` object. A `config` parameter lets you override `Config.pm` settings for all operations performed by the object, as in the following example:

  ```
  # Use a different compiler than Config.pm says
  my $b = ExtUtils::CBuilder->new( config =>
                                   { ld => 'gcc' } );
  ```

  A `quiet` parameter tells `CBuilder` to not print its `system()` commands before executing them:

  ```
  # Be quieter than normal
  my $b = ExtUtils::CBuilder->new( quiet => 1 );
  ```

- have_compiler

  Returns true if the current system has a working C compiler and linker, false otherwise. To determine this, we actually compile and link a sample C library. The sample will be compiled in the system tempdir or, if that fails for some reason, in the current directory.

- have_cplusplus

  Just like have_compiler but for C++ instead of C.

- compile

  Compiles a C source file and produces an object file. The name of the object file is returned. The source file is specified in a `source` parameter, which is required; the other parameters listed below are optional.

- `object_file`

  Specifies the name of the output file to create. Otherwise the `object_file()` method will be consulted, passing it the name of the `source` file.

- `include_dirs`

  Specifies any additional directories in which to search for header files. May be given as a string indicating a single directory, or as a list reference indicating multiple directories.

- `extra_compiler_flags`

  Specifies any additional arguments to pass to the compiler. Should be given as a list reference containing the arguments individually, or if this is not possible, as a string containing all the arguments together.

- `C++`

  Specifies that the source file is a C++ source file and sets appropriate compiler flags

The operation of this method is also affected by the `archlibexp`, `cccdlflags`, `ccflags`, `optimize`, and `cc` entries in `Config.pm`.

- link

Invokes the linker to produce a library file from object files. In scalar context, the name of the library file is returned. In list context, the library file and any temporary files created are returned. A required `objects` parameter contains the name of the object files to process, either in a string (for one object file) or list reference (for one or more files). The following parameters are optional:

- lib_file

  Specifies the name of the output library file to create. Otherwise the `lib_file()` method will be consulted, passing it the name of the first entry in `objects`.

- module_name

  Specifies the name of the Perl module that will be created by linking. On platforms that need to do prelinking (Win32, OS/2, etc.) this is a required parameter.

- extra_linker_flags

  Any additional flags you wish to pass to the linker.

On platforms where `need_prelink()` returns true, `prelink()` will be called automatically.

The operation of this method is also affected by the `lddlflags`, `shrpenv`, and `ld` entries in `Config.pm`.

- link_executable

Invokes the linker to produce an executable file from object files. In scalar context, the name of the executable file is returned. In list context, the executable file and any temporary files created are returned. A required `objects` parameter contains the name of the object files to process, either in a string (for one object file) or list

reference (for one or more files). The optional parameters are the same as `link` with exception for

  - exe_file

    Specifies the name of the output executable file to create. Otherwise the `exe_file()` method will be consulted, passing it the name of the first entry in `objects`.

- object_file

```
my $object_file = $b->object_file($source_file);
```

Converts the name of a C source file to the most natural name of an output object file to create from it. For instance, on Unix the source file `foo.c` would result in the object file `foo.o`.

- lib_file

```
my $lib_file = $b->lib_file($object_file);
```

Converts the name of an object file to the most natural name of a output library file to create from it. For instance, on Mac OS X the object file `foo.o` would result in the library file `foo.bundle`.

- exe_file

```
my $exe_file = $b->exe_file($object_file);
```

Converts the name of an object file to the most natural name of an executable file to create from it. For instance, on Mac OS X the object file `foo.o` would result in the executable file `foo`, and on Windows it would result in `foo.exe`.

- prelink

On certain platforms like Win32, OS/2, VMS, and AIX, it is necessary to perform some actions before invoking the linker. The `ExtUtils::Mksymlists` module does this, writing files used by the linker during the creation of shared libraries for dynamic extensions. The names of any files written will be returned as a list.

Several parameters correspond to `ExtUtils::Mksymlists::Mksymlists()` options, as follows:

```
 Mksymlists()   prelink()          type
-------------|-------------------|-------------------
 NAME        |  dl_name          | string (required)
 DLBASE      |  dl_base          | string
 FILE        |  dl_file          | string
 DL_VARS     |  dl_vars          | array reference
 DL_FUNCS    |  dl_funcs         | hash reference
 FUNCLIST    |  dl_func_list     | array reference
 IMPORTS     |  dl_imports       | hash reference
 VERSION     |  dl_version       | string
```

Please see the documentation for `ExtUtils::Mksymlists` for the details of what these parameters do.

- need_prelink

  Returns true on platforms where `prelink()` should be called during linking, and false otherwise.

- extra_link_args_after_prelink

  Returns list of extra arguments to give to the link command; the arguments are the same as for prelink(), with addition of array reference to the results of prelink(); this reference is indexed by key `prelink_res`.

TO DO

Currently this has only been tested on Unix and doesn't contain any of the Windows-specific code from the `Module::Build` project. I'll do that next.

HISTORY

This module is an outgrowth of the `Module::Build` project, to which there have been many contributors. Notably, Randy W. Sims submitted lots of code to support 3 compilers on Windows and helped with various other platform-specific issues. Ilya Zakharevich has contributed fixes for OS/2; John E. Malmberg and Peter Prymmer have done likewise for VMS.

SUPPORT

ExtUtils::CBuilder is maintained as part of the Perl 5 core. Please submit any bug reports via the `perlbug` tool included with Perl 5. Bug reports will be included in the Perl 5 ticket system at https://rt.perl.org.

The Perl 5 source code is available at https://perl5.git.perl.org/perl.git and ExtUtils-CBuilder may be found in the `dist/ExtUtils-CBuilder` directory of the repository.

AUTHOR

Ken Williams, kwilliams@cpan.org

Additional contributions by The Perl 5 Porters.

COPYRIGHT

Copyright (c) 2003-2005 Ken Williams. All rights reserved.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

SEE ALSO

perl(1), Module::Build(3)

ExtUtils::Command

utilities to replace common UNIX commands in Makefiles etc.

SYNOPSIS

```
perl -MExtUtils::Command -e cat files... > destination
perl -MExtUtils::Command -e mv source... destination
perl -MExtUtils::Command -e cp source... destination
perl -MExtUtils::Command -e touch files...
perl -MExtUtils::Command -e rm_f files...
perl -MExtUtils::Command -e rm_rf directories...
perl -MExtUtils::Command -e mkpath directories...
perl -MExtUtils::Command -e eqtime source destination
perl -MExtUtils::Command -e test_f file
perl -MExtUtils::Command -e test_d directory
perl -MExtUtils::Command -e chmod mode files...
...
```

DESCRIPTION

The module is used to replace common UNIX commands. In all cases the functions work from @ARGV rather than taking arguments. This makes them easier to deal with in Makefiles. Call them like this:

```
perl -MExtUtils::Command -e some_command some files to work on
```

and *NOT* like this:

```
perl -MExtUtils::Command -e 'some_command qw(some files to work on)'
```

For that use Shell::Command.

Filenames with * and ? will be glob expanded.

FUNCTIONS

- cat

  ```
  cat file ...
  ```

  Concatenates all files mentioned on command line to STDOUT.

- eqtime

  ```
  eqtime source destination
  ```

  Sets modified time of destination to that of source.

- rm_rf

  ```
  rm_rf files or directories ...
  ```

Removes files and directories - recursively (even if readonly)

- rm_f

```
rm_f file ...
```

Removes files (even if readonly)

- touch

```
touch file ...
```

Makes files exist, with current timestamp

- mv

```
mv source_file destination_file
mv source_file source_file destination_dir
```

Moves source to destination. Multiple sources are allowed if destination is an existing directory.

Returns true if all moves succeeded, false otherwise.

- cp

```
cp source_file destination_file
cp source_file source_file destination_dir
```

Copies sources to the destination. Multiple sources are allowed if destination is an existing directory.

Returns true if all copies succeeded, false otherwise.

- chmod

```
chmod mode files ...
```

Sets UNIX like permissions 'mode' on all the files. e.g. 0666

- mkpath

```
mkpath directory ...
```

Creates directories, including any parent directories.

- test_f

```
test_f file
```

Tests if a file exists. *Exits* with 0 if it does, 1 if it does not (ie. shell's idea of true and false).

- test_d

```
test_d directory
```

Tests if a directory exists. *Exits* with 0 if it does, 1 if it does not (ie. shell's idea of true and false).

- dos2unix

```
dos2unix files or dirs ...
```

Converts DOS and OS/2 linefeeds to Unix style recursively.

SEE ALSO

Shell::Command which is these same functions but take arguments normally.

AUTHOR

Nick Ing-Simmons `ni-s@cpan.org`

Maintained by Michael G Schwern `schwern@pobox.com` within the ExtUtils-MakeMaker package and, as a separate CPAN package, by Randy Kobes `r.kobes@uwinnipeg.ca`.

Modules

- `ExtUtils::Command::MM` - Commands for the MM's to use in Makefiles

ExtUtils::Command::MM

Commands for the MM's to use in Makefiles

SYNOPSIS

```
perl "-MExtUtils::Command::MM" -e "function" "--" arguments...
```

DESCRIPTION

FOR INTERNAL USE ONLY! The interface is not stable.

ExtUtils::Command::MM encapsulates code which would otherwise have to be done with large "one" liners.

Any $(FOO) used in the examples are make variables, not Perl.

- **test_harness**

```
test_harness($verbose, @test_libs);
```

Runs the tests on @ARGV via Test::Harness passing through the $verbose flag. Any @test_libs will be unshifted onto the test's @INC.

@test_libs are run in alphabetical order.

- **pod2man**

```
pod2man( '--option=value',
         $podfile1 => $manpage1,
         $podfile2 => $manpage2,
         ...
       );

# or args on @ARGV
```

pod2man() is a function performing most of the duties of the pod2man program. Its arguments are exactly the same as pod2man as of 5.8.0 with the addition of:

```
--perm_rw   octal permission to set the resulting manpage to
```

And the removal of:

```
--verbose/-v
--help/-h
```

If no arguments are given to pod2man it will read from @ARGV.

If Pod::Man is unavailable, this function will warn and return undef.

- **warn_if_old_packlist**

```
perl "-MExtUtils::Command::MM" -e warn_if_old_packlist
 ↪<somefile>
```

Displays a warning that an old packlist file was found. Reads the filename from @ARGV.

- **perllocal_install**

```
perl "-MExtUtils::Command::MM" -e perllocal_install
    <type> <module name> <key> <value> ...

# VMS only, key|value pairs come on STDIN
perl "-MExtUtils::Command::MM" -e perllocal_install
    <type> <module name> < <key>|<value> ...
```

Prints a fragment of POD suitable for appending to perllocal.pod. Arguments are read from @ARGV.

'type' is the type of what you're installing. Usually 'Module'.

'module name' is simply the name of your module. (Foo::Bar)

Key/value pairs are extra information about the module. Fields include:

```
installed into      which directory your module was out into
LINKTYPE            dynamic or static linking
VERSION             module version number
EXE_FILES           any executables installed in a space␣
 ↪separated
                    list
```

- **uninstall**

```
perl "-MExtUtils::Command::MM" -e uninstall <packlist>
```

  A wrapper around ExtUtils::Install::uninstall(). Warns that uninstallation is deprecated and doesn't actually perform the uninstallation.

- **test_s**

```
perl "-MExtUtils::Command::MM" -e test_s <file>
```

  Tests if a file exists and is not empty (size > 0). *Exits* with 0 if it does, 1 if it does not.

- **cp_nonempty**

```
perl "-MExtUtils::Command::MM" -e cp_nonempty <srcfile>
↪<dstfile> <perm>
```

  Tests if the source file exists and is not empty (size > 0). If it is not empty it copies it to the given destination with the given permissions.

ExtUtils::Embed

Utilities for embedding Perl in C/C++ applications

SYNOPSIS

```
perl -MExtUtils::Embed -e xsinit
perl -MExtUtils::Embed -e ccopts
perl -MExtUtils::Embed -e ldopts
```

DESCRIPTION

`ExtUtils::Embed` provides utility functions for embedding a Perl interpreter and extensions in your C/C++ applications.
Typically, an application `Makefile` will invoke `ExtUtils::Embed` functions while building your application.

@EXPORT

`ExtUtils::Embed` exports the following functions:

xsinit(), ldopts(), ccopts(), perl_inc(), ccflags(), ccdlflags(), xsi_header(), xsi_protos(), xsi_body()

FUNCTIONS

- xsinit()

  When invoked as `` `perl -MExtUtils::Embed -e xsinit --` `` the following options are recognized:

  -o <output filename> (Defaults to **perlxsi.c**)

-o STDOUT will print to STDOUT.

-std (Write code for extensions that are linked with the current Perl.)

Any additional arguments are expected to be names of modules to generate code for.

When invoked with parameters the following are accepted and optional:

`xsinit($filename,$std,[@modules])`

Where,

$filename is equivalent to the **-o** option.

$std is boolean, equivalent to the **-std** option.

[@modules] is an array ref, same as additional arguments mentioned above.

- Examples

```
perl -MExtUtils::Embed -e xsinit -- -o xsinit.c Socket
```

This will generate code with an `xs_init` function that glues the perl `Socket::bootstrap` function to the C `boot_Socket` function and writes it to a file named `xsinit.c`.

Note that *DynaLoader* is a special case where it must call `boot_DynaLoader` directly.

```
perl -MExtUtils::Embed -e xsinit
```

This will generate code for linking with `DynaLoader` and each static extension found in `$Config{static_ext}`. The code is written to the default file name `perlxsi.c`.

```
perl -MExtUtils::Embed -e xsinit -- -o xsinit.c \
                            -std DBI DBD::Oracle
```

Here, code is written for all the currently linked extensions along with code for `DBI` and `DBD::Oracle`.

If you have a working `DynaLoader` then there is rarely any need to statically link in any other extensions.

- ldopts()

Output arguments for linking the Perl library and extensions to your application.

When invoked as `` `perl -MExtUtils::Embed -e ldopts --` `` the following options are recognized:

-std

Output arguments for linking the Perl library and any extensions linked with the current Perl.

-I <path1:path2>

Search path for ModuleName.a archives.
Default path is `@INC`. Library archives are expected to be found as `/some/path/auto/ModuleName/ModuleName.a` For example, when looking for `Socket.a` relative to a search path, we should find `auto/Socket/Socket.a`

When looking for `DBD::Oracle` relative to a search path, we should find `auto/DBD/Oracle/Oracle.a`

Keep in mind that you can always supply `/my/own/path/ModuleName.a` as an additional linker argument.

– <list of linker args>

Additional linker arguments to be considered.

Any additional arguments found before the – token are expected to be names of modules to generate code for.

When invoked with parameters the following are accepted and optional:

`ldopts($std,[@modules],[@link_args],$path)`

Where:

$std is boolean, equivalent to the **-std** option.

[@modules] is equivalent to additional arguments found before the – token.

[@link_args] is equivalent to arguments found after the – token.

$path is equivalent to the **-I** option.

In addition, when ldopts is called with parameters, it will return the argument string rather than print it to STDOUT.

- Examples

```
perl -MExtUtils::Embed -e ldopts
```

This will print arguments for linking with `libperl` and extensions found in `$Config{static_ext}`. This includes libraries found in `$Config{libs}` and the first ModuleName.a library for each extension that is found by searching `@INC` or the path specified by the **-I** option. In addition, when ModuleName.a is found, additional linker arguments are picked up from the `extralibs.ld` file in the same directory.

```
perl -MExtUtils::Embed -e ldopts -- -std Socket
```

This will do the same as the above example, along with printing additional arguments for linking with the `Socket` extension.

```
perl -MExtUtils::Embed -e ldopts -- -std Msql -- \
                       -L/usr/msql/lib -lmsql
```

Any arguments after the second '–' token are additional linker arguments that will be examined for potential conflict. If there is no conflict, the additional arguments will be part of the output.

- perl_inc()

For including perl header files this function simply prints:

```
-I$Config{archlibexp}/CORE
```

So, rather than having to say:

```
perl -MConfig -e 'print "-I$Config{archlibexp}/CORE"'
```

Just say:

```
perl -MExtUtils::Embed -e perl_inc
```

- ccflags(), ccdlflags()

  These functions simply print $Config{ccflags} and $Config{ccdlflags}

- ccopts()

  This function combines `perl_inc()`, `ccflags()` and `ccdlflags()` into one.

- xsi_header()

  This function simply returns a string defining the same `EXTERN_C` macro as `perlmain.c` along with #including `perl.h` and `EXTERN.h`.

- xsi_protos(@modules)

  This function returns a string of `boot_$ModuleName` prototypes for each @modules.

- xsi_body(@modules)

  This function returns a string of calls to `newXS()` that glue the module *bootstrap* function to *boot_ModuleName* for each @modules.

  `xsinit()` uses the xsi_* functions to generate most of its code.

EXAMPLES

For examples on how to use `ExtUtils::Embed` for building C/C++ applications with embedded perl, see perlembed.

SEE ALSO

perlembed

AUTHOR

Doug MacEachern <dougm@osf.org>

Based on ideas from Tim Bunce <Tim.Bunce@ig.co.uk> and `minimod.pl` by Andreas Koenig <k@anna.in-berlin.de> and Tim Bunce.

ExtUtils::HasCompiler

whether a distribution's compiled part will work (pperl)

SYNOPSIS

```
use ExtUtils::HasCompiler 0.014 'can_compile_loadable_object';
die 'This distribution requires a working compiler'
    unless can_compile_loadable_object(quiet => 1);
```

DESCRIPTION

The interface of ExtUtils::HasCompiler 0.025. pperl builds no C: the compiled parts it supports are built into it. So each function answers true when every module the distribution in the current directory loads through `XSLoader::load` or `bootstrap` is built into pperl, and false otherwise.

FUNCTIONS

can_compile_loadable_object(%opts)

can_compile_static_library(%opts)

can_compile_extension(%opts)

All three give the same answer. Without `quiet => 1` they print it. Other options are accepted and ignored.

ExtUtils::Install

install files from here to there

SYNOPSIS

```
use ExtUtils::Install;

install({ 'blib/lib' => 'some/install/dir' } );

uninstall($packlist);

pm_to_blib({ 'lib/Foo/Bar.pm' => 'blib/lib/Foo/Bar.pm' });
```

VERSION

2.22

DESCRIPTION

Handles the installing and uninstalling of perl modules, scripts, man pages, etc...

Both install() and uninstall() are specific to the way ExtUtils::MakeMaker handles the installation and deinstallation of perl modules. They are not designed as general purpose tools.

On some operating systems such as Win32 installation may not be possible until after a reboot has occurred. This can have varying consequences: removing an old DLL does not impact programs using the new one, but if a new DLL cannot be installed properly until reboot then anything depending on it must wait. The package variable

```
$ExtUtils::Install::MUST_REBOOT
```

is used to store this status.

If this variable is true then such an operation has occurred and anything depending on this module cannot proceed until a reboot has occurred.

If this value is defined but false then such an operation has occurred, but should not impact later operations.

Functions

install

```
# deprecated forms
install(\%from_to);
install(\%from_to, $verbose, $dry_run, $uninstall_shadows,
            $skip, $always_copy, \%result);

# recommended form as of 1.47
install([
    from_to => \%from_to,
    verbose => 1,
    dry_run => 0,
    uninstall_shadows => 1,
    skip => undef,
    always_copy => 1,
    result => \%install_results,
]);
```

Copies each directory tree of %from_to to its corresponding value preserving timestamps and permissions.

There are two keys with a special meaning in the hash: “read” and “write”. These contain packlist files. After the copying is done, install() will write the list of target files to $from_to{write}. If $from_to{read} is given the contents of this file will be merged into the written file. The read and the written file may be identical, but on AFS it is quite likely that people are installing to a different directory than the one where the files later appear.

If $verbose is true, will print out each file removed. Default is false. This is “make install VERBINST=1”. $verbose values going up to 5 show increasingly more diagnostics output.

If $dry_run is true it will only print what it was going to do without actually doing it. Default is false.

If $uninstall_shadows is true any differing versions throughout @INC will be uninstalled. This is "make install UNINST=1"

As of 1.37_02 install() supports the use of a list of patterns to filter out files that shouldn't be installed. If $skip is omitted or undefined then install will try to read the list from INSTALL.SKIP in the CWD. This file is a list of regular expressions and is just like the MANIFEST.SKIP file used by *ExtUtils::Manifest*.

A default site INSTALL.SKIP may be provided by setting then environment variable EU_INSTALL_SITE_SKIPFILE, this will only be used when there isn't a distribution specific INSTALL.SKIP. If the environment variable EU_INSTALL_IGNORE_SKIP is true then no install file filtering will be performed.

If $skip is undefined then the skip file will be autodetected and used if it is found. If $skip is a reference to an array then it is assumed the array contains the list of patterns, if $skip is a true non reference it is assumed to be the filename holding the list of patterns, any other value of $skip is taken to mean that no install filtering should occur.

Changes As of Version 1.47

As of version 1.47 the following additions were made to the install interface. Note that the new argument style and use of the %result hash is recommended.

The $always_copy parameter which when true causes files to be updated regardless as to whether they have changed, if it is defined but false then copies are made only if the files have changed, if it is undefined then the value of the environment variable EU_INSTALL_ALWAYS_COPY is used as default.

The %result hash will be populated with the various keys/subhashes reflecting the install. Currently these keys and their structure are:

```
install             => { $target    => $source },
install_fail        => { $target    => $source },
install_unchanged   => { $target    => $source },

install_filtered    => { $source    => $pattern },

uninstall           => { $uninstalled => $source },
uninstall_fail      => { $uninstalled => $source },
```

where `$source` is the filespec of the file being installed. `$target` is where it is being installed to, and `$uninstalled` is any shadow file that is in `@INC` or `$ENV{PERL5LIB}` or other standard locations, and `$pattern` is the pattern that caused a source file to be skipped. In future more keys will be added, such as to show created directories, however this requires changes in other modules and must therefore wait.

These keys will be populated before any exceptions are thrown should there be an error.

Note that all updates of the %result are additive, the hash will not be cleared before use, thus allowing status results of many installs to be easily aggregated.

NEW ARGUMENT STYLE

If there is only one argument and it is a reference to an array then the array is assumed to

contain a list of key-value pairs specifying the options. In this case the option "from_to" is mandatory. This style means that you do not have to supply a cryptic list of arguments and can use a self documenting argument list that is easier to understand.

This is now the recommended interface to install().

RETURN

If all actions were successful install will return a hashref of the results as described above for the $result parameter. If any action is a failure then install will die, therefore it is recommended to pass in the $result parameter instead of using the return value. If the result parameter is provided then the returned hashref will be the passed in hashref.

install_default

DISCOURAGED

```
install_default();
install_default($fullext);
```

Calls install() with arguments to copy a module from blib/ to the default site installation location.

$fullext is the name of the module converted to a directory (ie. Foo::Bar would be Foo/Bar). If $fullext is not specified, it will attempt to read it from @ARGV.

This is primarily useful for install scripts.

NOTE This function is not really useful because of the hard-coded install location with no way to control site vs core vs vendor directories and the strange way in which the module name is given. Consider its use discouraged.

uninstall

```
uninstall($packlist_file);
uninstall($packlist_file, $verbose, $dont_execute);
```

Removes the files listed in a $packlist_file.

If $verbose is true, will print out each file removed. Default is false.

If $dont_execute is true it will only print what it was going to do without actually doing it. Default is false.

pm_to_blib

```
pm_to_blib(\%from_to);
pm_to_blib(\%from_to, $autosplit_dir);
pm_to_blib(\%from_to, $autosplit_dir, $filter_cmd);
```

Copies each key of %from_to to its corresponding value efficiently. If an $autosplit_dir is provided, all .pm files will be autosplit into it. Any destination directories are created.

$filter_cmd is an optional shell command to run each .pm file through prior to splitting and copying. Input is the contents of the module, output the new module contents.

You can have an environment variable PERL_INSTALL_ROOT set which will be prepended as a directory to each installed file (and directory).

By default verbose output is generated, setting the PERL_INSTALL_QUIET environment variable will silence this output.

ENVIRONMENT

- **PERL_INSTALL_ROOT**

  Will be prepended to each install path.

- **EU_INSTALL_IGNORE_SKIP**

  Will prevent the automatic use of INSTALL.SKIP as the install skip file.

- **EU_INSTALL_SITE_SKIPFILE**

  If there is no INSTALL.SKIP file in the make directory then this value can be used to provide a default.

- **EU_INSTALL_ALWAYS_COPY**

  If this environment variable is true then normal install processes will always overwrite older identical files during the install process.

  Note that the alias EU_ALWAYS_COPY will be supported if EU_INSTALL_ALWAYS_COPY is not defined until at least the 1.50 release. Please ensure you use the correct EU_INSTALL_ALWAYS_COPY.

AUTHOR

Original author lost in the mists of time. Probably the same as Makemaker.

Production release currently maintained by demerphq `yves at cpan.org`, extensive changes by Michael G. Schwern.

Send bug reports via http://rt.cpan.org/. Please send your generated Makefile along with your report.

LICENSE

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See http://www.perl.com/perl/misc/Artistic.html

ExtUtils::Installed

Inventory management of installed modules

SYNOPSIS

```
use ExtUtils::Installed;
my ($inst) = ExtUtils::Installed->new( skip_cwd => 1 );
my (@modules) = $inst->modules();
my (@missing) = $inst->validate("DBI");
my $all_files = $inst->files("DBI");
my $files_below_usr_local = $inst->files("DBI", "all", "/usr/local
 ↪");
my $all_dirs = $inst->directories("DBI");
my $dirs_below_usr_local = $inst->directory_tree("DBI", "prog");
my $packlist = $inst->packlist("DBI");
```

DESCRIPTION

ExtUtils::Installed provides a standard way to find out what core and module files have been installed. It uses the information stored in .packlist files created during installation to provide this information. In addition it provides facilities to classify the installed files and to extract directory information from the .packlist files.

USAGE

The new() function searches for all the installed .packlists on the system, and stores their contents. The .packlists can be queried with the functions described below. Where it searches by default is determined by the settings found in `%Config::Config`, and what the value is of the PERL5LIB environment variable.

METHODS

Unless specified otherwise all method can be called as class methods, or as object methods. If called as class methods then the "default" object will be used, and if necessary created using the current processes %Config and @INC. See the 'default' option to new() for details.

- new()

  This takes optional named parameters. Without parameters, this searches for all the installed .packlists on the system using information from `%Config::Config` and the default module search paths `@INC`. The packlists are read using the *ExtUtils::Packlist* module.

  If the named parameter `skip_cwd` is true, the current directory `.` will be stripped from `@INC` before searching for .packlists. This keeps ExtUtils::Installed from finding modules installed in other perls that happen to be located below the current directory.

  If the named parameter `config_override` is specified, it should be a reference to a hash which contains all information usually found in `%Config::Config`. For example, you can obtain the configuration information for a separate perl installation and pass that in.

  ```
  my $yoda_cfg  = get_fake_config('yoda');
  my $yoda_inst =
            ExtUtils::Installed->new(config_override=>$yoda_cfg);
  ```

Similarly, the parameter `inc_override` may be a reference to an array which is used in place of the default module search paths from `@INC`.

```
use Config;
my @dirs = split(/\Q$Config{path_sep}\E/, $ENV{PERL5LIB});
my $p5libs = ExtUtils::Installed->new(inc_override=>\@dirs);
```

Note: You probably do not want to use these options alone, almost always you will want to set both together.

The parameter `extra_libs` can be used to specify **additional** paths to search for installed modules. For instance

```
my $installed =
        ExtUtils::Installed->new(extra_libs=>["/my/lib/path"]);
```

This should only be necessary if `/my/lib/path` is not in PERL5LIB.

Finally there is the 'default', and the related 'default_get' and 'default_set' options. These options control the "default" object which is provided by the class interface to the methods. Setting `default_get` to true tells the constructor to return the default object if it is defined. Setting `default_set` to true tells the constructor to make the default object the constructed object. Setting the `default` option is like setting both to true. This is used primarily internally and probably isn't interesting to any real user.

- modules()

  This returns a list of the names of all the installed modules. The perl 'core' is given the special name 'Perl'.

- files()

  This takes one mandatory parameter, the name of a module. It returns a list of all the filenames from the package. To obtain a list of core perl files, use the module name 'Perl'. Additional parameters are allowed. The first is one of the strings "prog", "doc" or "all", to select either just program files, just manual files or all files. The remaining parameters are a list of directories. The filenames returned will be restricted to those under the specified directories.

- directories()

  This takes one mandatory parameter, the name of a module. It returns a list of all the directories from the package. Additional parameters are allowed. The first is one of the strings "prog", "doc" or "all", to select either just program directories, just manual directories or all directories. The remaining parameters are a list of directories. The directories returned will be restricted to those under the specified directories. This method returns only the leaf directories that contain files from the specified module.

- directory_tree()

  This is identical in operation to directories(), except that it includes all the intermediate directories back up to the specified directories.

- validate()

This takes one mandatory parameter, the name of a module. It checks that all the files listed in the modules .packlist actually exist, and returns a list of any missing files. If an optional second argument which evaluates to true is given any missing files will be removed from the .packlist

- packlist()

  This returns the ExtUtils::Packlist object for the specified module.

- version()

  This returns the version number for the specified module.

EXAMPLE

See the example in *ExtUtils::Packlist*.

AUTHOR

Alan Burlison Alan.Burlison@uk.sun.com

ExtUtils::Liblist

determine libraries to use and how to use them

SYNOPSIS

```
require ExtUtils::Liblist;

$MM->ext($potential_libs, $verbose, $need_names);

# Usually you can get away with:
ExtUtils::Liblist->ext($potential_libs, $verbose, $need_names)
```

DESCRIPTION

This utility takes a list of libraries in the form `-llib1 -llib2 \-llib3` and returns lines suitable for inclusion in an extension Makefile. Extra library paths may be included with the form `-L/another/path` this will affect the searches for all subsequent libraries.

It returns an array of four or five scalar values: EXTRALIBS, BSLOADLIBS, LDLOADLIBS, LD_RUN_PATH, and, optionally, a reference to the array of the filenames of actual libraries. Some of these don't mean anything unless on Unix. See the details about those platform specifics below. The list of the filenames is returned only if $need_names argument is true.

Dependent libraries can be linked in one of three ways:

- For static extensions

  by the ld command when the perl binary is linked with the extension library. See EXTRALIBS below.

- For dynamic extensions at build/link time

  by the ld command when the shared object is built/linked. See LDLOADLIBS below.

- For dynamic extensions at load time

  by the DynaLoader when the shared object is loaded. See BSLOADLIBS below.

EXTRALIBS

List of libraries that need to be linked with when linking a perl binary which includes this extension. Only those libraries that actually exist are included. These are written to a file and used when linking perl.

LDLOADLIBS and LD_RUN_PATH

List of those libraries which can or must be linked into the shared library when created using ld. These may be static or dynamic libraries. LD_RUN_PATH is a colon separated list of the directories in LDLOADLIBS. It is passed as an environment variable to the process that links the shared library.

BSLOADLIBS

List of those libraries that are needed but can be linked in dynamically at run time on this platform. SunOS/Solaris does not need this because ld records the information (from LDLOADLIBS) into the object file. This list is used to create a .bs (bootstrap) file.

PORTABILITY

This module deals with a lot of system dependencies and has quite a few architecture specific `if`s in the code.

VMS implementation

The version of ext() which is executed under VMS differs from the Unix-OS/2 version in several respects:

- Input library and path specifications are accepted with or without the `-l` and `-L` prefixes used by Unix linkers. If neither prefix is present, a token is considered a directory to search if it is in fact a directory, and a library to search for otherwise. Authors who wish their extensions to be portable to Unix or OS/2 should use the Unix prefixes, since the Unix-OS/2 version of ext() requires them.
- Wherever possible, shareable images are preferred to object libraries, and object libraries to plain object files. In accordance with VMS naming conventions, ext() looks for files named _lib_shr and _lib_rtl; it also looks for *lib_lib and lib_lib* to accommodate Unix conventions used in some ported software.
- For each library that is found, an appropriate directive for a linker options file is generated. The return values are space-separated strings of these directives, rather than elements used on the linker command line.
- LDLOADLIBS contains both the libraries found based on `$potential_libs` and the CRTLs, if any, specified in Config.pm. EXTRALIBS contains just those libraries

found based on `$potential_libs`. BSLOADLIBS and LD_RUN_PATH are always empty.

In addition, an attempt is made to recognize several common Unix library names, and filter them out or convert them to their VMS equivalents, as appropriate.

In general, the VMS version of ext() should properly handle input from extensions originally designed for a Unix or VMS environment. If you encounter problems, or discover cases where the search could be improved, please let us know.

Win32 implementation

The version of ext() which is executed under Win32 differs from the Unix-OS/2 version in several respects:

- If `$potential_libs` is empty, the return value will be empty. Otherwise, the libraries specified by `$Config{perllibs}` (see Config.pm) will be appended to the list of `$potential_libs`. The libraries will be searched for in the directories specified in `$potential_libs`, `$Config{libpth}`, and in `$Config{installarchlib}/CORE`. For each library that is found, a space-separated list of fully qualified library pathnames is generated.
- Input library and path specifications are accepted with or without the `-l` and `-L` prefixes used by Unix linkers.

  An entry of the form `-La:\foo` specifies the `a:\foo` directory to look for the libraries that follow.

  An entry of the form `-lfoo` specifies the library `foo`, which may be spelled differently depending on what kind of compiler you are using. If you are using GCC, it gets translated to `libfoo.a`, but for other win32 compilers, it becomes `foo.lib`. If no files are found by those translated names, one more attempt is made to find them using either `foo.a` or `libfoo.lib`, depending on whether GCC or some other win32 compiler is being used, respectively.

  If neither the `-L` or `-l` prefix is present in an entry, the entry is considered a directory to search if it is in fact a directory, and a library to search for otherwise. The `$Config{lib_ext}` suffix will be appended to any entries that are not directories and don't already have the suffix.

  Note that the `-L` and `-l` prefixes are **not required**, but authors who wish their extensions to be portable to Unix or OS/2 should use the prefixes, since the Unix-OS/2 version of ext() requires them.
- Entries cannot be plain object files, as many Win32 compilers will not handle object files in the place of libraries.
- Entries in `$potential_libs` beginning with a colon and followed by alphanumeric characters are treated as flags. Unknown flags will be ignored.

  An entry that matches `/:nodefault/i` disables the appending of default libraries found in `$Config{perllibs}` (this should be only needed very rarely).

  An entry that matches `/:nosearch/i` disables all searching for the libraries specified after it. Translation of `-Lfoo` and `-lfoo` still happens as appropriate (depending on compiler being used, as reflected by `$Config{cc}`), but the entries are not verified to be valid files or directories.

An entry that matches `/:search/i` reenables searching for the libraries specified after it. You can put it at the end to enable searching for default libraries specified by `$Config{perllibs}`.

- The libraries specified may be a mixture of static libraries and import libraries (to link with DLLs). Since both kinds are used pretty transparently on the Win32 platform, we do not attempt to distinguish between them.
- LDLOADLIBS and EXTRALIBS are always identical under Win32, and BSLOADLIBS and LD_RUN_PATH are always empty (this may change in future).
- You must make sure that any paths and path components are properly surrounded with double-quotes if they contain spaces. For example, `$potential_libs` could be (literally):

```
"-Lc:\Program Files\vc\lib" msvcrt.lib "la test\foo bar.lib"
```

Note how the first and last entries are protected by quotes in order to protect the spaces.

- Since this module is most often used only indirectly from extension `Makefile.PL` files, here is an example `Makefile.PL` entry to add a library to the build process for an extension:

```
LIBS => ['-lgl']
```

When using GCC, that entry specifies that MakeMaker should first look for `libgl.a` (followed by `gl.a`) in all the locations specified by `$Config{libpth}`.

When using a compiler other than GCC, the above entry will search for `gl.lib` (followed by `libgl.lib`).

If the library happens to be in a location not in `$Config{libpth}`, you need:

```
LIBS => ['-Lc:\gllibs -lgl']
```

Here is a less often used example:

```
LIBS => ['-lgl', ':nosearch -Ld:\mesalibs -lmesa -luser32']
```

This specifies a search for library `gl` as before. If that search fails to find the library, it looks at the next item in the list. The `:nosearch` flag will prevent searching for the libraries that follow, so it simply returns the value as `-Ld:\mesalibs -lmesa -luser32`, since GCC can use that value as is with its linker.

When using the Visual C compiler, the second item is returned as `-libpath:d:\mesalibs mesa.lib user32.lib`.

When using the Borland compiler, the second item is returned as `-Ld:\mesalibs mesa.lib user32.lib`, and MakeMaker takes care of moving the `-Ld:\mesalibs` to the correct place in the linker command line.

SEE ALSO

ExtUtils::MakeMaker

ExtUtils::MM

OS adjusted ExtUtils::MakeMaker subclass

SYNOPSIS

```
require ExtUtils::MM;
my $mm = MM->new(...);
```

DESCRIPTION

FOR INTERNAL USE ONLY

ExtUtils::MM is a subclass of *ExtUtils::MakeMaker* which automatically chooses the appropriate OS specific subclass for you (ie. *ExtUtils::MM_Unix*, etc...).

It also provides a convenient alias via the MM class (I didn't want MakeMaker modules outside of ExtUtils/).

This class might turn out to be a temporary solution, but MM won't go away.

ExtUtils::MM_Any

Platform-agnostic MM methods

SYNOPSIS

```
FOR INTERNAL USE ONLY!

package ExtUtils::MM_SomeOS;

# Temporarily, you have to subclass both.  Put MM_Any first.
require ExtUtils::MM_Any;
require ExtUtils::MM_Unix;
@ISA = qw(ExtUtils::MM_Any ExtUtils::Unix);
```

DESCRIPTION

FOR INTERNAL USE ONLY!

ExtUtils::MM_Any is a superclass for the ExtUtils::MM_* set of modules. It contains methods which are either inherently cross-platform or are written in a cross-platform manner.

Subclass off of ExtUtils::MM_Any *and* *ExtUtils::MM_Unix*. This is a temporary solution.

THIS MAY BE TEMPORARY!

METHODS

Any methods marked *Abstract* must be implemented by subclasses.

Cross-platform helper methods

These are methods which help writing cross-platform code.

os_flavor *Abstract*

```
my @os_flavor = $mm->os_flavor;
```

@os_flavor is the style of operating system this is, usually corresponding to the MM_*.pm file we're using.

The first element of @os_flavor is the major family (ie. Unix, Windows, VMS, OS/2, etc...) and the rest are sub families.

Some examples:

```
Cygwin98       ('Unix',  'Cygwin', 'Cygwin9x')
Windows        ('Win32')
Win98          ('Win32', 'Win9x')
Linux          ('Unix',  'Linux')
MacOS X        ('Unix',  'Darwin', 'MacOS', 'MacOS X')
OS/2           ('OS/2')
```

This is used to write code for styles of operating system. See os_flavor_is() for use.

os_flavor_is

```
my $is_this_flavor = $mm->os_flavor_is($this_flavor);
my $is_this_flavor = $mm->os_flavor_is(@one_of_these_flavors);
```

Checks to see if the current operating system is one of the given flavors.

This is useful for code like:

```
if( $mm->os_flavor_is('Unix') ) {
    $out = `foo 2>&1`;
}
else {
    $out = `foo`;
}
```

can_load_xs

```
my $can_load_xs = $self->can_load_xs;
```

can_run

```
use ExtUtils::MM;
my $runnable = MM->can_run($Config{make});
```

If called in a scalar context it will return the full path to the binary you asked for if it was found, or `undef` if it was not.

If called in a list context, it will return a list of the full paths to instances of the binary where found in `PATH`, or an empty list if it was not found.

Copied from *IPC::Cmd*, but modified into a method (and removed `$INSTANCES` capability).

can_redirect_error

```
$useredirect = MM->can_redirect_error;
```

True if on an OS where qx operator (or backticks) can redirect `STDERR` onto `STDOUT`.

is_make_type

```
my $is_dmake = $self->is_make_type('dmake');
```

Returns true if `$self->make` is the given type; possibilities are:

```
gmake    GNU make
dmake
nmake
bsdmake  BSD pmake-derived
```

can_dep_space

```
my $can_dep_space = $self->can_dep_space;
```

Returns true if `make` can handle (probably by quoting) dependencies that contain a space. Currently known true for GNU make, false for BSD pmake derivative.

quote_dep

```
$text = $mm->quote_dep($text);
```

Method that protects Makefile single-value constants (mainly filenames), so that make will still treat them as single values even if they inconveniently have spaces in. If the make program being used cannot achieve such protection and the given text would need it, throws an exception.

split_command

```
my @cmds = $MM->split_command($cmd, @args);
```

Most OS have a maximum command length they can execute at once. Large modules can easily generate commands well past that limit. Its necessary to split long commands up into a series of shorter commands.

`split_command` will return a series of @cmds each processing part of the args. Collectively they will process all the arguments. Each individual line in @cmds will not be longer than the $self->max_exec_len being careful to take into account macro expansion.

$cmd should include any switches and repeated initial arguments. If it has newlines, they should be already escaped.

If no @args are given, no @cmds will be returned.

Pairs of arguments will always be preserved in a single command, this is a heuristic for things like pm_to_blib and pod2man which work on pairs of arguments. This makes things like this safe:

```
$self->split_command($cmd, %pod2man);
```

make_type

Returns a suitable string describing the type of makefile being written.

stashmeta

```
my @recipelines = $MM->stashmeta($text, $file);
```

Generates a set of `@recipelines` which will result in the literal `$text` ending up in literal `$file` when the recipe is executed. Call it once, with all the text you want in `$file`. Make macros will not be expanded, so the locations will be fixed at configure-time, not at build-time.

echo

```
my @commands = $MM->echo($text);
my @commands = $MM->echo($text, $file);
my @commands = $MM->echo($text, $file, \%opts);
```

Generates a set of @commands which print the $text to a $file.

If $file is not given, output goes to STDOUT.

If $opts{append} is true the $file will be appended to rather than overwritten. Default is to overwrite.

If $opts{allow_variables} is true, make variables of the form `$(...)` will not be escaped. Other $ will. Default is to escape all $.

Example of use:

```
my $make = join '', map "\t$_\n", $MM->echo($text, $file);
```

wraplist

```
my $args = $mm->wraplist(@list);
```

Takes an array of items and turns them into a well-formatted list of arguments. In most cases this is simply something like:

```
FOO \
BAR \
BAZ
```

maketext_filter

```
my $filter_make_text = $mm->maketext_filter($make_text);
```

The text of the Makefile is run through this method before writing to disk. It allows systems a chance to make portability fixes to the Makefile.

By default it does nothing.

This method is protected and not intended to be called outside of MakeMaker.

cd *Abstract*

```
my $subdir_cmd = $MM->cd($subdir, @cmds);
```

This will generate a make fragment which runs the @cmds in the given $dir. The rough equivalent to this, except cross platform.

```
cd $subdir && $cmd
```

Currently $dir can only go down one level. “foo” is fine. “foo/bar” is not. “../foo” is right out.

The resulting $subdir_cmd has no leading tab nor trailing newline. This makes it easier to embed in a make string. For example.

```
    my $make = sprintf <<'CODE', $subdir_cmd;
foo :
    $(ECHO) what
    %s
    $(ECHO) mouche
CODE
```

oneliner *Abstract*

```
my $oneliner = $MM->oneliner($perl_code);
my $oneliner = $MM->oneliner($perl_code, \@switches);
```

This will generate a perl one-liner safe for the particular platform you're on based on the given $perl_code and @switches (a -e is assumed) suitable for using in a make target. It will use the proper shell quoting and escapes.

$(PERLRUN) will be used as perl.

Any newlines in $perl_code will be escaped. Leading and trailing newlines will be stripped. Makes this idiom much easier:

```
    my $code = $MM->oneliner(<<'CODE', [...switches...]);
some code here
another line here
CODE
```

Usage might be something like:

```
# an echo emulation
$oneliner = $MM->oneliner('print "Foo\n"');
$make = '$oneliner > somefile';
```

Dollar signs in the $perl_code will be protected from make using the `quote_literal` method, unless they are recognised as being a make variable, `$(varname)`, in which case they will be left for make to expand. Remember to quote make macros else it might be used as a bareword. For example:

```
# Assign the value of the $(VERSION_FROM) make macro to $vf.
$oneliner = $MM->oneliner('$vf = "$(VERSION_FROM)"');
```

It's currently very simple, and may be expanded sometime in the future to include more flexible code and switches.

quote_literal *Abstract*

```
my $safe_text = $MM->quote_literal($text);
my $safe_text = $MM->quote_literal($text, \%options);
```

This will quote $text so it is interpreted literally in the shell.

For example, on Unix this would escape any single-quotes in $text and put single-quotes around the whole thing.

If $options{allow_variables} is true it will leave `'$(FOO)'` make variables untouched. If false they will be escaped like any other `$`. Defaults to true.

escape_dollarsigns

```
my $escaped_text = $MM->escape_dollarsigns($text);
```

Escapes stray $ so they are not interpreted as make variables.

It lets by $(...).

escape_all_dollarsigns

```
my $escaped_text = $MM->escape_all_dollarsigns($text);
```

Escapes all $ so they are not interpreted as make variables.

escape_newlines *Abstract*

```
my $escaped_text = $MM->escape_newlines($text);
```

Shell escapes newlines in $text.

max_exec_len *Abstract*

```
my $max_exec_len = $MM->max_exec_len;
```

Calculates the maximum command size the OS can exec. Effectively, this is the max size of a shell command line.

make

```
my $make = $MM->make;
```

Returns the make variant we're generating the Makefile for. This attempts to do some normalization on the information from %Config or the user.

Targets

These are methods which produce make targets.

all_target

Generate the default target 'all'.

blibdirs_target

```
my $make_frag = $mm->blibdirs_target;
```

Creates the blibdirs target which creates all the directories we use in blib/.

The blibdirs.ts target is deprecated. Depend on blibdirs instead.

clean (o)

Defines the clean target.

clean_subdirs_target

```
my $make_frag = $MM->clean_subdirs_target;
```

Returns the clean_subdirs target. This is used by the clean target to call clean on any subdirectories which contain Makefiles.

dir_target

```
my $make_frag = $mm->dir_target(@directories);
```

Generates targets to create the specified directories and set its permission to PERM_DIR.

Because depending on a directory to just ensure it exists doesn't work too well (the modified time changes too often) dir_target() creates a .exists file in the created directory. It is this you should depend on. For portability purposes you should use the $(DIRFILESEP) macro rather than a '/' to separate the directory from the file.

```
yourdirectory$(DIRFILESEP).exists
```

distdir

Defines the scratch directory target that will hold the distribution before tar-ing (or shar-ing).

dist_test

Defines a target that produces the distribution in the scratch directory, and runs 'perl Makefile.PL; make ;make test' in that subdirectory.

xs_dlsyms_arg

Returns command-line arg(s) to linker for file listing dlsyms to export. Defaults to returning empty string, can be overridden by e.g. AIX.

xs_dlsyms_ext

Returns file-extension for `xs_make_dlsyms` method's output file, including any "." character.

xs_dlsyms_extra

Returns any extra text to be prepended to the `$extra` argument of `xs_make_dlsyms`.

xs_dlsyms_iterator

Iterates over necessary shared objects, calling `xs_make_dlsyms` method for each with appropriate arguments.

xs_make_dlsyms

```
$self->xs_make_dlsyms(
    \%attribs, # hashref from %attribs in caller
    "$self->{BASEEXT}.def", # output file for Makefile target
    'Makefile.PL', # dependency
    $self->{NAME}, # shared object's "name"
    $self->{DLBASE}, # last ::-separated part of name
    $attribs{DL_FUNCS} || $self->{DL_FUNCS} || {}, # various params
    $attribs{FUNCLIST} || $self->{FUNCLIST} || [],
    $attribs{IMPORTS} || $self->{IMPORTS} || {},
    $attribs{DL_VARS} || $self->{DL_VARS} || [],
    # optional extra param that will be added as param to Mksymlists
);
```

Utility method that returns Makefile snippet to call `Mksymlists`.

dynamic (o)

Defines the dynamic target.

makemakerdflt_target

```
my $make_frag = $mm->makemakerdflt_target
```

Returns a make fragment with the makemakerdeflt_target specified. This target is the first target in the Makefile, is the default target and simply points off to 'all' just in case any make variant gets confused or something gets snuck in before the real 'all' target.

manifypods_target

```
my $manifypods_target = $self->manifypods_target;
```

Generates the manifypods target. This target generates man pages from all POD files in MAN1PODS and MAN3PODS.

metafile_target

```
my $target = $mm->metafile_target;
```

Generate the metafile target.

Writes the file META.yml (YAML encoded meta-data) and META.json (JSON encoded meta-data) about the module in the distdir. The format follows Module::Build's as closely as possible.

metafile_data

```
my $metadata_hashref = $mm->metafile_data(\%meta_add, \%meta_merge);
```

Returns the data which MakeMaker turns into the META.yml file and the META.json file. It is always in version 2.0 of the format.

Values of %meta_add will overwrite any existing metadata in those keys. %meta_merge will be merged with them.

metafile_file

```
my $meta_yml = $mm->metafile_file(@metadata_pairs);
```

Turns the @metadata_pairs into YAML.

This method does not implement a complete YAML dumper, being limited to dump a hash with values which are strings, undef's or nested hashes and arrays of strings. No quoting/escaping is done.

distmeta_target

```
my $make_frag = $mm->distmeta_target;
```

Generates the distmeta target to add META.yml and META.json to the MANIFEST in the distdir.

mymeta

```
my $mymeta = $mm->mymeta;
```

Generate MYMETA information as a hash either from an existing CPAN Meta file (META.json or META.yml) or from internal data.

write_mymeta

```
$self->write_mymeta( $mymeta );
```

Write MYMETA information to MYMETA.json and MYMETA.yml.

realclean (o)

Defines the realclean target.

realclean_subdirs_target

```
my $make_frag = $MM->realclean_subdirs_target;
```

Returns the realclean_subdirs target. This is used by the realclean target to call realclean on any subdirectories which contain Makefiles.

signature_target

```
my $target = $mm->signature_target;
```

Generate the signature target.

Writes the file SIGNATURE with “cpansign -s”.

distsignature_target

```
my $make_frag = $mm->distsignature_target;
```

Generates the distsignature target to add SIGNATURE to the MANIFEST in the distdir.

special_targets

```
my $make_frag = $mm->special_targets
```

Returns a make fragment containing any targets which have special meaning to make. For example, .SUFFIXES and .PHONY.

Init methods

Methods which help initialize the MakeMaker object and macros.

init_ABSTRACT

```
$mm->init_ABSTRACT
```

init_INST

```
$mm->init_INST;
```

Called by init_main. Those are handled in init_xs.

init_INSTALL

```
$mm->init_INSTALL;
```

Called by init_main. Sets up all INSTALL_* variables (except INSTALLDIRS) and *PREFIX.

init_INSTALL_from_PREFIX

```
$mm->init_INSTALL_from_PREFIX;
```

init_from_INSTALL_BASE

```
$mm->init_from_INSTALL_BASE
```

init_VERSION *Abstract*

```
$mm->init_VERSION
```

Initialize macros representing versions of MakeMaker and other tools

MAKEMAKER: path to the MakeMaker module.

MM_VERSION: ExtUtils::MakeMaker Version

MM_REVISION: ExtUtils::MakeMaker version control revision (for backwards compat)

VERSION: version of your module

VERSION_MACRO: which macro represents the version (usually 'VERSION')

VERSION_SYM: like version but safe for use as an RCS revision number

DEFINE_VERSION: -D line to set the module version when compiling

Defaults to $(VERSION)

Called by init_main.

init_tools

```
$MM->init_tools();
```

Initializes the simple macro definitions used by tools_other() and places them in the $MM object. These use conservative cross platform versions and should be overridden with platform specific versions for performance.

Defines at least these macros.

```
Macro             Description

NOOP              Do nothing
NOECHO            Tell make not to display the command itself

SHELL             Program used to run shell commands

ECHO              Print text adding a newline on the end
RM_F              Remove a file
RM_RF             Remove a directory
TOUCH             Update a file's timestamp
TEST_F            Test for a file's existence
TEST_S            Test the size of a file
CP                Copy a file
CP_NONEMPTY       Copy a file if it is not empty
MV                Move a file
```

(continues on next page)

(continued from previous page)

```
CHMOD             Change permissions on a file
FALSE             Exit with non-zero
TRUE              Exit with zero

UMASK_NULL        Nullify umask
DEV_NULL          Suppress all command output
```

init_others

```
$MM->init_others();
```

Initializes the macro definitions having to do with compiling and linking used by tools_other() and places them in the $MM object.

If there is no description, its the same as the parameter to WriteMakefile() documented in *ExtUtils::MakeMaker*.

tools_other

```
my $make_frag = $MM->tools_other;
```

Returns a make fragment containing definitions for the macros init_others() initializes.

init_DIRFILESEP *Abstract*

```
$MM->init_DIRFILESEP;
my $dirfilesep = $MM->{DIRFILESEP};
```

Initializes the DIRFILESEP macro which is the separator between the directory and filename in a filepath. ie. / on Unix, \ on Win32 and nothing on VMS.

For example:

```
# instead of $(INST_ARCHAUTODIR)/extralibs.ld
$(INST_ARCHAUTODIR)$(DIRFILESEP)extralibs.ld
```

Something of a hack but it prevents a lot of code duplication between MM_* variants.

Do not use this as a separator between directories. Some operating systems use different separators between subdirectories as between directories and filenames (for example: VOLUME:[dir1.dir2]file on VMS).

init_linker *Abstract*

```
$mm->init_linker;
```

Initialize macros which have to do with linking.

PERL_ARCHIVE: path to libperl.a equivalent to be linked to dynamic extensions.

PERL_ARCHIVE_AFTER: path to a library which should be put on the linker command line *after* the external libraries to be linked to dynamic extensions. This may be needed if the linker is one-pass, and Perl includes some overrides for C RTL functions, such as malloc().

EXPORT_LIST: name of a file that is passed to linker to define symbols to be exported.

Some OSes do not need these in which case leave it blank.

init_platform

```
$mm->init_platform
```

Initialize any macros which are for platform specific use only.

A typical one is the version number of your OS specific module. (ie. MM_Unix_VERSION or MM_VMS_VERSION).

init_MAKE

```
$mm->init_MAKE
```

Initialize MAKE from either a MAKE environment variable or $Config{make}.

Tools

A grab bag of methods to generate specific macros and commands.

manifypods

Defines targets and routines to translate the pods into manpages and put them into the INST_* directories.

POD2MAN_macro

```
my $pod2man_macro = $self->POD2MAN_macro
```

Returns a definition for the POD2MAN macro. This is a program which emulates the pod2man utility. You can add more switches to the command by simply appending them on the macro.

Typical usage:

```
$(POD2MAN) --section=3 --perm_rw=$(PERM_RW) podfile1 man_page1 ...
```

test_via_harness

```
my $command = $mm->test_via_harness($perl, $tests);
```

Returns a $command line which runs the given set of $tests with Test::Harness and the given $perl.

Used on the t/*.t files.

test_via_script

```
my $command = $mm->test_via_script($perl, $script);
```

Returns a $command line which just runs a single test without Test::Harness. No checks are done on the results, they're just printed.

Used for test.pl, since they don't always follow Test::Harness formatting.

tool_autosplit

Defines a simple perl call that runs autosplit. May be deprecated by pm_to_blib soon.

arch_check

```
my $arch_ok = $mm->arch_check(
    $INC{"Config.pm"},
    File::Spec->catfile($Config{archlibexp}, "Config.pm")
);
```

A sanity check that what Perl thinks the architecture is and what Config thinks the architecture is are the same. If they're not it will return false and show a diagnostic message.

When building Perl it will always return true, as nothing is installed yet.

The interface is a bit odd because this is the result of a quick refactoring. Don't rely on it.

File::Spec wrappers

ExtUtils::MM_Any is a subclass of *File::Spec*. The methods noted here override File::Spec.

catfile

File::Spec <= 0.83 has a bug where the file part of catfile is not canonicalized. This override fixes that bug.

Misc

Methods I can't really figure out where they should go yet.

find_tests

```
my $test = $mm->find_tests;
```

Returns a string suitable for feeding to the shell to return all tests in t/*.t.

find_tests_recursive

```
my $tests = $mm->find_tests_recursive;
```

Returns a string suitable for feeding to the shell to return all tests in t/ but recursively. Equivalent to

```
my $tests = $mm->find_tests_recursive_in('t');
```

find_tests_recursive_in

```
my $tests = $mm->find_tests_recursive_in($dir);
```

Returns a string suitable for feeding to the shell to return all tests in $dir recursively.

extra_clean_files

```
my @files_to_clean = $MM->extra_clean_files;
```

Returns a list of OS specific files to be removed in the clean target in addition to the usual set.

installvars

```
my @installvars = $mm->installvars;
```

A list of all the INSTALL* variables without the INSTALL prefix. Useful for iteration or building related variable sets.

libscan

```
my $wanted = $self->libscan($path);
```

Takes a path to a file or dir and returns an empty string if we don't want to include this file in the library. Otherwise it returns the the $path unchanged.

Mainly used to exclude version control administrative directories and base-level `README.pod` from installation.

platform_constants

```
my $make_frag = $mm->platform_constants
```

Returns a make fragment defining all the macros initialized in init_platform() rather than put them in constants().

post_constants (o)

Returns an empty string per default. Dedicated to overrides from within Makefile.PL after all constants have been defined.

post_initialize (o)

Returns an empty string per default. Used in Makefile.PLs to add some chunk of text to the Makefile after the object is initialized.

postamble (o)

Returns an empty string. Can be used in Makefile.PLs to write some text to the Makefile at the end.

AUTHOR

Michael G Schwern schwern@pobox.com and the denizens of makemaker@perl.org with code from ExtUtils::MM_Unix and ExtUtils::MM_Win32.

ExtUtils::MM_Unix

methods used by ExtUtils::MakeMaker

SYNOPSIS

```
require ExtUtils::MM_Unix;
```

DESCRIPTION

The methods provided by this package are designed to be used in conjunction with *ExtUtils::MakeMaker*. When MakeMaker writes a Makefile, it creates one or more objects that inherit their methods from a package *MM*. MM itself doesn't provide any methods, but it ISA ExtUtils::MM_Unix class. The inheritance tree of MM lets operating specific packages take the responsibility for all the methods provided by MM_Unix. We are trying to reduce the number of the necessary overrides by defining rather primitive operations within ExtUtils::MM_Unix.

If you are going to write a platform specific MM package, please try to limit the necessary overrides to primitive methods, and if it is not possible to do so, let's work out how to achieve that gain.

If you are overriding any of these methods in your Makefile.PL (in the MY class), please report that to the makemaker mailing list. We are trying to minimize the necessary method overrides and switch to data driven Makefile.PLs wherever possible. In the long run less methods will be overridable via the MY class.

METHODS

The following description of methods is still under development. Please refer to the code for not suitably documented sections and complain loudly to the makemaker@perl.org mailing list. Better yet, provide a patch.

Not all of the methods below are overridable in a Makefile.PL. Overridable methods are marked as (o). All methods are overridable by a platform specific MM_*.pm file.

Cross-platform methods are being moved into *MM_Any*. If you can't find something that used to be in here, look in MM_Any.

Methods

- os_flavor

  Simply says that we're Unix.

- c_o (o)

  Defines the suffix rules to compile different flavors of C files to object files.

- xs_obj_opt

  Takes the object file as an argument, and returns the portion of compile command-line that will output to the specified object file.

- dbgoutflag

  Returns a CC flag that tells the CC to emit a separate debugging symbol file when compiling an object file.

- cflags (o)

  Does very much the same as the cflags script in the perl distribution. It doesn't return the whole compiler command line, but initializes all of its parts. The const_cccmd method then actually returns the definition of the CCCMD macro which uses these parts.

- const_cccmd (o)

  Returns the full compiler call for C programs and stores the definition in CONST_CCCMD.

- const_config (o)

  Sets SHELL if needed, then defines a couple of constants in the Makefile that are imported from %Config.

- const_loadlibs (o)

  Defines EXTRALIBS, LDLOADLIBS, BSLOADLIBS, LD_RUN_PATH. See *ExtUtils::Liblist* for details.

- constants (o)

  ```
  my $make_frag = $mm->constants;
  ```

  Prints out macros for lots of constants.

- dep_constants (o)

```
my $make_frag = $mm->dep_constants;
```

Prints out macros for dependency constants.

- depend (o)

  Same as macro for the depend attribute.

- init_DEST

```
$mm->init_DEST
```

Defines the DESTDIR and DEST* variables paralleling the INSTALL*.

- init_dist

```
$mm->init_dist;
```

Defines a lot of macros for distribution support.

```
macro         description                     default

TAR           tar command to use              tar
TARFLAGS      flags to pass to TAR            cvf

ZIP           zip command to use              zip
ZIPFLAGS      flags to pass to ZIP            -r

COMPRESS      compression command to          gzip --best
              use for tarfiles
SUFFIX        suffix to put on                .gz
              compressed files

SHAR          shar command to use             shar

PREOP         extra commands to run before
              making the archive
POSTOP        extra commands to run after
              making the archive

TO_UNIX       a command to convert linefeeds
              to Unix style in your archive

CI            command to checkin your         ci -u
              sources to version control
RCS_LABEL     command to label your sources   rcs -Nv$(VERSION_
 ↪SYM): -q
              just after CI is run

DIST_CP       $how argument to manicopy()     best
              when the distdir is created

DIST_DEFAULT  default target to use to        tardist
```

(continues on next page)

(continued from previous page)

```
                create a distribution

DISTVNAME       name of the resulting archive   $(DISTNAME)-
 ↪$(VERSION)
                (minus suffixes)
```

- dist (o)

```
my $dist_macros = $mm->dist(%overrides);
```

Generates a make fragment defining all the macros initialized in init_dist.

- dist_basics (o)

Defines the targets distclean, distcheck, skipcheck, manifest, veryclean.

- dist_ci (o)

Defines a check in target for RCS.

- dist_core (o)

```
my $dist_make_fragment = $MM->dist_core;
```

Puts the targets necessary for 'make dist' together into one make fragment.

- **dist_target**

```
my $make_frag = $MM->dist_target;
```

Returns the 'dist' target to make an archive for distribution. This target simply checks to make sure the Makefile is up-to-date and depends on $(DIST_DEFAULT).

- **tardist_target**

```
my $make_frag = $MM->tardist_target;
```

Returns the 'tardist' target which is simply so 'make tardist' works. The real work is done by the dynamically named tardistfile_target() method, tardist should have that as a dependency.

- **zipdist_target**

```
my $make_frag = $MM->zipdist_target;
```

Returns the 'zipdist' target which is simply so 'make zipdist' works. The real work is done by the dynamically named zipdistfile_target() method, zipdist should have that as a dependency.

- **tarfile_target**

```
my $make_frag = $MM->tarfile_target;
```

The name of this target is the name of the tarball generated by tardist. This target does the actual work of turning the distdir into a tarball.

- zipfile_target

```
my $make_frag = $MM->zipfile_target;
```

The name of this target is the name of the zip file generated by zipdist. This target does the actual work of turning the distdir into a zip file.

- uutardist_target

```
my $make_frag = $MM->uutardist_target;
```

Converts the tarfile into a uuencoded file

- shdist_target

```
my $make_frag = $MM->shdist_target;
```

Converts the distdir into a shell archive.

- dlsyms (o)

Used by some OS' to define DL_FUNCS and DL_VARS and write the *.exp files.

Normally just returns an empty string.

- dynamic_bs (o)

Defines targets for bootstrap files.

- dynamic_lib (o)

Defines how to produce the *.so (or equivalent) files.

- xs_dynamic_lib_macros

Defines the macros for the `dynamic_lib` section.

- xs_make_dynamic_lib

Defines the recipes for the `dynamic_lib` section.

- exescan

Deprecated method. Use libscan instead.

- extliblist

Called by init_others, and calls ext ExtUtils::Liblist. See *ExtUtils::Liblist* for details.

- find_perl

Finds the executables PERL and FULLPERL

- fixin

```
$mm->fixin(@files);
```

Inserts the sharpbang or equivalent magic number to a set of @files.

- force (o)

Writes an empty FORCE: target.

- guess_name

  Guess the name of this package by examining the working directory's name. MakeMaker calls this only if the developer has not supplied a NAME attribute.

- has_link_code

  Does not descend into subdirectories as needs_linking() does.

  Under pperl this is always false. These two, needs_linking() and the strict.pm check in init_main() are the differences between this file and ExtUtils::MakeMaker's own.

- init_dirscan

  Called by init_main.

- init_MANPODS

  Determines if man pages should be generated and initializes MAN1PODS and MAN3PODS as appropriate.

- init_MAN1PODS

  Initializes MAN1PODS from the list of EXE_FILES.

- init_MAN3PODS

  Initializes MAN3PODS from the list of PM files.

- init_PM

  Initializes PMLIBDIRS and PM from PMLIBDIRS.

- init_DIRFILESEP

  Using / for Unix. Called by init_main.

- init_main

- init_tools

  Initializes tools to use their common (and faster) Unix commands.

- init_linker

  Unix has no need of special linker flags.

- init_PERL

```
$mm->init_PERL;
```

  Called by init_main. Sets up ABSPERL, PERL, FULLPERL and all the *PERLRUN* permutations.

```
PERL is allowed to be miniperl
FULLPERL must be a complete perl

ABSPERL is PERL converted to an absolute path

*PERLRUN contains everything necessary to run perl, find it's
     libraries, etc...
```

(continues on next page)

(continued from previous page)

```
*PERLRUNINST is *PERLRUN + everything necessary to find the
     modules being built.
```

- init_platform
- platform_constants

  Add MM_Unix_VERSION.
- init_PERM

```
$mm->init_PERM
```

  Called by init_main. Initializes PERL_*
- init_xs

```
$mm->init_xs
```

  Currently just INST_STATIC, INST_DYNAMIC and INST_BOOT.
- install (o)

  Defines the install target.
- installbin (o)

  Defines targets to make and to install EXE_FILES.
- linkext (o)

  Defines the linkext target which in turn defines the LINKTYPE.
- lsdir

  Takes as arguments a directory name and a regular expression. Returns all entries in the directory that match the regular expression.
- macro (o)

  Simple subroutine to insert the macros defined by the macro attribute into the Makefile.
- makeaperl (o)

  Called by staticmake. Defines how to write the Makefile to produce a static new perl.

  By default the Makefile produced includes all the static extensions in the perl library. (Purified versions of library files, e.g., DynaLoader_pure_p1_c0_032.a are automatically ignored to avoid link errors.)
- xs_static_lib_is_xs (o)

  Called by a utility method of makeaperl.
- makefile (o)

  Defines how to rewrite the Makefile.

- maybe_command

  Returns true, if the argument is likely to be a command.

- needs_linking (o)

  Does this module need linking? Looks into subdirectory objects (see also has_link_code())

  Under pperl nothing is linked, for the reason has_link_code() gives: neither this object nor a subdirectory's has link code, and there is no perl to relink (MAKEAPERL).

- parse_abstract

  parse a file and return what you think is the ABSTRACT

- parse_version

```
my $version = MM->parse_version($file);
```

  Parse a $file and return what $VERSION is set to by the first assignment. It will return the string "undef" if it can't figure out what $VERSION is. $VERSION should be for all to see, so `our $VERSION` or plain $VERSION are okay, but `my $VERSION` is not.

  `package Foo VERSION` is also checked for. The first version declaration found is used, but this may change as it differs from how Perl does it.

  parse_version() will try to `use version` before checking for `$VERSION` so the following will work.

```
$VERSION = qv(1.2.3);
```

- pasthru (o)

  Defines the string that is passed to recursive make calls in subdirectories. The variables like `PASTHRU_DEFINE` are used in each level, and passed downwards on the command-line with e.g. the value of that level's DEFINE. Example:

```
# Level 0 has DEFINE = -Dfunky
# This code will define level 0's PASTHRU=PASTHRU_DEFINE="
 ↪$(DEFINE)
#     $(PASTHRU_DEFINE)"
# Level 0's $(CCCMD) will include macros $(DEFINE) and
 ↪$(PASTHRU_DEFINE)
# So will level 1's, so when level 1 compiles, it will get␣
 ↪right values
# And so ad infinitum
```

- perl_script

  Takes one argument, a file name, and returns the file name, if the argument is likely to be a perl script. On MM_Unix this is true for any ordinary, readable file.

- perldepend (o)

  Defines the dependency from all *.h files that come with the perl distribution.

- pm_to_blib

  Defines target that copies all files in the hash PM to their destination and autosplits them. See *"DESCRIPTION" in ExtUtils::Install*

- ppd

  Defines target that creates a PPD (Perl Package Description) file for a binary distribution.

- prefixify

```
$MM->prefixify($var, $prefix, $new_prefix, $default);
```

  Using either $MM->{uc $var} || $Config{lc $var}, it will attempt to replace it's $prefix with a $new_prefix.

  Should the $prefix fail to match *AND* a PREFIX was given as an argument to WriteMakefile() it will set it to the $new_prefix + $default. This is for systems whose file layouts don't neatly fit into our ideas of prefixes.

  This is for heuristics which attempt to create directory structures that mirror those of the installed perl.

  For example:

```
$MM->prefixify('installman1dir', '/usr', '/home/foo', 'man/man1
 ↪');
```

  this will attempt to remove '/usr' from the front of the $MM->{INSTALLMAN1DIR} path (initializing it to $Config{installman1dir} if necessary) and replace it with '/home/foo'. If this fails it will simply use '/home/foo/man/man1'.

- processPL (o)

  Defines targets to run *.PL files.

- specify_shell

  Specify SHELL if needed - not done on Unix.

- quote_paren

  Backslashes parentheses `()` in command line arguments. Doesn't handle recursive Makefile `$(...)` constructs, but handles simple ones.

- replace_manpage_separator

```
my $man_name = $MM->replace_manpage_separator($file_path);
```

  Takes the name of a package, which may be a nested package, in the form 'Foo/Bar.pm' and replaces the slash with `::` or something else safe for a man page file name. Returns the replacement.

- cd

  On BSD make, will add a countervailing `cd ..` on each command since parallel builds run all the commands in a recipe in the same shell.

- oneliner

- quote_literal

  Quotes macro literal value suitable for being used on a command line so that when expanded by make, will be received by command as given to this method:

```
my $quoted = $mm->quote_literal(q{it isn't});
# returns:
#   'it isn'\''t'
print MAKEFILE "target:\n\techo $quoted\n";
# when run "make target", will output:
#   it isn't
```

- escape_newlines
- max_exec_len

  Using *POSIX*::ARG_MAX. Otherwise falling back to 4096.

- static (o)

  Defines the static target.

- xs_make_static_lib

  Defines the recipes for the `static_lib` section.

- static_lib_closures

  Records `$(EXTRALIBS)` in `extralibs.ld` and `$(PERL_SRC)/ext.libs`.

- static_lib_fixtures

  Handles copying `$(MYEXTLIB)` as starter for final static library that then gets added to.

- static_lib_pure_cmd

  Defines how to run the archive utility.

- staticmake (o)

  Calls makeaperl.

- subdir_x (o)

  Helper subroutine for subdirs

- subdirs (o)

  Defines targets to process subdirectories.

- test (o)

  Defines the test targets.

- test_via_harness (override)

  For some reason which I forget, Unix machines like to have PERL_DL_NONLAZY set for tests.

- test_via_script (override)

  Again, the PERL_DL_NONLAZY thing.

- tool_xsubpp (o)

  Determines typemaps, xsubpp version, prototype behaviour.

- all_target

  Build man pages, too

- top_targets (o)

  Defines the targets all, subdirs, config, and O_FILES

- writedoc

  Obsolete, deprecated method. Not used since Version 5.21.

- xs_c (o)
- xs_cpp (o)
- xs_o (o)

SEE ALSO

ExtUtils::MakeMaker

ExtUtils::MY

ExtUtils::MakeMaker subclass for customization

SYNOPSIS

```
# in your Makefile.PL
sub MY::whatever {
    ...
}
```

DESCRIPTION

FOR INTERNAL USE ONLY

ExtUtils::MY is a subclass of *ExtUtils::MM*. It is provided in your Makefile.PL for you to add and override MakeMaker functionality.

It also provides a convenient alias via the MY class.

ExtUtils::MY might turn out to be a temporary solution, but MY won't go away.

ExtUtils::MakeMaker

Create a module Makefile

SYNOPSIS

```
use ExtUtils::MakeMaker;

WriteMakefile(
    NAME              => "Foo::Bar",
    VERSION_FROM      => "lib/Foo/Bar.pm",
);
```

DESCRIPTION

This utility is designed to write a Makefile for an extension module from a Makefile.PL. It is based on the Makefile.SH model provided by Andy Dougherty and the perl5-porters.

It splits the task of generating the Makefile into several subroutines that can be individually overridden. Each subroutine returns the text it wishes to have written to the Makefile.

As there are various Make programs with incompatible syntax, which use operating system shells, again with incompatible syntax, it is important for users of this module to know which flavour of Make a Makefile has been written for so they'll use the correct one and won't have to face the possibly bewildering errors resulting from using the wrong one.

On POSIX systems, that program will likely be GNU Make; on Microsoft Windows, it will be either Microsoft NMake, DMake or GNU Make. See the section on the "MAKE" parameter for details.

ExtUtils::MakeMaker (EUMM) is object oriented. Each directory below the current directory that contains a Makefile.PL is treated as a separate object. This makes it possible to write an unlimited number of Makefiles with a single invocation of WriteMakefile().

All inputs to WriteMakefile are Unicode characters, not just octets. EUMM seeks to handle all of these correctly. It is currently still not possible to portably use Unicode characters in module names, because this requires Perl to handle Unicode filenames, which is not yet the case on Windows.

See ExtUtils::MakeMaker::FAQ for details of the design and usage.

How To Write A Makefile.PL

See ExtUtils::MakeMaker::Tutorial.

The long answer is the rest of the manpage :-)

Default Makefile Behaviour

The generated Makefile enables the user of the extension to invoke

```
perl Makefile.PL # optionally "perl Makefile.PL verbose"
make
make test        # optionally set TEST_VERBOSE=1
make install     # See below
```

The Makefile to be produced may be altered by adding arguments of the form KEY=VALUE. E.g.

```
perl Makefile.PL INSTALL_BASE=~
```

Other interesting targets in the generated Makefile are

```
make config     # to check if the Makefile is up-to-date
make clean      # delete local temp files (Makefile gets renamed)
make realclean  # delete derived files (including ./blib)
make ci         # check in all the files in the MANIFEST file
make dist       # see below the Distribution Support section
```

make test

MakeMaker checks for the existence of a file named `test.pl` in the current directory, and if it exists it executes the script with the proper set of perl `-I` options.

MakeMaker also checks for any files matching glob("t/*.t"). It will execute all matching files in alphabetical order via the *Test::Harness* module with the `-I` switches set correctly.

You can also organize your tests within subdirectories in the `t/` directory. To do so, use the `test` directive in your *Makefile.PL*. For example, if you had tests in:

```
t/foo
t/foo/bar
```

You could tell make to run tests in both of those directories with the following directives:

```
test => {TESTS => 't/*/*.t t/*/*/*.t'}
test => {TESTS => 't/foo/*.t t/foo/bar/*.t'}
```

The first will run all test files in all first-level subdirectories and all subdirectories they contain. The second will run tests in only the `t/foo` and `t/foo/bar`.

If you'd like to see the raw output of your tests, set the `TEST_VERBOSE` variable to true.

```
make test TEST_VERBOSE=1
```

If you want to run particular test files, set the `TEST_FILES` variable. It is possible to use globbing with this mechanism.

```
make test TEST_FILES='t/foobar.t t/dagobah*.t'
```

Windows users who are using `nmake` should note that due to a bug in `nmake`, when specifying `TEST_FILES` you must use back-slashes instead of forward-slashes.

```
nmake test TEST_FILES='t\foobar.t t\dagobah*.t'
```

make testdb

A useful variation of the above is the target `testdb`. It runs the test under the Perl debugger (see perldebug). If the file `test.pl` exists in the current directory, it is used for the test.

If you want to debug some other testfile, set the `TEST_FILE` variable thusly:

```
make testdb TEST_FILE=t/mytest.t
```

By default the debugger is called using `-d` option to perl. If you want to specify some other option, set the `TESTDB_SW` variable:

```
make testdb TESTDB_SW=-Dx
```

make install

make alone puts all relevant files into directories that are named by the macros INST_LIB, INST_ARCHLIB, INST_SCRIPT, INST_MAN1DIR and INST_MAN3DIR. All these default to something below ./blib if you are *not* building below the perl source directory. If you *are* building below the perl source, INST_LIB and INST_ARCHLIB default to ../../lib, and INST_SCRIPT is not defined.

The *install* target of the generated Makefile copies the files found below each of the INST_* directories to their INSTALL* counterparts. Which counterparts are chosen depends on the setting of INSTALLDIRS according to the following table:

```
                             INSTALLDIRS set to
                        perl         site          vendor

               PERLPREFIX      SITEPREFIX          VENDORPREFIX
INST_ARCHLIB   INSTALLARCHLIB  INSTALLSITEARCH     INSTALLVENDORARCH
INST_LIB       INSTALLPRIVLIB  INSTALLSITELIB      INSTALLVENDORLIB
INST_BIN       INSTALLBIN      INSTALLSITEBIN      INSTALLVENDORBIN
INST_SCRIPT    INSTALLSCRIPT   INSTALLSITESCRIPT   ␣
 ↪INSTALLVENDORSCRIPT
INST_MAN1DIR   INSTALLMAN1DIR  INSTALLSITEMAN1DIR ␣
 ↪INSTALLVENDORMAN1DIR
INST_MAN3DIR   INSTALLMAN3DIR  INSTALLSITEMAN3DIR ␣
 ↪INSTALLVENDORMAN3DIR
```

The INSTALL... macros in turn default to their %Config ($Config{installprivlib}, $Config{installarchlib}, etc.) counterparts.

You can check the values of these variables on your system with

```
perl '-V:install.*'
```

And to check the sequence in which the library directories are searched by perl, run

```
perl -le 'print join $/, @INC'
```

Sometimes older versions of the module you're installing live in other directories in @INC. Because Perl loads the first version of a module it finds, not the newest, you might

accidentally get one of these older versions even after installing a brand new version. To delete *all other versions of the module you're installing* (not simply older ones) set the `UNINST` variable.

```
make install UNINST=1
```

INSTALL_BASE

INSTALL_BASE can be passed into Makefile.PL to change where your module will be installed. INSTALL_BASE is more like what everyone else calls "prefix" than PREFIX is.

To have everything installed in your home directory, do the following.

```
# Unix users, INSTALL_BASE=~ works fine
perl Makefile.PL INSTALL_BASE=/path/to/your/home/dir
```

Like PREFIX, it sets several INSTALL* attributes at once. Unlike PREFIX it is easy to predict where the module will end up. The installation pattern looks like this:

```
INSTALLARCHLIB     INSTALL_BASE/lib/perl5/$Config{archname}
INSTALLPRIVLIB     INSTALL_BASE/lib/perl5
INSTALLBIN         INSTALL_BASE/bin
INSTALLSCRIPT      INSTALL_BASE/bin
INSTALLMAN1DIR     INSTALL_BASE/man/man1
INSTALLMAN3DIR     INSTALL_BASE/man/man3
```

INSTALL_BASE in MakeMaker and `--install_base` in Module::Build (as of 0.28) install to the same location. If you want MakeMaker and Module::Build to install to the same location simply set INSTALL_BASE and `--install_base` to the same location.

INSTALL_BASE was added in 6.31.

PREFIX and LIB attribute

PREFIX and LIB can be used to set several INSTALL* attributes in one go. Here's an example for installing into your home directory.

```
# Unix users, PREFIX=~ works fine
perl Makefile.PL PREFIX=/path/to/your/home/dir
```

This will install all files in the module under your home directory, with man pages and libraries going into an appropriate place (usually ~/man and ~/lib). How the exact location is determined is complicated and depends on how your Perl was configured. INSTALL_BASE works more like what other build systems call "prefix" than PREFIX and we recommend you use that instead.

Another way to specify many INSTALL directories with a single parameter is LIB.

```
perl Makefile.PL LIB=~/lib
```

This will install the module's architecture-independent files into ~/lib, the architecture-dependent files into ~/lib/$archname.

Note, that in both cases the tilde expansion is done by MakeMaker, not by perl by default, nor by make.

Conflicts between parameters LIB, PREFIX and the various INSTALL* arguments are resolved so that:

- setting LIB overrides any setting of INSTALLPRIVLIB, INSTALLARCHLIB, INSTALLSITELIB, INSTALLSITEARCH (and they are not affected by PREFIX);
- without LIB, setting PREFIX replaces the initial `$Config{prefix}` part of those INSTALL* arguments, even if the latter are explicitly set (but are set to still start with `$Config{prefix}`).

If the user has superuser privileges, and is not working on AFS or relatives, then the defaults for INSTALLPRIVLIB, INSTALLARCHLIB, INSTALLSCRIPT, etc. will be appropriate, and this incantation will be the best:

```
perl Makefile.PL;
make;
make test
make install
```

make install by default writes some documentation of what has been done into the file `$(INSTALLARCHLIB)/perllocal.pod`. This feature can be bypassed by calling make pure_install.

AFS users

will have to specify the installation directories as these most probably have changed since perl itself has been installed. They will have to do this by calling

```
perl Makefile.PL INSTALLSITELIB=/afs/here/today \
    INSTALLSCRIPT=/afs/there/now INSTALLMAN3DIR=/afs/for/manpages
make
```

Be careful to repeat this procedure every time you recompile an extension, unless you are sure the AFS installation directories are still valid.

Static Linking of a new Perl Binary

An extension that is built with the above steps is ready to use on systems supporting dynamic loading. On systems that do not support dynamic loading, any newly created extension has to be linked together with the available resources. MakeMaker supports the linking process by creating appropriate targets in the Makefile whenever an extension is built. You can invoke the corresponding section of the makefile with

```
make perl
```

That produces a new perl binary in the current directory with all extensions linked in that can be found in INST_ARCHLIB, SITELIBEXP, and PERL_ARCHLIB. To do that, MakeMaker writes a new Makefile, on UNIX, this is called `Makefile.aperl` (may be system dependent). If you want to force the creation of a new perl, it is recommended that you delete this `Makefile.aperl`, so the directories are searched through for linkable libraries again.

The binary can be installed into the directory where perl normally resides on your machine with

```
make inst_perl
```

To produce a perl binary with a different name than `perl`, either say

```
perl Makefile.PL MAP_TARGET=myperl
make myperl
make inst_perl
```

or say

```
perl Makefile.PL
make myperl MAP_TARGET=myperl
make inst_perl MAP_TARGET=myperl
```

In any case you will be prompted with the correct invocation of the `inst_perl` target that installs the new binary into INSTALLBIN.

make inst_perl by default writes some documentation of what has been done into the file `$(INSTALLARCHLIB)/perllocal.pod`. This can be bypassed by calling make pure_inst_perl.

Warning: the inst_perl: target will most probably overwrite your existing perl binary. Use with care!

Sometimes you might want to build a statically linked perl although your system supports dynamic loading. In this case you may explicitly set the linktype with the invocation of the Makefile.PL or make:

```
perl Makefile.PL LINKTYPE=static    # recommended
```

or

```
make LINKTYPE=static                # works on most systems
```

Determination of Perl Library and Installation Locations

MakeMaker needs to know, or to guess, where certain things are located. Especially INST_LIB and INST_ARCHLIB (where to put the files during the make(1) run), PERL_LIB and PERL_ARCHLIB (where to read existing modules from), and PERL_INC (header files and `libperl*.*`).

Extensions may be built either using the contents of the perl source directory tree or from the installed perl library. The recommended way is to build extensions after you have run 'make install' on perl itself. You can do that in any directory on your hard disk that is not below the perl source tree. The support for extensions below the ext directory of the perl distribution is only good for the standard extensions that come with perl.

If an extension is being built below the `ext/` directory of the perl source then MakeMaker will set PERL_SRC automatically (e.g., `../..`). If PERL_SRC is defined and the extension is recognized as a standard extension, then other variables default to the following:

```
PERL_INC     = PERL_SRC
PERL_LIB     = PERL_SRC/lib
PERL_ARCHLIB = PERL_SRC/lib
INST_LIB     = PERL_LIB
INST_ARCHLIB = PERL_ARCHLIB
```

If an extension is being built away from the perl source then MakeMaker will leave PERL_SRC undefined and default to using the installed copy of the perl library. The other variables default to the following:

```
PERL_INC     = $archlibexp/CORE
PERL_LIB     = $privlibexp
PERL_ARCHLIB = $archlibexp
INST_LIB     = ./blib/lib
INST_ARCHLIB = ./blib/arch
```

If perl has not yet been installed then PERL_SRC can be defined on the command line as shown in the previous section.

Which architecture dependent directory?

If you don't want to keep the defaults for the INSTALL* macros, MakeMaker helps you to minimize the typing needed: the usual relationship between INSTALLPRIVLIB and INSTALLARCHLIB is determined by Configure at perl compilation time. MakeMaker supports the user who sets INSTALLPRIVLIB. If INSTALLPRIVLIB is set, but INSTALLARCHLIB not, then MakeMaker defaults the latter to be the same subdirectory of INSTALLPRIVLIB as Configure decided for the counterparts in %Config, otherwise it defaults to INSTALLPRIVLIB. The same relationship holds for INSTALLSITELIB and INSTALLSITEARCH.

MakeMaker gives you much more freedom than needed to configure internal variables and get different results. It is worth mentioning that make(1) also lets you configure most of the variables that are used in the Makefile. But in the majority of situations this will not be necessary, and should only be done if the author of a package recommends it (or you know what you're doing).

Using Attributes and Parameters

The following attributes may be specified as arguments to WriteMakefile() or as NAME=VALUE pairs on the command line. Attributes that became available with later versions of MakeMaker are indicated.

A computer-readable list of recognized attributes is available as `%ExtUtils::MakeMakers::Recognized_Att_Keys`, supported since 7.72. You can check whether a particular parameter is supported by the current version of ExtUtils::MakeMaker by checking whether it exists in the hash.

In order to maintain portability of attributes with older versions of MakeMaker you may want to use App::EUMM::Upgrade with your `Makefile.PL`.

- ABSTRACT

  One line description of the module. Will be included in PPD file.

- ABSTRACT_FROM

  Name of the file that contains the package description. MakeMaker looks for a line in the POD matching /^($package\s-\s)(.*)/. This is typically the first line in the "=head1 NAME" section. $2 becomes the abstract.

- AUTHOR

  Array of strings containing name (and email address) of package author(s). Is used in CPAN Meta files (META.yml or META.json) and PPD (Perl Package Description) files for PPM (Perl Package Manager).

- BINARY_LOCATION

  Used when creating PPD files for binary packages. It can be set to a full or relative path or URL to the binary archive for a particular architecture. For example:

  ```
  perl Makefile.PL BINARY_LOCATION=x86/Agent.tar.gz
  ```

  builds a PPD package that references a binary of the `Agent` package, located in the `x86` directory relative to the PPD itself.

- BUILD_REQUIRES

  Available in version 6.55_03 and above.

  A hash of modules that are needed to build your module but not run it.

  This will go into the `build_requires` field of your `META.yml` and the `build` of the `prereqs` field of your `META.json`.

  Defaults to `{ "ExtUtils::MakeMaker" => 0 }` if this attribute is not specified.

  The format is the same as PREREQ_PM.

- C

  Ref to array of *.c file names. This is not currently used by MakeMaker but may be handy in Makefile.PLs.

- CCFLAGS

  String that will be included in the compiler call command line between the arguments INC and OPTIMIZE. Note that setting this will overwrite its default value (`$Config::Config{ccflags}`); to preserve that, include the default value directly, e.g.:

  ```
  CCFLAGS => "$Config::Config{ccflags} ..."
  ```

- CONFIG

  Arrayref. E.g. [qw(archname manext)] defines ARCHNAME & MANEXT from config.sh. MakeMaker will add to CONFIG the following values anyway: ar cc cccdlflags ccdlflags cpprun dlext dlsrc ld lddlflags ldflags libc lib_ext obj_ext ranlib sitelibexp sitearchexp so

- CONFIGURE

  CODE reference. The subroutine should return a hash reference. The hash may contain further attributes, e.g. {LIBS => ...}, that have to be determined by some evaluation method.

- CONFIGURE_REQUIRES

  Available in version 6.52 and above.

  A hash of modules that are required to run Makefile.PL itself, but not to run your distribution.

  This will go into the `configure_requires` field of your `META.yml` and the `configure` of the `prereqs` field of your `META.json`.

  Defaults to `{ "ExtUtils::MakeMaker" => 0 }` if this attribute is not specified.

  The format is the same as PREREQ_PM.

- DEFINE

  Something like `"-DHAVE_UNISTD_H"`

- DESTDIR

  This is the root directory into which the code will be installed. It *prepends itself to the normal prefix*. For example, if your code would normally go into `/usr/local/lib/perl` you could set DESTDIR=~/tmp/ and installation would go into `~/tmp/usr/local/lib/perl`.

  This is primarily of use for people who repackage Perl modules.

  NOTE: Due to the nature of make, it is important that you put the trailing slash on your DESTDIR. `~/tmp/` not `~/tmp`.

- DIR

  Ref to array of subdirectories containing Makefile.PLs e.g. ['sdbm'] in ext/SDBM_File

- DISTNAME

  A safe filename for the package.

  Defaults to NAME below but with :: replaced with -.

  For example, Foo::Bar becomes Foo-Bar.

- DISTVNAME

  Your name for distributing the package with the version number included. This is used by 'make dist' to name the resulting archive file.

  Defaults to DISTNAME-VERSION.

  For example, version 1.04 of Foo::Bar becomes Foo-Bar-1.04.

  On some OS's where . has special meaning VERSION_SYM may be used in place of VERSION.

- DLEXT

  Specifies the extension of the module's loadable object. For example:

  ```
  DLEXT => 'unusual_ext', # Default value is $Config{so}
  ```

  NOTE: When using this option to alter the extension of a module's loadable object, it is also necessary that the module's pm file specifies the same change:

```
local $DynaLoader::dl_dlext = 'unusual_ext';
```

- DL_FUNCS

  Hashref of symbol names for routines to be made available as universal symbols. Each key/value pair consists of the package name and an array of routine names in that package. Used only under AIX, OS/2, VMS and Win32 at present. Defaults to

```
{"$(NAME)" => ["boot_$(NAME)" ] }
```

  e.g.

```
{"RPC" => [qw( boot_rpcb rpcb_gettime getnetconfigent )],
 "NetconfigPtr" => [ 'DESTROY'] }
```

  Please see the ExtUtils::Mksymlists documentation for more information about the DL_FUNCS, DL_VARS and FUNCLIST attributes.

- DL_VARS

  Array of symbol names for variables to be made available as universal symbols. Used only under AIX, OS/2, VMS and Win32 at present. Defaults to []. (e.g. [qw(Foo_version Foo_numstreams Foo_tree)])

- EXCLUDE_EXT

  Array of extension names to exclude when doing a static build. This is ignored if INCLUDE_EXT is present. Consult INCLUDE_EXT for more details. (e.g. [qw(Socket POSIX)])

  This attribute may be most useful when specified as a string on the command line: perl Makefile.PL EXCLUDE_EXT='Socket Safe'

- EXE_FILES

  Ref to array of executable files. The files will be copied to the INST_SCRIPT directory. Make realclean will delete them from there again.

  If your executables start with something like #!perl or #!/usr/bin/perl MakeMaker will change this to the path of the perl 'Makefile.PL' was invoked with so the programs will be sure to run properly even if perl is not in /usr/bin/perl.

- FIRST_MAKEFILE

  The name of the Makefile to be produced. This is used for the second Makefile that will be produced for the MAP_TARGET.

  Defaults to 'Makefile' or 'Descrip.MMS' on VMS.

  (Note: we couldn't use MAKEFILE because dmake uses this for something else).

- FULLPERL

- FULLPERLRUN

  Like PERLRUN, except it uses FULLPERL.

- FULLPERLRUNINST

  Like PERLRUNINST, except it uses FULLPERL.

- FUNCLIST

  This provides an alternate means to specify function names to be exported from the extension. Its value is a reference to an array of function names to be exported by the extension. These names are passed through unaltered to the linker options file.

- H

  Ref to array of *.h file names. Similar to C.

- IMPORTS

  This attribute is used to specify names to be imported into the extension. Takes a hash ref.

  It is only used on OS/2 and Win32.

- INC

  Include file dirs eg: `"-I/usr/5include -I/path/to/inc"`

- INCLUDE_EXT

  Array of extension names to be included when doing a static build. MakeMaker will normally build with all of the installed extensions when doing a static build, and that is usually the desired behavior. If INCLUDE_EXT is present then MakeMaker will build only with those extensions which are explicitly mentioned. (e.g. [qw(Socket POSIX)])

  It is not necessary to mention DynaLoader or the current extension when filling in INCLUDE_EXT. If the INCLUDE_EXT is mentioned but is empty then only DynaLoader and the current extension will be included in the build.

  This attribute may be most useful when specified as a string on the command line: perl Makefile.PL INCLUDE_EXT='POSIX Socket Devel::Peek'

- INSTALLARCHLIB

  Used by 'make install', which copies files from INST_ARCHLIB to this directory if INSTALLDIRS is set to perl.

- INSTALLBIN

  Directory to install binary files (e.g. tkperl) into if INSTALLDIRS=perl.

- INSTALLDIRS

  Determines which of the sets of installation directories to choose: perl, site or vendor. Defaults to site.

- INSTALLMAN1DIR

- INSTALLMAN3DIR

  These directories get the man pages at 'make install' time if INSTALLDIRS=perl. Defaults to $Config{installman*dir}.

  If set to 'none', no man pages will be installed.

- INSTALLPRIVLIB

Used by 'make install', which copies files from INST_LIB to this directory if INSTALLDIRS is set to perl.

Defaults to $Config{installprivlib}.

- INSTALLSCRIPT

  Available in version 6.30_02 and above.

  Used by 'make install' which copies files from INST_SCRIPT to this directory if INSTALLDIRS=perl.

- INSTALLSITEARCH

  Used by 'make install', which copies files from INST_ARCHLIB to this directory if INSTALLDIRS is set to site (default).

- INSTALLSITEBIN

  Used by 'make install', which copies files from INST_BIN to this directory if INSTALLDIRS is set to site (default).

- INSTALLSITELIB

  Used by 'make install', which copies files from INST_LIB to this directory if INSTALLDIRS is set to site (default).

- INSTALLSITEMAN1DIR
- INSTALLSITEMAN3DIR

  These directories get the man pages at 'make install' time if INSTALLDIRS=site (default). Defaults to $(SITEPREFIX)/man/man$(MAN*EXT).

  If set to 'none', no man pages will be installed.

- INSTALLSITESCRIPT

  Used by 'make install' which copies files from INST_SCRIPT to this directory if INSTALLDIRS is set to site (default).

- INSTALLVENDORARCH

  Used by 'make install', which copies files from INST_ARCHLIB to this directory if INSTALLDIRS is set to vendor. Note that if you do not set this, the value of INSTALLVENDORLIB will be used, which is probably not what you want.

- INSTALLVENDORBIN

  Used by 'make install', which copies files from INST_BIN to this directory if INSTALLDIRS is set to vendor.

- INSTALLVENDORLIB

  Used by 'make install', which copies files from INST_LIB to this directory if INSTALLDIRS is set to vendor.

- INSTALLVENDORMAN1DIR
- INSTALLVENDORMAN3DIR

  These directories get the man pages at 'make install' time if INSTALLDIRS=vendor. Defaults to $(VENDORPREFIX)/man/man$(MAN*EXT).

If set to 'none', no man pages will be installed.

- INSTALLVENDORSCRIPT

  Available in version 6.30_02 and above.

  Used by 'make install' which copies files from INST_SCRIPT to this directory if INSTALLDIRS is set to vendor.

- INST_ARCHLIB

  Same as INST_LIB for architecture dependent files.

- INST_BIN

  Directory to put real binary files during 'make'. These will be copied to INSTALLBIN during 'make install'

- INST_LIB

  Directory where we put library files of this extension while building it.

- INST_MAN1DIR

  Directory to hold the man pages at 'make' time

- INST_MAN3DIR

  Directory to hold the man pages at 'make' time

- INST_SCRIPT

  Directory where executable files should be installed during 'make'. Defaults to "./blib/script", just to have a dummy location during testing. make install will copy the files in INST_SCRIPT to INSTALLSCRIPT.

- LD

  Program to be used to link libraries for dynamic loading.

  Defaults to $Config{ld}.

- LDDLFLAGS

  Any special flags that might need to be passed to ld to create a shared library suitable for dynamic loading. It is up to the makefile to use it. (See *"lddlflags" in Config*)

  Defaults to $Config{lddlflags}.

- LDFROM

  Defaults to "$(OBJECT)" and is used in the ld command to specify what files to link/load from (also see dynamic_lib below for how to specify ld flags)

- LIB

  LIB should only be set at `perl Makefile.PL` time but is allowed as a MakeMaker argument. It has the effect of setting both INSTALLPRIVLIB and INSTALLSITELIB to that value regardless any explicit setting of those arguments (or of PREFIX). INSTALLARCHLIB and INSTALLSITEARCH are set to the corresponding architecture subdirectory.

- LIBPERL_A

The filename of the perllibrary that will be used together with this extension. Defaults to libperl.a.

- LIBS

  An anonymous array of alternative library specifications to be searched for (in order) until at least one library is found. E.g.

```
'LIBS' => ["-lgdbm", "-ldbm -lfoo", "-L/path -ldbm.nfs"]
```

  Mind, that any element of the array contains a complete set of arguments for the ld command. So do not specify

```
'LIBS' => ["-ltcl", "-ltk", "-lX11"]
```

  See ODBM_File/Makefile.PL for an example, where an array is needed. If you specify a scalar as in

```
'LIBS' => "-ltcl -ltk -lX11"
```

  MakeMaker will turn it into an array with one element.

- LICENSE

  Available in version 6.31 and above.

  The licensing terms of your distribution. Generally it's "perl_5" for the same license as Perl itself.

  See *CPAN::Meta::Spec* for the list of options.

  Defaults to "unknown".

- LINKTYPE

  'static' or 'dynamic' (default unless usedl=undef in config.sh). Should only be used to force static linking (also see linkext below).

- MAGICXS

  Available in version 6.8305 and above.

  When this is set to 1, `OBJECT` will be automagically derived from `O_FILES`.

- MAKE

  Available in version 6.30_01 and above.

  Variant of make you intend to run the generated Makefile with. This parameter lets Makefile.PL know what make quirks to account for when generating the Makefile.

  MakeMaker also honors the MAKE environment variable. This parameter takes precedence.

  Currently the only significant values are 'dmake' and 'nmake' for Windows users, instructing MakeMaker to generate a Makefile in the flavour of DMake ("Dennis Vadura's Make") or Microsoft NMake respectively.

  Defaults to $Config{make}, which may go looking for a Make program in your environment.

How are you supposed to know what flavour of Make a Makefile has been generated for if you didn't specify a value explicitly? Search the generated Makefile for the definition of the MAKE variable, which is used to recursively invoke the Make utility. That will tell you what Make you're supposed to invoke the Makefile with.

- MAKEAPERL

  Boolean which tells MakeMaker that it should include the rules to make a perl. This is handled automatically as a switch by MakeMaker. The user normally does not need it.

- MAKEFILE_OLD

  When 'make clean' or similar is run, the $(FIRST_MAKEFILE) will be backed up at this location.

  Defaults to $(FIRST_MAKEFILE).old or $(FIRST_MAKEFILE)_old on VMS.

- MAN1PODS

  Hashref of pod-containing files. MakeMaker will default this to all EXE_FILES files that include POD directives. The files listed here will be converted to man pages and installed as was requested at Configure time.

  This hash should map POD files (or scripts containing POD) to the man file names under the `blib/man1/` directory, as in the following example:

```
MAN1PODS            => {
  'doc/command.pod'    => 'blib/man1/command.1',
  'scripts/script.pl'  => 'blib/man1/script.1',
}
```

- MAN3PODS

  Hashref that assigns to *.pm and *.pod files the files into which the manpages are to be written. MakeMaker parses all *.pod and *.pm files for POD directives. Files that contain POD will be the default keys of the MAN3PODS hashref. These will then be converted to man pages during `make` and will be installed during `make install`.

  Example similar to MAN1PODS.

- MAP_TARGET

  If it is intended that a new perl binary be produced, this variable may hold a name for that binary. Defaults to perl

- META_ADD

- META_MERGE

  Available in version 6.46 and above.

  A hashref of items to add to the CPAN Meta file (`META.yml` or `META.json`).

  They differ in how they behave if they have the same key as the default metadata. META_ADD will override the default value with its own. META_MERGE will merge its value with the default.

  Unless you want to override the defaults, prefer META_MERGE so as to get the advantage of any future defaults.

Where prereqs are concerned, if META_MERGE is used, prerequisites are merged with their counterpart `WriteMakefile()` argument (PREREQ_PM is merged into {prereqs}{runtime}{requires}, BUILD_REQUIRES into `{prereqs}{build}{requires}`, CONFIGURE_REQUIRES into `{prereqs}{configure}{requires}`, and TEST_REQUIRES into `{prereqs}{test}{requires}`). When prereqs are specified with META_ADD, the only prerequisites added to the file come from the metadata, not `WriteMakefile()` arguments.

Note that these configuration options are only used for generating `META.yml` and `META.json` – they are NOT used for `MYMETA.yml` and `MYMETA.json`. Therefore data in these fields should NOT be used for dynamic (user-side) configuration.

By default CPAN Meta specification `1.4` is used. In order to use CPAN Meta specification `2.0`, indicate with `meta-spec` the version you want to use.

```
META_MERGE         => {

  "meta-spec" => { version => 2 },

  resources => {

    repository => {
        type => 'git',
        url => 'git://github.com/Perl-Toolchain-Gang/ExtUtils-
↪MakeMaker.git',
        web => 'https://github.com/Perl-Toolchain-Gang/ExtUtils-
↪MakeMaker',
    },

  },

},
```

- MIN_PERL_VERSION

  Available in version 6.48 and above.

  The minimum required version of Perl for this distribution.

  Either the 5.006001 or the 5.6.1 format is acceptable.

- MYEXTLIB

  If the extension links to a library that it builds, set this to the name of the library (see SDBM_File)

- NAME

  The package representing the distribution. For example, `Test::More` or `ExtUtils::MakeMaker`.

  `NAME` *must* be a valid Perl package name and it *must* have an associated `.pm` file. For example, `Foo::Bar` is a valid `NAME` and there must exist `Foo/Bar.pm`.

  Your distribution **must** have a `NAME`.

- NEEDS_LINKING

MakeMaker will figure out if an extension contains linkable code anywhere down the directory tree, and will set this variable accordingly, but you can speed it up a very little bit if you define this boolean variable yourself.

- NOECHO

  Command so make does not print the literal commands it's running.

  By setting it to an empty string you can generate a Makefile that prints all commands. Mainly used in debugging MakeMaker itself.

  Defaults to @.

- NORECURS

  Boolean. Attribute to inhibit descending into subdirectories.

- NO_META

  When true, suppresses the generation and addition to the MANIFEST of the META.yml and META.json module meta-data files during 'make distdir'.

  Defaults to false.

- NO_MYMETA

  Available in version 6.57_02 and above.

  When true, suppresses the generation of MYMETA.yml and MYMETA.json module meta-data files during 'perl Makefile.PL'.

  Defaults to false.

- NO_PACKLIST

  Available in version 6.7501 and above.

  When true, suppresses the writing of `packlist` files for installs.

  Defaults to false.

- NO_PERLLOCAL

  Available in version 6.7501 and above.

  When true, suppresses the appending of installations to `perllocal`.

  Defaults to false.

- NO_VC

  In general, any generated Makefile checks for the current version of MakeMaker and the version the Makefile was built under. If NO_VC is set, the version check is neglected. Do not write this into your Makefile.PL, use it interactively instead.

- OBJECT

  List of object files, defaults to '$(BASEEXT)$(OBJ_EXT)', but can be a long string or an array containing all object files, e.g. "tkpBind.o tkpButton.o tkpCanvas.o" or ["tkpBind.o", "tkpButton.o", "tkpCanvas.o"]

  (Where BASEEXT is the last component of NAME, and OBJ_EXT is $Config{obj_ext}.)

- OPTIMIZE

  Defaults to -O. Set it to -g to turn debugging on. The flag is passed to subdirectory makes.

- PERL

  Perl binary for tasks that can be done by miniperl. If it contains spaces or other shell metacharacters, it needs to be quoted in a way that protects them, since this value is intended to be inserted in a shell command line in the Makefile. E.g.:

```
# Perl executable lives in "C:/Program Files/Perl/bin"
# Normally you don't need to set this yourself!
$ perl Makefile.PL PERL='"C:/Program Files/Perl/bin/perl.exe" -w
 ↪'
```

- PERL_CORE

  Set only when MakeMaker is building the extensions of the Perl core distribution.

- PERLMAINCC

  The call to the program that is able to compile perlmain.c. Defaults to $(CC).

- PERL_ARCHLIB

  Same as for PERL_LIB, but for architecture dependent files.

  Used only when MakeMaker is building the extensions of the Perl core distribution (because normally $(PERL_ARCHLIB) is automatically in @INC, and adding it would get in the way of PERL5LIB).

- PERL_LIB

  Directory containing the Perl library to use.

  Used only when MakeMaker is building the extensions of the Perl core distribution (because normally $(PERL_LIB) is automatically in @INC, and adding it would get in the way of PERL5LIB).

- PERL_MALLOC_OK

  defaults to 0. Should be set to TRUE if the extension can work with the memory allocation routines substituted by the Perl malloc() subsystem. This should be applicable to most extensions with exceptions of those

  – with bugs in memory allocations which are caught by Perl's malloc();

  – which interact with the memory allocator in other ways than via malloc(), realloc(), free(), calloc(), sbrk() and brk();

  – which rely on special alignment which is not provided by Perl's malloc().

  NOTE. Neglecting to set this flag in *any one* of the loaded extension nullifies many advantages of Perl's malloc(), such as better usage of system resources, error detection, memory usage reporting, catchable failure of memory allocations, etc.

- PERLPREFIX

  Directory under which core modules are to be installed.

Defaults to $Config{installprefixexp}, falling back to $Config{installprefix}, $Config{prefixexp} or $Config{prefix} should $Config{installprefixexp} not exist.

Overridden by PREFIX.

- PERLRUN

  Use this instead of $(PERL) when you wish to run perl. It will set up extra necessary flags for you.

- PERLRUNINST

  Use this instead of $(PERL) when you wish to run perl to work with modules. It will add things like -I$(INST_ARCH) and other necessary flags so perl can see the modules you're about to install.

- PERL_SRC

  Directory containing the Perl source code (use of this should be avoided, it may be undefined)

- PERM_DIR

  Available in version 6.51_01 and above.

  Desired permission for directories. Defaults to 755.

- PERM_RW

  Desired permission for read/writable files. Defaults to 644.

- PERM_RWX

  Desired permission for executable files. Defaults to 755.

- PL_FILES

  MakeMaker can run programs to generate files for you at build time. By default any file named *.PL (except Makefile.PL and Build.PL) in the top level directory will be assumed to be a Perl program and run passing its own basename in as an argument. This basename is actually a build target, and there is an intention, but not a requirement, that the *.PL file make the file passed to to as an argument. For example...

```
perl foo.PL foo
```

  This behavior can be overridden by supplying your own set of files to search. PL_FILES accepts a hash ref, the key being the file to run and the value is passed in as the first argument when the PL file is run.

```
PL_FILES => {'bin/foobar.PL' => 'bin/foobar'}

PL_FILES => {'foo.PL' => 'foo.c'}
```

  Would run bin/foobar.PL like this:

```
perl bin/foobar.PL bin/foobar
```

  If multiple files from one program are desired an array ref can be used.

```
PL_FILES => {'bin/foobar.PL' => [qw(bin/foobar1 bin/foobar2)]}
```

In this case the program will be run multiple times using each target file.

```
perl bin/foobar.PL bin/foobar1
perl bin/foobar.PL bin/foobar2
```

If an output file depends on extra input files beside the script itself, a hash ref can be used in version 7.36 and above:

```
PL_FILES => { 'foo.PL' => {
    'foo.out' => 'foo.in',
    'bar.out' => [qw(bar1.in bar2.in)],
}
```

In this case the extra input files will be passed to the program after the target file:

```
perl foo.PL foo.out foo.in
perl foo.PL bar.out bar1.in bar2.in
```

PL files are normally run **after** pm_to_blib and include INST_LIB and INST_ARCH in their `@INC`, so the just built modules can be accessed... unless the PL file is making a module (or anything else in PM) in which case it is run **before** pm_to_blib and does not include INST_LIB and INST_ARCH in its `@INC`. This apparently odd behavior is there for backwards compatibility (and it's somewhat DWIM). The argument passed to the .PL is set up as a target to build in the Makefile. In other sections such as `postamble` you can specify a dependency on the filename/argument that the .PL is supposed (or will have, now that that is is a dependency) to generate. Note the file to be generated will still be generated and the .PL will still run even without an explicit dependency created by you, since the `all` target still depends on running all eligible to run.PL files.

- PM

  Hashref of .pm files and *.pl files to be installed. e.g.

  ```
  {'name_of_file.pm' => '$(INST_LIB)/install_as.pm'}
  ```

  By default this will include *.pm and *.pl and the files found in the PMLIBDIRS directories. Defining PM in the Makefile.PL will override PMLIBDIRS.

- PMLIBDIRS

  Ref to array of subdirectories containing library files. Defaults to ['lib', $(BASEEXT)]. The directories will be scanned and *any* files they contain will be installed in the corresponding location in the library. A libscan() method can be used to alter the behaviour. Defining PM in the Makefile.PL will override PMLIBDIRS.

  (Where BASEEXT is the last component of NAME.)

- PM_FILTER

  A filter program, in the traditional Unix sense (input from stdin, output to stdout) that is passed on each .pm file during the build (in the pm_to_blib() phase). It is empty by default, meaning no filtering is done. You could use:

```
PM_FILTER => 'perl -ne "print unless /^#/"',
```

to remove all the leading comments on the fly during the build. In order to be as portable as possible, please consider using a Perl one-liner rather than Unix (or other) utilities, as above. MakeMaker will escape the # for the Makefile, since what goes in the Makefile will depend on which `make` implementation is being targeted.

You will almost certainly be better off using the `PL_FILES` system, instead. See above, or the ExtUtils::MakeMaker::FAQ entry.

- POLLUTE

  Prior to 5.6 various interpreter variables were available without a `PL_` prefix, eg. `PL_undef` was available as `undef`. As of release 5.6, these are only defined if the POLLUTE flag is enabled:

  ```
  perl Makefile.PL POLLUTE=1
  ```

  Please inform the module author if this is necessary to successfully install a module under 5.6 or later.

- PPM_INSTALL_EXEC

  Name of the executable used to run `PPM_INSTALL_SCRIPT` below. (e.g. perl)

- PPM_INSTALL_SCRIPT

  Name of the script that gets executed by the Perl Package Manager after the installation of a package.

- PPM_UNINSTALL_EXEC

  Available in version 6.8502 and above.

  Name of the executable used to run `PPM_UNINSTALL_SCRIPT` below. (e.g. perl)

- PPM_UNINSTALL_SCRIPT

  Available in version 6.8502 and above.

  Name of the script that gets executed by the Perl Package Manager before the removal of a package.

- PREFIX

  This overrides all the default install locations. Man pages, libraries, scripts, etc... MakeMaker will try to make an educated guess about where to place things under the new PREFIX based on your Config defaults. Failing that, it will fall back to a structure which should be sensible for your platform.

  If you specify LIB or any INSTALL* variables they will not be affected by the PREFIX.

- PREREQ_FATAL

  Bool. If this parameter is true, failing to have the required modules (or the right versions thereof) will be fatal. `perl Makefile.PL` will `die` instead of simply informing the user of the missing dependencies.

  It is *extremely* rare to have to use `PREREQ_FATAL`. Its use by module authors is *strongly discouraged* and should never be used lightly.

For dependencies that are required in order to run `Makefile.PL`, see `CONFIGURE_REQUIRES`.

Module installation tools have ways of resolving unmet dependencies but to do that they need a `Makefile`. Using `PREREQ_FATAL` breaks this. That's bad.

Assuming you have good test coverage, your tests should fail with missing dependencies informing the user more strongly that something is wrong. You can write a `t/00compile.t` test which will simply check that your code compiles and stop "make test" prematurely if it doesn't. See *"BAIL_OUT" in Test::More* for more details.

- PREREQ_PM

  A hash of modules that are needed to run your module. The keys are the module names ie. Test::More, and the minimum version is the value. If the required version number is 0 any version will do. The versions given may be a Perl v-string (see *version*) or a range (see *CPAN::Meta::Requirements*).

  This will go into the `requires` field of your `META.yml` and the `runtime` of the `prereqs` field of your `META.json`.

```
PREREQ_PM => {
    # Require Test::More at least 0.47
    "Test::More" => "0.47",

    # Require any version of Acme::Buffy
    "Acme::Buffy" => 0,
}
```

- PREREQ_PRINT

  Bool. If this parameter is true, the prerequisites will be printed to stdout and MakeMaker will exit. The output format is an evalable hash ref.

```
$PREREQ_PM = {
               'A::B' => Vers1,
               'C::D' => Vers2,
               ...
             };
```

  If a distribution defines a minimal required perl version, this is added to the output as an additional line of the form:

```
$MIN_PERL_VERSION = '5.008001';
```

  If BUILD_REQUIRES is not empty, it will be dumped as $BUILD_REQUIRES hashref.

- PRINT_PREREQ

  RedHatism for `PREREQ_PRINT`. The output format is different, though:

```
perl(A::B)>=Vers1 perl(C::D)>=Vers2 ...
```

  A minimal required perl version, if present, will look like this:

```
perl(perl)>=5.008001
```

- SITEPREFIX

  Like PERLPREFIX, but only for the site install locations.

  Defaults to $Config{siteprefixexp}. Perls prior to 5.6.0 didn't have an explicit siteprefix in the Config. In those cases $Config{installprefix} will be used.

  Overridable by PREFIX

- SIGN

  Available in version 6.18 and above.

  When true, perform the generation and addition to the MANIFEST of the SIGNATURE file in the distdir during 'make distdir', via 'cpansign -s'.

  Note that you need to install the Module::Signature module to perform this operation.

  Defaults to false.

- SKIP

  Arrayref. E.g. [qw(name1 name2)] skip (do not write) sections of the Makefile. Caution! Do not use the SKIP attribute for the negligible speedup. It may seriously damage the resulting Makefile. Only use it if you really need it.

- TEST_REQUIRES

  Available in version 6.64 and above.

  A hash of modules that are needed to test your module but not run or build it.

  This will go into the `build_requires` field of your `META.yml` and the `test` of the `prereqs` field of your `META.json`.

  The format is the same as PREREQ_PM.

- TYPEMAPS

  Ref to array of typemap file names. Use this when the typemaps are in some directory other than the current directory or when they are not named **typemap**. The last typemap in the list takes precedence. A typemap in the current directory has highest precedence, even if it isn't listed in TYPEMAPS. The default system typemap has lowest precedence.

- VENDORPREFIX

  Like PERLPREFIX, but only for the vendor install locations.

  Defaults to $Config{vendorprefixexp}.

  Overridable by PREFIX

- VERBINST

  If true, make install will be verbose

- VERSION

  Your version number for distributing the package. This defaults to 0.1.

- VERSION_FROM

  Instead of specifying the VERSION in the Makefile.PL you can let MakeMaker parse a file to determine the version number. The parsing routine requires that the file named by VERSION_FROM contains one single line to compute the version number. The first line in the file that contains something like a $VERSION assignment or `package Name VERSION` will be used. The following lines will be parsed o.k.:

```
# Good
package Foo::Bar 1.23;                          # 1.23
$VERSION   = '1.00';                            # 1.00
*VERSION   = \'1.01';                           # 1.01
($VERSION) = q$Revision$ =~ /(\d+)/g;           # The digits in
 ↪$Revision$
$FOO::VERSION = '1.10';                         # 1.10
*FOO::VERSION = \'1.11';                        # 1.11
```

  but these will fail:

```
# Bad
my $VERSION         = '1.01';
local $VERSION      = '1.02';
local $FOO::VERSION = '1.30';
```

  (Putting `my` or `local` on the preceding line will work o.k.)

  "Version strings" are incompatible and should not be used.

```
# Bad
$VERSION = 1.2.3;
$VERSION = v1.2.3;
```

  version objects are fine. As of MakeMaker 6.35 version.pm will be automatically loaded, but you must declare the dependency on version.pm. For compatibility with older MakeMaker you should load on the same line as $VERSION is declared.

```
# All on one line
use version; our $VERSION = qv(1.2.3);
```

  The file named in VERSION_FROM is not added as a dependency to Makefile. This is not really correct, but it would be a major pain during development to have to rewrite the Makefile for any smallish change in that file. If you want to make sure that the Makefile contains the correct VERSION macro after any change of the file, you would have to do something like

```
depend => { Makefile => '$(VERSION_FROM)' }
```

  See attribute depend below.

- VERSION_SYM

  A sanitized VERSION with . replaced by _. For places where . has special meaning (some filesystems, RCS labels, etc...)

- XSBUILD

Available in version 7.12 and above.

Hashref with options controlling the operation of `XSMULTI`:

```
{
  xs => {
      all => {
          # options applying to all .xs files for this␣
↪distribution
      },
      'lib/Class/Name/File' => { # specifically for this file
          DEFINE => '-Dfunktastic', # defines for only this file
          INC => "-I$funkyliblocation", # include flags for␣
↪only this file
          # OBJECT => 'lib/Class/Name/File$(OBJ_EXT)', # default
          LDFROM => "lib/Class/Name/File\$(OBJ_EXT) $otherfile\
↪$(OBJ_EXT)", # what's linked
      },
  },
}
```

Note `xs` is the file-extension. More possibilities may arise in the future.

`LDFROM` defaults to the same as `OBJECT`.

The distinction between `OBJECT` and `LDFROM`: `OBJECT` is the make target, so make will try to build it.

- XSMULTI

  Available in version 7.12 and above.

  This feature should be considered experimental, and details of it may change.

  This feature was inspired by, and small portions of code copied from, ExtUtils::MakeMaker::BigHelper. Hopefully this feature will render that module mainly obsolete.

- XSOPT

  String of options to pass to xsubpp. This might include `-C++` or `-extern`. Do not include typemaps here; the TYPEMAP parameter exists for that purpose.

- XSPROTOARG

  May be set to `-prototypes`, `-noprototypes` or the empty string. The empty string is equivalent to the xsubpp default, or `-noprototypes`. See the xsubpp documentation for details. MakeMaker defaults to the empty string.

Additional lowercase attributes

can be used to pass parameters to the methods which implement that part of the Makefile. Parameters are specified as a hash ref but are passed to the method as a hash.

- clean

```
{FILES => "*.xyz foo"}
```

- depend

```
{ANY_TARGET => ANY_DEPENDENCY, ...}
```

(ANY_TARGET must not be given a double-colon rule by MakeMaker.)

- dist

```
{TARFLAGS => 'cvfF', COMPRESS => 'gzip', SUFFIX => '.gz',
SHAR => 'shar -m', DIST_CP => 'ln', ZIP => '/bin/zip',
ZIPFLAGS => '-rl', DIST_DEFAULT => 'private tardist' }
```

If you specify COMPRESS, then SUFFIX should also be altered, as it is needed to tell make the target file of the compression. Setting DIST_CP to ln can be useful, if you need to preserve the timestamps on your files. DIST_CP can take the values 'cp', which copies the file, 'ln', which links the file, and 'best' which copies symbolic links and links the rest. Default is 'best'.

- dynamic_lib

```
{ARMAYBE => 'ar', OTHERLDFLAGS => '...', INST_DYNAMIC_DEP => '..
↪.'}
```

- linkext

```
{LINKTYPE => 'static', 'dynamic' or ''}
```

NB: Extensions that have nothing but *.pm files had to say

```
{LINKTYPE => ''}
```

with Pre-5.0 MakeMakers. Since version 5.00 of MakeMaker such a line can be deleted safely. MakeMaker recognizes when there's nothing to be linked.

- macro

```
{ANY_MACRO => ANY_VALUE, ...}
```

- postamble

Anything put here will be passed to *MY::postamble()* if you have one.

- realclean

```
{FILES => '$(INST_ARCHAUTODIR)/*.xyz'}
```

- test

Specify the targets for testing.

```
{TESTS => 't/*.t'}
```

RECURSIVE_TEST_FILES can be used to include all directories recursively under t that contain .t files. It will be ignored if you provide your own TESTS attribute, defaults to false.

```
{RECURSIVE_TEST_FILES=>1}
```

This is supported since 6.76

- tool_autosplit

```
{MAXLEN => 8}
```

Overriding MakeMaker Methods

If you cannot achieve the desired Makefile behaviour by specifying attributes you may define private subroutines in the Makefile.PL. Each subroutine returns the text it wishes to have written to the Makefile. To override a section of the Makefile you can either say:

```
sub MY::c_o { "new literal text" }
```

or you can edit the default by saying something like:

```
package MY; # so that "SUPER" works right
sub c_o {
    my $inherited = shift->SUPER::c_o(@_);
    $inherited =~ s/old text/new text/;
    $inherited;
}
```

If you are running experiments with embedding perl as a library into other applications, you might find MakeMaker is not sufficient. You'd better have a look at *ExtUtils::Embed* which is a collection of utilities for embedding.

If you still need a different solution, try to develop another subroutine that fits your needs and submit the diffs to makemaker@perl.org

For a complete description of all MakeMaker methods see *ExtUtils::MM_Unix*.

Here is a simple example of how to add a new target to the generated Makefile:

```
sub MY::postamble {
    return <<'MAKE_FRAG';
$(MYEXTLIB): sdbm/Makefile
        cd sdbm && $(MAKE) all

MAKE_FRAG
}
```

The End Of Cargo Cult Programming

WriteMakefile() now does some basic sanity checks on its parameters to protect against typos and malformatted values. This means some things which happened to work in the past will now throw warnings and possibly produce internal errors.

Some of the most common mistakes:

- `MAN3PODS => ' '`

  This is commonly used to suppress the creation of man pages. MAN3PODS takes a hash ref not a string, but the above worked by accident in old versions of MakeMaker.

  The correct code is `MAN3PODS => { }`.

Hintsfile support

MakeMaker.pm uses the architecture-specific information from Config.pm. In addition it evaluates architecture specific hints files in a `hints/` directory. The hints files are expected to be named like their counterparts in `PERL_SRC/hints`, but with an `.pl` file name extension (eg. `next_3_2.pl`). They are simply `eval`ed by MakeMaker within the WriteMakefile() subroutine, and can be used to execute commands as well as to include special variables. The rules which hintsfile is chosen are the same as in Configure.

The hintsfile is eval()ed immediately after the arguments given to WriteMakefile are stuffed into a hash reference $self but before this reference becomes blessed. So if you want to do the equivalent to override or create an attribute you would say something like

```
$self->{LIBS} = ['-ldbm -lucb -lc'];
```

Distribution Support

For authors of extensions MakeMaker provides several Makefile targets. Most of the support comes from the *ExtUtils::Manifest* module, where additional documentation can be found.

- make distcheck

  reports which files are below the build directory but not in the MANIFEST file and vice versa. (See *"fullcheck" in ExtUtils::Manifest* for details)

- make skipcheck

  reports which files are skipped due to the entries in the `MANIFEST.SKIP` file (See *"skipcheck" in ExtUtils::Manifest* for details)

- make distclean

  does a realclean first and then the distcheck. Note that this is not needed to build a new distribution as long as you are sure that the MANIFEST file is ok.

- make veryclean

  does a realclean first and then removes backup files such as `*~`, `*.bak`, `*.old` and `*.orig`

- make manifest

  rewrites the MANIFEST file, adding all remaining files found (See *"mkmanifest" in ExtUtils::Manifest* for details)

- make distdir

  Copies all the files that are in the MANIFEST file to a newly created directory with the name `$(DISTNAME)-$(VERSION)`. If that directory exists, it will be removed first.

  Additionally, it will create META.yml and META.json module meta-data file in the distdir and add this to the distdir's MANIFEST. You can shut this behavior off with the NO_META flag.

- make disttest

  Makes a distdir first, and runs a `perl Makefile.PL`, a make, and a make test in that directory.

- make tardist

  First does a distdir. Then a command $(PREOP) which defaults to a null command, followed by $(TO_UNIX), which defaults to a null command under UNIX, and will convert files in distribution directory to UNIX format otherwise. Next it runs `tar` on that directory into a tarfile and deletes the directory. Finishes with a command $(POSTOP) which defaults to a null command.

- make dist

  Defaults to $(DIST_DEFAULT) which in turn defaults to tardist.

- make uutardist

  Runs a tardist first and uuencodes the tarfile.

- make shdist

  First does a distdir. Then a command $(PREOP) which defaults to a null command. Next it runs `shar` on that directory into a sharfile and deletes the intermediate directory again. Finishes with a command $(POSTOP) which defaults to a null command. Note: For shdist to work properly a `shar` program that can handle directories is mandatory.

- make zipdist

  First does a distdir. Then a command $(PREOP) which defaults to a null command. Runs `$(ZIP) $(ZIPFLAGS)` on that directory into a zipfile. Then deletes that directory. Finishes with a command $(POSTOP) which defaults to a null command.

- make ci

  Does a $(CI) and a $(RCS_LABEL) on all files in the MANIFEST file.

Customization of the dist targets can be done by specifying a hash reference to the dist attribute of the WriteMakefile call. The following parameters are recognized:

```
CI           ('ci -u')
COMPRESS     ('gzip --best')
POSTOP       ('@ :')
PREOP        ('@ :')
TO_UNIX      (depends on the system)
RCS_LABEL    ('rcs -q -Nv$(VERSION_SYM):')
SHAR         ('shar')
```

(continues on next page)

(continued from previous page)

```
SUFFIX       ('.gz')
TAR          ('tar')
TARFLAGS     ('cvf')
ZIP          ('zip')
ZIPFLAGS     ('-r')
```

An example:

```
WriteMakefile(
    ...other options...
    dist => {
        COMPRESS => "bzip2",
        SUFFIX   => ".bz2"
    }
);
```

Module Meta-Data (META and MYMETA)

Long plaguing users of MakeMaker based modules has been the problem of getting basic information about the module out of the sources *without* running the `Makefile.PL` and doing a bunch of messy heuristics on the resulting `Makefile`. Over the years, it has become standard to keep this information in one or more CPAN Meta files distributed with each distribution.

The original format of CPAN Meta files was *YAML* and the corresponding file was called `META.yml`. In 2010, version 2 of the *CPAN::Meta::Spec* was released, which mandates JSON format for the metadata in order to overcome certain compatibility issues between YAML serializers and to avoid breaking older clients unable to handle a new version of the spec. The *CPAN::Meta* library is now standard for accessing old and new-style Meta files.

If *CPAN::Meta* is installed, MakeMaker will automatically generate `META.json` and `META.yml` files for you and add them to your `MANIFEST` as part of the 'distdir' target (and thus the 'dist' target). This is intended to seamlessly and rapidly populate CPAN with module meta-data. If you wish to shut this feature off, set the `NO_META WriteMakefile()` flag to true.

At the 2008 QA Hackathon in Oslo, Perl module toolchain maintainers agreed to use the CPAN Meta format to communicate post-configuration requirements between toolchain components. These files, `MYMETA.json` and `MYMETA.yml`, are generated when `Makefile.PL` generates a `Makefile` (if *CPAN::Meta* is installed). Clients like *CPAN* or CPANPLUS will read these files to see what prerequisites must be fulfilled before building or testing the distribution. If you wish to shut this feature off, set the `NO_MYMETA WriteMakefile()` flag to true.

Disabling an extension

If some events detected in `Makefile.PL` imply that there is no way to create the Module, but this is a normal state of things, then you can create a `Makefile` which does nothing, but succeeds on all the "usual" build targets. To do so, use

```
use ExtUtils::MakeMaker qw(WriteEmptyMakefile);
WriteEmptyMakefile();
```

instead of WriteMakefile().

This may be useful if other modules expect this module to be *built* OK, as opposed to *work* OK (say, this system-dependent module builds in a subdirectory of some other distribution, or is listed as a dependency in a CPAN::Bundle, but the functionality is supported by different means on the current architecture).

Other Handy Functions

- prompt

  ```
  my $value = prompt($message);
  my $value = prompt($message, $default);
  ```

  The `prompt()` function provides an easy way to request user input used to write a makefile. It displays the $message as a prompt for input. If a $default is provided it will be used as a default. The function returns the $value selected by the user.

  If `prompt()` detects that it is not running interactively and there is nothing on STDIN or if the PERL_MM_USE_DEFAULT environment variable is set to true, the $default will be used without prompting. This prevents automated processes from blocking on user input.

  If no $default is provided an empty string will be used instead.

- os_unsupported

  ```
  os_unsupported();
  os_unsupported if $^O eq 'MSWin32';
  ```

  The `os_unsupported()` function provides a way to correctly exit your `Makefile.PL` before calling `WriteMakefile`. It is essentially a `die` with the message "OS unsupported".

  This is supported since 7.26

Supported versions of Perl

Please note that while this module works on Perl 5.6, it is no longer being routinely tested on 5.6 - the earliest Perl version being routinely tested, and expressly supported, is 5.8.1. However, patches to repair any breakage on 5.6 are still being accepted.

ENVIRONMENT

- PERL_MM_OPT

  Command line options used by `MakeMaker->new()`, and thus by `WriteMakefile()`. The string is split as the shell would, and the result is processed before any actual command line arguments are processed.

```
PERL_MM_OPT='CCFLAGS="-Wl,-rpath -Wl,/foo/bar/lib" LIBS="-
↪lwibble -lwobble"'
```

- PERL_MM_USE_DEFAULT

  If set to a true value then MakeMaker's prompt function will always return the default without waiting for user input.

- PERL_CORE

  Same as the PERL_CORE parameter. The parameter overrides this.

SEE ALSO

Module::Build is a pure-Perl alternative to MakeMaker which does not rely on make or any other external utility. It may be easier to extend to suit your needs.

Module::Build::Tiny is a minimal pure-Perl alternative to MakeMaker that follows the Build.PL protocol of Module::Build but without its complexity and cruft, implementing only the installation of the module and leaving authoring to mbtiny or other authoring tools.

Module::Install is a (now discouraged) wrapper around MakeMaker which adds features not normally available.

ExtUtils::ModuleMaker and Module::Starter are both modules to help you setup your distribution.

CPAN::Meta and *CPAN::Meta::Spec* explain CPAN Meta files in detail.

File::ShareDir::Install makes it easy to install static, sometimes also referred to as 'shared' files. *File::ShareDir* helps accessing the shared files after installation. Test::File::ShareDir helps when writing tests to use the shared files both before and after installation.

Dist::Zilla is an authoring tool which allows great customization and extensibility of the author experience, relying on the existing install tools like ExtUtils::MakeMaker only for installation.

Dist::Milla is a Dist::Zilla bundle that greatly simplifies common usage.

Minilla is a minimal authoring tool that does the same things as Dist::Milla without the overhead of Dist::Zilla.

AUTHORS

Andy Dougherty doughera@lafayette.edu, Andreas König andreas.koenig@mind.de, Tim Bunce timb@cpan.org. VMS support by Charles Bailey bailey@newman.upenn.edu. OS/2 support by Ilya Zakharevich ilya@math.ohio-state.edu.

Currently maintained by Michael G Schwern schwern@pobox.com

Send patches and ideas to makemaker@perl.org.

Send bug reports via http://rt.cpan.org/. Please send your generated Makefile along with your report.

For more up-to-date information, see https://metacpan.org/release/ExtUtils-MakeMaker.

Repository available at https://github.com/Perl-Toolchain-Gang/ExtUtils-MakeMaker.

LICENSE

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See http://www.perl.com/perl/misc/Artistic.html

Modules

- `ExtUtils::MakeMaker::Config` - Wrapper around Config.pm
- `ExtUtils::MakeMaker::Locale` - bundled Encode::Locale

ExtUtils::MakeMaker::Config

Wrapper around Config.pm

SYNOPSIS

```
use ExtUtils::MakeMaker::Config;
print $Config{installbin};  # or whatever
```

DESCRIPTION

FOR INTERNAL USE ONLY

A very thin wrapper around Config.pm so MakeMaker is easier to test.

ExtUtils::MakeMaker::Locale

bundled Encode::Locale

SYNOPSIS

```
use Encode::Locale;
use Encode;

$string = decode(locale => $bytes);
$bytes = encode(locale => $string);

if (-t) {
    binmode(STDIN, ":encoding(console_in)");
    binmode(STDOUT, ":encoding(console_out)");
    binmode(STDERR, ":encoding(console_out)");
}

# Processing file names passed in as arguments
my $uni_filename = decode(locale => $ARGV[0]);
```

(continues on next page)

(continued from previous page)

```
open(my $fh, "<", encode(locale_fs => $uni_filename))
   || die "Can't open '$uni_filename': $!";
binmode($fh, ":encoding(locale)");
...
```

DESCRIPTION

In many applications it's wise to let Perl use Unicode for the strings it processes. Most of the interfaces Perl has to the outside world are still byte based. Programs therefore need to decode byte strings that enter the program from the outside and encode them again on the way out.

The POSIX locale system is used to specify both the language conventions requested by the user and the preferred character set to consume and output. The `Encode::Locale` module looks up the charset and encoding (called a CODESET in the locale jargon) and arranges for the *Encode* module to know this encoding under the name "locale". It means bytes obtained from the environment can be converted to Unicode strings by calling `Encode::encode(locale => $bytes)` and converted back again with `Encode::decode(locale => $string)`.

Where file systems interfaces pass file names in and out of the program we also need care. The trend is for operating systems to use a fixed file encoding that don't actually depend on the locale; and this module determines the most appropriate encoding for file names. The *Encode* module will know this encoding under the name "locale_fs". For traditional Unix systems this will be an alias to the same encoding as "locale".

For programs running in a terminal window (called a "Console" on some systems) the "locale" encoding is usually a good choice for what to expect as input and output. Some systems allows us to query the encoding set for the terminal and `Encode::Locale` will do that if available and make these encodings known under the `Encode` aliases "console_in" and "console_out". For systems where we can't determine the terminal encoding these will be aliased as the same encoding as "locale". The advice is to use "console_in" for input known to come from the terminal and "console_out" for output to the terminal.

In addition to arranging for various Encode aliases the following functions and variables are provided:

- decode_argv( )
- decode_argv(Encode::FB_CROAK)

  This will decode the command line arguments to perl (the `@ARGV` array) in-place.

  The function will by default replace characters that can't be decoded by "\x{FFFD}", the Unicode replacement character.

  Any argument provided is passed as CHECK to underlying Encode::decode() call. Pass the value `Encode::FB_CROAK` to have the decoding croak if not all the command line arguments can be decoded. See *"Handling Malformed Data" in Encode* for details on other options for CHECK.
- env($uni_key)
- env($uni_key => $uni_value)

Interface to get/set environment variables. Returns the current value as a Unicode string. The $uni_key and $uni_value arguments are expected to be Unicode strings as well. Passing undef as $uni_value deletes the environment variable named $uni_key.

The returned value will have the characters that can't be decoded replaced by "\x{FFFD}", the Unicode replacement character.

There is no interface to request alternative CHECK behavior as for decode_argv(). If you need that you need to call encode/decode yourself. For example:

```
my $key = Encode::encode(locale => $uni_key, Encode::FB_CROAK);
my $uni_value = Encode::decode(locale => $ENV{$key}, Encode::FB_
↪CROAK);
```

- reinit( )
- reinit($encoding)

  Reinitialize the encodings from the locale. You want to call this function if you changed anything in the environment that might influence the locale.

  This function will croak if the determined encoding isn't recognized by the Encode module.

  With argument force $ENCODING_... variables to set to the given value.

- $ENCODING_LOCALE

  The encoding name determined to be suitable for the current locale. *Encode* know this encoding as "locale".

- $ENCODING_LOCALE_FS

  The encoding name determined to be suitable for file system interfaces involving file names. *Encode* know this encoding as "locale_fs".

- $ENCODING_CONSOLE_IN
- $ENCODING_CONSOLE_OUT

  The encodings to be used for reading and writing output to the a console. *Encode* know these encodings as "console_in" and "console_out".

NOTES

This table summarizes the mapping of the encodings set up by the Encode::Locale module:

```
Encode      |         |              |
Alias       | Windows | Mac OS X     | POSIX
------------+---------+--------------+------------
locale      | ANSI    | nl_langinfo  | nl_langinfo
locale_fs   | ANSI    | UTF-8        | nl_langinfo
console_in  | OEM     | nl_langinfo  | nl_langinfo
console_out | OEM     | nl_langinfo  | nl_langinfo
```

Windows

Windows has basically 2 sets of APIs. A wide API (based on passing UTF-16 strings) and a byte based API based a character set called ANSI. The regular Perl interfaces to the OS currently only uses the ANSI APIs. Unfortunately ANSI is not a single character set.

The encoding that corresponds to ANSI varies between different editions of Windows. For many western editions of Windows ANSI corresponds to CP-1252 which is a character set similar to ISO-8859-1. Conceptually the ANSI character set is a similar concept to the POSIX locale CODESET so this module figures out what the ANSI code page is and make this available as $ENCODING_LOCALE and the "locale" Encoding alias.

Windows systems also operate with another byte based character set. It's called the OEM code page. This is the encoding that the Console takes as input and output. It's common for the OEM code page to differ from the ANSI code page.

Mac OS X

On Mac OS X the file system encoding is always UTF-8 while the locale can otherwise be set up as normal for POSIX systems.

File names on Mac OS X will at the OS-level be converted to NFD-form. A file created by passing a NFC-filename will come in NFD-form from readdir(). See *Unicode::Normalize* for details of NFD/NFC.

Actually, Apple does not follow the Unicode NFD standard since not all character ranges are decomposed. The claim is that this avoids problems with round trip conversions from old Mac text encodings. See Encode::UTF8Mac for details.

POSIX (Linux and other Unixes)

File systems might vary in what encoding is to be used for filenames. Since this module has no way to actually figure out what the is correct it goes with the best guess which is to assume filenames are encoding according to the current locale. Users are advised to always specify UTF-8 as the locale charset.

SEE ALSO

I18N::Langinfo, *Encode*, Term::Encoding

AUTHOR

Copyright 2010 Gisle Aas gisle@aas.no.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

ExtUtils::Manifest

Utilities to write and check a MANIFEST file

VERSION

version 1.75

SYNOPSIS

```
use ExtUtils::Manifest qw(...funcs to import...);

mkmanifest();

my @missing_files     = manicheck;
my @skipped           = skipcheck;
my @extra_files       = filecheck;
my($missing, $extra) = fullcheck;

my $found    = manifind();

my $manifest = maniread();

manicopy($read,$target);

maniadd({$file => $comment, ...});
```

DESCRIPTION

...

FUNCTIONS

ExtUtils::Manifest exports no functions by default. The following are exported on request:

mkmanifest

```
mkmanifest();
```

Writes all files in and below the current directory to your `MANIFEST`. It works similar to the result of the Unix command

```
find . > MANIFEST
```

All files that match any regular expression in a file `MANIFEST.SKIP` (if it exists) are ignored.

Any existing `MANIFEST` file will be saved as `MANIFEST.bak`.

manifind

```
my $found = manifind();
```

returns a hash reference. The keys of the hash are the files found below the current directory.

manicheck

```
my @missing_files = manicheck();
```

checks if all the files within a MANIFEST in the current directory really do exist. If MANIFEST and the tree below the current directory are in sync it silently returns an empty list. Otherwise it returns a list of files which are listed in the MANIFEST but missing from the directory, and by default also outputs these names to STDERR.

filecheck

```
my @extra_files = filecheck();
```

finds files below the current directory that are not mentioned in the MANIFEST file. An optional file MANIFEST.SKIP will be consulted. Any file matching a regular expression in such a file will not be reported as missing in the MANIFEST file. The list of any extraneous files found is returned, and by default also reported to STDERR.

fullcheck

```
my($missing, $extra) = fullcheck();
```

does both a manicheck() and a filecheck(), returning then as two array refs.

skipcheck

```
my @skipped = skipcheck();
```

lists all the files that are skipped due to your MANIFEST.SKIP file.

maniread

```
my $manifest = maniread();
my $manifest = maniread($manifest_file);
```

reads a named MANIFEST file (defaults to MANIFEST in the current directory) and returns a HASH reference with files being the keys and comments being the values of the HASH. Blank lines and lines which start with # in the MANIFEST file are discarded.

maniskip

```
my $skipchk = maniskip();
my $skipchk = maniskip($manifest_skip_file);

if ($skipchk->($file)) { .. }
```

reads a named `MANIFEST.SKIP` file (defaults to `MANIFEST.SKIP` in the current directory) and returns a CODE reference that tests whether a given filename should be skipped.

manicopy

```
manicopy(\%src, $dest_dir);
manicopy(\%src, $dest_dir, $how);
```

Copies the files that are the keys in %src to the $dest_dir. %src is typically returned by the maniread() function.

```
manicopy( maniread(), $dest_dir );
```

This function is useful for producing a directory tree identical to the intended distribution tree.

$how can be used to specify a different methods of "copying". Valid values are `cp`, which actually copies the files, `ln` which creates hard links, and `best` which mostly links the files but copies any symbolic link to make a tree without any symbolic link. `cp` is the default.

maniadd

```
maniadd({ $file => $comment, ...});
```

Adds an entry to an existing `MANIFEST` unless its already there.

$file will be normalized (ie. Unixified). **UNIMPLEMENTED**

MANIFEST

A list of files in the distribution, one file per line. The MANIFEST always uses Unix filepath conventions even if you're not on Unix. This means `foo/bar` style not `foo\bar`.

Anything between white space and an end of line within a `MANIFEST` file is considered to be a comment. Any line beginning with # is also a comment. Beginning with ExtUtils::Manifest 1.52, a filename may contain whitespace characters if it is enclosed in single quotes; single quotes or backslashes in that filename must be backslash-escaped.

```
# this a comment
some/file
some/other/file           comment about some/file
'some/third file'         comment
```

MANIFEST.SKIP

The file MANIFEST.SKIP may contain regular expressions of files that should be ignored by mkmanifest() and filecheck(). The regular expressions should appear one on each line. Blank lines and lines which start with # are skipped. Use \# if you need a regular expression to start with a #.

For example:

```
# Version control files and dirs.
\bRCS\b
\bCVS\b
,v$
\B\.svn\b

# Makemaker generated files and dirs.
^MANIFEST\.
^Makefile$
^blib/
^MakeMaker-\d

# Temp, old and emacs backup files.
~$
\.old$
^#.*#$
^\.#
```

If no MANIFEST.SKIP file is found, a default set of skips will be used, similar to the example above. If you want nothing skipped, simply make an empty MANIFEST.SKIP file.

In one's own MANIFEST.SKIP file, certain directives can be used to include the contents of other MANIFEST.SKIP files. At present two such directives are recognized.

- #!include_default

  This tells ExtUtils::Manifest to read the default `MANIFEST.SKIP` file and skip files accordingly, but *not* to include it in the local `MANIFEST.SKIP`. This is intended to skip files according to a system default, which can change over time without requiring further changes to the distribution's `MANIFEST.SKIP`.

- #!include /Path/to/another/manifest.skip

  This inserts the contents of the specified external file in the local `MANIFEST.SKIP`. This is intended for authors to have a central `MANIFEST.SKIP` file, and to include it with their various distributions.

The included contents will be inserted into the MANIFEST.SKIP file in between *#!start included /path/to/manifest.skip* and *#!end included /path/to/manifest.skip* markers. The original MANIFEST.SKIP is saved as MANIFEST.SKIP.bak.

EXPORT_OK

`&mkmanifest`, `&manicheck`, `&filecheck`, `&fullcheck`, `&maniread`, and `&manicopy` are exportable.

GLOBAL VARIABLES

`$ExtUtils::Manifest::MANIFEST` defaults to `MANIFEST`. Changing it results in both a different `MANIFEST` and a different `MANIFEST.SKIP` file. This is useful if you want to maintain different distributions for different audiences (say a user version and a developer version including RCS).

`$ExtUtils::Manifest::Quiet` defaults to 0. If set to a true value, all functions act silently.

`$ExtUtils::Manifest::Debug` defaults to 0. If set to a true value, or if PERL_MM_MANIFEST_DEBUG is true, debugging output will be produced.

DIAGNOSTICS

All diagnostic output is sent to `STDERR`.

- `Not in MANIFEST:` *file*

  is reported if a file is found which is not in `MANIFEST`.

- `Skipping` *file*

  is reported if a file is skipped due to an entry in `MANIFEST.SKIP`.

- `No such file:` *file*

  is reported if a file mentioned in a `MANIFEST` file does not exist.

- `MANIFEST:` *$!*

  is reported if `MANIFEST` could not be opened.

- `Added to MANIFEST:` *file*

  is reported by mkmanifest() if $Verbose is set and a file is added to MANIFEST. $Verbose is set to 1 by default.

ENVIRONMENT

- **PERL_MM_MANIFEST_DEBUG**

  Turns on debugging

SEE ALSO

ExtUtils::MakeMaker which has handy targets for most of the functionality.

AUTHOR

Andreas Koenig andreas.koenig@anima.de

Currently maintained by the Perl Toolchain Gang.

COPYRIGHT AND LICENSE

This software is copyright (c) 1996- by Andreas Koenig.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

ExtUtils::Miniperl

write the C code for miniperlmain.c and perlmain.c

SYNOPSIS

```
use ExtUtils::Miniperl;
writemain(@directories);
# or
writemain($fh, @directories);
# or
writemain(\$filename, @directories);
```

DESCRIPTION

`writemain()` takes an argument list of zero or more directories containing archive libraries that relate to perl modules and should be linked into a new perl binary. It writes a corresponding `miniperlmain.c` or `perlmain.c` file that is a plain C file containing all the bootstrap code to make the modules associated with the libraries available from within perl. If the first argument to `writemain()` is a reference to a scalar it is used as the filename to open for output. Any other reference is used as the filehandle to write to. Otherwise output defaults to `STDOUT`.

The typical usage is from within perl's own Makefile (to build `perlmain.c`) or from `regen/miniperlmain.pl` (to build miniperlmain.c). So under normal circumstances you won't have to deal with this module directly.

SEE ALSO

ExtUtils::MakeMaker

ExtUtils::PL2Bat

Batch file creation to run perl scripts on Windows

VERSION

version 0.005

OVERVIEW

This module converts a perl script into a batch file that can be executed on Windows/DOS-like operating systems. This is intended to allow you to use a Perl script like regular programs and batch files where you just enter the name of the script [probably minus the extension] plus any command-line arguments and the script is found in your **PATH** and run.

FUNCTIONS

pl2bat(%opts)

This function takes a perl script and write a batch file that contains the script. This is sometimes necessary

- `in`

  The name of the script that is to be batchified. This argument is mandatory.

- `out`

  The name of the output batch file. If not given, it will be generated using `in` and `stripsuffix`.

- `ntargs`

  Arguments to invoke perl with in generated batch file when run from Windows NT. Defaults to '-x -S %0 %*'.

- `otherargs`

  Arguments to invoke perl with in generated batch file except when run from Windows NT (ie. when run from DOS, Windows 3.1, or Windows 95). Defaults to '-x -S "%0" %1 %2 %3 %4 %5 %6 %7 %8 %9'.

- `stripsuffix`

  Strip a suffix string from file name before appending a ".bat" suffix. The suffix is not case-sensitive. It can be a regex or a string and a trailing $ is always assumed). Defaults to `qr/\.plx?/`.

- `usewarnings`

  With the `usewarnings` option, `" -w"` is added after the value of `$Config{startperl}`. If a line matching `/^#!.*perl/` already exists in the script, then it is not changed and the **-w** option is ignored.

- `update`

  If the script appears to have already been processed by **pl2bat**, then the script is skipped and not processed unless `update` was specified. If `update` is specified, the existing preamble is replaced.

ACKNOWLEDGEMENTS

This code was taken from Module::Build and then modified; which had taken it from perl's pl2bat script. This module is an attempt at unifying all three implementations.

AUTHOR

Leon Timmermans leont@cpan.org

COPYRIGHT AND LICENSE

This software is copyright (c) 2015 by Leon Timmermans.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

ExtUtils::Packlist

manage .packlist files

SYNOPSIS

```
use ExtUtils::Packlist;
my ($pl) = ExtUtils::Packlist->new('.packlist');
$pl->read('/an/old/.packlist');
my @missing_files = $pl->validate();
$pl->write('/a/new/.packlist');

$pl->{'/some/file/name'}++;
   or
$pl->{'/some/other/file/name'} = { type => 'file',
                                   from => '/some/file' };
```

DESCRIPTION

ExtUtils::Packlist provides a standard way to manage .packlist files. Functions are provided to read and write .packlist files. The original .packlist format is a simple list of absolute pathnames, one per line. In addition, this package supports an extended format, where as well as a filename each line may contain a list of attributes in the form of a space separated list of key=value pairs. This is used by the installperl script to differentiate between files and links, for example.

USAGE

The hash reference returned by the new() function can be used to examine and modify the contents of the .packlist. Items may be added/deleted from the .packlist by modifying the hash. If the value associated with a hash key is a scalar, the entry written to the .packlist by any subsequent write() will be a simple filename. If the value is a hash, the entry written will be the filename followed by the key=value pairs from the hash. Reading back the .packlist will recreate the original entries.

FUNCTIONS

- new()

  This takes an optional parameter, the name of a .packlist. If the file exists, it will be opened and the contents of the file will be read. The new() method returns a reference to a hash. This hash holds an entry for each line in the .packlist. In the case of old-style .packlists, the value associated with each key is undef. In the case of new-style .packlists, the value associated with each key is a hash containing the key=value pairs following the filename in the .packlist.

- read()

  This takes an optional parameter, the name of the .packlist to be read. If no file is specified, the .packlist specified to new() will be read. If the .packlist does not exist, Carp::croak will be called.

- write()

  This takes an optional parameter, the name of the .packlist to be written. If no file is specified, the .packlist specified to new() will be overwritten.

- validate()

  This checks that every file listed in the .packlist actually exists. If an argument which evaluates to true is given, any missing files will be removed from the internal hash. The return value is a list of the missing files, which will be empty if they all exist.

- packlist_file()

  This returns the name of the associated .packlist file

EXAMPLE

Here's `modrm`, a little utility to cleanly remove an installed module.

```
#!/usr/local/bin/perl -w

use strict;
use IO::Dir;
use ExtUtils::Packlist;
use ExtUtils::Installed;

sub emptydir($) {
    my ($dir) = @_;
    my $dh = IO::Dir->new($dir) || return(0);
    my @count = $dh->read();
    $dh->close();
    return(@count == 2 ? 1 : 0);
}

# Find all the installed packages
print("Finding all installed modules...\n");
my $installed = ExtUtils::Installed->new();
```

(continues on next page)

(continued from previous page)

```
foreach my $module (grep(!/^Perl$/, $installed->modules())) {
   my $version = $installed->version($module) || "???";
   print("Found module $module Version $version\n");
   print("Do you want to delete $module? [n] ");
   my $r = <STDIN>; chomp($r);
   if ($r && $r =~ /^y/i) {
      # Remove all the files
      foreach my $file (sort($installed->files($module))) {
         print("rm $file\n");
         unlink($file);
      }
      my $pf = $installed->packlist($module)->packlist_file();
      print("rm $pf\n");
      unlink($pf);
      foreach my $dir (sort($installed->directory_tree($module))) {
         if (emptydir($dir)) {
            print("rmdir $dir\n");
            rmdir($dir);
         }
      }
   }
}
```

AUTHOR

Alan Burlison Alan.Burlison@uk.sun.com

ExtUtils::testlib

add blib/* directories to @INC

SYNOPSIS

```
use ExtUtils::testlib;
```

DESCRIPTION

After an extension has been built and before it is installed it may be desirable to test it bypassing `make test`. By adding

```
use ExtUtils::testlib;
```

to a test program the intermediate directories used by `make` are added to @INC.

5.2.41 FFI

Namespace containing the following modules:

- `FFI::Build` - Build shared libraries for use with FFI
- `FFI::CheckLib` - Check that a library is available for FFI
- `FFI::Platypus` - Call C functions in shared libraries from Perl, with no compiler.
- `FFI::Probe` - System detection and probing for FFI extensions.
- `FFI::Temp` - Temp Dir support for FFI::Platypus

FFI::Build

Build shared libraries for use with FFI

VERSION

version 2.12

SYNOPSIS

```
use FFI::Platypus 2.00;
use FFI::Build;

my $build = FFI::Build->new(
  'frooble',
  source => 'ffi/*.c',
);

# $lib is an instance of FFI::Build::File::Library
my $lib = $build->build;

my $ffi = FFI::Platypus->new( api => 2 );
# The filename will be platform dependant, but something like␣
↪libfrooble.so or frooble.dll
$ffi->lib($lib->path);

... # use $ffi to attach functions in ffi/*.c
```

DESCRIPTION

Sometimes, however, you need to bundle a little C code with your FFI module, but this has never been that easy to use. Module::Build::FFI was an early attempt to address this use case, but it uses the now out of fashion Module::Build.

This module itself doesn't directly integrate with CPAN installers like *ExtUtils::MakeMaker* or Module::Build, but there is a light weight layer *FFI::Build::MM* that will allow you to easily use this module with *ExtUtils::MakeMaker*. If you are using Dist::Zilla as your dist builder, then there is also Dist::Zilla::Plugin::FFI::Build, which will help with the connections.

There is some functional overlap with *ExtUtils::CBuilder*, which was in fact used by Module::Build::FFI.

Supported languages out of the box are C, C++ and Fortran. Rust is supported via a language plugin, see FFI::Platypus::Lang::Rust.

CONSTRUCTOR

new

```
my $build = FFI::Build->new($name, %options);
```

Create an instance of this class. The `$name` argument is used when computing the file name for the library. The actual name will be something like `lib$name.so` or `$name.dll`. The following options are supported:

- alien

  List of Aliens to compile/link against. *FFI::Build* will work with any Alien::Base based alien, or modules that provide a compatible API.

- buildname

  Directory name that will be used for building intermediate files, such as object files. This is `_build` by default.

- cflags

  Extra compiler flags to use. Things like `-I/foo/include` or `-DFOO=1`.

- dir

  The directory where the library will be written. This is `.` by default.

- export

  Functions that should be exported (Windows + Visual C++ only)

- file

  An instance of *FFI::Build::File::Library* to which the library will be written. Normally not needed.

- libs

  Extra library flags to use. Things like `-L/foo/lib -lfoo`.

- platform

  An instance of *FFI::Build::Platform*. Usually you want to omit this and use the default instance.

- source

  List of source files. You can use wildcards supported by `bsd_glob` from *File::Glob*.

- verbose

  By default this class does not print out the actual compiler and linker commands used in building the library unless there is a failure. You can alter this behavior with this option. Set to one of these values:

- zero (0)

  Default, quiet unless there is a failure.

- one (1)

  Output the operation (compile, link, etc) and the file, but nothing else

- two (2)

  Output the complete commands run verbatim.

If the environment variable `V` is set to a true value then the verbosity will be set to 2 regardless of what is passed in.

METHODS

dir

```
my $dir = $build->dir;
```

Returns the directory where the library will be written.

buildname

```
my $builddir = $build->builddir;
```

Returns the build name. This is used in computing a directory to save intermediate files like objects. For example, if you specify a file like `ffi/foo.c`, then the object file will be stored in `ffi/_build/foo.o` by default. `_build` in this example (the default) is the build name.

export

```
my $exports = $build->export;
```

Returns a array reference of the exported functions (Windows + Visual C++ only)

file

```
my $file = $build->file;
```

Returns an instance of *FFI::Build::File::Library* corresponding to the library being built. This is also returned by the `build` method below.

platform

```
my $platform = $build->platform;
```

An instance of *FFI::Build::Platform*, which contains information about the platform on which you are building. The default is usually reasonable.

verbose

```
my $verbose = $build->verbose;
```

Returns the verbose flag.

cflags

```
my @cflags = @{ $build->cflags };
```

Returns the compiler flags.

cflags_I

```
my @cflags_I = @{ $build->cflags_I };
```

Returns the `-I` cflags.

libs

```
my @libs = @{ $build->libs };
```

Returns the library flags.

libs_L

```
my @libs = @{ $build->libs };
```

Returns the `-L` library flags.

alien

```
my @aliens = @{ $build->alien };
```

Returns a the list of aliens being used.

source

```
$build->source(@files);
```

Add the `@files` to the list of source files that will be used in building the library. The format is the same as with the `source` attribute above.

build

```
my $lib = $build->build;
```

This compiles the source files and links the library. Files that have already been compiled or linked may be reused without recompiling/linking if the timestamps are newer than the source files. An instance of *FFI::Build::File::Library* is returned which can be used to get the path to the library, which can be feed into *FFI::Platypus* or similar.

clean

```
$build->clean;
```

Removes the library and intermediate files.

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

Modules

- `FFI::Build::File`
- `FFI::Build::MM` - FFI::Build installer code for ExtUtils::MakeMaker
- `FFI::Build::Platform` - Platform specific configuration.
- `FFI::Build::Plugin` - Platform and local customizations of FFI::Build
- `FFI::Build::PluginData` - Platform and local customizations of FFI::Build

FFI::Build::File

Namespace containing the following modules:

- `FFI::Build::File::Base` - Base class for File::Build files
- `FFI::Build::File::C` - Class to track C source file in FFI::Build
- `FFI::Build::File::CXX` - Class to track C source file in FFI::Build
- `FFI::Build::File::Library` - Class to track object file in FFI::Build
- `FFI::Build::File::Object` - Class to track object file in FFI::Build

FFI::Build::File::Base

Base class for File::Build files

VERSION

version 2.12

SYNOPSIS

Create your own file class

```
package FFI::Build::File::Foo;
use parent qw( FFI::Build::File::Base );
use constant default_suffix => '.foo';
use constant default_encoding => ':utf8';
```

Use it:

```
# use an existing file in the filesystem
my $file = FFI::Build::File::Foo->new('src/myfile.foo');
```

(continues on next page)

(continued from previous page)

```
# generate a temp file with provided content
# file will be deletd when $file falls out of scope.
my $file = FFI::Build::File::Foo->new(\'content for a temp foo');
```

DESCRIPTION

This class is the base class for other `FFI::Build::File::*` classes.

CONSTRUCTOR

new

```
my $file = FFI::Build::File::Base->new(\$content, %options);
my $file = FFI::Build::File::Base->new($filename, %options);
```

Create a new instance of the file class. You may provide either the content of the file as a scalar reference, or the path to an existing filename. Options:

- base

  The base name for any temporary file `ffi_build_` by default.

- build

  The *FFI::Build* instance to use.

- dir

  The directory to store any temporary file.

- platform

  The *FFI::Build::Platform* instance to use.

METHODS

default_suffix

```
my $suffix = $file->default_suffix;
```

MUST be overridden in the subclass. This is the standard extension for the file type. `.c` for a C file, `.o` or `.obj` for an object file depending on platform. etc.

default_encoding

```
my $encoding = $file->default_encoding;
```

MUST be overridden in the subclass. This is the passed to `binmode` when the file is opened for reading or writing.

accept_suffix

```
my @suffix_list = $file->accept_suffix;
```

Returns a list of regexes that recognize the file type.

path

```
my $path = $file->path;
```

The full or relative path to the file.

basename

```
my $basename = $file->basename;
```

The base filename part of the path.

dirname

```
my $dir = $file->dirname;
```

The directory part of the path.

is_temp

```
my $bool = $file->is_temp;
```

Returns true if the file is temporary, that is, it will be deleted when the file object falls out of scope. You can call keep, to keep the file.

platform

```
my $platform = $file->platform;
```

The *FFI::Build::Platform* instance used for this file object.

build

```
my $build = $file->build;
```

The *FFI::Build* instance used for this file object, if any.

native

```
my $path = $file->native;
```

Returns the operating system native version of the filename path. On Windows, this means that forward slash `\` is used instead of backslash `/`.

slurp

```
my $content = $file->slurp;
```

Returns the content of the file.

keep

```
$file->keep;
```

Turns off the temporary flag on the file object, meaning it will not automatically be deleted when the file object is deallocated or falls out of scope.

build_item

```
$file->build_item;
```

Builds the file into its natural output type, usually an object file. It returns a new file instance, or if the file is an object file then it returns empty list.

build_all

```
$file->build_all;
```

If implemented the file in question can directly create a shared or dynamic library without needing a link step. This is useful for languages that have their own build systems.

needs_rebuild

```
my $bool = $file->needs_rebuild
```

ld

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::Build::File::C

Class to track C source file in FFI::Build

VERSION

version 2.12

SYNOPSIS

```
use FFI::Build::File::C;

my $c = FFI::Build::File::C->new('src/foo.c');
```

DESCRIPTION

File class for C source files.

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::Build::File::CXX

Class to track C source file in FFI::Build

VERSION

version 2.12

SYNOPSIS

```
use FFI::Build::File::CXX;

my $c = FFI::Build::File::CXX->new('src/foo.cxx');
```

DESCRIPTION

File class for C++ source files.

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::Build::File::Library

Class to track object file in FFI::Build

VERSION

version 2.12

SYNOPSIS

```
use FFI::Build;

my $build = FFI::Build->new(source => 'src/*.c');
# $lib is an instance of FFI::Build::File::Library
my $lib = $build->build;
```

DESCRIPTION

This is a class to track a library generated by *FFI::Build*. This is returned by *FFI::Build*'s build method. This class is a subclass of *FFI::Build::File::Base*. The most important method is probably `path`, which returns the path to the library which can be passed into *FFI::Platypus* for immediate use.

METHODS

path

```
my $path = $lib->path;
```

Returns the path of the library.

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::Build::File::Object

Class to track object file in FFI::Build

VERSION

version 2.12

SYNOPSIS

```
use FFI::Build::File::Object;
my $o = FFI::Build::File::Object->new('src/_build/foo.o');
```

DESCRIPTION

This class represents an object file. You normally do not need to use it directly.

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::Build::MM

FFI::Build installer code for ExtUtils::MakeMaker

VERSION

version 2.12

SYNOPSIS

In your Makefile.PL:

```
use ExtUtils::MakeMaker;
use FFI::Build::MM;

my $fbmm = FFI::Build::MM->new;

```

(continues on next page)

(continued from previous page)

```
WriteMakefile($fbmm->mm_args(
  ABSTRACT     => 'My FFI extension',
  DISTNAME     => 'Foo-Bar-Baz-FFI',
  NAME         => 'Foo::Bar::Baz::FFI',
  VERSION_FROM => 'lib/Foo/Bar/Baz/FFI.pm',
  ...
));

sub MY::postamble {
  $fbmm->mm_postamble;
}
```

Then put the C, C++ or Fortran files in `./ffi` for your runtime library and `./t/ffi` for your test time library.

DESCRIPTION

This module provides a thin layer between *FFI::Build* and *ExtUtils::MakeMaker*. Its interface is influenced by the design of Alien::Build::MM. The idea is that for your distribution you throw some C, C++ or Fortran source files into a directory called `ffi` and these files will be compiled and linked into a library that can be used by your module. There is a control file `ffi/*.fbx` which can be used to control the compiler and linker options. (options passed directly into *FFI::Build*). The interface for this file is still under development.

CONSTRUCTOR

new

```
my $fbmm = FFI::Build::MM->new;
```

Create a new instance of *FFI::Build::MM*.

METHODS

mm_args

```
my %new_args = $fbmm->mm_args(%old_args);
```

This method does two things:

- reads the arguments to determine sensible defaults (library name, install location, etc).
- adjusts the arguments as necessary and returns an updated set of arguments.

mm_postamble

```
my $postamble = $fbmm->mm_postamble;
```

This returns the Makefile postamble used by *ExtUtils::MakeMaker*. The synopsis above for how to invoke it properly. It adds the following Make targets:

- fbx_build / ffi

  build the main runtime library in `./ffi`.

- fbx_test / ffi-test

  Build the test library in `./t/ffi`.

- fbx_clean / ffi-clean

  Clean any runtime or test libraries already built.

Normally you do not need to build these targets manually, they will be built automatically at the appropriate stage.

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::Build::Platform

Platform specific configuration.

VERSION

version 2.12

SYNOPSIS

```
use FFI::Build::Platform;
```

DESCRIPTION

This class is used to abstract out the platform specific parts of the *FFI::Build* system. You shouldn't need to use it directly in most cases, unless you are working on *FFI::Build* itself.

CONSTRUCTOR

new

```
my $platform = FFI::Build::Platform->new;
```

Create a new instance of *FFI::Build::Platform*.

default

```
my $platform = FFI::Build::Platform->default;
```

Returns the default instance of *FFI::Build::Platform*.

METHODS

All of these methods may be called either as instance or classes methods. If called as a class method, the default instance will be used.

osname

```
my $osname = $platform->osname;
```

The "os name" as understood by Perl. This is the same as `$^O`.

object_suffix

```
my $suffix = $platform->object_suffix;
```

The object suffix for the platform. On UNIX this is usually `.o`. On Windows this is usually `.obj`.

library_suffix

```
my(@suffix) = $platform->library_suffix;
my $suffix  = $platform->library_suffix;
```

The library suffix for the platform. On Linux and some other UNIX this is often `.so`. On OS X, this is `.dylib` and `.bundle`. On Windows this is `.dll`.

library_prefix

```
my $prefix = $platform->library_prefix;
```

The library prefix for the platform. On Unix this is usually `lib`, as in `libfoo`.

cc

```
my @cc = @{ $platform->cc };
```

The C compiler

cpp

```
my @cpp = @{ $platform->cpp };
```

The C pre-processor

cxx

```
my @cxx = @{ $platform->cxx };
```

The C++ compiler that naturally goes with the C compiler.

cxxld

```
my @cxxld = @{ $platform->cxxld };
```

The C++ linker that naturally goes with the C compiler.

for

```
my @for = @{ $platform->for };
```

The Fortran compiler that naturally goes with the C compiler.

ld

```
my $ld = $platform->ld;
```

The C linker

shellwords

```
my @words = $platform->shellwords(@strings);
```

This is a wrapper around *Text::ParseWords*'s `shellwords` with some platform workarounds applied.

ccflags

```
my @ccflags = @{ $platform->cflags};
```

The compiler flags, including those needed to compile object files that can be linked into a dynamic library. On Linux, for example, this is usually includes `-fPIC`.

ldflags

```
my @ldflags = @{ $platform->ldflags };
```

The linker flags needed to link object files into a dynamic library. This is NOT the `libs` style library flags that specify the location and name of a library to link against, this is instead the flags that tell the linker to generate a dynamic library. On Linux, for example, this is usually `-shared`.

cc_mm_works

```
my $bool = $platform->cc_mm_works;
```

Returns the flags that can be passed into the C compiler to compute dependencies.

flag_object_output

```
my @flags = $platform->flag_object_output($object_filename);
```

Returns the flags that the compiler recognizes as being used to write out to a specific object filename.

flag_library_output

```
my @flags = $platform->flag_library_output($library_filename);
```

Returns the flags that the compiler recognizes as being used to write out to a specific library filename.

flag_exe_output

```
my @flags = $platform->flag_exe_output($library_filename);
```

Returns the flags that the compiler recognizes as being used to write out to a specific exe filename.

flag_export

```
my @flags = $platform->flag_export(@symbols);
```

Returns the flags that the linker recognizes for exporting functions.

which

```
my $path = $platform->which($command);
```

Returns the full path of the given command, if it is available, otherwise undef is returned.

run

```
$platform->run(@command);
```

diag

Diagnostic for the platform as a string. This is for human consumption only, and the format may and will change over time so do not attempt to use is programmatically.

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::Build::Plugin

Platform and local customizations of FFI::Build

VERSION

version 2.12

SYNOPSIS

```
perldoc FFI::Build
```

DESCRIPTION

This class is experimental, but may do something useful in the future.

SEE ALSO

- *FFI::Platypus*
- *FFI::Build*

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::Build::PluginData

Platform and local customizations of FFI::Build

VERSION

version 2.12

SYNOPSIS

```
perldoc FFI::Build
```

DESCRIPTION

This class is experimental, but may do something useful in the future.

SEE ALSO

- *FFI::Platypus*
- *FFI::Build*

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::CheckLib

Check that a library is available for FFI

VERSION

version 0.31

SYNOPSIS

```
use FFI::CheckLib;

check_lib_or_exit( lib => 'jpeg', symbol => 'jinit_memory_mgr' );
check_lib_or_exit( lib => [ 'iconv', 'jpeg' ] );

# or prompt for path to library and then:
print "where to find jpeg library: ";
my $path = <STDIN>;
check_lib_or_exit( lib => 'jpeg', libpath => $path );
```

DESCRIPTION

This module checks whether a particular dynamic library is available for FFI to use. It is modeled heavily on Devel::CheckLib, but will find dynamic libraries even when development packages are not installed. It also provides a *find_lib* function that will return the full path to the found dynamic library, which can be feed directly into *FFI::Platypus* or another FFI system.

Although intended mainly for FFI modules via *FFI::Platypus* and similar, this module does not actually use any FFI to do its detection and probing. This module does not have any non-core runtime dependencies. The test suite does depend on *Test2::Suite*.

FUNCTIONS

All of these take the same named parameters and are exported by default.

find_lib

```
my(@libs) = find_lib(%args);
```

This will return a list of dynamic libraries, or empty list if none were found.

[version 0.05]

If called in scalar context it will return the first library found.

Arguments are key value pairs with these keys:

- lib

  Must be either a string with the name of a single library or a reference to an array of strings of library names. Depending on your platform, `CheckLib` will prepend `lib` or append `.dll` or `.so` when searching.

  [version 0.11]

  As a special case, if * is specified then any libs found will match.

- libpath

  A string or array of additional paths to search for libraries.

- systempath

  [version 0.11]

  A string or array of system paths to search for instead of letting *FFI::CheckLib* determine the system path. You can set this to `[]` in order to not search *any* system paths.

- symbol

  A string or a list of symbol names that must be found.

- verify

  A code reference used to verify a library really is the one that you want. It should take two arguments, which is the name of the library and the full path to the library pathname. It should return true if it is acceptable, and false otherwise. You can use this in conjunction with *FFI::Platypus* to determine if it is going to meet your needs. Example:

```
use FFI::CheckLib;
use FFI::Platypus;

my($lib) = find_lib(
  lib => 'foo',
  verify => sub {
    my($name, $libpath) = @_;

    my $ffi = FFI::Platypus->new;
    $ffi->lib($libpath);

    my $f = $ffi->function('foo_version', [] => 'int');
```

(continues on next page)

(continued from previous page)

```
    return $f->call() >= 500; # we accept version 500 or better
  },
);
```

- recursive

  [version 0.11]

  Recursively search for libraries in any non-system paths (those provided via `lib-path` above).

- try_linker_script

  [version 0.24]

  Some vendors provide `.so` files that are linker scripts that point to the real binary shared library. These linker scripts can be used by gcc or clang, but are not directly usable by *FFI::Platypus* and friends. On select platforms, this options will use the linker command (`ld`) to attempt to resolve the real `.so` for non-binary files. Since there is extra overhead this is off by default.

  An example is libyaml on Red Hat based Linux distributions. On Debian these are handled with symlinks and no trickery is required.

- alien

  [version 0.25]

  If no libraries can be found, try the given aliens instead. The Alien classes specified must provide the Alien::Base interface for dynamic libraries, which is to say they should provide a method called `dynamic_libs` that returns a list of dynamic libraries.

  [version 0.28]

  In 0.28 and later, if the Alien is not installed then it will be ignored and this module will search in system or specified directories only. This module *will* still throw an exception, if the Alien doesn't look like a module name or if it does not provide a `dynamic_libs` method (which is implemented by all Alien::Base subclasses).

  [version 0.30] [breaking change]

  Starting with version 0.30, libraries provided by Aliens is preferred over the system libraries. The original thinking was that you want to prefer the system libraries because they are more likely to get patched with regular system updates. Unfortunately, the reason a module needs to install an Alien is likely because the system library is not new enough, so we now prefer the Aliens instead.

assert_lib

```
assert_lib(%args);
```

This behaves exactly the same as *find_lib*, except that instead of returning empty list of failure it throws an exception.

check_lib_or_exit

```
check_lib_or_exit(%args);
```

This behaves exactly the same as *assert_lib*, except that instead of dying, it warns (with exactly the same error message) and exists. This is intended for use in `Makefile.PL` or `Build.PL`

find_lib_or_exit

[version 0.05]

```
my(@libs) = find_lib_or_exit(%args);
```

This behaves exactly the same as *find_lib*, except that if the library is not found, it will call exit with an appropriate diagnostic.

find_lib_or_die

[version 0.06]

```
my(@libs) = find_lib_or_die(%args);
```

This behaves exactly the same as *find_lib*, except that if the library is not found, it will die with an appropriate diagnostic.

check_lib

```
my $bool = check_lib(%args);
```

This behaves exactly the same as *find_lib*, except that it returns true (1) on finding the appropriate libraries or false (0) otherwise.

which

[version 0.17]

```
my $path = which($name);
```

Return the path to the first library that matches the given name.

Not exported by default.

where

[version 0.17]

```
my @paths = where($name);
```

Return the paths to all the libraries that match the given name.

Not exported by default.

has_symbols

[version 0.17]

```
my $bool = has_symbols($path, @symbol_names);
```

Returns true if *all* of the symbols can be found in the dynamic library located at the given path. Can be useful in conjunction with `verify` with `find_lib` above.

Not exported by default.

system_path

[version 0.20]

```
my $path = FFI::CheckLib::system_path;
```

Returns the system path as a list reference. On some systems, this is `PATH` on others it might be `LD_LIBRARY_PATH` on still others it could be something completely different. So although you *may* add items to this list, you should probably do some careful consideration before you do so.

This function is not exportable, even on request.

ENVIRONMENT

FFI::CheckLib responds to these environment variables:

- FFI_CHECKLIB_PACKAGE

  On macOS platforms with Homebrew and/or MacPorts installed, their corresponding lib paths will be automatically appended to `$system_path`. In case of having both managers installed, Homebrew will appear before.

  This behaviour can be overridden using the environment variable `FFI_CHECKLIB_PACKAGE`.

  Allowed values are:

  - none: Won't use either Homebrew's path nor MacPorts - homebrew: Will append `$(brew --prefix)/lib` to the system paths - `macports`: Will append `port`'s default lib path

  A comma separated list is also valid:

  ```
  export FFI_CHECKLIB_PACKAGE=macports,homebrew
  ```

  Order matters. So in this example, MacPorts' lib path appears before Homebrew's path.

- FFI_CHECKLIB_PATH

  List of directories that will be considered by *FFI::CheckLib* as additional "system directories". They will be searched before other system directories but after `libpath`. The variable is colon separated on Unix and semicolon separated on Windows. If you use this variable, `FFI_CHECKLIB_PACKAGE` will be ignored.

- PATH

  On Windows the PATH environment variable will be used as a search path for libraries.

On some operating systems LD_LIBRARY_PATH, DYLD_LIBRARY_PATH, DYLD_FALLBACK_LIBRARY_PATH or others *may* be used as part of the search for dynamic libraries and *may* be used (indirectly) by *FFI::CheckLib* as well.

FAQ

- Why not just use dlopen?

  Calling dlopen on a library name and then dlclose immediately can tell you if you have the exact name of a library available on a system. It does have a number of drawbacks as well.

  – No absolute or relative path

    It only tells you that the library is *somewhere* on the system, not having the absolute or relative path makes it harder to generate useful diagnostics.

  – POSIX only

    This doesn't work on non-POSIX systems like Microsoft Windows. If you are using a POSIX emulation layer on Windows that provides dlopen, like Cygwin, there are a number of gotchas there as well. Having a layer written in Perl handles this means that developers on Unix can develop FFI that will more likely work on these platforms without special casing them.

  – inconsistent implementations

    Even on POSIX systems you have inconsistent implementations. OpenBSD for example don't usually include symlinks for .so files meaning you need to know the exact .so version.

  – non-system directories

    By default dlopen only works for libraries in the system paths. Most platforms have a way of configuring the search for different non-system paths, but none of them are portable, and are usually discouraged anyway. Alien and friends need to do searches for dynamic libraries in non-system directories for share installs.

- My 64-bit Perl is misconfigured and has 32-bit libraries in its search path. Is that a bug in *FFI::CheckLib*?

  Nope.

- The way *FFI::CheckLib* is implemented it won't work on AIX, HP-UX, OpenVMS or Plan 9.

  I know for a fact that it doesn't work on AIX *as currently implemented* because I used to develop on AIX in the early 2000s, and I am aware of some of the technical challenges. There are probably other systems that it won't work on. I would love to add support for these platforms. Realistically these platforms have a tiny market share, and absent patches from users or the companies that own these operating systems (patches welcome), or hardware / CPU time donations, these platforms are unsupportable anyway.

SEE ALSO

- *FFI::Platypus*

  Call library functions dynamically without a compiler.

- Dist::Zilla::Plugin::FFI::CheckLib

  Dist::Zilla plugin for this module.

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dan Book (grinnz, DBOOK)

Ilya Pavlov (Ilya, ILUX)

Shawn Laffan (SLAFFAN)

Petr Písař (ppisar)

Michael R. Davis (MRDVT)

Shawn Laffan (SLAFFAN)

Carlos D. Álvaro (cdalvaro)

COPYRIGHT AND LICENSE

This software is copyright (c) 2014-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::Platypus

Call C functions in shared libraries from Perl, with no compiler.

`FFI::Platypus->new(api => 2, lib => ...)` finds a library; `$ffi->attach(name => [argument types] => return type)` makes a C function a Perl sub, `$ffi->function(...)` an object to call. Types cover the C integers and floats, `long double` and the complex types, strings, pointers and fixed arrays of them, records (C structs) by pointer and by value, closures (Perl subs C can call back), objects and custom types written in Perl.

This is a native implementation of FFI::Platypus 2.12's compiled part, `Platypus.xs` with its `xs/*.xs` and helpers; its Perl modules are used unchanged. Its calls, types, records and closures are those of pperl's own *Peta::FFI*, which can express everything FFI::Platypus can.

Synopsis

```
use FFI::Platypus 2.00;

my $ffi = FFI::Platypus->new( api => 2, lib => undef );
$ffi->attach( puts   => ['string'] => 'int' );
$ffi->attach( strlen => ['string'] => 'size_t' );

puts("hello");
my $n = strlen("hello");                # 5

my $qsort = $ffi->function( qsort =>
    ['int*', 'size_t', 'size_t', '(opaque,opaque)->int'] => 'void'␣
↪);
```

Differences from upstream

- Reading an array member of a record does not leak the array; perl leaks it and the reference on every read.
- An index past an opaque `[N]` member warns "illegal index" and returns nothing, as for the other array members; perl reads beyond the member.
- A reference to a hash or code for a pointer argument under api 2 warns and passes NULL; perl crashes.
- An argument with get-magic - a regex capture, a tied scalar - passes its value; perl passes 0 for a number and NULL for a string, which crashes `strlen($1)`.
- A closure that a library calls from a thread of its own stops the program with a message saying so; perl runs Perl on that thread and crashes ("longjmp causes uninitialized stack frame").

Modules

- `FFI::Platypus::API` - Platypus arguments and return value API for custom types
- `FFI::Platypus::Buffer` - Convert scalars to C buffers
- `FFI::Platypus::Bundle` - Bundle foreign code with your Perl module
- `FFI::Platypus::Closure` - Platypus closure object
- `FFI::Platypus::Constant` - Define constants in C space for Perl
- `FFI::Platypus::DL` - Slightly non-portable interface to libdl
- `FFI::Platypus::Function` - An FFI function object
- `FFI::Platypus::Internal` - For internal use only
- `FFI::Platypus::Lang` - Language specific customizations
- `FFI::Platypus::Legacy` - Legacy Platypus interfaces
- `FFI::Platypus::Memory` - Memory functions for FFI
- `FFI::Platypus::Record` - FFI support for structured records data

- `FFI::Platypus::ShareConfig` - FFI::Platypus::ShareConfig
- `FFI::Platypus::Type` - Defining types for FFI::Platypus
- `FFI::Platypus::TypeParser` - FFI Type Parser

FFI::Platypus::API

Platypus arguments and return value API for custom types

VERSION

version 2.12

SYNOPSIS

```
package FFI::Platypus::Type::MyCustomType;

use FFI::Platypus::API;

sub ffi_custom_type_api_1
{
  {
    native_type => 'uint32',
    perl_to_native => sub {
      my($value, $i) = @_;
      # Translates ($value) passed in from Perl
      # into ($value+1, $value+2)
      arguments_set_uint32($i, $value+1);
      arguments_set_uint32($i+1, $value+2);
    },
    argument_count => 2,
  }
}
```

DESCRIPTION

NOTE: I added this interface early on to *FFI::Platypus*, but haven't used it much, generally finding function wrappers to be a more powerful (although possibly not as fast) interface. It has thus not been tested as much as the rest of Platypus. If you feel the need to use this interface please coordinate with the Platypus developers.

The custom types API for *FFI::Platypus* allows you to set multiple C arguments from a single Perl argument as a common type. This is sometimes useful for pointer / size pairs which are a common pattern in C, but are usually represented by a single value (a string scalar) in Perl.

The custom type API is somewhat experimental, and you should expect some changes as needs arise (I won't break compatibility lightly, however).

FUNCTIONS

These functions are only valid within a custom type callback.

arguments_count

```
my $count = argument_count;
```

Returns the total number of native arguments.

arguments_get_sint8

```
my $sint8 = arguments_get_sint8 $i;
```

Get the 8 bit signed integer argument from position *$i*.

arguments_set_sint8

```
arguments_set_sint8 $i, $sint8;
```

Set the 8 bit signed integer argument at position *$i* to *$sint8*.

arguments_get_uint8

```
my $uint8 = arguments_get_uint8 $i;
```

Get the 8 bit unsigned integer argument from position *$i*.

arguments_set_uint8

```
arguments_set_uint8 $i, $uint8;
```

Set the 8 bit unsigned integer argument at position *$i* to *$uint8*.

arguments_get_sint16

```
my $sint16 = arguments_get_sint16 $i;
```

Get the 16 bit signed integer argument from position *$i*.

arguments_set_sint16

```
arguments_set_sint16 $i, $sint16;
```

Set the 16 bit signed integer argument at position *$i* to *$sint16*.

arguments_get_uint16

```
my $uint16 = arguments_get_uint16 $i;
```

Get the 16 bit unsigned integer argument from position *$i*.

arguments_set_uint16

```
arguments_set_uint16 $i, $uint16;
```

Set the 16 bit unsigned integer argument at position *$i* to *$uint16*.

arguments_get_sint32

```
my $sint32 = arguments_get_sint32 $i;
```

Get the 32 bit signed integer argument from position *$i*.

arguments_set_sint32

```
arguments_set_sint32 $i, $sint32;
```

Set the 32 bit signed integer argument at position *$i* to *$sint32*.

arguments_get_uint32

```
my $uint32 = arguments_get_uint32 $i;
```

Get the 32 bit unsigned integer argument from position *$i*.

arguments_set_uint32

```
arguments_set_uint32 $i, $uint32;
```

Set the 32 bit unsigned integer argument at position *$i* to *$uint32*.

arguments_get_sint64

```
my $sint64 = arguments_get_sint64 $i;
```

Get the 64 bit signed integer argument from position *$i*.

arguments_set_sint64

```
arguments_set_sint64 $i, $sint64;
```

Set the 64 bit signed integer argument at position *$i* to *$sint64*.

arguments_get_uint64

```
my $uint64 = arguments_get_uint64 $i;
```

Get the 64 bit unsigned integer argument from position *$i*.

arguments_set_uint64

```
arguments_set_uint64 $i, $uint64;
```

Set the 64 bit unsigned integer argument at position *$i* to *$uint64*.

arguments_get_float

```
my $float = arguments_get_float $i;
```

Get the floating point argument from position *$i*.

arguments_set_float

```
arguments_set_float $i, $float;
```

Set the floating point argument at position *$i* to *$float*

arguments_get_double

```
my $double = arguments_get_double $i;
```

Get the double precision floating point argument from position *$i*.

arguments_set_double

```
arguments_set_double $i, $double;
```

Set the double precision floating point argument at position *$i* to *$double*

arguments_get_pointer

```
my $pointer = arguments_get_pointer $i;
```

Get the pointer argument from position *$i*.

arguments_set_pointer

```
arguments_set_pointer $i, $pointer;
```

Set the pointer argument at position *$i* to *$pointer*.

arguments_get_string

```
my $string = arguments_get_string $i;
```

Get the string argument from position *$i*.

arguments_set_string

```
arguments_set_string $i, $string;
```

Set the string argument at position *$i* to *$string*.

SEE ALSO

- *FFI::Platypus*

Examples of use:

- *FFI::Platypus::Type::PointerSizeBuffer*

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::Platypus::Buffer

Convert scalars to C buffers

VERSION

version 2.12

SYNOPSIS

```
use FFI::Platypus::Buffer;
my($pointer, $size) = scalar_to_buffer $scalar;
my $scalar2 = buffer_to_scalar $pointer, $size;
```

DESCRIPTION

A common pattern in C is to pass a "buffer" or region of memory into a function with a pair of arguments, an opaque pointer and the size of the memory region. In Perl the equivalent structure is a scalar containing a string of bytes. This module provides portable functions for converting a Perl string or scalar into a buffer and back.

These functions are implemented using pack and unpack and so they should be relatively fast.

Both functions are exported by default, but you can explicitly export one or neither if you so choose.

A better way to do this might be with custom types see *FFI::Platypus::API* and *FFI::Platypus::Type*. These functions were taken from the now obsolete FFI::Util module, as they may be useful in some cases.

Caution: This module provides great power in the way that you interact with C code, but with that power comes great responsibility. Since you are dealing with blocks of memory you need to take care to understand the underlying ownership model of these pointers.

FUNCTIONS

scalar_to_buffer

```
my($pointer, $size) = scalar_to_buffer $scalar;
```

Convert a string scalar into a buffer. Returned in order are a pointer to the start of the string scalar's memory region and the size of the region.

You should NEVER try to free `$pointer`.

When you pass this pointer and size into a C function, it has direct access to the data stored in your scalar, so it is important that you not resize or free the scalar while it is in use by the C code. Typically if you are passing a buffer into a C function which reads or writes to the buffer, but does not keep the pointer for later use you are okay. If the buffer is in use long term by the C code, then you should consider copying the buffer instead. For example:

```
use FFI::Platypus::Buffer qw( scalar_to_buffer );
use FFI::Platypus::Memory qw( malloc memcpy free )

my($ptr, $size) = scalar_to_buffer $string;
c_function_that_does_not_keep_ptr( $ptr, $size); # okay

my($ptr, $size) = scalar_to_buffer $string;
my $ptr_copy = malloc($size);
memcpy($ptr_copy, $ptr, $size);
c_function_that_DOES_keep_ptr( $ptr_copy, $size); # also okay

...

# later when you know that the c code is no longer using the pointer
# Since you allocated the copy, you are responsible for free'ing it.
free($ptr_copy);
```

scalar_to_pointer

```
my $pointer = scalar_to_pointer $scalar;
```

Get the pointer to the scalar. (Similar to `scalar_to_buffer` above, but the size of the scalar is not computed or returned).

Not exported by default, but may be exported on request.

buffer_to_scalar

```
my $scalar = buffer_to_scalar $pointer, $size;
```

Convert the buffer region defined by the pointer and size into a string scalar.

Because of the way memory management works in Perl, the buffer is copied from the buffer into the scalar. If this pointer was returned from C land, then you should only free it if you allocated it.

grow

```
grow $scalar, $size, \%options;
```

Ensure that the scalar can contain at least `$size` bytes. The following are recognized:

- clear => *boolean*

  If true, `$scalar` is cleared prior to being enlarged. This avoids copying the existing contents to the reallocated memory if they are not needed.

  ```
  For example, after

   $scalar = "my string";
   grow $scalar, 100, { clear => 0 };
  ```

  `$scalar == "my string"`, while after

  ```
  $scalar = "my string";
  grow $scalar, 100;
  ```

  `length($scalar) == 0`

  It defaults to `true`.

- set_length => *boolean*

  If true, the length of the *string* in the `$scalar` is set to `$size`. (See the discussion in "set_used_length".) This is useful if a foreign function writes exactly `$size` bytes to `$scalar`, as it avoids a subsequent call to `set_used_length`. Contrast this

  ```
  grow my $scalar, 100;
  read_exactly_100_bytes_into_scalar( scalar_to_pointer($scalar)␣
   ↪);
  @chars = unpack( 'c*', $scalar );
  ```

  with this:

  ```
  grow my $scalar, 100, { set_length => 0 };
  read_exactly_100_bytes_into_scalar( scalar_to_pointer($scalar)␣
   ↪);
  set_used_length( $scalar, 100 );
  @chars = unpack( 'c*', $scalar );
  ```

  It defaults to `true`.

Any pointers obtained with `scalar_to_pointer` or `scalar_to_buffer` are no longer valid after growing the scalar.

Not exported by default, but may be exported on request.

set_used_length

```
set_used_length $scalar, $length;
```

Update Perl's notion of the length of the string in the scalar. A string scalar keeps track of two lengths: the number of available bytes and the number of used bytes. When a string scalar is used as a buffer by a foreign function, it is necessary to indicate to Perl how many bytes were actually written to it so that Perl's string functions (such as `substr` or `unpack`) will work correctly.

If `$length` is larger than what the scalar can hold, it is set to the maximum possible size.

In the following example, the foreign routine `read_doubles` may fill the buffer with up to a set number of doubles, returning the number actually written.

```
my $sizeof_double = $ffi->sizeof( 'double' );
my $max_doubles = 100;
my $max_length = $max_doubles * $sizeof_double;

my $buffer;                    # length($buffer) == 0
grow $buffer, $max_length;     # length($buffer) is still  0
my $pointer = scalar_to_pointer($buffer);

my $num_read = read_doubles( $pointer, $max_doubles );
                               # length($buffer) is still == 0

set_used_length $buffer, $num_read * $sizeof_double;
                               # length($buffer) is finally != 0

# unpack the native doubles into a Perl array
my @doubles = unpack( 'd*', $buffer );  # @doubles == $num_read
```

Not exported by default, but may be exported on request.

window

```
window $scalar, $pointer;
window $scalar, $pointer, $size;
window $scalar, $pointer, $size, $utf8;
```

This makes the scalar a read-only window into the arbitrary region of memory defined by `$pointer`, pointing to the start of the region and `$size`, the size of the region. If `$size` is omitted then it will assume a C style string and use the C `strlen` function to determine the size (the terminating `'\0'` will not be included).

This can be useful if you have a C function that returns a buffer pair (pointer, size), and want to access it from Perl without having to copy the data. This can also be useful when interfacing with programming languages that store strings as a address/length pair instead of a pointer to null-terminated sequence of bytes.

You can specify `$utf8` to set the UTF-8 flag on the scalar. Note that the behavior of setting the UTF-8 flag on a buffer that does not contain UTF-8 as understood by the version of Perl that you are running is undefined.

Hint: If you have a buffer that needs to be free'd by C once the scalar falls out of scope you can use Variable::Magic to apply magic to the scalar and free the pointer once it falls out of scope.

```
use FFI::Platypus::Buffer qw( scalar_to_pointer );
use FFI::Platypus::Memory qw( strdup free );
use Variable::Magic qw( wizard cast );

my $free_when_out_of_scope = wizard(
  free => sub {
    my $ptr = scalar_to_pointer ${$_[0]};
    free $ptr;
  }
);

my $ptr = strdup "Hello Perl";
my $scalar;
window $scalar, $ptr, 10;
cast $scalar, $free_when_out_of_scope;
undef $ptr;  # don't need to track the pointer anymore.

# we can now use scalar as a regular read-only Perl variable
print $scalar, "\n";  # prints "Hello Perl" without the \0

# this will free the C pointer
undef $scalar;
```

Hint: Returning a scalar string from a Perl function actually copies the value. If you want to return a string without copying then you need to return a reference.

```
sub c_string
{
  my $ptr = strdup "Hello Perl";
  my $scalar;
  window $scalar, $ptr, 10;
  cast $scalar, $free_when_out_of_scope;
  \$scalar;
}

my $ref = c_string();
print $$ref, "\n";  # prints "Hello Perl" without the \0
```

Not exported by default, but may be exported on request.

raw_scalar

```
my $buffer = raw_scalar $size;
```

Returns a new scalar of exactly `$size` bytes, suitable for use as a buffer. The contents of the returned scalar are **not** initialized, so it will contain whatever happened to be in the memory that was allocated for it. This makes it cheaper than the equivalent

```
my $buffer = "\0" x $size;
```

when you are about to have a foreign function overwrite the whole thing anyway. If you need the buffer zeroed out, use the `"\0" x $size` form instead.

Not exported by default, but may be exported on request.

SEE ALSO

- *FFI::Platypus*

  Main Platypus documentation.

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::Platypus::Bundle

Bundle foreign code with your Perl module

VERSION

version 2.12

SYNOPSIS

`ffi/foo.c`:

```
#include <ffi_platypus_bundle.h>
#include <string.h>

typedef struct {
  char *name;
  int value;
} foo_t;

foo_t*
foo__new(const char *class_name, const char *name, int value) {
  (void)class_name;
  foo_t *self = malloc( sizeof( foo_t ) );
  self->name = strdup(name);
  self->value = value;
  return self;
}

const char *
foo__name(foo_t *self) {
  return self->name;
}

int
foo__value(foo_t *self) {
  return self->value;
}

void
foo__DESTROY(foo_t *self) {
  free(self->name);
  free(self);
}
```

lib/Foo.pm:

```
package Foo;

use strict;
use warnings;
use FFI::Platypus 2.00;

my $ffi = FFI::Platypus->new( api => 2 );

$ffi->type('object(Foo)' => 'foo_t');
$ffi->mangler(sub {
  my $name = shift;
  $name =~ s/^/foo__/;
  $name;
});

$ffi->bundle;

$ffi->attach( new =>     [ 'string', 'string', 'int' ] => 'foo_t' ␣
↪);
$ffi->attach( name =>    [ 'foo_t' ]                   => 'string'␣
↪);
$ffi->attach( value =>   [ 'foo_t' ]                   => 'int'   ␣
↪);
$ffi->attach( DESTROY => [ 'foo_t' ]                   => 'void'  ␣
↪);

1;
```

t/foo.t

```
use Test2::V0;
use Foo;

my $foo = Foo->new("platypus", 10);
isa_ok $foo, 'Foo';
is $foo->name, "platypus";
is $foo->value, 10;

done_testing;
```

Makefile.PL:

```
use ExtUtils::MakeMaker;
use FFI::Build::MM;
my $fbmm = FFI::Build::MM->new;
WriteMakefile(
  $fbmm->mm_args(
    NAME     => 'Foo',
    DISTNAME => 'Foo',
```

(continues on next page)

(continued from previous page)

```
    VERSION  => '1.00',
    # ...
  )
);

sub MY::postamble
{
  $fbmm->mm_postamble;
}
```

or `dist.ini`:

```
name    = Foo
version = 0.01
...

[FFI::Build]
version = 1.04
```

DESCRIPTION

This document serves as a tutorial for using the new bundling interface provided by *FFI::Platypus* as of api version 1. It requires *FFI::Platypus* of at least 1.00.

Sometimes when writing FFI bindings you need to include a little C code (or your favorite compiled language) to finish things off. Alternatively, you might just want to write some C code (or your favorite compiled language) to include with your Perl module to make a tight loop faster. The bundling interface has you covered.

Basic example

To illustrate we will go through the files in the synopsis and explain how and why they work. To start with we have some C code which emulates object oriented code using `foo__` as a prefix. We use a C struct that we call `foo_t` to store our object data. On the C level the struct acts as a class, when combined with its functions that act as methods. The constructor just allocates the memory it needs for the `foo_t` instance, fills in the appropriate fields and returns the pointer:

```
foo_t*
foo__new(const char *class_name, const char *name, int value)
{
  (void) class_name;
  foo_t *self = malloc( sizeof( foo_t ) );
  self->name = strdup(name);
  self->value = value;
  return self;
}
```

We include a class name as the first argument, because Perl will include that when calling the constructor, but we do not use it here. An exercise for the reader would be to

add hierarchical inheritance.

There are also some methods which return member values. This class has only read only members, but you could have read/write or other methods depending on your needs.

```
const char *
foo__name(foo_t *self)
{
  return self->name;
}
```

We also include a destructor so that the memory owned by the object can be freed when it is no longer needed.

```
void
foo__DESTROY(foo_t *self)
{
  free(self->name);
  free(self);
}
```

This might start to look a little like a Perl module, and when we look at the Perl code that binds to this code, you will see why. First lets prepare the *FFI::Platypus* instance and specify the correct api version:

```
my $ffi = FFI::Platypus->new( api => 2 );
```

The bundle interface is only supported with api version 1, so if you try to use version 0 it will not work. Next we define an object type for `foo_t` which will associate it with the Perl class `Foo`.

```
$ffi->type('object(Foo)' => 'foo_t');
```

As object type is a blessed reference to an opaque (default) or integer type which can be used as a Perl object. Platypus does the translating of Perl object to and from the foo_t pointers that the C code understands. For more details on Platypus types see *FFI::Platypus::Type*.

Next we set the mangler on the Platypus instance so that we can refer to function names without the `foo__` prefix. You could just not use the prefix in your C code and skip this step, or you could refer to the function names in their full in your Perl code, however, this saves extra typing and allows you to bundle more than one class with your Perl code without having to worry about name conflicts.

```
$ffi->mangler(sub {
  my $name = shift;
  $name =~ s/^/foo__/;
  $name;
});
```

Finally we let Platypus know that we will be bundling code.

```
$ffi->bundle;
```

By default, this searches for the appropriate place for your dynamic libraries using the current package. In some cases you may need to override this, for example if your dist is named `Foo-Bar` but your specific class is named `Foo::Bar::Baz`, you'd want something like this:

```
package Foo::Bar::Baz;
use FFI::Platypus 2.00;
my $ffi = FFI::Platypus->new( api => 2 );
$ffi->bundle('Foo::Bar');
...
```

Now, finally we can attach the methods for our class:

```
$ffi->attach( new =>      [ 'string', 'int' ] => 'foo_t'  );
$ffi->attach( name =>     [ 'foo_t' ]         => 'string' );
$ffi->attach( value =>    [ 'foo_t' ]         => 'int'    );
$ffi->attach( DESTROY => [ 'foo_t' ]          => 'void'   );
```

Note that we do not have to include the `foo__` prefix because of the way we set up the mangler. If we hadn't done that then we could instead attach with the full names:

```
$ffi->attach( [ 'foo__new'  => 'new' ]  => [ 'string', 'int' ] =>
 ↪'foo_t'  );
$ffi->attach( [ 'foo__name' => 'name' ] => [ 'foo_t' ]         =>
 ↪'string' );
...
```

You're done! You can now use this class. Lets write a test to make sure it works,

```
use Test2::V0;
use Foo;

my $foo = Foo->new("platypus", 10);
isa_ok $foo, 'Foo';
is $foo->name, "platypus";
is $foo->value, 10;

done_testing;
```

and use `prove` to check that it works:

```
% prove -lvm
t/foo.t ..
ok 1 - An object of class 'Foo' isa 'Foo'
ok 2
ok 3
1..3
ok
All tests successful.
Files=1, Tests=3,  0 wallclock secs ( 0.02 usr  0.00 sys +  0.14␣
 ↪cusr  0.03 csys =  0.19 CPU)
Result: PASS
```

Platypus automatically compiles and links the dynamic library for you:

```
% ls ffi/_build
foo.c.o  libFoo.so
```

The C code will be rebuilt next time if the source code is newer than the object or dynamic libraries files. If the source files are not changed, then it won't be rebuilt to save time. If you are using the code without MakeMaker, or another build system you are responsible for cleaning up these files. This is intended as a convenience to allow you to test your code without having to invoke MakeMaker, or `dzil` or whatever build system you are using.

When you distribute your module though, you will want the dynamic library built just once at build-time and installed correctly so that it can be found at run-time. You don't need to make any changes to your C or Perl code, but you do need to tell MakeMaker to build and install the appropriate files using *FFI::Build::MM*:

```
use ExtUtils::MakeMaker;
use FFI::Build::MM;
my $fbmm = FFI::Build::MM->new;
WriteMakefile(
  $fbmm->mm_args(
    NAME     => 'Foo',
    DISTNAME => 'Foo',
    VERSION  => '1.00',
    # ...
  )
);

sub MY::postamble
{
  $fbmm->mm_postamble;
}
```

And we can invoke all the normal MakeMaker style stuff and our C code will be compiled, linked and installed at the appropriate steps.

```
% perl Makefile.PL
Generating a Unix-style Makefile
Writing Makefile for Foo
Writing MYMETA.yml and MYMETA.json
% make
cp lib/Foo.pm blib/lib/Foo.pm
"/Users/ollisg/perl5/perlbrew/perls/perl-5.30.0/bin/perl" -
 ↪MFFI::Build::MM=cmd -e fbx_build
CC ffi/foo.c
LD blib/lib/auto/share/dist/Foo/lib/libFoo.dylib
% make test
"/Users/ollisg/perl5/perlbrew/perls/perl-5.30.0/bin/perl" -
 ↪MFFI::Build::MM=cmd -e fbx_build
"/Users/ollisg/perl5/perlbrew/perls/perl-5.30.0/bin/perl" -
 ↪MFFI::Build::MM=cmd -e fbx_test
```

(continues on next page)

(continued from previous page)

```
PERL_DL_NONLAZY=1 "/Users/ollisg/perl5/perlbrew/perls/perl-5.30.0/
 ↪bin/perl" "-MExtUtils::Command::MM" "-MTest::Harness" "-e" "undef␣
 ↪*Test::Harness::Switches; test_harness(0, 'blib/lib', 'blib/arch')
 ↪" t/*.t
t/foo.t .. ok
All tests successful.
Files=1, Tests=3,  0 wallclock secs ( 0.01 usr  0.00 sys +  0.06␣
 ↪cusr  0.01 csys =  0.08 CPU)
Result: PASS
```

If the `Makefile.PL` file above looks overly complicated, you can use the Dist::Zilla::Plugin::FFI::Build plugin to simplify your life if you are using Dist::Zilla:

```
[FFI::Build]
version = 1.04
```

Specifying version 1.04 will ensure that any `.o` or `.so` files are pruned from your build tree and not distributed by mistake.

Initialization example

The bundle interface also gives you entry points which will be called automatically when your code is loaded and unloaded if they are found.

- `ffi_pl_bundle_init`

  ```
  void ffi_pl_bundle_init(const char *package, int argc, void␣
   ↪*argv[]);
  ```

  Called when the dynamic library is loaded. `package` is the Perl package that called `bundle` from Perl space. `argc` and `argv` represents an array of opaque pointers that can be passed as an array to bundle as the last argument. (the count `argc` is a little redundant because `argv` is also NULL terminated).

- `ffi_pl_bundle_constant`

  ```
  void ffi_pl_bundle_constant(const char *package, ffi_platypus_
   ↪constant_t *c);
  ```

  Called immediately after `ffi_pl_bundle_init`, and is intended to allow you to set Perl constants from C space. For details on how this works and what methods you can call on the `ffi_platypus_constant_t` instance, see *FFI::Platypus::Constant*.

- `ffi_pl_bundle_fini`

  ```
  void ffi_pl_bundle_fini(const char *package);
  ```

  Called when the dynamic library is unloaded. `package` is the Perl package that called `bundle` from Perl space when the library was loaded. **CAVEAT**: if you attach any functions then this will never be called, because attaching functions locks the Platypus instance into memory along with the libraries which it is using.

Here is an example that passes the version and a callback back into Perl space that emulates the Perl 5.10 say feature.

ffi/init.c:

```
#include <ffi_platypus_bundle.h>

char buffer[512];
const char *version;
void (*say)(const char *);

void
ffi_pl_bundle_init(const char *package, int argc, void *argv[])
{
  version = argv[0];
  say     = argv[1];

  say("in init!");

  snprintf(buffer, 512, "package = %s, version = %s", package,
↪version);
  say(buffer);

  snprintf(buffer, 512, "args = %d", argc);
  say(buffer);
}

void
ffi_pl_bundle_fini(const char *package)
{
  say("in fini!");
}
```

lib/Init.pm:

```
package Init;

use strict;
use warnings;
use FFI::Platypus 2.00;

our $VERSION = '1.00';

{
  my $ffi = FFI::Platypus->new( api => 2 );

  my $say = $ffi->closure(sub {
    my $string = shift;
    print "$string\n";
  });
```

(continues on next page)

(continued from previous page)

```
  $ffi->bundle([
    $ffi->cast( 'string' => 'opaque', $VERSION ),
    $ffi->cast( '(string)->void' => 'opaque', $say ),
  ]);

  undef $ffi;
  undef $say;
}

1;
```

The deinitialization order for the `$say` callback and the `$ffi` instance is essential here, so we do it manually with `undef`:

```
undef $ffi;
undef $say;
```

First we deallocate `$ffi` which calls `ffi_pl_bundle_fini`, which calls `$say`, so we want to make sure the latter is still allocated. Once `ffi_pl_bundle_fini` is done, we can safely deallocate `$say`.

If `ffi_pl_bundle_fini` didn't call back into Perl space like this then we don't have to be as careful about deallocating things in Perl space.

Compiler or linker flags example

There are times when you will want to specify your own compiler and linker flags for the C code that you are bundling. The `TL;DR` is that you can put a `.fbx` file in your `ffi` directory. This is a Perl script that returns a hash reference that is passed into the *FFI::Build* constructor. This allows you to set a number of options, including compiler and linker flags. A more detailed example follows:

You may want or need to set compiler and linker flags for your bundled C code. For example, say we have a header file, but instead of putting it in the `ffi` directory we want to put it in a separate directory called `include`.

`include/answer.h`:

```
#ifndef ANSWER_H
#define ANSWER_H

int answer(void);

#endif
```

`ffi/answer.c`:

```
int
answer(void)
{
  /* the answer to life the universe and everything */
```

(continues on next page)

(continued from previous page)

```
  return 42;
}
```

lib/Answer.pm:

```
package Answer;

use strict;
use warnings;
use FFI::Platypus 2.00;
use Exporter qw( import );

our @EXPORT = qw( answer );

my $ffi = FFI::Platypus->new( api => 2 );
$ffi->bundle;
$ffi->attach( answer => [] => 'int' );

1;
```

If you try to use this module just as-is you will get an error, about not being able to find the header file. Probably something like this:

```
ffi/answer.c:1:10: fatal error: 'answer.h' file not found
```

So we put a `answer.fbx` file in the `ffi` directory. (In case you are wondering FBX stands for "Ffi Build and file eXtensions should whenever possible be three characters long"). The name of the file can be anything so long as it ends in `.fbx`, we just choose `answer` here because that is the name of the project.

ffi/answer.fbx:

```
our $DIR;

return {
  cflags => "-I/include",
  source => "$DIR/*.c",
}
```

The `$DIR` variable is provided by the builder code. It is the root of the distribution, and is helpful if you need a fully qualified path. In this case you could have also used `ffi/*.c`.

The script returns a hash reference which is passed into the *FFI::Build* constructor, so you can use any of the options supported by that class. Now we should be able to use our bundled module:

```
% perl -Ilib -MAnswer=answer -E 'say answer'
42
```

Using bundled code with Alien.

A useful technique is to use Platypus with Alien technology. The Alien namespace is reserved for providing external non-Perl dependencies for CPAN modules. The nominal Alien module when installed looks for the library locally, and if it can't be found it fetches it from the internet, builds it, and installs it in a private directory so that it can be used by other CPAN modules. For Aliens that provide shared libraries, and that have simple interfaces that do not require additional C code you can easily just pass the shared libraries to Platypus directly. For modules that require some bundled C code and an Alien you have to link the Alien library with your bundled code. If the Alien uses the Alien::Base interface then all you have to do is give the name of the Alien to *FFI::Build*.

For example, the `bzip2` library provides an interface that requires the caller to allocate a C `struct` and then pass it to its various functions. The `struct` is actually pretty simple and you could use FFI::C or *FFI::Platypus::Record*, but here is an example of how you would connect bundled C code with an Alien.

`ffi/compress.c`:

```
#include <bzlib.h>
#include <stdlib.h>

int
bzip2__new(bz_stream **stream, int blockSize100k, int verbosity, int␣
↪workFactor )
{
  *stream = calloc(1, sizeof(bz_stream));

  return BZ2_bzCompressInit(*stream, blockSize100k, verbosity,␣
↪workFactor );
}
```

`lib/Bzip2.pm`:

```
package Bzip2;

use strict;
use warnings;
use FFI::Platypus 2.00;
use FFI::Platypus::Memory qw( free );

my $ffi = FFI::Platypus->new( api => 2 );
$ffi->bundle;

$ffi->mangler(sub {
  my $name = shift;
  $name =~ s/^/bzip2__/ unless $name =~ /^BZ2_/;
  $name;
});

=head2 new
```

(continues on next page)

(continued from previous page)

```
 my $bzip2 = Bzip2->new($block_size_100k, $verbosity, $work_flow);

=cut

$ffi->attach( new => ['opaque*', 'int', 'int', 'int'] => 'int' =>␣
 ↪sub {
  my $xsub = shift;
  my $class = shift;
  my $ptr;
  my $ret = $xsub->(\$ptr, @_);
  return bless \$ptr, $class;
});

$ffi->attach( [ BZ2_bzCompressEnd => 'DESTROY' ] => ['opaque'] =>
 ↪'int' => sub {
  my $xsub = shift;
  my $self = shift;
  my $ret = $xsub->($$self);
  free $$self;
});

1;
```

The `.fbx` file that goes with this to make it work with Alien::Libbz2 is now pretty trivial:

`ffi/bz2.fbx`:

```
{
  alien => ['Alien::Libbz2'],
  source => ['ffi/*.c'],
};
```

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::Platypus::Closure

Platypus closure object

VERSION

version 2.12

SYNOPSIS

create closure with OO interface

```
use FFI::Platypus::Closure;
my $closure = FFI::Platypus::Closure->new(sub { print "hello world\n
 ↪" });
```

create closure from Platypus object

```
use FFI::Platypus 2.00;
my $ffi = FFI::Platypus->new( api => 2 );
my $closure = $ffi->closure(sub { print "hello world\n" });
```

use closure

```
$ffi->function(foo => ['()->void'] => 'void')->call($closure);
```

DESCRIPTION

This class represents a Perl code reference that can be called from compiled code. When you create a closure object, you can pass it into any function that expects a function pointer. Care needs to be taken with closures because compiled languages typically have a different way of handling lifetimes of objects. You have to make sure that if the compiled code is going to call a closure that the closure object is still in scope somewhere, or has been made sticky, otherwise you may get a segment violation or other mysterious crash.

CONSTRUCTOR

new

```
my $closure = FFI::Platypus::Closure->new($coderef);
```

Create a new closure object; `$coderef` must be a subroutine code reference.

METHODS

call

```
$closure->call(@arguments);
$closure->(@arguments);
```

Call the closure from Perl space. May also be invoked by treating the closure object as a code reference.

sticky

```
$closure->sticky;
```

Mark the closure sticky, meaning that it won't be free'd even if all the reference of the object fall out of scope.

[version 2.12]

Returns `$self`, so this method may be chained.

unstick

```
$closure->unstick;
```

Unmark the closure as sticky.

[version 2.12]

Returns `$self`, so this method may be chained.

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::Platypus::Constant

Define constants in C space for Perl

VERSION

version 2.12

SYNOPSIS

`ffi/foo.c`:

```
#include <ffi_platypus_bundle.h>

void
ffi_pl_bundle_constant(const char *package, ffi_platypus_constant_t
 ↪*c)
{
  c->set_str("FOO", "BAR");       /* sets $package::FOO to "BAR" */
  c->set_str("ABC::DEF", "GHI");  /* sets ABC::DEF to GHI        */
}
```

`lib/Foo.pm`:

```
package Foo;

use strict;
use warnings;
use FFI::Platypus 2.00;
use Exporter qw( import );

my $ffi = FFI::Platypus->new( api => 2 );
# sets constants Foo::FOO and ABC::DEF from C
$ffi->bundle;

1;
```

DESCRIPTION

The Platypus bundle interface (see *FFI::Platypus::Bundle*) has an entry point `ffi_pl_bundle_constant` that lets you define constants in Perl space from C.

```
void ffi_pl_bundle_constant(const char *package, ffi_platypus_
 ↪constant_t *c);
```

The first argument `package` is the name of the Perl package. The second argument `c` is a struct with function pointers that lets you define constants of different types. The first argument for each function is the name of the constant and the second is the value. If `::` is included in the constant name then it will be defined in that package space. If it isn't then the constant will be defined in whichever package called `bundle`.

- set_str

  ```
  c->set_str(name, value);
  ```

  Sets a string constant.

- set_sint

```
c->set_sint(name, value);
```

  Sets a 64-bit signed integer constant.

- set_uint

```
c->set_uint(name, value);
```

  Sets a 64-bit unsigned integer constant.

- set_double

```
c->set_double(name, value);
```

  Sets a double precision floating point constant.

Example

Suppose you have a header file `myheader.h`:

```
#ifndef MYHEADER_H
#define MYHEADER_H

#define MYVERSION_STRING "1.2.3"
#define MYVERSION_MAJOR 1
#define MYVERSION_MINOR 2
#define MYVERSION_PATCH 3

enum {
  MYBAD = -1,
  MYOK  = 1
};

#define MYPI 3.14

#endif
```

You can define these constants from C:

```
#include <ffi_platypus_bundle.h>
#include "myheader.h"

void ffi_pl_bundle_constant(const char *package, ffi_platypus_
 ↪constant_t *c)
{
  c->set_str("MYVERSION_STRING", MYVERSION_STRING);
  c->set_uint("MYVERSION_MAJOR", MYVERSION_MAJOR);
  c->set_uint("MYVERSION_MINOR", MYVERSION_MINOR);
  c->set_uint("MYVERSION_PATCH", MYVERSION_PATCH);
  c->set_sint("MYBAD", MYBAD);
```

(continues on next page)

(continued from previous page)

```
  c->set_sint("MYOK", MYOK);
  c->set_double("MYPI", MYPI);
}
```

Your Perl code doesn't have to do anything when calling bundle:

```
package Const;

use strict;
use warnings;
use FFI::Platypus 2.00;

{
  my $ffi = FFI::Platypus->new( api => 2 );
  $ffi->bundle;
}

1;
```

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::Platypus::DL

Slightly non-portable interface to libdl

VERSION

version 2.12

SYNOPSIS

```
use FFI::Platypus 2.00;
use FFI::Platypus::DL;

my $handle = dlopen("./libfoo.so", RTLD_PLATYPUS_DEFAULT);
my $address = dlsym($handle, "my_function_named_foo");
my $ffi = FFI::Platypus->new( api => 2 );
$ffi->function($address => [] => 'void')->call;
dlclose($handle);
```

DESCRIPTION

This module provides an interface to libdl, the dynamic loader on UNIX. The underlying interface has always been used by *FFI::Platypus*, but it wasn't a public interface until version 0.52. The name was changed with that version when it became a public interface, so be sure to specify that version if you are going to use it.

It is somewhat non-portable for these reasons:

- GNU extensions

  It provides some GNU extensions to platforms such as Linux that support them.

- Windows

  It provides an emulation layer on Windows. The emulation layer only supports `RTLD_PLATYPUS_DEFAULT` as a flag. The emulation layer emulates the convention described below of passing `undef` as the dynamic library name to mean, use the currently running executable. I've used it without any problems for years, but Windows is not my main development platform.

FUNCTIONS

dlopen

```
my $handle = dlopen($filename, $flags);
```

This opens a dynamic library in the context of the dynamic loader. `$filename` is the full or relative path to a dynamic library (usually a `.so` on Linux and some other UNIXen, a `.dll` on Windows and a `.dylib` on OS X). `$flags` are flags that can be used to alter the behavior of the library and the symbols it contains. The return value is an opaque pointer or `$handle` which can be used to look up symbols with `dlsym`. The handle should be closed with `dlclose` when you are done with it.

By convention if you pass in `undef` for the filename, the currently loaded executable will be used instead of a separate dynamic library. This is the easiest and most portable way to find the address of symbols in the standard C library. This convention is baked into most UNIXen, but this capability is emulated in Windows which doesn't come with the capability out of the box.

If there is an error in opening the library then `undef` will be returned and the diagnostic for the failure can be retrieved with `dlerror` as described below.

Not all flags are supported on all platforms. You can test if a flag is available using can:

```
if(FFI::Platypus::DL->can('RTLD_LAZY'))
{
  ...
}
```

Typically where flags are not mutually exclusive, they can be or'd together:

```
my $handle = dlopen("libfoo.so", RTLD_LAZY | RTLD_GLOBAL);
```

Check your operating system documentation for detailed descriptions of these flags.

- RTLD_PLATYPUS_DEFAULT

  This is the *FFI::Platypus* default for `dlopen` (NOTE: NOT the libdl default). This is the only flag supported on Windows. For historical reasons, this is usually `RTLD_LAZY` on Unix and `0` on Windows.

- RTLD_LAZY

  Perform lazy binding.

- RTLD_NOW

  Resolve all symbols before returning from `dlopen`. Error if all symbols cannot resolve.

- RTLD_GLOBAL

  Symbols are shared.

- RTLD_LOCAL

  Symbols are NOT shared.

- RTLD_NODELETE

  glibc 2.2 extension.

- RTLD_NOLOAD

  glibc 2.2 extension.

- RTLD_DEEPBIND

  glibc 2.3.4 extension.

dlsym

```
my $opaque = dlsym($handle, $symbol);
```

This looks up the given `$symbol` in the library pointed to by `$handle`. If the symbol is found, the address for that symbol is returned as an opaque pointer. This pointer can be passed into the *FFI::Platypus* `function` and `attach` methods instead of a function name.

If the symbol cannot be found then `undef` will be returned and the diagnostic for the failure can be retrieved with `dlerror` as described below.

dlclose

```
my $status = dlclose($handle);
```

On success, `dlclose` returns 0; on error, it returns a nonzero value, and the diagnostic for the failure can be retrieved with `dlerror` as described below.

dlerror

```
my $error_string = dlerror;
```

Returns the human readable diagnostic for the reason for the failure for the most recent `dl` prefixed function call.

CAVEATS

Some flags for `dlopen` are not portable. This module may not be supported platforms added to *FFI::Platypus* in the future. It does work as far as I know on all of the currently supported platforms.

SEE ALSO

- *FFI::Platypus*

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::Platypus::Function

An FFI function object

VERSION

version 2.12

SYNOPSIS

```
use FFI::Platypus 2.00;

# call directly
my $ffi = FFI::Platypus->new( api => 2 );
my $f = $ffi->function(puts => ['string'] => 'int');
$f->call("hello there");

# attach as xsub and call (faster for repeated calls)
$f->attach('puts');
puts('hello there');
```

DESCRIPTION

This class represents an unattached platypus function. For more context and better examples see *FFI::Platypus*.

METHODS

attach

```
$f->attach($name);
$f->attach($name, $prototype);
```

Attaches the function as an xsub (similar to calling attach directly from an *FFI::Platypus* instance). You may optionally include a prototype.

call

```
my $ret = $f->call(@arguments);
my $ret = $f->(@arguments);
```

Calls the function and returns the result. You can also use the function object **like** a code reference.

sub_ref

```
my $code = $f->sub_ref;
```

Returns an anonymous code reference. This will usually be faster than using the `call` method above.

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::Platypus::Internal

For internal use only

VERSION

version 2.12

SYNOPSIS

```
perldoc FFI::Platypus
```

DESCRIPTION

This module is for internal use only. Do not rely on it having any particular behavior, or even existing in future versions. You have been warned.

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::Platypus::Lang

Language specific customizations

VERSION

version 2.12

SYNOPSIS

```
perldoc FFI::Platypus::Lang;
```

DESCRIPTION

This namespace is reserved for language specific customizations of *FFI::Platypus*. This usually involves providing native type maps. It can also involve computing mangled names. The default language is C, and is defined in *FFI::Platypus::Lang::C*.

This package itself doesn't do anything, it serves only as documentation.

SEE ALSO

- *FFI::Platypus*

  Platypus itself.

- *FFI::Platypus::Lang::ASM*

  This language plugin provides no type aliases, and is intended for use with assembly language or for when no other language plugin is appropriate.

- *FFI::Platypus::Lang::C*

  Language plugin for the C programming language.

- FFI::Platypus::Lang::Fortran

  Non-core language plugin for the Fortran programming language.

- FFI::Platypus::Lang::CPP

  Non-core language plugin for the C++ programming language.

- FFI::Platypus::Lang::Go

  Non-core language plugin for the Go programming language.

- FFI::Platypus::Lang::Pascal

  Non-core language plugin for the Pascal programming language.

- FFI::Platypus::Lang::Rust

  Non-core language plugin for the Rust programming language.

- *FFI::Platypus::Lang::Win32*

  Language plugin for use with the Win32 API.

- FFI::Platypus::Lang::Zig

  Non-core language plugin for the Zig programming language.

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

Modules

- `FFI::Platypus::Lang::ASM` - Documentation and tools for using Platypus with the Assembly
- `FFI::Platypus::Lang::C` - Documentation and tools for using Platypus with the C programming language
- `FFI::Platypus::Lang::Win32` - Documentation and tools for using Platypus with the Windows API

FFI::Platypus::Lang::ASM

Documentation and tools for using Platypus with the Assembly

VERSION

version 2.12

SYNOPSIS

```
use FFI::Platypus 2.00;
my $ffi = FFI::Platypus->new( api => 2 );
$ffi->lang('ASM');
```

DESCRIPTION

Setting your lang to `ASM` includes no native type aliases, so types like `int` or `unsigned long` will not work. You need to specify instead `sint32` or `sint64`. Although intended for use with Assembly it could also be used for other languages if you did not want to use the normal C aliases for native types.

This document will one day include information on bundling Assembly with your Perl / FFI / Platypus distribution. Pull requests welcome!

METHODS

native_type_map

```
my $hashref = FFI::Platypus::Lang::ASM->native_type_map;
```

This returns an empty hash reference. For other languages it returns a hash reference that defines the aliases for the types normally used for that language.

SEE ALSO

- *FFI::Platypus*

  The Core Platypus documentation.

- *FFI::Platypus::Lang*

  Includes a list of other language plugins for Platypus.

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::Platypus::Lang::C

Documentation and tools for using Platypus with the C programming language

VERSION

version 2.12

SYNOPSIS

```
use FFI::Platypus 2.00;
my $ffi = FFI::Platypus->new( api => 2 );
$ffi->lang('C'); # the default
```

DESCRIPTION

This module provides some hooks for Platypus to interact with the C programming language. It is generally used by default if you do not specify another foreign programming language with the *FFI::Platypus#lang* attribute.

METHODS

native_type_map

```
my $hashref = FFI::Platypus::Lang::C->native_type_map;
```

This returns a hash reference containing the native aliases for the C programming languages. That is the keys are native C types and the values are libffi native types.

SEE ALSO

- *FFI::Platypus*

  The Core Platypus documentation.

- *FFI::Platypus::Lang*

  Includes a list of other language plugins for Platypus.

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::Platypus::Lang::Win32

Documentation and tools for using Platypus with the Windows API

VERSION

version 2.12

SYNOPSIS

```
use utf8;
use FFI::Platypus 2.00;

my $ffi = FFI::Platypus->new(
  api  => 2,
  lib  => [undef],
);

# load this plugin
$ffi->lang('Win32');

# Pass two double word integer values to the Windows API Beep␣
 ↪function.
$ffi->attach( Beep => ['DWORD','DWORD'] => 'BOOL');
```

(continues on next page)

(continued from previous page)

```
Beep(262, 300);

# Send a Unicode string to the Windows API MessageBoxW function.
use constant MB_OK                   => 0x00000000;
use constant MB_DEFAULT_DESKTOP_ONLY => 0x00020000;
$ffi->attach( [MessageBoxW => 'MessageBox'] => [ 'HWND', 'LPCWSTR',
 ↪'LPCWSTR', 'UINT'] => 'int' );
MessageBox(undef, "I ❤ Platypus", "Confession", MB_OK|MB_DEFAULT_
 ↪DESKTOP_ONLY);

# Get a Unicode string from the Windows API GetCurrentDirectoryW␣
 ↪function.
$ffi->attach( [GetCurrentDirectoryW => 'GetCurrentDirectory'] => [
 ↪'DWORD', 'LPWSTR'] => 'DWORD');
my $buf_size = GetCurrentDirectory(0,undef);
my $dir = "\0\0" x $buf_size;
GetCurrentDirectory($buf_size, \$dir) or die $^E;
print "$dir\n";
```

DESCRIPTION

This module provides the Windows datatypes used by the Windows API. This means that you can use things like `DWORD` as an alias for `uint32`. The full list of type aliases is not documented here as it may change over time or be dynamic. You can get the list for your current environment with this one-liner:

```
perl -MFFI::Platypus::Lang::Win32 -E "say for sort keys %{␣
 ↪FFI::Platypus::Lang::Win32->native_type_map }"
```

This plugin will also set the correct ABI for use with Win32 API functions. (On 32 bit systems a different ABI is used for Win32 API than what is used by the C library, on 32 bit systems the same ABI is used). Most of the time this exactly what you want, but if you need to use functions that are using the standard C calling convention, but need the Win32 types, you can do that by setting the ABI back immediately after loading the language plugin:

```
$ffi->lang('Win32');
$ffi->abi('default_abi');
```

Most of the types should be pretty self-explanatory or at least provided in the Microsoft documentation on the internet, but the use of Unicode strings probably requires some more detail:

[version 1.35]

This plugin also provides `LPCWSTR` and `LPWSTR` "wide" string types which are implemented using *FFI::Platypus::Type::WideString*. For full details, please see the documentation for that module, and note that `LPCWSTR` is a wide string in the read-only string mode and `LPWSTR` is a wide string in the read-write buffer mode.

The `LPCWSTR` is handled fairly transparently by the plugin, but for when using read-write

buffers (LPWSTR) with the Win32 API you typically need to allocate a buffer string of the right size. These examples will use GetCurrentDirectoryW attached as GetCurrentDirectory as in the synopsis above. These are illustrative only, you would normally want to use the *Cwd* module to get the current working directory.

- default buffer size 2048

  The simplest way is to fallback on the rather arbitrary default buffer size of 2048.

```
my $dir;
GetCurrentDirectory(1024, \$dir);
print "I am in the directory: $dir\n";
```

  Discussion: This only works if you know the API that you are using will not ever use more than 2048 bytes. The author believes this to be the case for GetCurrentDirectoryW since directory paths in windows have a maximum of 260 characters. If every character was outside the Basic Multilingual Plane (BMP) they would take up exactly 4 characters each. (This is probably not ever the case since the disk volume at least will be a Latin letter). Taking account of the NULL termination you would need 260 * 4 + 2 bytes or 1048 bytes.

  We pass in a reference to our scalar so that the Win32 API can write into it.

  We are passing in half the number of bytes as the first argument because the API expects the number of WCHAR (or wchar_t), not the number of bytes or the technically the number of characters since characters can take up either 2 or 4 bytes in UTF-16.

- allocate your buffer to your own size.

  If possible it is of course always best to allocate exactly the size of buffer that you need.

```
my $size = GetCurrentDirectory(0, undef);
my $dir = "\0\0" x $size;
GetCurrentDirectory($size, \$dir);
print "I am in the directory: $dir\n";
```

  Discussion: In this case the API provides a way of getting the exact size of buffer that you need. We allocate this in Perl by creating a string of NULLs of the right length. The Perl string "\0" is exactly on byte, so we double that before using the x operator to multiple that by the size returned by the API.

  Now, somewhat unexpectedly what is returned is not the same buffer, but a new string in new UTF-8 encoded Perl string. This is what you want most of the time.

- initialize your read-write buffer

  Some APIs might be modifying an existing string rather than just writing an entirely new one. In that case you still want to allocate a buffer, but you want to initialize it with a value. You can do this by passing an array reference instead of a scalar reference. The firs element of the array is the buffer, and the second is the initialization.

```
my $dir;
GetCurrentDirectory($size, [ \$dir, "I ❤ Perl + Platypus" ]);
```

Discussion: Note that this particular API ignores the string passed in and writes over it, but this demonstrates how you would initialize a buffer string. Once again, if `$dir` is not initialized (is `undef`), then a buffer of the default size of 2048 bytes will be created internally. You can also allocate a specific number of bytes as in the previous example.

- allocate memory using `malloc` etc.

  You can also allocate memory using `malloc` (see *FFI::Platypus::Memory*) and encode your string using *Encode* and copy it using `wcscpy`. This may be appropriate in some cases, but it is beyond the scope of this document.

METHODS

abi

```
my $abi = FFI::Platypus::Lang::Win32->abi;
```

This is called internally when the type plugin is loaded by Platypus. It selects the appropriate ABI to make Win32 API function calls.

native_type_map

```
my $hashref = FFI::Platypus::Lang::Win32->native_type_map;
```

This is called internally when the type plugin is loaded by Platypus. It provides types aliases useful on the Windows platform, so it may also be useful for introspection.

This returns a hash reference containing the native aliases for the Windows API. That is the keys are native Windows API C types and the values are libffi native types.

This will includes types like `DWORD` and `HWND`, and others. The full list may be adjusted over time and may be computed dynamically. To get the full list for your install you can use this one-liner:

```
perl -MFFI::Platypus::Lang::Win32 -E "say for sort keys %{␣
↪FFI::Platypus::Lang::Win32->native_type_map }"
```

load_custom_types

```
FFI::Platypus::Lang::Win32->load_custom_types($ffi);
```

This is called internally when the type plugin is loaded by Platypus. It provides custom types useful on the Windows platform. For now that means the `LPWSTR` and `LPCWSTR` types.

CAVEATS

The Win32 API isn't a different computer language in the same sense that the other language plugins (those for Fortran or Rust for example). But implementing these types as a language plugin is the most convenient way to do it.

Prior to version 1.35, this plugin didn't provide an implementation for `LPWSTR` or `LPCWSTR`, so in the likely event that you need those types make sure you also require at least that version of Platypus.

SEE ALSO

- *FFI::Platypus*

  The Core Platypus documentation.

- *FFI::Platypus::Lang*

  Includes a list of other language plugins for Platypus.

- *FFI::Platypus::Type::WideString*

  The wide string type plugin use for `LPWSTR` and `LPCWSTR` types.

- Win32::API

  Another FFI, but for Windows only.

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::Platypus::Legacy

Legacy Platypus interfaces

VERSION

version 2.12

DESCRIPTION

This class is private to *FFI::Platypus*.

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::Platypus::Memory

Memory functions for FFI

VERSION

version 2.12

SYNOPSIS

```
use FFI::Platypus::Memory;

# allocate 64 bytes of memory using the
# libc malloc function.
my $pointer = malloc 64;

# use that memory wisely
...

# free the memory when you are done.
free $pointer;
```

DESCRIPTION

This module provides an interface to common memory functions provided by the standard C library. They may be useful when constructing interfaces to C libraries with FFI. It works mostly with the opaque type and it is worth reviewing the section on opaque pointers in *FFI::Platypus::Type*.

Allocating memory and forgetting to free it is a common source of memory leaks in C and when using this module. Very recent Perls have a `defer` keyword that lets you automatically call functions like `free` when a block ends. This can be especially handy when you have multiple code paths or possible exceptions to keep track of.

```
use feature 'defer';
use FFI::Platypus::Memory qw( malloc free );
```

(continues on next page)

(continued from previous page)

```
sub run {
  my $ptr = malloc 66;
  defer { free $ptr };

  my $data = do_something($ptr);

  # do not need to remember to place free $ptr here, as it will
  # run through defer.

  return $data;
}
```

If you are not lucky enough to have the `defer` feature in your version of Perl you may be able to use Feature::Compat::Defer, which will use the feature if available, and provides its own mostly compatible version if not.

FUNCTIONS

calloc

```
my $pointer = calloc $count, $size;
```

The `calloc` function contiguously allocates enough space for *$count* objects that are *$size* bytes of memory each.

free

```
free $pointer;
```

The `free` function frees the memory allocated by `malloc`, `calloc`, `realloc` or `strdup`. It is important to only free memory that you yourself have allocated. A good way to crash your program is to try and free a pointer that some C library has returned to you.

malloc

```
my $pointer = malloc $size;
```

The `malloc` function allocates *$size* bytes of memory.

memcpy

```
memcpy $dst_pointer, $src_pointer, $size;
```

The `memcpy` function copies *$size* bytes from *$src_pointer* to *$dst_pointer*. It also returns *$dst_pointer*.

memset

```
memset $buffer, $value, $length;
```

The `memset` function writes *$length* bytes of *$value* to the address specified by *$buffer*.

realloc

```
my $new_pointer = realloc $old_pointer, $size;
```

The `realloc` function reallocates enough memory to fit *$size* bytes. It copies the existing data and frees *$old_pointer*.

If you pass `undef` in as *$old_pointer*, then it behaves exactly like `malloc`:

```
my $pointer = realloc undef, 64; # same as malloc 64
```

strcpy

```
strcpy $opaque, $string;
```

Copies the string to the memory location pointed to by `$opaque`.

strdup

```
my $pointer = strdup $string;
```

The `strdup` function allocates enough memory to contain *$string* and then copies it to that newly allocated memory. This version of `strdup` returns an opaque pointer type, not a string type. This may seem a little strange, but returning a string type would not be very useful in Perl.

strndup

```
my $pointer = strndup $string, $max;
```

The same as `strdup` above, except at most `$max` characters will be copied in the new string.

SEE ALSO

- *FFI::Platypus*

  Main Platypus documentation.

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::Platypus::Record

FFI support for structured records data

VERSION

version 2.12

SYNOPSIS

C:

```
struct my_person {
  int         age;
  const char  title[3];
  const char *name
};

void process_person(struct my_person *person)
{
  /* ... */
}
```

Perl:

```
package MyPerson;

use FFI::Platypus::Record;

record_layout_1(
  'int'       => 'age',
  'string(3)' => 'title',
  'string rw' => 'name',
);

package main;

use FFI::Platypus 2.00;

my $ffi = FFI::Platypus->new( api => 2 );
$ffi->lib("myperson.so");
$ffi->type("record(MyPerson)" => 'MyPerson');

my $person = MyPerson->new(
  age   => 40,
  title => "Mr.",
  name  => "John Smith",
);

$ffi->attach( process_person => [ 'MyPerson*' ] => 'void' );

process_person($person);

$person->age($person->age + 1); # another year older

process_person($person);
```

DESCRIPTION

[version 0.21]

This module provides a mechanism for building classes that can be used to mange structured data records (known as C as “structs” and in some languages as “records”). A

structured record is a series of bytes that have structure understood by the C or other foreign language library that you are interfacing with. It is designed for use with FFI and *FFI::Platypus*, though it may have other applications.

Before you get to deep into using this class you should also consider the FFI::C, which provides some overlapping functionality. Briefly, it comes down to this:

(The tl;dr is: use this class when you need to pass by value (since FFI::C does not support pass by value) and use FFI::C in all other circumstances).

- *FFI::Platypus::Record*

  Supports:

  - C pointers to `struct` types
  - Passing C `structs` by-value.

  Does not support:

  - C `union` types.
  - C arrays of `struct` and `union` types.

- FFI::C

  Supports:

  - C `struct` and`union` types
  - C arrays of `struct` and `union` types.

  Does not support:

  - Passing C `structs` by-value.

  String members are as of this writing a TODO for FFI::C, but should be coming soon!

FUNCTIONS

record_layout_1

```
record_layout_1($ffi, $type => $name, ... );
record_layout_1(\@ffi_args, $type => $name, ... );
record_layout_1($type => $name, ... );
```

Define the layout of the record. You may optionally provide an instance of *FFI::Platypus* as the first argument in order to use its type aliases. Alternatively you may provide constructor arguments that will be passed to the internal platypus instance. Thus this is the same:

```
my $ffi = FFI::Platypus->new( lang => 'Rust', api => 2 );
record_layout_1( $ffi, ... );
# same as:
record_layout_1( [ lang => 'Rust' ], ... );
```

and this is the same:

```
my $ffi = FFI::Platypus->new( api => 2 );
record_layout_1( $ffi, ... );
# same as:
record_layout_1( ... );
```

Then you provide members as type/name pairs.

For each member you declare, `record_layout_1` will create an accessor which can be used to read and write its value. For example imagine a class `Foo`:

```
package Foo;

use FFI::Platypus::Record;

record_layout_1(
  int          => 'bar',  #  int bar;
  'string(10)' => 'baz',  #  char baz[10];
);
```

You can get and set its fields with like named `bar` and `baz` accessors:

```
my $foo = Foo->new;

$foo->bar(22);
my $value = $foo->bar;

$foo->baz("grimlock\0\0"); # should be 10 characters long
my $string_value = $foo->baz; # includes the trailing \0\0
```

You can also pass initial values in to the constructor, either passing as a list of key value pairs or by passing a hash reference:

```
$foo = Foo->new(
  bar => 22,
  baz => "grimlock\0\0",
);

# same as:

$foo = Foo->new( {
  bar => 22,
  baz => "grimlock\0\0",
} );
```

If there are members of a record that you need to account for in terms of size and alignment, but do not want to have an accessor for, you can use `:` as a place holder for its name:

```
record_layout_1(
  'int'        => ':',
  'string(10)' => 'baz',
);
```

strings

So far I've shown fixed length strings. These are declared with the word `string` followed by the length of the string in parentheticals. Fixed length strings are included inside the record itself and do not need to be allocated or deallocated separately from the record. Variable length strings must be allocated on the heap, and thus require a sense of "ownership", that is whomever allocates variable length strings should be responsible for also free'ing them. To handle this, you can add a `ro` or `rw` trait to a string field. The default is `ro`, means that you can get, but not set its value:

```
package Foo;

record_layout_1(
  'string ro' => 'bar',  # same type as 'string' and 'string_ro'
);

package main;

my $foo = Foo->new;

my $string = $foo->bar;  # GOOD
$foo->bar("starscream"); # BAD
```

If you specify a field is `rw`, then you can set its value:

```
package Foo;

record_layout_1(
  'string rw' => 'bar',  # same type as 'string_rw'
);

package main;

my $foo = Foo->new;

my $string = $foo->bar;  # GOOD
$foo->bar("starscream"); # GOOD
```

Any string value that is pointed to by the record will be free'd when it falls out of scope, so you must be very careful that any `string rw` fields are not set or modified by C code. You should also take care not to copy any record that has a `rw` string in it because its values will be free'd twice!

```
use Clone qw( clone );

my $foo2 = clone $foo;  # BAD  bar will be free'd twice
```

arrays

Arrays of integer, floating points and opaque pointers are supported.

```
package Foo;

record_layout_1(
  'int[10]' => 'bar',
);

my $foo = Foo->new;

$foo->bar([1,2,3,4,5,6,7,8,9,10]); # sets the values for the array
my $list = $foo->bar;  # returns a list reference

$foo->bar(5, -6); # sets the 5th element in the array to -6
my $item = $foo->bar(5); gets the 5th element in the array
```

record_layout

```
record_layout($ffi, $type => $name, ... );
record_layout(\@ffi_args, $type => $name, ... );
record_layout($type => $name, ... );
```

This function works like `record_layout` except that `api => 0` is used instead of `api => 1`. All new code should use `record_layout_1` instead.

CAVEATS

These useful features (and probably more) are missing, and unlikely to be added.

- Unions
- Nested records

If you need these features, consider using FFI::C instead.

SEE ALSO

- *FFI::Platypus*

  The main platypus documentation.

- FFI::C

  Another interface for constructing structured data. It includes support for `union` and array types (which this module does not), but lacks support for passing records by-value.

- *FFI::Platypus::Record::TieArray*

  Tied array interface for record array members.

- Convert::Binary::C

  Another method for constructing and dissecting structured data records.

- pack and unpack

  Built-in Perl functions for constructing and dissecting structured data records.

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

Modules

- `FFI::Platypus::Record::Meta` - FFI support for structured records data
- `FFI::Platypus::Record::TieArray` - Tied array interface for record array members

FFI::Platypus::Record::Meta

FFI support for structured records data

VERSION

version 2.12

DESCRIPTION

This class is private to FFI::Platypus. See *FFI::Platypus::Record* for the public interface to Platypus records.

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::Platypus::Record::TieArray

Tied array interface for record array members

VERSION

version 2.12

SYNOPSIS

```
package Foo;

use FFI::Platypus::Record;
use FFI::Platypus::Record::TieArray;

record_layout(qw(
  int[20]  _bar
));

sub bar
{
  my($self, $arg) = @_;
  $self->_bar($arg) if ref($arg) eq ' ARRAY';
  tie my @list, 'FFI::Platypus::Record::TieArray',
    $self, '_bar', 20;
}

package main;

my $foo = Foo->new;

my $bar5 = $foo->bar->[5];  # get the 5th element of the bar array
$foo->bar->[5] = 10;        # set the 5th element of the bar array
@{ $foo->bar } = ();        # set all elements in bar to 0
@{ $foo->bar } = (1..5);    # set the first five elements of the bar
 ↪array
```

DESCRIPTION

WARNING: This module is considered EXPERIMENTAL. It may go away or be changed in incompatible ways, possibly without notice, but not without a good reason.

This class provides a tie interface for record array members.

In the future a short cut for using this with *FFI::Platypus::Record* directly may be provided.

SEE ALSO

- *FFI::Platypus*

  The main Platypus documentation.

- *FFI::Platypus::Record*

  Documentation on Platypus records.

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::Platypus::ShareConfig

FFI::Platypus::ShareConfig

VERSION

version 2.12

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::Platypus::Type

Defining types for FFI::Platypus

VERSION

version 2.12

SYNOPSIS

OO Interface:

```
use FFI::Platypus 2.00;
my $ffi = FFI::Platypus->new( api => 2 );
$ffi->type('int' => 'my_int');
```

DESCRIPTION

Note: This document assumes that you are using `api => 2`, which you should be using for all new code.

This document describes how to define types using *FFI::Platypus*. Types may be "defined" ahead of time, or simply used when defining or attaching functions.

```
# Example of defining types
use FFI::Platypus 2.00;
my $ffi = FFI::Platypus->new( api => 2 );
$ffi->type('int');
$ffi->type('string');

# Example of simply using types in function declaration or␣
 ↪attachment
my $f = $ffi->function(puts => ['string'] => 'int');
$ffi->attach(puts => ['string'] => 'int');
```

Unless you are using aliases the *FFI::Platypus#type* method is not necessary, but they will throw an exception if the type is incorrectly specified or not supported, which may be helpful for determining if the types are available or not.

Note: This document sometimes uses the term "C Function" as short hand for function implemented in a compiled language. Unless the term is referring literally to a C function example code, you can assume that it should also work with another compiled language.

meta information about types

You can get the size of a type using the *FFI::Platypus#sizeof* method.

```
my $intsize = $ffi->sizeof('int');           # usually 4
my $intarraysize = $ffi->sizeof('int[64]');  # usually 256
```

converting types

Sometimes it is necessary to convert types. In particular various pointer types often need to be converted for consumption in Perl. For this purpose the *FFI::Platypus#cast* method is provided. It needs to be used with care though, because not all type combinations are supported. Here are some useful ones:

```
my $address = $ffi->cast('string' => 'opaque', $string);
```

This converts a Perl string to a pointer address that can be used by functions that take an opaque type. Be carefully though that the Perl string is not resized or free'd while in use from C code.

```
my $string  = $ffi->cast('opaque' => 'string', $pointer);
```

This does the opposite, converting a null terminated string (the type of strings used by C) into a Perl string. In this case the string is copied, so the other language is free to deallocate or otherwise manipulate the string after the conversion without adversely affecting the Perl.

aliases

Some times using alternate names is useful for documenting the purpose of an argument or return type. For this "aliases" can be helpful. The second argument to the *FFI::Platypus#type* method can be used to define a type alias that can later be used by function declaration and attachment.

```
use FFI::Platypus 2.00;
my $ffi = FFI::Platypus->new( api => 2 );
$ffi->type('int'    => 'myint');
$ffi->type('string' => 'mystring');
my $f = $ffi->function( puts => ['mystring'] => 'myint' );
$ffi->attach( puts => ['mystring'] => 'myint' );
```

Aliases are contained without the *FFI::Platypus* object, so feel free to define your own crazy types without stepping on the toes of other CPAN developers using Platypus.

One useful application of an alias is when you know types are different on two different platforms:

```
if($^O eq 'MSWin32')
{
  $type->type('sint16' => 'foo_t');
} elsif($^O eq 'linux')
```

(continues on next page)

(continued from previous page)

```
{
  $type->type('sint32' => 'foo_t');
}

# function foo takes 16 bit signed integer on Windows
# and a 32 bit signed integer on Linux.
$ffi->attach( foo => [ 'foo_t' ] => 'void' );
```

TYPE CATEGORIES

Native types

So called native types are the types that the CPU understands that can be passed on the argument stack or returned by a function. It does not include more complicated types like arrays or structs, which can be passed via pointers (see the opaque type below). Generally native types include void, integers, floats and pointers.

the void type

This can be used as a return value to indicate a function does not return a value (or if you want the return value to be ignored).

```
$ffi->type( foo => [] => 'void' );
```

Newer versions of Platypus also allow you to omit the return type and `void` is assumed.

```
$ffi->type( foo => [] );
```

It doesn't really make sense to use `void` in any other context. However, because of historical reasons involving older versions of Perl.

It doesn't really make sense for `void` to be passed in as an argument. However, because C functions that take no arguments frequently are specified as taking `void` as this was required by older C compilers, as a special case you can specify a function's arguments as taking a single `void` to mean it takes no arguments.

```
# C: void foo(void);
$ffi->type( foo => ['void'] );
# same (but probably better)
$ffi->type( foo => [] );
```

integer types

The following native integer types are always available (parentheticals indicates the usual corresponding C type):

- sint8

  Signed 8 bit byte (`signed char`, `int8_t`).

- uint8

Unsigned 8 bit byte (`unsigned char`, `uint8_t`).

- sint16

  Signed 16 bit integer (`short`, `int16_t`)

- uint16

  Unsigned 16 bit integer (`unsigned short`, `uint16_t`)

- sint32

  Signed 32 bit integer (`int`, `int32_t`)

- uint32

  Unsigned 32 bit integer (`unsigned int`, `uint32_t`)

- sint64

  Signed 64 bit integer (`long long`, `int64_t`)

- uint64

  Unsigned 64 bit integer (`unsigned long long`, `uint64_t`)

You may also use `uchar`, `ushort`, `uint` and `ulong` as short names for `unsigned char`, `unsigned short`, `unsigned int` and `unsigned long`.

These integer types are also available, but there actual size and sign may depend on the platform.

- char

  Somewhat confusingly, `char` is an integer type! This is really an alias for either `sint8_t` or `uint8_t` depending on your platform. If you want to pass a character (not integer) in to a C function that takes a character you want to use the perl *ord* function. Here is an example that uses the standard libc `isalpha`, `isdigit` type functions:

```
use FFI::Platypus 2.00;

my $ffi = FFI::Platypus->new( api => 2 );
$ffi->lib(undef);
$ffi->type('int' => 'character');

my @list = qw(
  alnum alpha ascii blank cntrl digit lower print punct
  space upper xdigit
);

$ffi->attach("is$_" => ['character'] => 'int') for @list;

my $char = shift(@ARGV) || 'a';

no strict 'refs';
printf "'%s' is %s %s\n", $char, $_, &{'is'.$_}(ord $char) for␣
↪@list;
```

- size_t

  This is usually an `unsigned long`, but it is up to the compiler to decide. The `malloc` function is defined in terms of `size_t`:

  ```
  $ffi->attach( malloc => ['size_t'] => 'opaque';
  ```

  (Note that you can get `malloc` from *FFI::Platypus::Memory*).

- long, unsigned long

  On 64 bit systems, this is usually a 64 bit integer. On 32 bit systems this is frequently a 32 bit integer (and `long long` or `unsigned long long` are for 64 bit).

There are a number of other types that may or may not be available if they are detected when *FFI::Platypus* is installed. This includes things like `wchar_t`, `off_t`, `wint_t`. You can use this script to list all the integer types that *FFI::Platypus* knows about, plus how they are implemented.

```
use FFI::Platypus 2.00;

my $ffi = FFI::Platypus->new( api => 2 );

foreach my $type_name (sort $ffi->types)
{
  my $meta = $ffi->type_meta($type_name);
  next unless defined $meta->{element_type} && $meta->{element_type}
 ↪eq 'int';
  printf "%20s %s\n", $type_name, $meta->{ffi_type};
}
```

If you need a common system type that is not provided, please open a ticket in the Platypus project's GitHub issue tracker. Be sure to include the usual header file the type can be found in.

Enum types

C provides enumerated types, which are typically implemented as integer types.

```
enum {
  BAR = 1,
  BAZ = 2
} foo_t;

void f(enum foo_t foo);
```

Platypus provides `enum` and `senum` types for the integer types used to represent enum and signed enum types respectively.

```
use constant BAR => 1;
use constant BAZ => 2;
$ffi->attach( f => [ 'enum' ] => 'void' );
f(BAR);
f(BAZ);
```

When do you use `senum`? Anytime the enum has negative values:

```
enum {
  BAR = -1;
  BAZ = 2;
} foo_t;

void f(enum foo_t foo);
```

Perl:

```
use constant BAR => -1;
use constant BAZ => 2;
$ffi->attach( f => [ 'senum' ] => 'void' );
f(BAR);
f(BAZ);
```

Dealing with enumerated values with FFI can be tricky because these are usually defined in C header files and cannot be found in dynamic libraries. For trivial usage you can do as illustrated above, simply define your own Perl constants. For more complicated usage, or where the values might vary from platform to platform you may want to consider the new Platypus bundle interface to define Perl constants (essentially the same as an enumerated value) from C space. This is more reliable, but does require a compiler at install time. See *FFI::Platypus::Constant* for details.

The main FAQ (*"FAQ" in FFI::Platypus*) also has a discussion on dealing with constants and enumerated types.

There is also a type plugin (FFI::Platypus::Type::Enum) that can be helpful in writing interfaces that use enums.

Boolean types

At install time Platypus attempts to detect the correct type for `bool` for your platform, and you can use that. `bool` is really an integer type, but the type used varies from platform to platform.

C header:

```
#include <stdbool.h>
bool foo();
```

Platypus

```
$ffi->attach( foo => [] => 'bool' );
```

If you get an exception when trying to use this type it means you either have a very old version of Platypus, or for some reason it was unable to detect the correct type at install time. Please open a ticket if that is the case.

floating point types

The following native floating point types are always available (parentheticals indicates the usual corresponding C type):

- float

  Single precision floating point (*float*)

- double

  Double precision floating point (*double*)

- longdouble

  Floating point that may be larger than `double` (*longdouble*). This type is only available if supported by the C compiler used to build *FFI::Platypus*. There may be a performance penalty for using this type, even if your Perl uses long doubles internally for its number value (NV) type, because of the way *FFI::Platypus* interacts with `libffi`.

  As an argument type either regular number values (NV) or instances of Math::LongDouble are accepted. When used as a return type, Math::LongDouble will be used, if you have that module installed. Otherwise the return type will be downgraded to whatever your Perl's number value (NV) is.

- complex_float

  Complex single precision floating point (*float complex*)

- complex_double

  Complex double precision floating point (*double complex*)

  `complex_float` and `complex_double` are only available if supported by your C compiler and by libffi. Complex numbers are only supported in very recent versions of libffi, and as of this writing the latest production version doesn't work on x86_64. It does seem to work with the latest production version of libffi on 32 bit Intel (x86), and with the latest libffi version in git on x86_64.

opaque pointers

Opaque pointers are simply a pointer to a region of memory that you do not manage, and do not know or care about its structure. It is like a `void *` in C. These types are represented in Perl space as integers and get converted to and from pointers by *FFI::Platypus*. You may use `pointer` as an alias for `opaque`, although this is discouraged. (The Platypus documentation uses the convention of using "pointer" to refer to pointers to known types (see below) and "opaque" as short hand for opaque pointer).

As an example, libarchive defines `struct archive` type in its header files, but does not define its content. Internally it is defined as a `struct` type, but the caller does not see this. It is therefore opaque to its caller. There are `archive_read_new` and `archive_write_new` functions to create a new instance of this opaque object and `archive_read_free` and `archive_write_free` to destroy this objects when you are done.

C header:

```
struct archive;
struct archive *archive_read_new(void);
struct archive *archive_write_new(void);
int archive_free(struct archive *);
int archive_write_free(struct archive *);
```

Perl code:

```
$lib->find_lib( lib => 'archive' );
$ffi->attach(archive_read_new   => []         => 'opaque');
$ffi->attach(archive_write_new  => []         => 'opaque');
$ffi->attach(archive_read_free  => ['opaque'] => 'int');
$ffi->attach(archive_write_free => ['opaque'] => 'int');
```

It is often useful to alias an opaque type like this so that you know what the object represents:

```
$lib->find_lib( lib => 'archive' );
$ffi->type('opaque' => 'archive');
$ffi->attach(archive_read_new   => [] => 'archive');
$ffi->attach(archive_read_free  => ['archive'] => 'int');
...
```

As a special case, when you pass `undef` into a function that takes an opaque type it will be translated into `NULL` for C. When a C function returns a NULL pointer, it will be translated back to `undef`.

For functions that take a pointer to a void pointer (that is a `void **`), you can use a pointer to an opaque type. Consider the C code:

```
struct archive_entry;
int archive_read_next_header(struct archive *, struvct archive_entry
↪**);
```

Once again the internals of `archive_entry` are not provided. Perl code:

```
$ffi->type('opaque' => 'archive_entry');
$ffi->attach(archive_read_next_header => [ 'archive', 'archive_
↪entry*' ] => 'int');
```

Now we can call this function

```
my $archive = archive_read_new();
...  # additional prep for $active is required
while(1) {
  my $entry;
  archive_read_next_header($archive, \$entry);
  last unless defined $entry;
  # can now use $entry for other archive_entry_ methods.
}
```

The way `archive_read_next_header` works, it will return a pointer to the next

`archive_entry` object until it gets to the end, when it will return a pointer to `NULL` which will be represented in Perl by a `undef`.

There are a number of useful utility functions for dealing with opaque types in the *FFI::Platypus::Memory* module.

Objects

Object types are thin wrappers around two native types: integer and opaque types. They are just blessed references around either of those two types so that methods can be defined on them, but when they get passed to a Platypus xsub they are converted into the native integer or opaque types. This type is most useful when a API provides an OO style interface with an integer or opaque value acting as an instance of a class. There are two detailed examples in the main Platypus documentation using libarchive and unix open:

- *"libarchive" in FFI::Platypus*
- *"unix open" in FFI::Platypus*

Strings

```
# used when you need a char * or const char *
$ffi->attach( puts => [ 'string' ] => 'int' );
```

The `string` type is a series of bytes that usually represent a series of characters. They will be NULL terminated for C and passed in as a pointer. This will typically work for APIs that take ASCII or UTF-8 strings which are common in Unix environments.

(Note if you need to handle the native "wide" string for example if you need to talk UTF-16 on Windows see *FFI::Platypus::Type::WideString*).

(Note if you need to pass in a fixed length string by value (not as a pointer) then you can do so using *FFI::Platypus::Record*).

(Note that languages like Go and Rust do not use NULL terminated strings and need their own string types; see the appropriate language plugins for details)

```
# can also be used when you need a void * or const void *
$ffi->attach( write => ['int', 'string', 'size_t' ] => 'ssizet' );
```

The `string` type can also be used to pass in the start of a buffer of arbitrary bytes stored in a Perl scalar. Because a `string` is passed just as a pointer you will typically need to also pass the length of the buffer as a separate argument. This is necessary because buffers could potentially have a NULL in them.

The pointer passed into C (or other language) is to the content of the actual scalar, which means it can modify the content of a scalar.

NOTE: When used as a return type, the string is *copied* into a new scalar rather than using the original address. This is due to the ownership model of scalars in Perl, but it is also most of the time what you want.

This can be problematic when a function returns a string that the callee is expected to free. Consider the functions:

```
char *
get_string()
{
  char *buffer;
  buffer = malloc(20);
  strcpy(buffer, "Perl");
}

void
free_string(char *buffer)
{
  free(buffer);
}
```

This API returns a string that you are expected to free when you are done with it. (At least they have provided an API for freeing the string instead of expecting you to call libc free)! A simple binding to get the string would be:

```
$ffi->attach( get_string => [] => 'string' );  # memory leak
my $str = get_string();
```

Which will work to a point, but the memory allocated by get_string will leak. Instead you need to get the opaque pointer, cast it to a string and then free it.

```
$ffi->attach( get_string => [] => 'opaque' );
$ffi->attach( free_string => ['opaque'] => 'void' );
my $ptr = get_string();
my $str = $ffi->cast( 'opaque' => 'string', $ptr );  # copies the␣
↪string
free_string($ptr);
```

If you are doing this sort of thing a lot, it can be worth adding a custom type:

```
$ffi->attach( free_string => ['opaque'] => 'void' );
$ffi->custom_type( 'my_string' => {
  native_type => 'opaque',
  native_to_perl => sub {
    my($ptr) = @_;
    my $str = $ffi->cast( 'opaque' => 'string', $ptr ); # copies the␣
↪string
    free_string($ptr);
    $str;
  }
});

$ffi->attach( get_string => [] => 'my_string' );
my $str = get_string();
```

Since version 0.62, pointers and arrays to strings are supported as a first class type. Prior to that *FFI::Platypus::Type::StringArray* and *FFI::Platypus::Type::StringPointer* could be used, though their use in new code is discouraged.

```
$ffi->attach( foo => ['string[]'] => 'void' );
foo( [ 'array', 'of', 'strings' ] );

$ffi->attach( bar => ['string*'] => 'void' );
my $string = 'baz';
bar( \$string );  # $string may be modified.
```

Strings are not allowed as return types from closure. This, again is due to the ownership model of scalars in Perl. (There is no way for Perl to know when calling language is done with the memory allocated to the string). Consider the API:

```
typedef const char *(*get_message_t)(void);

void
print_message(get_message_t get_message)
{
  const char *str;
  str = get_message();
  printf("message = %s\n", str);
}
```

It feels like this should be able to work:

```
$ffi->type('()->string' => 'get_message_t'); # not ok
$ffi->attach( print_message => ['get_message_t'] => 'void' );
my $get_message = $ffi->closure(sub {
  return "my message";
});
print_message($get_message);
```

If the type declaration for `get_message_t` were legal, then this script would likely segfault or in the very least corrupt memory. The problem is that once `"my message"` is returned from the closure Perl doesn't have a reference to it anymore and will free it. To do this safely, you have to keep a reference to the scalar around and return an opaque pointer to the string using a cast.

```
$ffi->type('()->opaque' => 'get_message_t');
$ffi->attach( print_message => ['get_message_t'] => 'void' );
my $get_message => $ffi->closure(sub {
  our $message = "my message";  # needs to be our so that it doesn't
                                # get free'd
  my $ptr = $ffi->cast('string' => 'opaque', $message);
  return $ptr;
});
print_message($get_message);
```

Another type of string that you may run into with some APIs is the so called "wide" string. In your C code if you see `wchar_t*` or `const wchar_t*` or if in Win32 API code you see `LPWSTR` or `LPCWSTR`. Most commonly you will see these types when working with the Win32 API, but you may see them in Unix as well. These types are intended for dealing with Unicode, but they do not use the same UTF-8 format used by Perl internally, so they need to be converted. You can do this manually by allocating the memory and using

the *Encode* module, but the easier way is to use either *FFI::Platypus::Type::WideString* or *FFI::Platypus::Lang::Win32*, which handle the memory allocation and conversion for you.

String types can be defined to have a fixed length using a trailing parenthetical like so `string(10)`. For arguments this has little practical effect since the strings are passed as pointers anyway, but does impact return values. If a function that returns a `string(10)` type returns a string that is not NULL terminated, only the first ten bytes will be returned in the result.

Internally fixed length strings are implemented the same as classless record types (that is to say `string(10)` is identically internally to `record(10)*`).

For the 1.00 Platypus API, the `string(10)` type was specified as a pointer (that is `string(10)*`). This was a mistake, but you can still use the latter as an alias for the correct form in the 2.00 API.

Pointers and Arrays of Strings

As of the 1.00 Platypus API, you can specify pointers to strings (`string*`) and arrays of strings (`string[10]`). Since strings themselves are passed as pointers, this means these types are passed in as pointers to pointers. If the pointer to the string is changed then when the function returns the scalar or array will be updated as well.

Pointer / References

In C you can pass a pointer to a variable to a function in order accomplish the task of pass by reference. In Perl the same task is accomplished by passing a reference (although you can also modify the argument stack thus Perl supports proper pass by reference as well).

With *FFI::Platypus* you can define a pointer to any native, string or record type. You cannot (at least not yet) define a pointer to a pointer or a pointer to an array or any other type not otherwise supported. When passing in a pointer to something you must make sure to pass in a reference to a scalar, or `undef` (`undef` will be translated int `NULL`).

If the C code makes a change to the value pointed to by the pointer, the scalar will be updated before returning to Perl space. Example, with C code.

```
/* foo.c */
void increment_int(int *value)
{
  if(value != NULL)
    (*value)++;
  else
    fprintf(stderr, "NULL pointer!\n");
}

# foo.pl
use FFI::Platypus 2.00;
my $ffi = FFI::Platypus->new( api => 2 );
$ffi->lib('libfoo.so'); # change to reflect the dynamic lib
                        # that contains foo.c
$ffi->type('int*' => 'int_p');
```

(continues on next page)

(continued from previous page)

```
$ffi->attach(increment_int => ['int_p'] => 'void');
my $i = 0;
increment_int(\$i);   # $i == 1
increment_int(\$i);   # $i == 2
increment_int(\$i);   # $i == 3
increment_int(undef); # prints "NULL pointer!\n"
```

Older versions of Platypus did not support pointers to strings or records.

Records

Records are structured data of a fixed length. In C they are called `structs`.

For most C structured data, as long as you do not need to a record by value, FFI::C is the better choice. Briefly, FFI::C supports `struct`, `union`, and arrays of `struct` and `unions`. FFI::C does not support passing by value. The reminder of this section will discuss only the `record` type.

To declare a record type, use `record`:

```
$ffi->type( 'record (42)' => 'my_record_of_size_42_bytes' );
```

The easiest way to mange records with Platypus is by using *FFI::Platypus::Record* to define a record layout for a record class. Here is a brief example:

```
package Unix::TimeStruct;

use FFI::Platypus 2.00;
use FFI::Platypus::Record;

record_layout_1(qw(
    int    tm_sec
    int    tm_min
    int    tm_hour
    int    tm_mday
    int    tm_mon
    int    tm_year
    int    tm_wday
    int    tm_yday
    int    tm_isdst
    long   tm_gmtoff
    string tm_zone
));

my $ffi = FFI::Platypus->new( api => 2 );
$ffi->lib(undef);
# define a record class Unix::TimeStruct and alias it to "tm"
$ffi->type("record(Unix::TimeStruct)*" => 'tm');

# attach the C localtime function as a constructor
```

(continues on next page)

(continued from previous page)

```
$ffi->attach( localtime => ['time_t*'] => 'tm', sub {
  my($inner, $class, $time) = @_;
  $time = time unless defined $time;
  $inner->(\$time);
});

package main;

# now we can actually use our Unix::TimeStruct class
my $time = Unix::TimeStruct->localtime;
printf "time is %d:%d:%d %s\n",
  $time->tm_hour,
  $time->tm_min,
  $time->tm_sec,
  $time->tm_zone;
```

For more detailed usage, see *FFI::Platypus::Record*.

Platypus does not manage the structure of a record (that is up to you), it just keeps track of their size and makes sure that they are copied correctly when used as a return type. A record in Perl is just a string of bytes stored as a scalar. In addition to defining a record layout for a record class, there are a number of tools you can use manipulate records in Perl, two notable examples are pack and unpack and Convert::Binary::C.

Here is an example with commentary that uses Convert::Binary::C to extract the component time values from the C `localtime` function, and then smushes them back together to get the original `time_t` (an integer).

```
use Convert::Binary::C;
use FFI::Platypus 2.00;
use Data::Dumper qw( Dumper );

my $c = Convert::Binary::C->new;

# Alignment of zero (0) means use
# the alignment of your CPU
$c->configure( Alignment => 0 );

# parse the tm record structure so
# that Convert::Binary::C knows
# what to spit out and suck in
$c->parse(<<ENDC);
struct tm {
  int tm_sec;
  int tm_min;
  int tm_hour;
  int tm_mday;
  int tm_mon;
  int tm_year;
  int tm_wday;
```

(continues on next page)

(continued from previous page)

```
  int tm_yday;
  int tm_isdst;
  long int tm_gmtoff;
  const char *tm_zone;
};
ENDC

# get the size of tm so that we can give it
# to Platypus
my $tm_size = $c->sizeof("tm");

# create the Platypus instance and create the appropriate
# types and functions
my $ffi = FFI::Platypus->new( api => 2 );
$ffi->lib(undef);
$ffi->type("record($tm_size)*" => 'tm');
$ffi->attach( [ localtime => 'my_localtime' ] => ['time_t*'] => 'tm'
 ↪    );
$ffi->attach( [ time      => 'my_time'      ] => ['tm']      =>
 ↪'time_t' );

# ================================================
# get the tm struct from the C localtime function
# note that we pass in a reference to the value that time
# returns because localtime takes a pointer to time_t
# for some reason.
my $time_hashref = $c->unpack( tm => my_localtime(\time) );

# tm_zone comes back from Convert::Binary::C as an opaque,
# cast it into a string.  We localize it to just this do
# block so that it will be a pointer when we pass it back
# to C land below.
do {
  local $time_hashref->{tm_zone} = $ffi->cast(opaque => string =>
 ↪$time_hashref->{tm_zone});
  print Dumper($time_hashref);
};

# ================================================
# convert the tm struct back into an epoch value
my $time = my_time( $c->pack( tm => $time_hashref ) );

print "time      = $time\n";
print "perl time = ", time, "\n";
```

You can also link a record type to a class. It will then be accepted when blessed into that class as an argument passed into a C function, and when it is returned from a C function it will be blessed into that class. Basically:

```
$ffi->type( 'record(My::Class)*' => 'my_class' );
$ffi->attach( my_function1 => [ 'my_class' ] => 'void' );
$ffi->attach( my_function2 => [ ] => 'my_class' );
```

The only thing that your class MUST provide is either a `ffi_record_size` or `_ffi_record_size` class method that returns the size of the record in bytes.

Contrast a record type which is stored as a scalar string of bytes in Perl to an opaque pointer which is stored as an integer in Perl. Both are treated as pointers in C functions. The situations when you usually want to use a record are when you know ahead of time what the size of the object that you are working with and probably something about its structure. Because a function that returns a structure copies the structure into a Perl data structure, you want to make sure that it is okay to copy the record objects that you are dealing with if any of your functions will be returning one of them.

Opaque pointers should be used when you do not know the size of the object that you are using, or if the objects are created and free'd through an API interface other than `malloc` and `free`.

The examples in this section actually use pointers to records (note the trailing star * in the declarations). Most programming languages allow you to pass or return a record as either pass-by-value or as a pointer (pass-by-reference).

C code:

```
struct { int a; } foo_t;
void pass_by_value_example( struct foo_t foo );
void pass_by_reference_example( struct foo_t *foo );
```

Perl code:

```
{
  package Foo;
  use FFI::Platypus::Record;
  record_layout_1( int => 'a' );
}
$ffi->type( 'Record(Foo)' => 'foo_t' );
$ffi->attach( pass_by_value_example => [ 'foo_t' ] => 'void' );
$ffi->attach( pass_by_reference_example => [ 'foo_t*' ] => 'void' );
```

As with strings, functions that return a pointer to a record are actually copied.

C code:

```
struct foo_t *return_struct_pointer_example();
```

Perl code:

```
$ffi->attach( return_struct_pointer_example => [] => 'foo_t*' );
my $foo = return_struct_pointer_example();
# $foo is a copy of the record returned by the function.
```

As with strings, if the API expects you to free the record it returns (it is misbehaving a little, but lets set that aside), then you can work around this by returning an opaque

type, casting to the record, and finally freeing the original pointer.

```
use FFI::Platypus::Memory qw( free );
$ffi->attach( return_struct_pointer_example => [] => 'opaque' );
my $foo_ptr = return_struct_pointer_example();
my $foo = $ffi->cast( 'opaque' => 'foo_t*', $foo_ptr );
free $foo_ptr;
```

You can pass records into a closure, but care needs to be taken. Records passed into a closure are read-only inside the closure, including `string rw` members. Although you can pass a "pointer" to a record into a closure, because of limitations of the implementation you actually have a copy, so all records passed into closures are passed by-value.

Note that a record that does not have a class (classless) and is defined instead using a length is internally identical to fixed strings. That is to say `string(10)` and `record(10)*` are identical.

Fixed length arrays

Fixed length arrays of native types and strings are supported by *FFI::Platypus*. Like pointers, if the values contained in the array are updated by the C function these changes will be reflected when it returns to Perl space. An example of using this is the Unix `pipe` command which returns a list of two file descriptors as an array.

```
use FFI::Platypus 2.00;

my $ffi = FFI::Platypus->new( api => 2 );
$ffi->lib(undef);
$ffi->attach([pipe=>'mypipe'] => ['int[2]'] => 'int');

my @fd = (0,0);
mypipe(\@fd);
my($fd1,$fd2) = @fd;

print "$fd1 $fd2\n";
```

Because of the way records are implemented, an array of records does not make sense and is not currently supported.

Variable length arrays

[version 0.22]

Variable length arrays are supported for argument types can also be specified by using the `[]` notation but by leaving the size empty:

```
$ffi->type('int[]' => 'var_int_array');
```

When used as an argument type it will probe the array reference that you pass in to determine the correct size. Usually you will need to communicate the size of the array to the C code. One way to do this is to pass the length of the array in as an additional argument. For example the C code:

```
int
sum(int *array, int size)
{
  int total, i;
  for (i = 0, total = 0; i < size; i++)
  {
    total += array[i];
  }
  return total;
}
```

Can be called from Perl like this:

```
use FFI::Platypus 2.00;

my $ffi = FFI::Platypus->new( api => 2 );
$ffi->lib('./var_array.so');

$ffi->attach( sum => [ 'int[]', 'int' ] => 'int' );

my @list = (1..100);

print sum(\@list, scalar @list), "\n";
```

Another method might be to have a special value, such as 0 or NULL indicate the termination of the array.

Because of the way records are implemented, an array of records does not make sense and is not currently supported.

Closures

A closure (sometimes called a "callback", we use the `libffi` terminology) is a Perl subroutine that can be called from C. In order to be called from C it needs to be passed to a C function. To define the closure type you need to provide a list of argument types and a return type. Currently only native types (integers, floating point values, opaque), strings and records (by-value; you can pass a pointer to a record, but due to limitations of the record implementation this is actually a copy) are supported as closure argument types, and only native types and records (by-value; pointer records and records with string pointers cannot be returned from a closure) are supported as closure return types. Inside the closure any records passed in are read-only.

We plan to add other types, though they can be converted using the Platypus `cast` or `attach_cast` methods.

Here is an example, with C code:

```
/*
 * closure.c - on Linux compile with: gcc closure.c -shared -o␣
↪closure.so -fPIC
 */
```

(continues on next page)

(continued from previous page)

```
#include <stdio.h>

typedef int (*closure_t)(int);
closure_t my_closure = NULL;

void set_closure(closure_t value)
{
  my_closure = value;
}

int call_closure(int value)
{
  if(my_closure != NULL)
    return my_closure(value);
  else
    fprintf(stderr, "closure is NULL\n");
}
```

And the Perl code:

```
use FFI::Platypus 2.00;

my $ffi = FFI::Platypus->new( api => 2 );
$ffi->lib('./closure.so');
$ffi->type('(int)->int' => 'closure_t');

$ffi->attach(set_closure => ['closure_t'] => 'void');
$ffi->attach(call_closure => ['int'] => 'int');

my $closure1 = $ffi->closure(sub { $_[0] * 2 });
set_closure($closure1);
print  call_closure(2), "\n"; # prints "4"

my $closure2 = $ffi->closure(sub { $_[0] * 4 });
set_closure($closure2);
print call_closure(2), "\n"; # prints "8"
```

If you have a pointer to a function in the form of an opaque type, you can pass this in place of a closure type:

```
use FFI::Platypus 2.00;

my $ffi = FFI::Platypus->new( api => 2 );
$ffi->lib('./closure.so');
$ffi->type('(int)->int' => 'closure_t');

$ffi->attach(set_closure => ['closure_t'] => 'void');
$ffi->attach(call_closure => ['int'] => 'int');

my $closure = $ffi->closure(sub { $_[0] * 6 });
```

(continues on next page)

(continued from previous page)

```
my $opaque = $ffi->cast(closure_t => 'opaque', $closure);
set_closure($opaque);
print call_closure(2), "\n"; # prints "12"
```

The syntax for specifying a closure type is a list of comma separated types in parentheticals followed by a narrow arrow `->`, followed by the return type for the closure. For example a closure that takes a pointer, an integer and a string and returns an integer would look like this:

```
$ffi->type('(opaque, int, string) -> int' => 'my_closure_type');
```

Care needs to be taken with scoping and closures, because of the way Perl and C handle responsibility for allocating memory differently. Perl keeps reference counts and frees objects when nothing is referencing them. In C the code that allocates the memory is considered responsible for explicitly free'ing the memory for objects it has created when they are no longer needed. When you pass a closure into a C function, the C code has a pointer or reference to that object, but it has no way up letting Perl know when it is no longer using it. As a result, if you do not keep a reference to your closure around it will be free'd by Perl and if the C code ever tries to call the closure it will probably SIGSEGV. Thus supposing you have a C function `set_closure` that takes a Perl closure, this is almost always wrong:

```
set_closure($ffi->closure({ $_[0] * 2 }));  # BAD
```

In some cases, you may want to create a closure shouldn't ever be free'd. For example you are passing a closure into a C function that will retain it for the lifetime of your application. You can use the sticky method to keep the closure, without the need to keep a reference of the closure:

```
{
  my $closure = $ffi->closure(sub { $_[0] * 2 });
  $closure->sticky;
  set_closure($closure); # OKAY
}
# closure still exists and is accesible from C, but
# not from Perl land.
```

Custom Types

Custom Types in Perl

They offer a method for converting Perl types into native types that the `libffi` can understand and pass on to the C code.

Example 1: Integer constants

Say you have a C header file like this:

```
/* possible foo types: */
#define FOO_STATIC  1
```

(continues on next page)

(continued from previous page)

```
#define FOO_DYNAMIC 2
#define FOO_OTHER   3

typedef int foo_t;

void foo(foo_t foo);
foo_t get_foo();
```

The challenge is here that once the source is processed by the C pre-processor the name/value mappings for these FOO_ constants are lost. There is no way to fetch them from the library once it is compiled and linked.

One common way of implementing this would be to create and export constants in your Perl module, like this:

```
package Foo;

use FFI::Platypus 2.00;
use Exporter qw( import );

our @EXPORT_OK = qw( FOO_STATIC FOO_DYNAMIC FOO_OTHER foo get_foo );

use constant FOO_STATIC  => 1;
use constant FOO_DYNAMIC => 2;
use constant FOO_OTHER   => 3;

my $ffi = FFI::Platypus->new( api => 2 );
$ffi->attach(foo     => ['int'] => 'void');
$ffi->attach(get_foo => []      => 'int');
```

Then you could use the module thus:

```
use Foo qw( foo FOO_STATIC );
foo(FOO_STATIC);
```

If you didn't want to rely on integer constants or exports, you could also define a custom type, and allow strings to be passed into your function, like this:

```
package Foo;

use FFI::Platypus 2.00;

our @EXPORT_OK = qw( foo get_foo );

my %foo_types = (
  static  => 1,
  dynamic => 2,
  other   => 3,
);
my %foo_types_reverse = reverse %foo_types;
```

(continues on next page)

(continued from previous page)

```
my $ffi = FFI::Platypus->new( api => 2 );
$ffi->custom_type(foo_t => {
  native_type    => 'int',
  native_to_perl => sub {
    $foo_types{$_[0]};
  },
  perl_to_native => sub {
    $foo_types_reverse{$_[0]};
  },
});

$ffi->attach(foo     => ['foo_t'] => 'void');
$ffi->attach(get_foo => []        => 'foo_t');
```

Now when an argument of type `foo_t` is called for it will be converted from an appropriate string representation, and any function that returns a `foo_t` type will return a string instead of the integer representation:

```
use Foo;
foo('static');
```

If the library that you are using has a lot of these constants you can try using Convert::Binary::C or another C header parser to obtain the appropriate name/value pairings for the constants that you need.

Example 2: Blessed references

Supposing you have a C library that uses an opaque pointer with a pseudo OO interface, like this:

```
typedef struct foo_t;

foo_t *foo_new();
void foo_method(foo_t *, int argument);
void foo_free(foo_t *);
```

One approach to adapting this to Perl would be to create a OO Perl interface like this:

```
package Foo;

use FFI::Platypus 2.00;
use FFI::Platypus::API qw( arguments_get_string );

my $ffi = FFI::Platypus->new( api => 2 );
$ffi->custom_type(foo_t => {
  native_type    => 'opaque',
  native_to_perl => sub {
    my $class = arguments_get_string(0);
    bless \$_[0], $class;
```

(continues on next page)

(continued from previous page)

```
  }
  perl_to_native => sub { ${$_[0]} },
});

$ffi->attach([ foo_new => 'new' ] => [ 'string' ] => 'foo_t' );
$ffi->attach([ foo_method => 'method' ] => [ 'foo_t', 'int' ] =>
↪'void');
$ffi->attach([ foo_free => 'DESTROY' ] => [ 'foo_t' ] => 'void');

my $foo = Foo->new;
```

Here we are blessing a reference to the opaque pointer when we return the custom type for `foo_t`, and dereferencing that reference before we pass it back in. The function `arguments_get_string` queries the C arguments to get the class name to make sure the object is blessed into the correct class (for more details on the custom type API see *FFI::Platypus::API*), so you can inherit and extend this class like a normal Perl class. This works because the C "constructor" ignores the class name that we pass in as the first argument. If you have a C "constructor" like this that takes arguments you'd have to write a wrapper for new.

A good example of a C library that uses this pattern, including inheritance is `libarchive`. Platypus comes with a more extensive example in `examples/archive.pl` that demonstrates this.

Example 3: Pointers with pack / unpack

TODO

See example *FFI::Platypus::Type::StringPointer*.

Example 4: Custom Type modules and the Custom Type API

TODO

See example *FFI::Platypus::Type::PointerSizeBuffer*.

Example 5: Custom Type on CPAN

You can distribute your own Platypus custom types on CPAN, if you think they may be applicable to others. The default namespace is prefix with `FFI::Platypus::Type::`, though you can stick it anywhere (under your own namespace may make more sense if the custom type is specific to your application).

A good example and pattern to follow is *FFI::Platypus::Type::StringArray*.

SEE ALSO

- *FFI::Platypus*

  Main platypus documentation.

- *FFI::Platypus::API*

  Custom types API.

- *FFI::Platypus::Type::StringPointer*

  String pointer type.

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

Modules

- `FFI::Platypus::Type::PointerSizeBuffer` - Convert string scalar to a buffer as a pointer / size_t combination
- `FFI::Platypus::Type::StringArray` - Platypus custom type for arrays of strings
- `FFI::Platypus::Type::StringPointer` - Convert a pointer to a string and back
- `FFI::Platypus::Type::WideString` - Platypus custom type for Unicode "wide" strings

FFI::Platypus::Type::PointerSizeBuffer

Convert string scalar to a buffer as a pointer / size_t combination

VERSION

version 2.12

SYNOPSIS

In your C code:

```
void
function_with_buffer(void *pointer, size_t size)
{
  ...
}
```

In your Platypus::FFI code:

```
use FFI::Platypus 2.00;

my $ffi = FFI::Platypus->new( api => 2 );
$ffi->load_custom_type('::PointerSizeBuffer' => 'buffer');

$ffi->attach(function_with_buffer => ['buffer'] => 'void');
my $string = "content of buffer";
function_with_buffer($string);
```

DESCRIPTION

A common pattern in C code is to pass in a region of memory as a buffer, consisting of a pointer and a size of the memory region. In Perl, string scalars also point to a contiguous series of bytes that has a size, so when interfacing with C libraries it is handy to be able to pass in a string scalar as a pointer / size buffer pair.

SEE ALSO

- *FFI::Platypus*

  Main Platypus documentation.

- *FFI::Platypus::Type*

  Platypus types documentation.

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::Platypus::Type::StringArray

Platypus custom type for arrays of strings

VERSION

version 2.12

SYNOPSIS

In your C code:

```
void
takes_string_array(const char **array)
{
  ...
}

void
takes_fixed_string_array(const char *array[5])
{
  ...
}
```

In your *FFI::Platypus* code:

```
use FFI::Platypus 2.00;

my $ffi = FFI::Platypus->new( api => 2 );
$ffi->load_custom_type('::StringArray' => 'string_array');
$ffi->load_custom_type('::StringArray' => 'string_5' => 5);

$ffi->attach(takes_string_array => ['string_array'] => 'void');
$ffi->attach(takes_fixed_string_array => ['string_5'] => 'void');

my @list = qw( foo bar baz );

takes_string_array(\@list);
takes_fixed_string_array([qw( s1 s2 s3 s4 s5 )]);
```

DESCRIPTION

NOTE: The primary motivation for this custom type was originally to fill the void left by the fact that *FFI::Platypus* did not support arrays of strings by itself. Since 0.62 this support has been added, and that is probably what you want to use, but the semantics and feature set are slightly different, so there are cases where you might want to use this custom type.

This module provides a *FFI::Platypus* custom type for arrays of strings. The array is always NULL terminated. Return types are supported!

This custom type takes two optional arguments. The first is the size of arrays and the second is a default value to fill in any values that aren't provided when the function is called. If not default is provided then `NULL` will be passed in for those values.

SUPPORT

If something does not work the way you think it should, or if you have a feature request, please open an issue on this project's GitHub Issue tracker:

https://github.com/plicease/FFI-Platypus-Type-StringArray/issues

CONTRIBUTING

If you have implemented a new feature or fixed a bug then you may make a pull request on this project's GitHub repository:

https://github.com/plicease/FFI-Platypus-Type-StringArray/pulls

This project's GitHub issue tracker listed above is not Write-Only. If you want to contribute then feel free to browse through the existing issues and see if there is something you feel you might be good at and take a whack at the problem. I frequently open issues myself that I hope will be accomplished by someone in the future but do not have time to immediately implement myself.

Another good area to help out in is documentation. I try to make sure that there is good document coverage, that is there should be documentation describing all the public features and warnings about common pitfalls, but an outsider's or alternate view point on such things would be welcome; if you see something confusing or lacks sufficient detail I encourage documentation only pull requests to improve things.

SEE ALSO

- *FFI::Platypus*
- *FFI::Platypus::Type::StringPointer*

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::Platypus::Type::StringPointer

Convert a pointer to a string and back

VERSION

version 2.12

SYNOPSIS

In your C code:

```
void
string_pointer_argument(const char **string)
{
  ...
}
const char **
string_pointer_return(void)
{
  ...
}
```

In your Platypus::FFI code:

```
use FFI::Platypus 2.00;

my $ffi = FFI::Platypus->new( api => 2 );
$ffi->load_custom_type('::StringPointer' => 'string_pointer');

$ffi->attach(string_pointer_argument => ['string_pointer'] => 'void
 ↪');
$ffi->attach(string_pointer_return   => [] => 'string_pointer');

my $string = "foo";

string_pointer_argument(\$string); # $string may be modified

$ref = string_pointer_return();

print $$ref;  # print the string pointed to by $ref
```

DESCRIPTION

NOTE: As of version 0.61, this custom type is now deprecated since pointers to strings are supported in the *FFI::Platypus* directly without custom types.

This module provides a *FFI::Platypus* custom type for pointers to strings.

SEE ALSO

- *FFI::Platypus*

  Main Platypus documentation.

- *FFI::Platypus::Type*

  Platypus types documentation.

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::Platypus::Type::WideString

Platypus custom type for Unicode “wide” strings

VERSION

version 2.12

SYNOPSIS

```
use FFI::Platypus 2.00;

my $ffi = FFI::Platypus->new( api => 2, lib => [undef] );
$ffi->load_custom_type('::WideString' => 'wstring', access => 'read'␣
↪);
$ffi->load_custom_type('::WideString' => 'wstring_w', access =>
↪'write' );

# call function that takes a constant wide string
$ffi->attach( wcscmp => ['wstring', 'wstring'] => 'int' );
my $diff = wcscmp("I ❤ perl + Platypus", "I ❤ perl + Platypus"); #␣
↪returns 0

# call a function that takes a wide string for writing
$ffi->attach( wcscpy => ['wstring_w', 'wstring'] );
my $buf;
```

(continues on next page)

(continued from previous page)

```
wcscpy(\$buf, "I ❤ perl + Platypus");
print $buf, "\n";  # prints "I ❤ perl + Platypus"

# call a function that takes a wide string for modification
$ffi->attach( wcscat => ['wstring_w', 'wstring'] );
my $buf;
wcscat( [ \$buf, "I ❤ perl" ], " + Platypus");
print $buf, "\n";  # prints "I ❤ perl + Platypus"
```

On Windows use with LPCWSTR:

```
use FFI::Platypus 2.00;

my $ffi = FFI::Platypus->new( api => 2, lib => [undef] );

# define some custom Win32 Types
# to get these automatically see FFI::Platypus::Lang::Win32
$ffi->load_custom_type('::WideString' => 'LPCWSTR', access => 'read'
 ↪);
$ffi->type('opaque' => 'HWND');
$ffi->type('uint'   => 'UINT');

use constant MB_OK                    => 0x00000000;
use constant MB_DEFAULT_DESKTOP_ONLY => 0x00020000;

$ffi->attach( [MessageBoxW => 'MessageBox'] => [ 'HWND', 'LPCWSTR',
 ↪'LPCWSTR', 'UINT'] => 'int' );

MessageBox(undef, "I ❤ Platypus", "Confession", MB_OK|MB_DEFAULT_
 ↪DESKTOP_ONLY);
```

DESCRIPTION

This custom type plugin for *FFI::Platypus* provides support for the native “wide” string type on your platform, if it is available.

Wide strings are made of up wide characters (wchar_t, also known as WCHAR on Windows) and have enough bits to represent character sets that require larger than the traditional one byte char.

These strings are most commonly used on Windows where they are referred to as LPWSTR and LPCWSTR (The former for read/write buffers and the latter for const read-only strings), where they are encoded as UTF-16LE.

They are also supported by libc on many modern Unix systems where they are usually UTF-32 of the native byte-order of the system. APIs on Unix systems more commonly use UTF-8 which provides some compatibility with ASCII, but you may occasionally find APIs that talk in wide strings. (libarchive, for example, can work in both).

This plugin will detect the native wide string format for you and transparently convert Perl strings, which are typically encoded internally as UTF-8. If for some reason it cannot detect the correct encoding, or if your platform is currently supported, an exception will

be thrown (please open a ticket if this is the case). It can be used either for read/write buffers, for const read-only strings, and for return values. It supports these options:

Options:

- access

  Either `read` or `write` depending on if you are using a read/write buffer or a const read-only string.

- size

  For read/write buffer, the size of the buffer to create, if not provided by the caller.

read-only

Read-only strings are the easiest of all, are converted to the native wide string format in a buffer and are freed after that function call completes.

```
$ffi->load_custom_type('::WideString' => 'wstring' );
$ffi->function( wprintf => [ 'wstring' ] => [ 'wstring' ] => 'int' )
     ->call("I %s perl + Platypus", "❤");
```

This is the mode that you want to use when you are calling a function that takes a `const wchar_t*` or a `LPCWSTR`.

return value

For return values the `access` and `size` options are ignored. The string is simply copied into a Perl native string.

```
$ffi->load_custom_type('::WideString' => 'wstring' );
# see note below in CAVEATS about wcsdup
my $str = $ffi->function( wcsdup => [ 'wstring' ] => 'wstring' )
               ->call("I ❤ perl + Platypus");
```

This is the mode that you want to use when you are calling a function that returns a `const wchar_t*`, `wchar_t`, `LPWSTR` or `LPCWSTR`.

read/write

Read/write strings can be passed in one of two ways. Which you choose depends on if you want to initialize the read/write buffer or not.

- default buffer size

  The simplest way is to fallback on the default buffer size, which can be specified using the `size` option when creating the custom type.

  ```
  my $ffi = FFI::Platypus->new( api => 2, lib => [undef] );
  $ffi->load_custom_type('::WideString' => 'wstring',   access =>
   ↪'read' );
  $ffi->load_custom_type('::WideString' => 'wstring_w', access =>
   ↪'write', size => 512 );
  ```

(continues on next page)

(continued from previous page)

```
$ffi->attach( wcscpy => ['wstring_w', 'wstring'] );
my $buf;
wcscpy(\$buf, "I ❤ perl + Platypus");
print $buf, "\n";  # prints "I ❤ perl + Platypus"
```

Discussion: This is the most sensical approach when the exact size of the buffer is known for all usages of the string type. It can also be sensical if the buffer size is larger than any possible output, though care should be taken since this may be hard to determine reliably.

The default size if none is specified when creating the custom type is 2048, which is probably large enough for many uses, but also probably wastes memory for many of them.

- allocate your buffer of a specific size

  The safest and most memory efficient method is of course to allocate exactly the amount of memory that you need.

```
my $ffi = FFI::Platypus->new( api => 2, lib => [undef] );
$ffi->load_custom_type('::WideString' => 'wstring',   access =>
 ↪'read'  );
$ffi->load_custom_type('::WideString' => 'wstring_w', access =>
 ↪'write' );

$ffi->attach( wcscpy => ['wstring_w', 'wstring'] );
my $width = $ffi->sizeof('wchar_t');
my $buf = "\0" x ( (length ("I ❤ perl + Platypus") + 1)*$width);
wcscpy(\$buf, "I ❤ perl + Platypus");
print $buf, "\n";  # prints "I ❤ perl + Platypus"
```

Discussion: By assigning `$buf` to a string of null characters the length of the source string, plus one (for the null at the end) and then multiplying that by the size of `wchar_t`, you get the exact number of bytes needed for the destination buffer.

Note that although we pass in a reference to a buffer, what comes back is converted to a Perl string, which will be internally UTF-8, not stored at the original buffer location. This is slightly awkward, but what you need most of the time.

- initialize the read/write buffer

  Some functions don't expect empty null padded buffers though, in this case you will want to initialize the buffer.

```
my $ffi = FFI::Platypus->new( api => 2, lib => [undef] );
$ffi->load_custom_type('::WideString' => 'wstring',   access =>
 ↪'read'  );
$ffi->load_custom_type('::WideString' => 'wstring_w', access =>
 ↪'write' );

$ffi->attach( wcscat => ['wstring_w', 'wstring'] );
my $buf;
```

(continues on next page)

(continued from previous page)

```
wcscat( [ \$buf, "I ❤ perl" ], " + Platypus");
print $buf, "\n";  # prints "I ❤ perl + Platypus"
```

Discussion: To initialize we pass in an array reference instead of a scalar reference. The first element is a scalar reference to the buffer (which can be pre-allocated or not; if it is not allocated then it will be allocated to the default size for the type). The second argument is what the buffer should be initialized to before the underlying C function is called. The Perl string is encoded into wide string format before being used to initialize the buffer.

As before a reference to the translated string is returned, and the buffer that was used to pass in is freed.

- allocate memory using `malloc` or `wcsdup` etc.

  You can also allocate memory using `malloc` or `wcsdup` to return an opaque type and manipulate it using the libc `wcs*` functions. It will still probably be useful to use this plugin to cast the opaque back to a Perl string. The CAVEATS section below includes several examples.

This is the mode that you want to use when you are calling a function that takes a <wchar_t*> or a `LPWSTR`.

CAVEATS

As with the Platypus built in `string` type, return values are copied into a Perl scalar. This is usually what you want anyway, but some APIs expect the caller to take responsibility for freeing the pointer to the wide string that it returns. For example, `wcsdup` works in this way. The workaround is to return an opaque pointer, cast it from a wide string and free the pointer.

```
use FFI::Platypus::Memory qw( free );
$ffi->load_custom_type('::WideString' => 'wstring' );
my $ptr = $ffi->function( wcsdup => [ 'wstring' ] => 'opaque' )
               ->call("I ❤ perl + Platypus");
my $str = $ffi->cast('opaque', 'wstring', $ptr);
free $ptr;
```

Because of the order in which objects are freed you cannot return a wide string if it is also a wide string argument to a function. For example `wcscpy` may crash if you specify the return value as a wide string:

```
# wchar_t *wcscpy(wchar_t *dest, const wchar_t *src);
$ffi->attach( wcscpy => [ 'wstring_w', 'wstring' ] => 'wstring' ); #␣
↪no
my $str;
wcscpy( \$str, "I ❤ perl + Platypus");  # may crash on memory error
```

This is because the order in which things are done here are 1. `$str` is allocated 2. `$str` is re-encoded as utf and the old buffer is freed 3. the return value is computed based on the `$str` buffer that was freed.

If you look at `wcscpy` though you don't actually need the return value. To make this code work, you can just ignore the return value:

```
$ffi->attach( wcscpy => [ 'wstring_w', 'wstring' ] => 'void' ); #␣
↪yes
my $str;
wcscpy( \$str, "I ❤ perl + Platypus"); # good!
```

On the other hand you do care about the return value from `wcschr`, which returns a pointer to the first occurrence of a character in an argument string:

```
# wchar_t *wcschr(const wchar_t *wcs, wchar_t wc);
$ffi->attach( wcschr => [ 'wstring', 'wchar_t' ] => 'wstring' ); #␣
↪no
# this may crash on memory error or return the wrong value
my $str = wcschr("I ❤ perl + Platypus", ord("❤"));
```

Instead you need to work with pointers and casts to use this function:

```
use FFI::Platypus 2.00;
use FFI::Platypus::Memory qw( free );

my $ffi = FFI::Platypus->new( api => 2, lib => [undef] );

$ffi->attach( wcsdup => ['wstring'] => 'opaque' );
$ffi->attach( strchr => [ opaque', 'wchar_t' ] => 'wstring' );

# create a wcs string in memory using wcsdup
my $haystack = wcsdup("I ❤ perl + Platypus");
# find the heart and return as a wide string
my $needle = strchr($haystack, ord("❤"));
# safe to free the pointer to the larger string now
free $haystack;
```

SEE ALSO

- *FFI::Platypus*

  Core Platypus documentation.

- *FFI::Platypus::Type*

  Includes documentation on handling "normal" 8 bit C strings among others.

- *FFI::Platypus::Lang::Win32*

  Documentation for using Platypus with `LPWSTR` and `LPCWSTR` types on Microsoft Windows. These types are just aliases for the standard C wide strings.

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::Platypus::TypeParser

FFI Type Parser

VERSION

version 2.12

DESCRIPTION

This class is private to FFI::Platypus. See *FFI::Platypus* for the public interface to Platypus types.

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

Modules

- `FFI::Platypus::TypeParser::Version0` - FFI Type Parser Version Zero
- `FFI::Platypus::TypeParser::Version1` - FFI Type Parser Version One
- `FFI::Platypus::TypeParser::Version2` - FFI Type Parser Version Two

FFI::Platypus::TypeParser::Version0

FFI Type Parser Version Zero

VERSION

version 2.12

SYNOPSIS

```
use FFI::Platypus;
my $ffi = FFI::Platypus->new( api => 0 );
$ffi->type('record(Foo::Bar)' => 'foo_bar_t');
$ffi->type('opaque' => 'baz_t');
$ffi->type('opaque*' => 'baz_ptr');
```

DESCRIPTION

This documents the original *FFI::Platypus* type parser. It was the default and only type parser used by *FFI::Platypus* starting with version `0.02`. Starting with version `1.00` *FFI::Platypus* comes with a new type parser with design fixes that are not backward compatibility.

Interface differences

- Pass-by-value records are not allowed

  Originally *FFI::Platypus* only supported passing records as a pointer. The type `record(Foo::Bar)` actually passes a pointer to the record. In the version 1.00 parser allows `record(Foo::Bar)` which is pass-by-value (the contents of the record is copied onto the stack) and `record(Foo::Bar)*` which is pass-by-reference or pointer (a pointer to the record is passed to the callee so that it can make modifications to the record).

  TL;DR `record(Foo::Bar)` in version 0 is equivalent to `record(Foo::Bar)*` in the version 1 API. There is no equivalent to `record(Foo::Bar)*` in the version 0 API.

- decorate aliases of basic types

  This is not allowed in the version 0 API:

  ```
  $ffi->type('opaque' => 'foo_t');    # ok!
  $ffi->type('foo_t*' => 'foo_ptr');  # not ok! in version 0, ok!
   ↪in version 1
  ```

Instead you need to use the basic type in the second type definition:

```
$ffi->type('opaque' => 'foo_t');    # ok!
$ffi->type('opaque*' => 'foo_ptr'); # ok!
```

- object types are not allowed

```
$ffi->type('object(Foo::Bar)');   # not ok! in version 0, ok!
 ↪in version 1
```

SEE ALSO

- *FFI::Platypus*

  The core *FFI::Platypus* documentation.

- *FFI::Platypus::TypeParser::Version1*

  The API `1.00` type parser.

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::Platypus::TypeParser::Version1

FFI Type Parser Version One

VERSION

version 2.12

SYNOPSIS

```
use FFI::Platypus 1.00;
my $ffi = FFI::Platypus->new( api => 1 );
$ffi->type('record(Foo::Bar)' => 'foo_bar_t');
$ffi->type('record(Foo::Bar)*' => 'foo_bar_ptr');
$ffi->type('opaque' => 'baz_t');
$ffi->type('bar_t*' => 'baz_ptr');
```

DESCRIPTION

This documents the second (version 1) type parser for *FFI::Platypus*. This type parser was included with *FFI::Platypus* starting with version `0.91` in an experimental capability, and `1.00` as a stable interface. Starting with version `1.00` the main *FFI::Platypus* documentation describes the version 1 API and you can refer to *FFI::Platypus::TypeParser::Version0* for details on the version0 API.

SEE ALSO

- *FFI::Platypus*

  The core *FFI::Platypus* documentation.

- *FFI::Platypus::TypeParser::Version0*

  The API `0.02` type parser.

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::Platypus::TypeParser::Version2

FFI Type Parser Version Two

VERSION

version 2.12

SYNOPSIS

```
use FFI::Platypus 2.00;
my $ffi = FFI::Platypus->new( api => 2 );
$ffi->type('string(10)');
```

DESCRIPTION

This documents the third (version 2) type parser for *FFI::Platypus*. This type parser was included with *FFI::Platypus* starting with version `1.58` in an experimental capability, and `2.00` as a stable interface. Starting with version `1.00` the main *FFI::Platypus* documentation describes the version 2 API and you can refer to *FFI::Platypus::TypeParser::Version1* for details on the version1 API.

SEE ALSO

- *FFI::Platypus*

  The core *FFI::Platypus* documentation.

- *FFI::Platypus::TypeParser::Version0*

  The API `0.02` type parser.

- *FFI::Platypus::TypeParser::Version1*

  The API `1.00` type parser.

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::Probe

System detection and probing for FFI extensions.

VERSION

version 2.12

SYNOPSIS

```
use FFI::Probe;

my $probe = FFI::Probe->new;
$probe->check_header('foo.h');
...
```

DESCRIPTION

This class provides an interface for probing for system capabilities. It is used internally as part of the *FFI::Platypus* build process, but it may also be useful for extensions that use Platypus as well.

CONSTRUCTOR

new

```
my $probe = FFI::Probe->new(%args);
```

Creates a new instance.

- log

  Path to a log or file handle to write to.

- data_filename

  Path to a file which will be used to store/cache results.

METHODS

check_header

```
my $bool = $probe->check_header($header);
```

Checks that the given C header file is available. Stores the result, and returns a true/false value.

check_cpp

check_eval

```
my $bool = $probe>check_eval(%args);
```

- headers

  Any additional headers.

- decl

  Any C declarations that need to be made before the `dlmain` function.

- stmt

  Any C statements that should be made before the evaluation.

- eval

  Any evaluations that should be returned.

check

check_type_int

```
my $type = $probe->check_type_int($type);
```

check_type_enum

```
my $type = $probe->check_type_enum;
```

check_type_enum

```
my $type = $probe->check_type_enum;
```

check_type_float

```
my $type = $probe->check_type_float($type);
```

check_type_pointer

```
my $type = $probe->check_type_pointer;
```

set

```
$probe->set(@key, $value);
```

Used internally to store a value.

save

```
$probe->save;
```

Saves the values already detected.

data

```
my $data = $probe->data;
```

Returns a hashref of the data already detected.

log

```
$probe->log($string);
```

Sends the given string to the log.

log_code

```
$prbe->log_code($string);
```

Sends the given multi-line code block to the log.

template

```
my $template = $probe->template;
```

Returns the C code template used for `check_eval` and other `check_` methods.

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

Modules

- `FFI::Probe::Runner` - Probe runner for FFI

FFI::Probe::Runner

Probe runner for FFI

VERSION

version 2.12

SYNOPSIS

```
use FFI::Probe::Runner;

my $runner = FFI::Probe::Runner->new;
$runner->run('foo.so');
```

DESCRIPTION

This class executes code in a dynamic library for probing and detecting platform properties.

CONSTRUCTOR

new

```
my $runner = FFI::Probe::Runner->new(%args);
```

Creates a new instance.

- exe

  The path to the dlrun wrapper. The default is usually correct.

- flags

  The flags to pass into `dlopen`. The default is `RTLD_LAZY` on Unix and `0` on windows..

METHODS

exe

```
my $exe = $runner->exe;
```

The path to the dlrun wrapper.

flags

```
my $flags = $runner->flags;
```

The flags to pass into `dlopen`.

verify

```
$runner->verify;
```

Verifies the dlrun wrapper is working. Throws an exception in the event of failure.

run

```
$runner->run($dll, @args);
```

Runs the `dlmain` function in the given dynamic library, passing in the given arguments. Returns a *FFI::Probe::Runner::Result* object which contains the results.

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

Modules

- `FFI::Probe::Runner::Builder` - Probe runner builder for FFI
- `FFI::Probe::Runner::Result` - The results from a probe run.

FFI::Probe::Runner::Builder

Probe runner builder for FFI

VERSION

version 2.12

SYNOPSIS

```
use FFI::Probe::Runner::Builder;
my $builder = FFI::Probe::Runner::Builder->new
  dir => "/foo/bar",
);
my $exe = $builder->build;
```

DESCRIPTION

This is a builder class for the FFI probe runner. It is mostly only of interest if you are hacking on *FFI::Platypus* itself.

The interface may and will change over time without notice. Use in external dependencies at your own peril.

CONSTRUCTORS

new

```
my $builder = FFI::Probe::Runner::Builder->new(%args);
```

Create a new instance.

- dir

  The root directory for where to place the probe runner files. Will be created if it doesn't already exist. The default makes sense for when *FFI::Platypus* is being built.

METHODS

dir

```
my $dir = $builder->dir(@subdirs);
```

Returns a subdirectory from the builder root. Directory will be created if it doesn't already exist.

cc

```
my @cc = @{ $builder->cc };
```

The C compiler to use. Returned as an array reference so that it may be modified.

ccflags

```
my @ccflags = @{ $builder->ccflags };
```

The C compiler flags to use. Returned as an array reference so that it may be modified.

optimize

The C optimize flags to use. Returned as an array reference so that it may be modified.

ld

```
my @ld = @{ $builder->ld };
```

The linker to use. Returned as an array reference so that it may be modified.

ldflags

```
my @ldflags = @{ $builder->ldflags };
```

The linker flags to use. Returned as an array reference so that it may be modified.

libs

```
my @libs = @{ $builder->libs };
```

The library flags to use. Returned as an array reference so that it may be modified.

file

```
my $file = $builder->file(@subdirs, $filename);
```

Returns a file in a subdirectory from the builder root. Directory will be created if it doesn't already exist. File will not be created.

exe

```
my $exe = $builder->exe;
```

The name of the executable, once it is built.

source

```
my $source = $builder->source;
```

The C source for the probe runner.

extract

```
$builder->extract;
```

Extract the source for the probe runner.

run

```
$builder->run($type, @command);
```

Runs the given command. Dies if the command fails.

run_list

```
$builder->run($type, \@command, \@command, ...);
```

Runs the given commands in order until one succeeds. Dies if they all fail.

build

```
my $exe = $builder->build;
```

Builds the probe runner. Returns the path to the executable.

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::Probe::Runner::Result

The results from a probe run.

VERSION

version 2.12

SYNOPSIS

DESCRIPTION

CONSTRUCTOR

new

```
my $result = FFI::Probe::Runner::Result->new(%args);
```

Creates a new instance of the class.

METHODS

stdout

```
my $stdout = $result->stdout;
```

stderr

```
my $stderr = $result->stderr;
```

rv

```
my $rv = $result->rv;
```

signal

```
my $signal = $result->signal;
```

pass

```
my $pass = $result->pass;
```

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

FFI::Temp

Temp Dir support for FFI::Platypus

VERSION

version 2.12

DESCRIPTION

This class is private to *FFI::Platypus*.

AUTHOR

Author: Graham Ollis plicease@cpan.org

Contributors:

Bakkiaraj Murugesan (bakkiaraj)

Dylan Cali (calid)

pipcet

Zaki Mughal (zmughal)

Fitz Elliott (felliott)

Vickenty Fesunov (vyf)

Gregor Herrmann (gregoa)

Shlomi Fish (shlomif)

Damyan Ivanov

Ilya Pavlov (Ilya33)

Petr Písař (ppisar)

Mohammad S Anwar (MANWAR)

Håkon Hægland (hakonhagland, HAKONH)

Meredith (merrilymeredith, MHOWARD)

Diab Jerius (DJERIUS)

Eric Brine (IKEGAMI)

szTheory

José Joaquín Atria (JJATRIA)

Pete Houston (openstrike, HOUSTON)

Lukas Mai (MAUKE)

COPYRIGHT AND LICENSE

This software is copyright (c) 2015-2022 by Graham Ollis.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

5.2.42 Fatal

Replace functions with equivalents which succeed or die

SYNOPSIS

```
use Fatal qw(open close);

open(my $fh, "<", $filename);  # No need to check errors!

use File::Copy qw(move);
use Fatal qw(move);

move($file1, $file2); # No need to check errors!

sub juggle { . . . }
Fatal->import('juggle');
```

BEST PRACTICE

Fatal has been obsoleted by the new *autodie* pragma. Please use *autodie* in preference to `Fatal`. *autodie* supports lexical scoping, throws real exception objects, and provides much nicer error messages.

The use of `:void` with Fatal is discouraged.

DESCRIPTION

`Fatal` provides a way to conveniently replace functions which normally return a false value when they fail with equivalents which raise exceptions if they are not successful. This lets you use these functions without having to test their return values explicitly on each call. Exceptions can be caught using `eval{}`. See *perlfunc* and *perlvar* for details.

The do-or-die equivalents are set up simply by calling Fatal's `import` routine, passing it the names of the functions to be replaced. You may wrap both user-defined functions and overridable CORE operators (except `exec`, `system`, `print`, or any other built-in that cannot be expressed via prototypes) in this way.

If the symbol `:void` appears in the import list, then functions named later in that import list raise an exception only when these are called in void context–that is, when their return values are ignored. For example

```
use Fatal qw/:void open close/;

# properly checked, so no exception raised on error
if (not open(my $fh, '<', '/bogotic') {
    warn "Can't open /bogotic: $!";
}

# not checked, so error raises an exception
close FH;
```

The use of `:void` is discouraged, as it can result in exceptions not being thrown if you *accidentally* call a method without void context. Use *autodie* instead if you need to be able to disable autodying/Fatal behaviour for a small block of code.

DIAGNOSTICS

- Bad subroutine name for Fatal: %s

  You've called `Fatal` with an argument that doesn't look like a subroutine name, nor a switch that this version of Fatal understands.

- You've asked `Fatal` to try and replace a subroutine which does not exist, or has not yet been defined.

- You've asked `Fatal` to replace a subroutine, but it's not a Perl built-in, and `Fatal` couldn't find it as a regular subroutine. It either doesn't exist or has not yet been defined.

- Cannot make the non-overridable %s fatal

  You've tried to use `Fatal` on a Perl built-in that can't be overridden, such as `print` or `system`, which means that `Fatal` can't help you, although some other modules might. See the "SEE ALSO" section of this documentation.

- Internal error: %s

  You've found a bug in `Fatal`. Please report it using the `perlbug` command.

BUGS

`Fatal` clobbers the context in which a function is called and always makes it a scalar context, except when the `:void` tag is used. This problem does not exist in *autodie*.

"Used only once" warnings can be generated when `autodie` or `Fatal` is used with package filehandles (eg, `FILE`). It's strongly recommended you use scalar filehandles instead.

AUTHOR

Original module by Lionel Cons (CERN).

Prototype updates by Ilya Zakharevich ilya@math.ohio-state.edu.

autodie support, bugfixes, extended diagnostics, `system` support, and major overhauling by Paul Fenwick pjf@perltraining.com.au

LICENSE

This module is free software, you may distribute it under the same terms as Perl itself.

SEE ALSO

autodie for a nicer way to use lexical Fatal.

IPC::System::Simple for a similar idea for calls to `system()` and backticks.

5.2.43 Fcntl

Symbolic constants for `fcntl(2)`, `flock(2)`, `lseek(2)`, and file-mode bits - everything you need to call `sysopen`, `fcntl`, `flock`, `seek`, and to interpret the mode word returned by `stat`.

Import individual names, or a named group via one of the export tags: `:DEFAULT` (open and file-descriptor flags), `:flock` (advisory locks), `:seek` (whence positions), `:mode` (mode bits and `S_IS*` predicates), `:Fcompat` (BSD-style `flock` aliases).

Open flags (sysopen)

- `O_RDONLY`, `O_WRONLY`, `O_RDWR` - access mode.
- `O_CREAT` - create the file if it does not exist.
- `O_EXCL` - with `O_CREAT`, fail if the file already exists.
- `O_TRUNC` - truncate to zero length on open.
- `O_APPEND` - every write seeks to end of file first.
- `O_NONBLOCK` - open without blocking; I/O returns `EAGAIN` instead.
- `O_NOCTTY`, `O_NOFOLLOW`, `O_BINARY` - additional platform flags (`O_BINARY` is a no-op on Linux).

File-descriptor control (fcntl)

- `F_DUPFD` - duplicate a descriptor.
- `F_GETFD`, `F_SETFD` - read / set the close-on-exec flag `FD_CLOEXEC`.
- `F_GETFL`, `F_SETFL` - read / set the open-file status flags.

Lock types (flock)

- `LOCK_SH` - shared lock.
- `LOCK_EX` - exclusive lock.
- `LOCK_UN` - release lock.
- `LOCK_NB` - OR with the above to request a non-blocking attempt.

Seek whence

- SEEK_SET - from start of file.
- SEEK_CUR - from current position.
- SEEK_END - from end of file.

File mode bits (stat)

- Type mask and values: S_IFMT, S_IFREG, S_IFDIR, S_IFCHR, S_IFBLK, S_IFIFO, S_IFLNK, S_IFSOCK.
- Permission triplets: S_IRWXU, S_IRWXG, S_IRWXO and the per-bit S_IRUSR / S_IWUSR / S_IXUSR (and the GRP, OTH variants).
- Setuid / setgid / sticky bits: S_ISUID, S_ISGID, S_ISVTX.

Mode test helpers

Function-form predicates over a stat-mode value: S_ISREG, S_ISDIR, S_ISCHR, S_ISBLK, S_ISFIFO, S_ISLNK, S_ISSOCK, plus S_IFMT (extract file-type bits) and S_IMODE (extract permission bits).

Performance

Every constant in this module is resolved at compile time - using O_CREAT in an expression costs exactly what writing the integer literal would cost. No per-call lookup.

A constant this system does not have dies when called: a name Fcntl knows but the platform lacks (O_EXLOCK, S_IFWHT) with "Your vendor has not defined Fcntl macro O_EXLOCK, used at FILE line N", any other name with "FOO is not a valid Fcntl macro at FILE line N".

```
use Fcntl qw(:DEFAULT :flock :seek :mode);

sysopen(my $fh, $file, O_WRONLY | O_CREAT | O_EXCL, 0644) or die
 ↪"sysopen: $!";
flock($fh, LOCK_EX | LOCK_NB) or die "flock: $!";
syswrite $fh, "data\n";
sysseek $fh, 0, SEEK_SET;
my $mode = (stat $fh)[2];
printf "%s, permissions %04o\n", S_ISREG($mode) ? "regular file" :
 ↪"other", S_IMODE($mode);
# regular file, permissions 0644
```

This is a native implementation of Fcntl 1.20; its Fcntl.pm is used unchanged, and use Fcntl alone exports the :DEFAULT set.

Differences from upstream

None.

Functions

Mode test helpers

S_IMODE

Extract the permission bits from a stat-mode value.

S_IFMT

Extract the file-type bits from a stat-mode value.

S_ISREG

True if a stat-mode value names a file of the predicate's type: `S_ISREG` a regular file, `S_ISDIR` a directory, `S_ISLNK` a symbolic link, `S_ISSOCK` a socket, `S_ISBLK` a block device, `S_ISCHR` a character device, `S_ISFIFO` a named pipe.

Also available as `Fcntl::S_ISDIR`, `Fcntl::S_ISLNK`, `Fcntl::S_ISSOCK`, `Fcntl::S_ISBLK`, `Fcntl::S_ISCHR`, `Fcntl::S_ISFIFO`.

S_IMODE

Extract the permission bits from a stat-mode value.

Synopsis

```
my $perm = S_IMODE($mode);
```

Returns `$mode & 07777` - the nine permission bits plus the setuid, setgid, and sticky bits. Strips the file-type bits that `S_IFMT` would keep. With no argument, warns "Use of uninitialized value" and returns `0`.

Examples

```
use Fcntl qw(:mode);
my $mode = (stat $path)[2];
printf "perms = %04o\n", S_IMODE($mode);    # perms = 0644
```

S_IFMT

Extract the file-type bits from a stat-mode value.

Synopsis

```
my $type = S_IFMT($mode);
my $mask = S_IFMT();           # the mask constant itself
```

Called with an argument, returns `$mode & S_IFMT` - the bits that classify the file (regular, directory, symlink, ...). Called with no argument, returns the mask constant, which is how Perl code commonly compares against `S_IFREG`, `S_IFDIR`, and friends.

Examples

```
use Fcntl qw(:mode);
my $mode = (stat $path)[2];
print "is a directory\n" if S_IFMT($mode) == S_IFDIR;
```

S_ISREG

True if a stat-mode value names a file of the predicate's type: `S_ISREG` a regular file, `S_ISDIR` a directory, `S_ISLNK` a symbolic link, `S_ISSOCK` a socket, `S_ISBLK` a block device, `S_ISCHR` a character device, `S_ISFIFO` a named pipe.

Also available as `Fcntl::S_ISDIR`, `Fcntl::S_ISLNK`, `Fcntl::S_ISSOCK`, `Fcntl::S_ISBLK`, `Fcntl::S_ISCHR`, `Fcntl::S_ISFIFO`.

Synopsis

```
my $mode = (lstat $path)[2];
print "plain file\n" if S_ISREG($mode);
print "symlink\n"    if S_ISLNK($mode);
```

Returns true (1) when (`$mode & S_IFMT`) is the predicate's `S_IF*` value, false (`""`) otherwise. With no argument, warns "Use of uninitialized value" and returns false. Use `lstat` rather than `stat` for `S_ISLNK`: plain `stat` follows symlinks and reports the target's mode. `S_ISWHT` exists only where the system has whiteouts, which Linux does not.

5.2.44 File

Namespace containing the following modules:

- `File::Basename` - Parse file paths into directory, filename and suffix.
- `File::Compare` - Compare files or filehandles
- `File::Copy` - Copy files or filehandles
- `File::DosGlob` - A DOS-style `glob`, whose iterators live per call site.
- `File::Fetch` - A generic file fetching mechanism
- `File::Find` - Traverse a directory tree.
- `File::Glob` - POSIX pathname globbing - expand shell-style wildcard patterns into lists of filenames.

- `File::GlobMapper` - Extend File Glob to Allow Input and Output Files
- `File::Map` - Memory-mapped scalars: make a file look like an ordinary string.
- `File::Path` - Create or remove directory trees
- `File::ShareDir` - Locate per-dist and per-module shared files
- `File::Spec` - portably perform operations on file names
- `File::Temp` - return name and handle of a temporary file safely
- `File::Which` - Perl implementation of the which utility as an API
- `File::stat` - by-name interface to Perl's built-in stat() functions

File::Basename

Parse file paths into directory, filename and suffix.

SYNOPSIS

```
use File::Basename;

my ($name, $path, $suffix) = fileparse($fullname, @suffixlist);
my $name = fileparse($fullname, @suffixlist);

my $basename = basename($fullname, @suffixlist);
my $dirname  = dirname($fullname);
```

DESCRIPTION

These routines allow you to parse file paths into their directory, filename and suffix.

NOTE: `dirname()` and `basename()` emulate the behaviours, and quirks, of the shell and C functions of the same name. See each function's documentation for details. If your concern is just parsing paths it is safer to use *File::Spec*'s `splitpath()` and `splitdir()` methods.

It is guaranteed that

```
# Where $path_separator is / for Unix, \ for Windows, etc...
dirname($path) . $path_separator . basename($path);
```

is equivalent to the original path for all systems but VMS.

- `fileparse`

  ```
  my($filename, $dirs, $suffix) = fileparse($path);
  my($filename, $dirs, $suffix) = fileparse($path, @suffixes);
  my $filename                  = fileparse($path, @suffixes);
  ```

  The `fileparse()` routine divides a file path into its $dirs, $filename and (optionally) the filename $suffix.

$dirs contains everything up to and including the last directory separator in the $path including the volume (if applicable). The remainder of the $path is the $filename.

```
# On Unix returns ("baz", "/foo/bar/", "")
fileparse("/foo/bar/baz");

# On Windows returns ("baz", 'C:\foo\bar\', "")
fileparse('C:\foo\bar\baz');

# On Unix returns ("", "/foo/bar/baz/", "")
fileparse("/foo/bar/baz/");
```

If @suffixes are given each element is a pattern (either a string or a `qr//`) matched against the end of the $filename. The matching portion is removed and becomes the $suffix.

```
# On Unix returns ("baz", "/foo/bar/", ".txt")
fileparse("/foo/bar/baz.txt", qr/\.[^.]*/);
```

If type is non-Unix (see "fileparse_set_fstype") then the pattern matching for suffix removal is performed case-insensitively, since those systems are not case-sensitive when opening existing files.

You are guaranteed that `$dirs . $filename . $suffix` will denote the same location as the original $path.

- `basename`

```
my $filename = basename($path);
my $filename = basename($path, @suffixes);
```

This function is provided for compatibility with the Unix shell command `basename(1)`. It does **NOT** always return the file name portion of a path as you might expect. To be safe, if you want the file name portion of a path use `fileparse()`.

`basename()` returns the last level of a filepath even if the last level is clearly directory. In effect, it is acting like `pop()` for paths. This differs from `fileparse()`'s behaviour.

```
# Both return "bar"
basename("/foo/bar");
basename("/foo/bar/");
```

@suffixes work as in `fileparse()` except all regex metacharacters are quoted.

```
# These two function calls are equivalent.
my $filename = basename("/foo/bar/baz.txt",  ".txt");
my $filename = fileparse("/foo/bar/baz.txt", qr/\Q.txt\E/);
```

Also note that in order to be compatible with the shell command, `basename()` does not strip off a suffix if it is identical to the remaining characters in the filename.

- `dirname`

This function is provided for compatibility with the Unix shell command `dirname(1)` and has inherited some of its quirks. In spite of its name it does **NOT** always return the directory name as you might expect. To be safe, if you want the directory name of a path use `fileparse()`.

Only on VMS (where there is no ambiguity between the file and directory portions of a path) and AmigaOS (possibly due to an implementation quirk in this module) does `dirname()` work like `fileparse($path)`, returning just the $dirs.

```
# On VMS and AmigaOS
my $dirs = dirname($path);
```

When using Unix or MSDOS syntax this emulates the `dirname(1)` shell function which is subtly different from how `fileparse()` works. It returns all but the last level of a file path even if the last level is clearly a directory. In effect, it is not returning the directory portion but simply the path one level up acting like `chop()` for file paths.

Also unlike `fileparse()`, `dirname()` does not include a trailing slash on its returned path.

```
# returns /foo/bar.  fileparse() would return /foo/bar/
dirname("/foo/bar/baz");

# also returns /foo/bar despite the fact that baz is clearly a
# directory.  fileparse() would return /foo/bar/baz/
dirname("/foo/bar/baz/");

# returns '.'.  fileparse() would return 'foo/'
dirname("foo/");
```

Under VMS, if there is no directory information in the $path, then the current default device and directory is used.

- `fileparse_set_fstype`

```
my $type = fileparse_set_fstype();
my $previous_type = fileparse_set_fstype($type);
```

Normally File::Basename will assume a file path type native to your current operating system (ie. /foo/bar style on Unix, \foo\bar on Windows, etc...). With this function you can override that assumption.

Valid $types are “MacOS”, “VMS”, “AmigaOS”, “OS2”, “RISCOS”, “MSWin32”, “DOS” (also “MSDOS” for backwards bug compatibility), “Epoc” and “Unix” (all case-insensitive). If an unrecognized $type is given “Unix” will be assumed.

If you’ve selected VMS syntax, and the file specification you pass to one of these routines contains a “/”, they assume you are using Unix emulation and apply the Unix syntax rules instead, for that function call only.

SEE ALSO

dirname(1), basename(1), *File::Spec*

File::Compare

Compare files or filehandles

SYNOPSIS

```
use File::Compare;

if (compare("file1", "file2") == 0) {
    print "They're equal\n";
}
```

DESCRIPTION

The `File::Compare::compare` function compares the contents of two sources, each of which can be a file or a file handle. It is exported from `File::Compare` by default.

`File::Compare::cmp` is a synonym for `File::Compare::compare`. It is exported from `File::Compare` only by request.

`File::Compare::compare_text` does a line by line comparison of the two files. It stops as soon as a difference is detected. `compare_text()` accepts an optional third argument: This must be a CODE reference to a line comparison function, which returns `0` when both lines are considered equal. For example:

```
compare_text($file1, $file2)
```

is basically equivalent to

```
compare_text($file1, $file2, sub {$_[0] ne $_[1]} )
```

RETURN

`File::Compare::compare` and its sibling functions return `0` if the files are equal, `1` if the files are unequal, or `-1` if an error was encountered.

AUTHOR

`File::Compare` was written by Nick Ing-Simmons. Its original documentation was written by Chip Salzenberg.

File::Copy

Copy files or filehandles

SYNOPSIS

```
use File::Copy;

copy("sourcefile", "destinationfile") or die "Copy failed: $!";
copy("Copy.pm", \*STDOUT);
move("/dev1/sourcefile", "/dev2/destinationfile");

use File::Copy "cp";

my $n = FileHandle->new("/a/file", "r");
cp($n, "x");
```

DESCRIPTION

The File::Copy module provides two basic functions, `copy` and `move`, which are useful for getting the contents of a file from one place to another.

- copy

  The `copy` function takes two parameters: a file to copy from and a file to copy to. Either argument may be a string, a FileHandle reference or a FileHandle glob. Obviously, if the first argument is a filehandle of some sort, it will be read from, and if it is a file *name* it will be opened for reading. Likewise, the second argument will be written to. If the second argument does not exist but the parent directory does exist, then it will be created. Trying to copy a file into a non-existent directory is an error. Trying to copy a file on top of itself is also an error. `copy` will not overwrite read-only files.

  If the destination (second argument) already exists and is a directory, and the source (first argument) is not a filehandle, then the source file will be copied into the directory specified by the destination, using the same base name as the source file. It's a failure to have a filehandle as the source when the destination is a directory.

  Note that passing in files as handles instead of names may lead to loss of information on some operating systems; it is recommended that you use file names whenever possible. Files are opened in binary mode where applicable. To get a consistent behaviour when copying from a filehandle to a file, use `binmode` on the filehandle.

  An optional third parameter can be used to specify the buffer size used for copying. This is the number of bytes from the first file, that will be held in memory at any given time, before being written to the second file. The default buffer size depends upon the file, but will generally be the whole file (up to 2MB), or 1k for filehandles that do not reference files (eg. sockets).

  You may use the syntax `use File::Copy "cp"` to get at the `cp` alias for this function. The syntax is *exactly* the same. The behavior is nearly the same as well: as of version 2.15, `cp` will preserve the source file's permission bits like the shell utility `cp(1)` would do with default options, while `copy` uses the default permissions for the target file (which may depend on the process' `umask`, file ownership, inherited ACLs, etc.). That is, if the destination file already exists, `cp` will leave its permissions unchanged; otherwise the permissions are taken from the source file and modified

by the `umask`. If an error occurs in setting permissions, `cp` will return 0, regardless of whether the file was successfully copied.

- move

  The `move` function also takes two parameters: the current name and the intended name of the file to be moved. If the destination already exists and is a directory, and the source is not a directory, then the source file will be renamed into the directory specified by the destination.

  If possible, move() will simply rename the file. Otherwise, it copies the file to the new location and deletes the original. If an error occurs during this copy-and-delete process, you may be left with a (possibly partial) copy of the file under the destination name.

  You may use the `mv` alias for this function in the same way that you may use the `cp` alias for `copy`.

- syscopy

  File::Copy also provides the `syscopy` routine, which copies the file specified in the first parameter to the file specified in the second parameter, preserving OS-specific attributes and file structure. For Unix systems, this is equivalent to the simple `copy` routine, which doesn't preserve OS-specific attributes. For VMS systems, this calls the `rmscopy` routine (see below). For Win32 systems, this calls `Win32::CopyFile`.

  Special behaviour if `syscopy` is defined (OS/2, VMS and Win32):

  If both arguments to `copy` are not file handles, then `copy` will perform a "system copy" of the input file to a new output file, in order to preserve file attributes, indexed file structure, *etc.* The buffer size parameter is ignored. If either argument to `copy` is a handle to an opened file, then data is copied using Perl operators, and no effort is made to preserve file attributes or record structure.

  The system copy routine may also be called directly under VMS and OS/2 as `File::Copy::syscopy` (or under VMS as `File::Copy::rmscopy`, which is the routine that does the actual work for syscopy).

- rmscopy($from,$to[,$date_flag])

  The first and second arguments may be strings, typeglobs, typeglob references, or objects inheriting from IO::Handle; they are used in all cases to obtain the *filespec* of the input and output files, respectively. The name and type of the input file are used as defaults for the output file, if necessary.

  A new version of the output file is always created, which inherits the structure and RMS attributes of the input file, except for owner and protections (and possibly timestamps; see below). All data from the input file is copied to the output file; if either of the first two parameters to `rmscopy` is a file handle, its position is unchanged. (Note that this means a file handle pointing to the output file will be associated with an old version of that file after `rmscopy` returns, not the newly created version.)

  The third parameter is an integer flag, which tells `rmscopy` how to handle timestamps. If it is < 0, none of the input file's timestamps are propagated to the output file. If it is > 0, then it is interpreted as a bitmask: if bit 0 (the LSB) is set, then timestamps other than the revision date are propagated; if bit 1 is set, the revision date is propagated. If the third parameter to `rmscopy` is 0, then it behaves much like

the DCL COPY command: if the name or type of the output file was explicitly specified, then no timestamps are propagated, but if they were taken implicitly from the input filespec, then all timestamps other than the revision date are propagated. If this parameter is not supplied, it defaults to 0.

`rmscopy` is VMS specific and cannot be exported; it must be referenced by its full name, e.g.:

```
File::Copy::rmscopy($from, $to) or die $!;
```

Like `copy`, `rmscopy` returns 1 on success. If an error occurs, it sets `$!`, deletes the output file, and returns 0.

RETURN

All functions return 1 on success, 0 on failure. `$!` will be set if an error was encountered.

NOTES

Before calling copy() or move() on a filehandle, the caller should close or flush() the file to avoid writes being lost. Note that this is the case even for move(), because it may actually copy the file, depending on the OS-specific implementation, and the underlying filesystem(s).

AUTHOR

File::Copy was written by Aaron Sherman *ajs@ajs.com* in 1995, and updated by Charles Bailey *bailey@newman.upenn.edu* in 1996.

File::DosGlob

A DOS-style `glob`, whose iterators live per call site.

```
use File::DosGlob 'glob';                 # overrides glob() in this␣
↪package

print join(" ", map { s{.*/}{}r } glob("$d/*.txt")), "\n";   # b.txt␣
↪a.txt
while (my $f = glob("$d/*.log")) { print "log: $f\n" }       # one␣
↪iterator per call site
```

It understands DOS-style paths as well as Unix ones - drive letters, `\` as a separator, spaces in names - and, unlike the built-in `glob`, returns names in directory order, unsorted. Importing `glob` replaces it in the importing package; `use File::DosGlob 'GLOBAL_glob'` replaces it everywhere.

This is a native implementation of the XS half of File::DosGlob 1.12: `File/DosGlob.pm` is used unchanged and does the globbing, keeping each call site's iterator in `%File::DosGlob::entries`, keyed by the op that called `glob`. The XS half supplies that key, `_callsite()`, and an op-free hook that deletes an op's entry when the op is freed, so a new op at the same address does not inherit it.

Differences from upstream

None.

File::Fetch

A generic file fetching mechanism

SYNOPSIS

```
use File::Fetch;

### build a File::Fetch object ###
my $ff = File::Fetch->new(uri => 'http://some.where.com/dir/a.txt');

### fetch the uri to cwd() ###
my $where = $ff->fetch() or die $ff->error;

### fetch the uri to /tmp ###
my $where = $ff->fetch( to => '/tmp' );

### parsed bits from the uri ###
$ff->uri;
$ff->scheme;
$ff->host;
$ff->path;
$ff->file;
```

DESCRIPTION

File::Fetch is a generic file fetching mechanism.

It allows you to fetch any file pointed to by a `ftp`, `http`, `file`, `git` or `rsync` uri by a number of different means.

See the `HOW IT WORKS` section further down for details.

ACCESSORS

A `File::Fetch` object has the following accessors

- $ff->uri

  The uri you passed to the constructor

- $ff->scheme

  The scheme from the uri (like 'file', 'http', etc)

- $ff->host

  The hostname in the uri. Will be empty if host was originally 'localhost' for a 'file://' url.

- $ff->vol

  On operating systems with the concept of a volume the second element of a file:// is considered to the be volume specification for the file. Thus on Win32 this routine returns the volume, on other operating systems this returns nothing.

  On Windows this value may be empty if the uri is to a network share, in which case the 'share' property will be defined. Additionally, volume specifications that use '|' as ':' will be converted on read to use ':'.

  On VMS, which has a volume concept, this field will be empty because VMS file specifications are converted to absolute UNIX format and the volume information is transparently included.

- $ff->share

  On systems with the concept of a network share (currently only Windows) returns the sharename from a file:/// url. On other operating systems returns empty.

- $ff->path

  The path from the uri, will be at least a single '/'.

- $ff->file

  The name of the remote file. For the local file name, the result of $ff->output_file will be used.

- $ff->file_default

  The name of the default local file, that $ff->output_file falls back to if it would otherwise return no filename. For example when fetching a URI like http://www.abc.net.au/ the contents retrieved may be from a remote file called 'index.html'. The default value of this attribute is literally 'file_default'.

- $ff->output_file

  The name of the output file. This is the same as $ff->file, but any query parameters are stripped off. For example:

  ```
  http://example.com/index.html?x=y
  ```

  would make the output file be `index.html` rather than `index.html?x=y`.

METHODS

$ff = File::Fetch->new(uri => 'http://some.where.com/dir/file.txt');

Parses the uri and creates a corresponding File::Fetch::Item object, that is ready to be `fetch`ed and returns it.

Returns false on failure.

$where = $ff->fetch([to => /my/output/dir/ | \$scalar])

Fetches the file you requested and returns the full path to the file.

By default it writes to `cwd()`, but you can override that by specifying the `to` argument:

```
### file fetch to /tmp, full path to the file in $where
$where = $ff->fetch( to => '/tmp' );

### file slurped into $scalar, full path to the file in $where
### file is downloaded to a temp directory and cleaned up at exit␣
 ↪time
$where = $ff->fetch( to => \$scalar );
```

Returns the full path to the downloaded file on success, and false on failure.

$ff->error([BOOL])

Returns the last encountered error as string. Pass it a true value to get the `Carp::longmess()` output instead.

HOW IT WORKS

File::Fetch is able to fetch a variety of uris, by using several external programs and modules.

Below is a mapping of what utilities will be used in what order for what schemes, if available:

```
file    => LWP, lftp, file
http    => LWP, HTTP::Tiny, wget, curl, lftp, fetch, HTTP::Lite,␣
 ↪lynx, iosock
ftp     => LWP, Net::FTP, wget, curl, lftp, fetch, ncftp, ftp
rsync   => rsync
git     => git
```

If you'd like to disable the use of one or more of these utilities and/or modules, see the `$BLACKLIST` variable further down.

If a utility or module isn't available, it will be marked in a cache (see the `$METHOD_FAIL` variable further down), so it will not be tried again. The `fetch` method will only fail when all options are exhausted, and it was not able to retrieve the file.

The `fetch` utility is available on FreeBSD. NetBSD and Dragonfly BSD may also have it from `pkgsrc`. We only check for `fetch` on those three platforms.

`iosock` is a very limited *IO::Socket::INET* based mechanism for retrieving `http` schemed urls. It doesn't follow redirects for instance.

`git` only supports `git://` style urls.

A special note about fetching files from an ftp uri:

By default, all ftp connections are done in passive mode. To change that, see the `$FTP_PASSIVE` variable further down.

Furthermore, ftp uris only support anonymous connections, so no named user/password pair can be passed along.

`/bin/ftp` is blacklisted by default; see the `$BLACKLIST` variable further down.

GLOBAL VARIABLES

The behaviour of File::Fetch can be altered by changing the following global variables:

$File::Fetch::FROM_EMAIL

This is the email address that will be sent as your anonymous ftp password.

Default is `File-Fetch@example.com`.

$File::Fetch::USER_AGENT

This is the useragent as `LWP` will report it.

Default is `File::Fetch/$VERSION`.

$File::Fetch::FTP_PASSIVE

This variable controls whether the environment variable `FTP_PASSIVE` and any passive switches to commandline tools will be set to true.

Default value is 1.

Note: When $FTP_PASSIVE is true, `ncftp` will not be used to fetch files, since passive mode can only be set interactively for this binary

$File::Fetch::TIMEOUT

When set, controls the network timeout (counted in seconds).

Default value is 0.

$File::Fetch::WARN

This variable controls whether errors encountered internally by `File::Fetch` should be `carp`'d or not.

Set to false to silence warnings. Inspect the output of the `error()` method manually to see what went wrong.

Defaults to `true`.

$File::Fetch::DEBUG

This enables debugging output when calling commandline utilities to fetch files. This also enables `Carp::longmess` errors, instead of the regular `carp` errors.

Good for tracking down why things don't work with your particular setup.

Default is 0.

$File::Fetch::BLACKLIST

This is an array ref holding blacklisted modules/utilities for fetching files with.

To disallow the use of, for example, `LWP` and `Net::FTP`, you could set $File::Fetch::BLACKLIST to:

```
$File::Fetch::BLACKLIST = [qw|lwp netftp|]
```

The default blacklist is [qw|ftp|], as `/bin/ftp` is rather unreliable.

See the note on `MAPPING` below.

$File::Fetch::METHOD_FAIL

This is a hashref registering what modules/utilities were known to fail for fetching files (mostly because they weren't installed).

You can reset this cache by assigning an empty hashref to it, or individually remove keys.

See the note on `MAPPING` below.

MAPPING

Here's a quick mapping for the utilities/modules, and their names for the $BLACKLIST, $METHOD_FAIL and other internal functions.

```
LWP          => lwp
HTTP::Lite   => httplite
HTTP::Tiny   => httptiny
Net::FTP     => netftp
wget         => wget
lynx         => lynx
ncftp        => ncftp
ftp          => ftp
curl         => curl
rsync        => rsync
lftp         => lftp
fetch        => fetch
IO::Socket   => iosock
```

FREQUENTLY ASKED QUESTIONS

So how do I use a proxy with File::Fetch?

`File::Fetch` currently only supports proxies with LWP::UserAgent. You will need to set your environment variables accordingly. For example, to use an ftp proxy:

```
$ENV{ftp_proxy} = 'foo.com';
```

Refer to the LWP::UserAgent manpage for more details.

I used 'lynx' to fetch a file, but its contents is all wrong!

`lynx` can only fetch remote files by dumping its contents to `STDOUT`, which we in turn capture. If that content is a 'custom' error file (like, say, a `404 handler`), you will get that contents instead.

Sadly, `lynx` doesn't support any options to return a different exit code on non-`200 OK` status, giving us no way to tell the difference between a 'successful' fetch and a custom error page.

Therefor, we recommend to only use `lynx` as a last resort. This is why it is at the back of our list of methods to try as well.

Files I'm trying to fetch have reserved characters or non-ASCII characters in them. What do I do?

`File::Fetch` is relatively smart about things. When trying to write a file to disk, it removes the `query parameters` (see the `output_file` method for details) from the file name before creating it. In most cases this suffices.

If you have any other characters you need to escape, please install the `URI::Escape` module from CPAN, and pre-encode your URI before passing it to `File::Fetch`. You can read about the details of URIs and URI encoding here:

https://datatracker.ietf.org/doc/html/rfc2396

TODO

- Implement $PREFER_BIN

  To indicate to rather use commandline tools than modules

BUG REPORTS

Please report bugs or other issues to <bug-file-fetch@rt.cpan.org<gt>.

AUTHOR

This module by Jos Boumans kane@cpan.org.

COPYRIGHT

This library is free software; you may redistribute and/or modify it under the same terms as Perl itself.

File::Find

Traverse a directory tree.

SYNOPSIS

```
use File::Find;
find(\&wanted, @directories_to_search);
sub wanted { ... }

use File::Find;
finddepth(\&wanted, @directories_to_search);
sub wanted { ... }

use File::Find;
find({ wanted => \&process, follow => 1 }, '.');
```

DESCRIPTION

These are functions for searching through directory trees doing work on each file found similar to the Unix find(1) command. `File::Find` exports two functions, `find` and `finddepth`. They work similarly but have subtle differences.

- **find**

  ```
  find(\&wanted,  @directories);
  find(\%options, @directories);
  ```

  `find()` does a depth-first search over the given `@directories` in the order they are given. For each file or directory found, it calls the `&wanted` subroutine. (See below for details on how to use the `&wanted` function). Additionally, for each directory found, it will `chdir()` into that directory and continue the search, invoking the `&wanted` function on each file or subdirectory in the directory.

- **finddepth**

  ```
  finddepth(\&wanted,  @directories);
  finddepth(\%options, @directories);
  ```

  `finddepth()` works just like `find()` except that it invokes the `&wanted` function for a directory *after* invoking it for the directory's contents. It does a postorder traversal instead of a preorder traversal, working from the bottom of the directory tree up where `find()` works from the top of the tree down.

Despite the name of the `finddepth()` function, both `find()` and `finddepth()` perform a depth-first search of the directory hierarchy.

%options

The first argument to `find()` is either a code reference to your `&wanted` function, or a hash reference describing the operations to be performed for each file. The code reference is described in "The wanted function" below.

Here are the possible **keys** for the hash:

- `wanted`

The value should be a code reference. This code reference is described in "The wanted function" below. The &wanted subroutine is mandatory.

- bydepth

  Reports the name of a directory only AFTER all its entries have been reported. Entry point finddepth() is a shortcut for specifying { bydepth => 1 } in the first argument of find().

- preprocess

  The value should be a code reference. This code reference is used to preprocess the current directory. The name of the currently processed directory is in $File::Find::dir. Your preprocessing function is called after readdir(), but before the loop that calls the wanted() function. It is called with a list of strings (actually file/directory names) and is expected to return a list of strings. The code can be used to sort the file/directory names alphabetically, numerically, or to filter out directory entries based on their name alone. When follow or follow_fast are in effect, preprocess is a no-op.

- postprocess

  The value should be a code reference. It is invoked just before leaving the currently processed directory. It is called in void context with no arguments. The name of the current directory is in $File::Find::dir. This hook is handy for summarizing a directory, such as calculating its disk usage. When follow or follow_fast are in effect, postprocess is a no-op.

- follow

  Causes symbolic links to be followed. Since directory trees with symbolic links (followed) may contain files more than once and may even have cycles, a hash has to be built up with an entry for each file. This might be expensive both in space and time for a large directory tree. See "follow_fast" and "follow_skip" below. If either follow or follow_fast is in effect:

  – It is guaranteed that an lstat has been called before the user's wanted() function is called. This enables fast file checks involving _. Note that this guarantee no longer holds if follow or follow_fast are not set.

  – There is a variable $File::Find::fullname which holds the absolute pathname of the file with all symbolic links resolved. If the link is a dangling symbolic link, then fullname will be set to undef.

- follow_fast

  This is similar to follow except that it may report some files more than once. It does detect cycles, however. Since only symbolic links have to be hashed, this is much cheaper both in space and time. If processing a file more than once (by the user's wanted() function) is worse than just taking time, the option follow should be used.

- follow_skip

  follow_skip==1, which is the default, causes all files which are neither directories nor symbolic links to be ignored if they are about to be processed a second time. If a directory or a symbolic link are about to be processed a second time, File::Find dies.

`follow_skip==0` causes `File::Find` to die if any file is about to be processed a second time.

`follow_skip==2` causes `File::Find` to ignore any duplicate files and directories but to proceed normally otherwise.

- `dangling_symlinks`

  Specifies what to do with symbolic links whose target doesn't exist. If true and a code reference, will be called with the symbolic link name and the directory it lives in as arguments. Otherwise, if true and warnings are on, a warning of the form `"symbolic_link_name is a dangling symbolic link\n"` will be issued. If false, the dangling symbolic link will be silently ignored.

- `no_chdir`

  Does not `chdir()` to each directory as it recurses. The `wanted()` function will need to be aware of this, of course. In this case, `$_` will be the same as `$File::Find::name`.

- `untaint`

  If find is used in *taint-mode* (`-T` command line switch or `if EUID != UID` or `if EGID != GID`), then internally directory names have to be untainted before they can be `chdir`'d to. Therefore they are checked against a regular expression `untaint_pattern`. Note that all names passed to the user's `wanted()` function are still tainted. If this option is used while not in taint-mode, `untaint` is a no-op.

- `untaint_pattern`

  See above. This should be set using the `qr` quoting operator. The default is set to `qr|^([-+@\w./]+)$|`. Note that the parentheses are vital.

- `untaint_skip`

  If set, a directory which fails the `untaint_pattern` is skipped, including all its sub-directories. The default is to `die` in such a case.

The wanted function

The `wanted()` function does whatever verifications you want on each file and directory. Note that despite its name, the `wanted()` function is a generic callback function, and does **not** tell `File::Find` if a file is "wanted" or not. In fact, its return value is ignored.

The `wanted` function takes no arguments but rather does its work through a collection of variables.

- `$File::Find::dir` is the current directory name,
- `$_` is the current filename within that directory
- `$File::Find::name` is the complete pathname to the file.

The above variables have all been localized and may be changed without affecting data outside of the wanted function.

For example, when examining the file `/some/path/foo.ext` you will have:

```
$File::Find::dir  = /some/path/
$_                = foo.ext
$File::Find::name = /some/path/foo.ext
```

You are `chdir()`'d to `$File::Find::dir` when the function is called, unless `no_chdir` was specified. Note that when changing to directories is in effect, the root directory (`/`) is a somewhat special case inasmuch as the concatenation of `$File::Find::dir`, `'/'` and `$_` is not literally equal to `$File::Find::name`. The table below summarizes all variants:

```
             $File::Find::name  $File::Find::dir  $_
default      /                  /                 .
no_chdir=>0  /etc               /                 etc
             /etc/x             /etc              x

no_chdir=>1  /                  /                 /
             /etc               /                 /etc
             /etc/x             /etc              /etc/x
```

When `follow` or `follow_fast` are in effect, there is also a `$File::Find::fullname`. The function may set `$File::Find::prune` to prune the tree unless `bydepth` was specified. Unless `follow` or `follow_fast` is specified, for compatibility reasons (`find.pl`, find2perl) there are in addition the following globals available: `$File::Find::topdir`, `$File::Find::topdev`, `$File::Find::topino`, `$File::Find::topmode` and `$File::Find::topnlink`.

This library is useful for the `find2perl` tool (distributed with the App::find2perl CPAN module), which when fed:

```
find2perl / -name .nfs\* -mtime +7 \
  -exec rm -f {} \; -o -fstype nfs -prune
```

produces something like:

```
sub wanted {
   /^\.nfs.*\z/s &&
   (($dev, $ino, $mode, $nlink, $uid, $gid) = lstat($_)) &&
   int(-M _) > 7 &&
   unlink($_)
   ||
   ($nlink || (($dev, $ino, $mode, $nlink, $uid, $gid) = lstat($_))
↪&&
   $dev < 0 &&
   ($File::Find::prune = 1);
}
```

Notice the `_` in the above `int(-M _)`: the `_` is a magical filehandle that caches the information from the preceding `stat()`, `lstat()`, or filetest.

Here's another interesting wanted function. It will find all symbolic links that don't resolve:

```
sub wanted {
     -l && !-e && print "bogus link: $File::Find::name\n";
}
```

Note that you may mix directories and (non-directory) files in the list of directories to be searched by the `wanted()` function.

```
find(\&wanted, "./foo", "./bar", "./baz/epsilon");
```

In the example above, no file in `./baz/` other than `./baz/epsilon` will be evaluated by `wanted()`.

See also the script `pfind` on CPAN for a nice application of this module.

WARNINGS

If you run your program with the `-w` switch, or if you use the `warnings` pragma, File::Find will report warnings for several weird situations. You can disable these warnings by putting the statement

```
no warnings 'File::Find';
```

in the appropriate scope. See *warnings* for more info about lexical warnings.

BUGS AND CAVEATS

- `$dont_use_nlink`

  You can set the variable `$File::Find::dont_use_nlink` to `0` if you are sure the filesystem you are scanning reflects the number of subdirectories in the parent directory's `nlink` count.

  If you do set `$File::Find::dont_use_nlink` to 0, you may notice an improvement in speed at the risk of not recursing into subdirectories if a filesystem doesn't populate `nlink` as expected.

  `$File::Find::dont_use_nlink` now defaults to 1 on all platforms.

- Symlinks

  Be aware that the option to follow symbolic links can be dangerous. Depending on the structure of the directory tree (including symbolic links to directories) you might traverse a given (physical) directory more than once (only if `follow_fast` is in effect). Furthermore, deleting or changing files in a symbolically linked directory might cause very unpleasant surprises, since you delete or change files in an unknown directory.

HISTORY

`File::Find` used to produce incorrect results if called recursively. During the development of perl 5.8 this bug was fixed. The first fixed version of `File::Find` was 1.01.

SEE ALSO

find(1), find2perl

File::Glob

POSIX pathname globbing - expand shell-style wildcard patterns into lists of filenames.

`File::Glob` is what backs Perl's built-in `glob` operator and its angle-bracket form `<*.pl>`. Give it a pattern with shell metacharacters (`*`, `?`, `[...]`, `{a,b,c}`, `~user`) and you get back the list of paths on disk that match. The same machinery is exposed directly as `bsd_glob` when you want explicit control over flags.

The mental model is simple: a pattern goes in, a list of real filenames comes out. Behaviour is tuned by a bitmask of `GLOB_*` flags - tilde expansion, brace expansion, case folding, sort order, and what happens when nothing matches are all flag-controlled. Every flag has a named constant (`GLOB_TILDE`, `GLOB_BRACE`, `GLOB_NOCASE`, `GLOB_NOMAGIC`, `GLOB_CSH`, etc.) and they OR together.

Three call styles, one engine: the built-in `glob`, the angle-bracket form `<*.pl>` and `bsd_glob` all run the same matcher, so they agree on flag handling, on sort order and on what `$File::Glob::DEFAULT_FLAGS` means. None of them forks a shell.

```
use File::Glob ':bsd_glob';
my @pl  = <*.pl>;                          # angle-bracket form
my @c   = bsd_glob('*.[ch]');              # function form,␣
 ↪default flags
my @hd  = bsd_glob('~gnat', GLOB_TILDE | GLOB_ERR);
```

The default flag set is `GLOB_CSH` - brace expansion, tilde expansion, backslash quoting, alphabetical sort, and the `GLOB_NOMAGIC` fallback that returns the literal pattern when it has no metacharacters and doesn't match. Two import tags nudge the default: `:nocase` turns `GLOB_NOCASE` on globally, `:case` turns it off.

This is a native implementation of the XS half of File::Glob 1.44, the BSD glob engine included; `File/Glob.pm` is used unchanged. `use File::Glob ':glob'` exports `bsd_glob`, `GLOB_ERROR` and the constants; `':bsd_glob'` exports the same and also replaces the caller's `glob` with one that takes its pattern whole instead of splitting it on whitespace; `':globally'` makes `csh_glob` the `glob` of every package.

Differences from upstream

None.

Functions

Globbing

bsd_glob_override

The `glob` that `use File::Glob ':bsd_glob'` installs in the calling package, replacing the built-in.

bsd_glob

Expand a shell-style pattern and return the list of matching paths.

csh_glob

Expand a pattern using the default flag set - the engine behind Perl's built-in `glob`.

GLOB_ERROR

Report the error status of the most recent `bsd_glob` call.

bsd_glob

Expand a shell-style pattern and return the list of matching paths.

Synopsis

```
my @list    = bsd_glob('*.[ch]');
my @homes   = bsd_glob('~gnat', GLOB_TILDE | GLOB_ERR);
my @latest  = bsd_glob('src/*.pl', GLOB_NOSORT);
```

What you get back

A list of filenames, possibly empty. In scalar context you get the count of matches. Filenames are returned as plain strings in the order determined by the flags (alphabetical under the default `GLOB_CSH`, unsorted under `GLOB_NOSORT`).

When the pattern matches nothing and neither `GLOB_NOCHECK` nor `GLOB_NOMAGIC` is in effect, the list is empty. With `GLOB_NOCHECK` the original pattern is returned as the sole element; with `GLOB_NOMAGIC` the pattern is returned only when it contains no metacharacters.

Examples

```
use File::Glob ':bsd_glob';
my @perl = bsd_glob('*.pl');          # ('a.pl', 'b.pl', ...)
```

```
# Brace expansion - default flags include GLOB_BRACE
my @src = bsd_glob('{lib,t}/*.pm');   # files under lib/ and t/
```

```
# Tilde expansion to another user's home
my ($home) = bsd_glob('~gnat', GLOB_TILDE);
```

```
# Case-insensitive match
my @readmes = bsd_glob('readme*', GLOB_NOCASE);
```

Edge cases

- No second argument: flags come from `$File::Glob::DEFAULT_FLAGS` (initialised to `GLOB_CSH`).
- `GLOB_APPEND`, `GLOB_DOOFFS`, `GLOB_ALTDIRFUNC` and `GLOB_MAGCHAR` are cleared from the caller's flags, as upstream clears them: they only make sense to a C caller.
- A pattern containing an embedded NUL byte returns the empty list, sets `$!` to ENOENT and warns "Invalid \0 character in pattern for bsd_glob" (a `syscalls` warning).
- Called with no pattern at all, croaks "Usage: File::Glob::bsd_glob(pattern_sv, ...)".
- Under taint mode (`-T`) every returned name is tainted, as with every other glob form: filenames come from outside the program.
- Brace alternatives are globbed one after the other, each sorted on its own: `bsd_glob('{b,a}.*')` lists the `b` names before the `a` names. A lone `{}` is kept as it is.
- In a set, `!` negates (`[!a]*`); `^` is an ordinary member.

See also

- `csh_glob` - the form Perl's built-in `glob` uses internally.
- `GLOB_ERROR` - non-zero after a traversal error.
- `GLOB_CSH` - the default flag bundle.
- `GLOB_NOCHECK` - return the pattern itself when nothing matches.

csh_glob

Expand a pattern using the default flag set - the engine behind Perl's built-in `glob`.

Synopsis

```
use File::Glob qw(csh_glob);
my @files = csh_glob('*.pl');
```

What you get back

A list of matching filenames. Flags always come from `$File::Glob::DEFAULT_FLAGS` (initially `GLOB_CSH`); `csh_glob` takes no flag argument. Sort order, brace expansion, tilde expansion, and the rest follow whatever `DEFAULT_FLAGS` currently holds.

You almost certainly want `bsd_glob` or the angle-bracket operator `<...>` instead. `csh_glob` exists because Perl's built-in `glob` dispatches through it; calling it directly is documented as "don't unless you know what you're doing."

Examples

```
use File::Glob qw(csh_glob);
my @matches = csh_glob('*.txt');        # uses DEFAULT_FLAGS
```

```
# Flip case sensitivity globally, then call
use File::Glob qw(csh_glob :nocase);
my @any = csh_glob('ReadMe*');          # matches README, readme,␣
↪ReadMe, ...
```

Edge cases

- Called with no argument: the pattern comes from `$_`, as it does for the built-in `glob`.
- The pattern is split on whitespace into several sub-patterns, so `csh_glob('*.c *.h')` globs both. Quote a token, or use `bsd_glob`, when the pattern itself contains a space.
- A pattern containing an embedded NUL byte warns and returns the empty list.

Differences from upstream

Fully compatible with upstream: patterns are tokenised on whitespace (honouring quotes and backslash escapes), and the per-op cache gives scalar context one name per call.

See also

- `bsd_glob` - the public, flag-accepting equivalent; prefer it.
- `GLOB_CSH` - the flag set `DEFAULT_FLAGS` is seeded with.
- `:nocase` / `:case` - import tags that toggle `GLOB_NOCASE` in `DEFAULT_FLAGS`.

GLOB_ERROR

Report the error status of the most recent `bsd_glob` call.

Synopsis

```
my @files = bsd_glob('~nobody', GLOB_TILDE | GLOB_ERR);
if (GLOB_ERROR) {
    warn "glob failed: $!";
}
```

What you get back

An integer. Zero means the last `bsd_glob` call completed without error. A non-zero value signals that an error interrupted traversal: `GLOB_NOSPACE` for allocation failure, `GLOB_ABEND` for a stopped traversal. When non-zero, `$!` carries the underlying errno.

Examples

```
use File::Glob ':bsd_glob';
my @out = bsd_glob('/root/*', GLOB_ERR);   # root-only dir
print "error\n" if GLOB_ERROR;              # usually prints under
 ↪non-root
```

Edge cases

- Reading `GLOB_ERROR` without a preceding glob returns `0`.
- It is `0` after a glob that completed, with or without matches.
- An unreadable directory stops a glob made with `GLOB_ERR` (`GLOB_ABEND`, -2); without `GLOB_ERR` one that cannot be read for want of permission, or that does not exist, is skipped.
- The built-in `glob` and `<*.c>` set it too.

See also

- `bsd_glob` - the call whose outcome this function reports.
- `GLOB_ERR` - the flag that makes `bsd_glob` stop (rather than skip) on unreadable directories.
- `GLOB_NOSPACE`, `GLOB_ABEND` - the two non-zero values it returns.

File::GlobMapper

Extend File Glob to Allow Input and Output Files

SYNOPSIS

```
use File::GlobMapper qw( globmap );

my $aref = globmap $input => $output
    or die $File::GlobMapper::Error ;

my $gm = File::GlobMapper->new( $input => $output )
    or die $File::GlobMapper::Error ;
```

DESCRIPTION

This module needs Perl5.005 or better.

This module takes the existing `File::Glob` module as a starting point and extends it to allow new filenames to be derived from the files matched by `File::Glob`.

This can be useful when carrying out batch operations on multiple files that have both an input filename and output filename and the output file can be derived from the input filename. Examples of operations where this can be useful include, file renaming, file copying and file compression.

Behind The Scenes

To help explain what `File::GlobMapper` does, consider what code you would write if you wanted to rename all files in the current directory that ended in `.tar.gz` to `.tgz`. So say these files are in the current directory

```
alpha.tar.gz
beta.tar.gz
gamma.tar.gz
```

and they need renamed to this

```
alpha.tgz
beta.tgz
gamma.tgz
```

Below is a possible implementation of a script to carry out the rename (error cases have been omitted)

```
foreach my $old ( glob "*.tar.gz" )
{
    my $new = $old;
    $new =~ s#(.*)\.tar\.gz$#$1.tgz# ;

    rename $old => $new
        or die "Cannot rename '$old' to '$new': $!\n;
}
```

Notice that a file glob pattern `*.tar.gz` was used to match the `.tar.gz` files, then a fairly similar regular expression was used in the substitute to allow the new filename to be created.

Given that the file glob is just a cut-down regular expression and that it has already done a lot of the hard work in pattern matching the filenames, wouldn't it be handy to be able to use the patterns in the fileglob to drive the new filename?

Well, that's *exactly* what `File::GlobMapper` does.

Here is same snippet of code rewritten using `globmap`

```
for my $pair (globmap '<*.tar.gz>' => '<#1.tgz>' )
{
```

(continues on next page)

(continued from previous page)

```
    my ($from, $to) = @$pair;
    rename $from => $to
        or die "Cannot rename '$old' to '$new': $!\n;
}
```

So how does it work?

Behind the scenes the `globmap` function does a combination of a file glob to match existing filenames followed by a substitute to create the new filenames.

Notice how both parameters to `globmap` are strings that are delimited by <>. This is done to make them look more like file globs - it is just syntactic sugar, but it can be handy when you want the strings to be visually distinctive. The enclosing <> are optional, so you don't have to use them - in fact the first thing globmap will do is remove these delimiters if they are present.

The first parameter to `globmap`, `*.tar.gz`, is an *Input File Glob*. Once the enclosing "< ... >" is removed, this is passed (more or less) unchanged to `File::Glob` to carry out a file match.

Next the fileglob `*.tar.gz` is transformed behind the scenes into a full Perl regular expression, with the additional step of wrapping each transformed wildcard metacharacter sequence in parenthesis.

In this case the input fileglob `*.tar.gz` will be transformed into this Perl regular expression

```
([^/]*)\.tar\.gz
```

Wrapping with parenthesis allows the wildcard parts of the Input File Glob to be referenced by the second parameter to `globmap`, `#1.tgz`, the *Output File Glob*. This parameter operates just like the replacement part of a substitute command. The difference is that the `#1` syntax is used to reference sub-patterns matched in the input fileglob, rather than the `$1` syntax that is used with perl regular expressions. In this case `#1` is used to refer to the text matched by the `*` in the Input File Glob. This makes it easier to use this module where the parameters to `globmap` are typed at the command line.

The final step involves passing each filename matched by the `*.tar.gz` file glob through the derived Perl regular expression in turn and expanding the output fileglob using it.

The end result of all this is a list of pairs of filenames. By default that is what is returned by `globmap`. In this example the data structure returned will look like this

```
( ['alpha.tar.gz' => 'alpha.tgz'],
  ['beta.tar.gz'  => 'beta.tgz' ],
  ['gamma.tar.gz' => 'gamma.tgz']
)
```

Each pair is an array reference with two elements - namely the *from* filename, that `File::Glob` has matched, and a *to* filename that is derived from the *from* filename.

Limitations

`File::GlobMapper` has been kept simple deliberately, so it isn't intended to solve all filename mapping operations. Under the hood `File::Glob` (or for older versions of Perl, `File::BSDGlob`) is used to match the files, so you will never have the flexibility of full Perl regular expression.

Input File Glob

The syntax for an Input FileGlob is identical to `File::Glob`, except for the following

1. No nested {}
2. Whitespace does not delimit fileglobs.
3. The use of parenthesis can be used to capture parts of the input filename.
4. If an Input glob matches the same file more than once, only the first will be used.

The syntax

- **~**
- **~user**
- **.**

  Matches a literal '.'. Equivalent to the Perl regular expression

  ```
  \.
  ```

- ***

  Matches zero or more characters, except '/'. Equivalent to the Perl regular expression

  ```
  [^/]*
  ```

- **?**

  Matches zero or one character, except '/'. Equivalent to the Perl regular expression

  ```
  [^/]?
  ```

- \

  Backslash is used, as usual, to escape the next character.

- **[]**

  Character class.

- **{,}**

  Alternation

- **()**

  Capturing parenthesis that work just like perl

Any other character it taken literally.

Output File Glob

The Output File Glob is a normal string, with 2 glob-like features.

The first is the '*' metacharacter. This will be replaced by the complete filename matched by the input file glob. So

```
*.c *.Z
```

The second is

Output FileGlobs take the

- "*"

  The "*" character will be replaced with the complete input filename.

- #1

  Patterns of the form /#\d/ will be replaced with the

Returned Data

EXAMPLES

A Rename script

Below is a simple "rename" script that uses `globmap` to determine the source and destination filenames.

```
use File::GlobMapper qw(globmap) ;
use File::Copy;

die "rename: Usage rename 'from' 'to'\n"
    unless @ARGV == 2 ;

my $fromGlob = shift @ARGV;
my $toGlob   = shift @ARGV;

my $pairs = globmap($fromGlob, $toGlob)
    or die $File::GlobMapper::Error;

for my $pair (@$pairs)
{
    my ($from, $to) = @$pair;
    move $from => $to ;
}
```

Here is an example that renames all c files to cpp.

```
$ rename '*.c' '#1.cpp'
```

A few example globmaps

Below are a few examples of globmaps

To copy all your .c file to a backup directory

```
'</my/home/*.c>'    '</my/backup/#1.c>'
```

If you want to compress all

```
'</my/home/*.[ch]>'    '<*.gz>'
```

To uncompress

```
'</my/home/*.[ch].gz>'    '</my/home/#1.#2>'
```

SEE ALSO

File::Glob

AUTHOR

The *File::GlobMapper* module was written by Paul Marquess, `pmqs@cpan.org`.

COPYRIGHT AND LICENSE

Copyright (c) 2005 Paul Marquess. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

File::Map

Memory-mapped scalars: make a file look like an ordinary string.

`File::Map` attaches a file - or a fresh anonymous region of memory - to a Perl scalar. Reads and writes to the scalar go straight to the kernel's mapping: the file appears in your variable as a string, and changes propagate back without you ever calling `read` or `write`. A multi-gigabyte file costs almost nothing to open this way, because pages are faulted in lazily as you touch them.

Once a scalar is mapped it behaves like a regular string for the operations that matter: `substr`, regex match and substitution, `index`, `length`, and slicing. Direct assignment (`$map = "..."`) is caught and fixed up so you never corrupt the mapping, though `substr` is the recommended way to write. The mapping is released automatically when the scalar goes out of scope; `unmap` is available for explicit control.

Synopsis

```
use File::Map qw(map_file map_anonymous unmap);

map_file my $map, 'data.bin', '+<';   # read/write
$map =~ s/foo/bar/g;                  # edits go to disk
```

(continues on next page)

(continued from previous page)

```
substr $map, 0, 4, "HEAD";            # in-place write

map_anonymous my $buf, 4096, 'shared'; # scratch region
unmap $map;                             # optional
```

Choosing a mapping function

- `map_file` - give a filename and a mode, get a mapped scalar. The common case.
- `map_handle` - same idea, but you already have an open filehandle.
- `map_anonymous` - no file; a scratch region shared with forked children or private to the process.
- `sys_map` - low-level wrapper around raw `mmap(2)` flags for when the convenience functions don't fit.

Managing a live mapping

- `sync` - flush pending writes back to disk.
- `advise` - hint the kernel about access patterns (`"sequential"`, `"random"`, `"willneed"`, and so on).
- `protect` - change read/write protection at runtime.
- `remap` - resize an existing mapping (Linux only).
- `unmap` - drop the mapping explicitly.
- `pin` / `unpin` - keep the mapping in physical memory.

This is a native implementation of the XS half of `File::Map` 0.71; its `File/Map.pm` is used unchanged. Nothing is exported by default. Name the functions, or take them by tag: `:map` (`map_handle`, `map_file`, `map_anonymous`, `sys_map`, `unmap`), `:extra` (`remap`, `sync`, `pin`, `unpin`, `advise`, `protect`), `:lock` (`lock_map`), `:constants` (`PROT_*`, `MAP_*`), or `:all`.

Differences from upstream

None. It is `File::Map` as a perl without threads builds it, so the thread functions `wait_until`, `notify` and `broadcast` are absent and `lock_map` locks nothing.

Functions

Mapping

`map_file`

Open a file by name and attach it to a scalar as a memory map.

map_handle

Attach an already-open filehandle to a scalar as a memory map.

map_anonymous

Allocate a fresh memory region backed by no file and attach it to a scalar.

sys_map

Low-level mapping with raw `mmap(2)` protection and flag constants.

remap

Resize an existing mapping in place.

Unmapping

unmap

Release a mapping explicitly before the scalar goes out of scope.

Synchronisation

sync

Flush pending writes from a mapped scalar back to disk.

Locking

pin

`pin $map`: lock the mapping in physical memory, so it is never paged out (`mlock(2)`). The system limits how much a process may pin (`ulimit -l`); beyond it `pin` dies with the error.

unpin

`unpin $map`: undo `pin`; the mapping may be paged out again.

lock_map

`lock_map $map`: take the mapping's lock until the end of the enclosing scope - the lock that serialises threads using the same mapping. Perl here has no threads, so it only checks that `$map` is a mapped scalar; the thread-only condition functions `wait_until`, `notify` and `broadcast` do not exist, as in any unthreaded perl.

Advice

advise

Tell the kernel how the mapped region will be accessed.

protect

Change the read/write/execute protection of a mapped region.

map_file

Open a file by name and attach it to a scalar as a memory map.

Synopsis

```
map_file my $map, $filename;           # read-only
map_file my $map, $filename, '+<';     # read/write
map_file my $map, $filename, '<', $offset, $length;
```

Arguments

- `$lvalue` - scalar to receive the mapping. Any previous value is discarded. The variable must not already be mapped.
- `$filename` - path to the file. The file is opened with a `:raw` layer added automatically, so encoding and newline translation never interfere with the mapping.
- `$mode` - defaults to `'<'`. Accepted values are `'<'`, `'+<'`, `'>'`, and `'+>'`, with the same meaning as open.
- `$offset` - byte position at which the mapping starts. Defaults to `0`.
- `$length` - length of the mapping in bytes. Defaults to `-s($filename) - $offset`, i.e. the rest of the file.

Examples

Read an entire file into a scalar without loading it eagerly:

```
map_file my $text, 'big.log';
print "lines: ", scalar(() = $text =~ /\n/g), "\n";
```

Patch bytes inside a binary file:

```
map_file my $map, 'image.bin', '+<';
substr $map, 0, 4, "MZ\0\0";
```

Map a window into a large file:

```
map_file my $chunk, 'archive.tar', '<', 1024 * 1024, 4096;
```

Edge cases

- Opening a zero-length file succeeds but produces a mapping that cannot be written to; writes are silently truncated to the empty string.
- `$offset` does not need to be page-aligned; the module adjusts internally.
- A filehandle with a non-binary PerlIO layer is rejected.
- Going out of scope unmaps the variable; an explicit `unmap` is rarely needed.

Differences from upstream

Fully compatible with upstream `File::Map` 0.71.

See also

- `map_handle` - same operation starting from an open filehandle.
- `map_anonymous` - allocate a scratch region with no backing file.
- `sys_map` - drop down to raw `mmap(2)` flags.
- `unmap` - release a mapping explicitly before scope exit.
- `sync` - flush writes to disk without waiting for scope exit.

map_handle

Attach an already-open filehandle to a scalar as a memory map.

Synopsis

```
open my $fh, '<:raw', $filename;
map_handle my $map, $fh;
map_handle my $map, $fh, '+<', $offset, $length;
```

Arguments

- `$lvalue` - scalar to receive the mapping.
- `$filehandle` - a real filehandle. Scalar-string handles and tied handles are rejected.
- `$mode` - defaults to `'<'`. Same accepted values as `map_file`.
- `$offset`, `$length` - same meaning and defaults as `map_file`.

Examples

Map the tail of a log file:

```
open my $fh, '<:raw', '/var/log/syslog' or die $!;
my $size = -s $fh;
map_handle my $tail, $fh, '<', $size - 4096, 4096;
```

Edge cases

- The filehandle must be binary. Encoding layers like `:utf8` or `:crlf` cause `map_handle` to reject the handle.
- The handle must refer to a regular file, block device, or character device. Pipes and sockets cannot be mapped.
- The handle stays valid after the call; closing it does not invalidate the mapping.

Differences from upstream

Fully compatible with upstream `File::Map` 0.71.

See also

- `map_file` - take a filename instead of a filehandle.
- `sys_map` - pass raw `mmap(2)` flags directly.
- `unmap` - release the mapping before scope exit.

map_anonymous

Allocate a fresh memory region backed by no file and attach it to a scalar.

Synopsis

```
map_anonymous my $buf, 4096;                # shared (default)
map_anonymous my $buf, 4096, 'shared';      # visible to forked
↪children
map_anonymous my $buf, 1 << 20, 'private'; # process-local
```

Arguments

- `$lvalue` - scalar to receive the mapping.
- `$length` - size of the region in bytes. Must be greater than zero.
- `$type` - `'shared'` or `'private'`. A shared region is inherited by child processes created with `fork`; a private region is not. Defaults to `'shared'`.

Examples

Share a buffer with a forked child:

```
map_anonymous my $counter, 16, 'shared';
substr $counter, 0, 16, pack('Q', 0) . pack('Q', 0);
if (fork() == 0) {
    substr $counter, 0, 8, pack('Q', 42);
    exit;
}
```

Allocate a private scratch region:

```
map_anonymous my $scratch, 1_000_000, 'private';
substr $scratch, 0, 5, "hello";
```

Edge cases

- Zero-length anonymous maps are rejected.
- Any `$type` other than `'shared'` or `'private'` raises an exception.

Differences from upstream

Fully compatible with upstream `File::Map` 0.71.

See also

- `map_file` - map a file instead of raw memory.
- `sys_map` - take raw `mmap(2)` flags.
- `remap` - grow or shrink a private anonymous mapping on Linux.

sys_map

Low-level mapping with raw `mmap(2)` protection and flag constants.

Synopsis

```
use File::Map qw(sys_map :constants);

sys_map my $map, $length, PROT_READ | PROT_WRITE,
        MAP_SHARED, $fh, 0;

sys_map my $buf, 4096, PROT_READ | PROT_WRITE,
        MAP_PRIVATE | MAP_ANONYMOUS;
```

Arguments

- `$lvalue` - scalar to receive the mapping.
- `$length` - size in bytes.
- `$protection` - bitwise-or of `PROT_NONE`, `PROT_READ`, `PROT_WRITE`, `PROT_EXEC`.
- `$flags` - bitwise-or of `MAP_SHARED`, `MAP_PRIVATE`, `MAP_ANONYMOUS`, `MAP_ANON`, `MAP_FILE`.
- `$filehandle` - omitted or `undef` when `MAP_ANONYMOUS` is set.
- `$offset` - byte offset into the file. Defaults to `0`.

When to reach for it

Use `sys_map` only when `map_file`, `map_handle`, or `map_anonymous` cannot express what you need - for example, to request `PROT_EXEC` on an executable page, or to combine flags that the convenience functions do not expose. If the `mmap(2)` man page does not feel familiar, `map_file` or `map_anonymous` is almost certainly the better choice.

Differences from upstream

Fully compatible with upstream `File::Map` 0.71.

See also

- `map_file`, `map_handle`, `map_anonymous` - the high-level alternatives.
- `protect` - change protection after the mapping exists.

sync

Flush pending writes from a mapped scalar back to disk.

Synopsis

```
sync $map;          # synchronous - wait for the kernel to finish
sync $map, 0;       # asynchronous - queue the flush and return
```

Arguments

- `$lvalue` - a currently mapped scalar.
- `$synchronous` - defaults to true. When true, `sync` does not return until the kernel reports the data on stable storage. When false, the flush is only queued.

The mapping is always flushed automatically when the scalar goes out of scope, so explicit calls to `sync` are only needed when you want other processes, or a crash-safe on-disk snapshot, to see the changes before then.

Edge cases

- Calling `sync` on an unmapped scalar raises an exception.
- Calling `sync` on an empty mapping is a no-op.

Differences from upstream

Fully compatible with upstream `File::Map` 0.71.

See also

- `unmap` - release the mapping (also triggers a flush).
- `advise` - influence the kernel's write-back strategy.

remap

Resize an existing mapping in place.

Synopsis

```
remap $map, $new_size;
```

Arguments

- `$lvalue` - a currently mapped scalar.
- `$new_size` - new length in bytes. Must be greater than zero.

When it applies

`remap` is Linux-specific. It wraps `mremap(2)` with `MREMAP_MAYMOVE`, so the scalar may end up pointing to a different virtual address after the call.

- For a file-backed mapping, the underlying file must be large enough to cover `$new_size`, or accessing the extra pages will produce a bus fault.
- Anonymous shared mappings cannot be remapped.

Use with caution on production data.

Differences from upstream

Fully compatible with upstream `File::Map` 0.71.

See also

- `map_file` - pick the right size up front if you can.
- `unmap` - drop the mapping entirely instead of resizing.

unmap

Release a mapping explicitly before the scalar goes out of scope.

Synopsis

```
unmap $map;
```

When to call it

Most programs never need to. A mapped scalar is unmapped automatically when it is freed - at scope exit for `my` variables, at the end of the program for globals - and the kernel flushes any pending writes as part of that. Call `unmap` explicitly when you want to release the memory (and close the mapping of the file) before the natural lifetime of the scalar ends.

After `unmap`, the scalar becomes an ordinary empty string and can be reused for any purpose, including being mapped again.

Edge cases

- Calling `unmap` on a scalar that is not mapped raises an exception.

Differences from upstream

Fully compatible with upstream `File::Map` 0.71.

See also

- `sync` - flush writes without releasing the mapping.

advise

Tell the kernel how the mapped region will be accessed.

Synopsis

```
advise $map, 'sequential';
advise $map, 'willneed';
```

Arguments

- `$lvalue` - a currently mapped scalar.
- `$advice` - a string naming an access pattern.

The portable values are:

- `'normal'` - no particular pattern (the default).
- `'random'` - access order is unpredictable.
- `'sequential'` - read start to end once.
- `'willneed'` - prefault pages that will be needed soon.
- `'dontneed'` - release pages that will not be needed soon.

On Linux additional values are accepted: `'remove'`, `'dontfork'`, `'dofork'`, `'mergeable'`, `'unmergeable'`, `'free'`. Unknown values are silently ignored.

Examples

Hint that a large log file will be scanned top to bottom:

```
map_file my $log, 'huge.log';
advise $log, 'sequential';
while ($log =~ /^(ERROR:.*)/mg) { ... }
```

Edge cases

- Advising an unmapped scalar raises an exception.
- Advising an empty mapping is a no-op.
- `advise` is a hint to the kernel; it may do nothing on some systems.

Differences from upstream

Fully compatible with upstream `File::Map` 0.71.

See also

- `protect` - change read/write permissions rather than access hints.
- `sync` - control flush timing explicitly.

protect

Change the read/write/execute protection of a mapped region.

Synopsis

```
protect $map, '<';              # read-only
protect $map, '+<';             # read/write
protect $map, PROT_READ;        # constants also work
```

Arguments

- `$lvalue` - a currently mapped scalar.
- `$mode` - either a string in the same format as open (`'<'`, `'+<'`, `'>'`, `'+>'`), or a bitmask built from the `PROT_*` constants.

If the new protection includes write permission the scalar becomes writable; otherwise it is marked read-only and direct assignment raises an exception.

Examples

Load a file read/write, patch it, then lock it down:

```
map_file my $map, 'data.bin', '+<';
substr $map, 0, 4, "HEAD";
sync $map;
protect $map, '<';             # prevent further writes
```

Edge cases

- Protecting an unmapped scalar raises an exception.
- Protecting an empty mapping updates the scalar's read-only flag but performs no kernel call.

Differences from upstream

Fully compatible with upstream `File::Map` 0.71.

See also

- `sys_map` - set protection when creating the mapping.
- `advise` - hint access patterns rather than enforce permissions.

File::Path

Create or remove directory trees

VERSION

2.18 - released November 4 2020.

SYNOPSIS

```
use File::Path qw(make_path remove_tree);

@created = make_path('foo/bar/baz', '/zug/zwang');
@created = make_path('foo/bar/baz', '/zug/zwang', {
    verbose => 1,
    mode => 0711,
});
make_path('foo/bar/baz', '/zug/zwang', {
    chmod => 0777,
});

$removed_count = remove_tree('foo/bar/baz', '/zug/zwang', {
    verbose => 1,
    error  => \my $err_list,
    safe => 1,
});
```

(continues on next page)

(continued from previous page)

```
# legacy (interface promoted before v2.00)
@created = mkpath('/foo/bar/baz');
@created = mkpath('/foo/bar/baz', 1, 0711);
@created = mkpath(['/foo/bar/baz', 'blurfl/quux'], 1, 0711);
$removed_count = rmtree('foo/bar/baz', 1, 1);
$removed_count = rmtree(['foo/bar/baz', 'blurfl/quux'], 1, 1);

# legacy (interface promoted before v2.06)
@created = mkpath('foo/bar/baz', '/zug/zwang', { verbose => 1, mode␣
↪=> 0711 });
$removed_count = rmtree('foo/bar/baz', '/zug/zwang', { verbose => 1,␣
↪mode => 0711 });
```

DESCRIPTION

This module provides a convenient way to create directories of arbitrary depth and to delete an entire directory subtree from the filesystem.

The following functions are provided:

- make_path($dir1, $dir2, ...)
- make_path($dir1, $dir2, ..., \%opts)

  The `make_path` function creates the given directories if they don't exist before, much like the Unix command `mkdir -p`.

  The function accepts a list of directories to be created. Its behaviour may be tuned by an optional hashref appearing as the last parameter on the call.

  The function returns the list of directories actually created during the call; in scalar context the number of directories created.

  The following keys are recognised in the option hash:

  – mode => $num

    The numeric permissions mode to apply to each created directory (defaults to `0777`), to be modified by the current `umask`. If the directory already exists (and thus does not need to be created), the permissions will not be modified.

    `mask` is recognised as an alias for this parameter.

  – chmod => $num

    Takes a numeric mode to apply to each created directory (not modified by the current `umask`). If the directory already exists (and thus does not need to be created), the permissions will not be modified.

  – verbose => $bool

    If present, will cause `make_path` to print the name of each directory as it is created. By default nothing is printed.

  – error => \$err

If present, it should be a reference to a scalar. This scalar will be made to reference an array, which will be used to store any errors that are encountered. See the "ERROR HANDLING" section for more information.

If this parameter is not used, certain error conditions may raise a fatal error that will cause the program to halt, unless trapped in an `eval` block.

- owner => $owner
- user => $owner
- uid => $owner

If present, will cause any created directory to be owned by `$owner`. If the value is numeric, it will be interpreted as a uid; otherwise a username is assumed. An error will be issued if the username cannot be mapped to a uid, the uid does not exist or the process lacks the privileges to change ownership.

Ownership of directories that already exist will not be changed.

`user` and `uid` are aliases of `owner`.

- group => $group

If present, will cause any created directory to be owned by the group `$group`. If the value is numeric, it will be interpreted as a gid; otherwise a group name is assumed. An error will be issued if the group name cannot be mapped to a gid, the gid does not exist or the process lacks the privileges to change group ownership.

Group ownership of directories that already exist will not be changed.

```
make_path '/var/tmp/webcache', {owner=>'nobody', group=>
↪'nogroup'};
```

- mkpath($dir)
- mkpath($dir, $verbose, $mode)
- mkpath([$dir1, $dir2,...], $verbose, $mode)
- mkpath($dir1, $dir2,..., \%opt)

The `mkpath()` function provide the legacy interface of `make_path()` with a different interpretation of the arguments passed. The behaviour and return value of the function is otherwise identical to `make_path()`.

- remove_tree($dir1, $dir2, ...)
- remove_tree($dir1, $dir2, ..., \%opts)

The `remove_tree` function deletes the given directories and any files and subdirectories they might contain, much like the Unix command `rm -rf` or the Windows commands `rmdir /s` and `rd /s`.

The function accepts a list of directories to be removed. (In point of fact, it will also accept filesystem entries which are not directories, such as regular files and symlinks. But, as its name suggests, its intent is to remove trees rather than individual files.)

`remove_tree()`'s behaviour may be tuned by an optional hashref appearing as the last parameter on the call. If an empty string is passed to `remove_tree`, an error will occur.

NOTE: For security reasons, we strongly advise use of the hashref-as-final-argument syntax – specifically, with a setting of the `safe` element to a true value.

```
remove_tree( $dir1, $dir2, ....,
    {
        safe => 1,
        ...          # other key-value pairs
    },
);
```

The function returns the number of files successfully deleted.

The following keys are recognised in the option hash:

– verbose => $bool

  If present, will cause `remove_tree` to print the name of each file as it is unlinked. By default nothing is printed.

– safe => $bool

  When set to a true value, will cause `remove_tree` to skip the files for which the process lacks the required privileges needed to delete files, such as delete privileges on VMS. In other words, the code will make no attempt to alter file permissions. Thus, if the process is interrupted, no filesystem object will be left in a more permissive mode.

– keep_root => $bool

  When set to a true value, will cause all files and subdirectories to be removed, except the initially specified directories. This comes in handy when cleaning out an application's scratch directory.

  ```
  remove_tree( '/tmp', {keep_root => 1} );
  ```

– result => \$res

  If present, it should be a reference to a scalar. This scalar will be made to reference an array, which will be used to store all files and directories unlinked during the call. If nothing is unlinked, the array will be empty.

  ```
  remove_tree( '/tmp', {result => \my $list} );
  print "unlinked $_\n" for @$list;
  ```

  This is a useful alternative to the `verbose` key.

– error => \$err

  If present, it should be a reference to a scalar. This scalar will be made to reference an array, which will be used to store any errors that are encountered. See the "ERROR HANDLING" section for more information.

Removing things is a much more dangerous proposition than creating things. As such, there are certain conditions that `remove_tree` may encounter that are so dangerous that the only sane action left is to kill the program.

Use `error` to trap all that is reasonable (problems with permissions and the like), and let it die if things get out of hand. This is the safest course of action.

- rmtree($dir)
- rmtree($dir, $verbose, $safe)
- rmtree([$dir1, $dir2,...], $verbose, $safe)
- rmtree($dir1, $dir2,..., \%opt)

The `rmtree()` function provide the legacy interface of `remove_tree()` with a different interpretation of the arguments passed. The behaviour and return value of the function is otherwise identical to `remove_tree()`.

NOTE: For security reasons, we strongly advise use of the hashref-as-final-argument syntax, specifically with a setting of the `safe` element to a true value.

```
rmtree( $dir1, $dir2, ....,
    {
        safe => 1,
        ...          # other key-value pairs
    },
);
```

ERROR HANDLING

- **NOTE:**

  The following error handling mechanism is consistent throughout all code paths EXCEPT in cases where the ROOT node is nonexistent. In version 2.11 the maintainers attempted to rectify this inconsistency but too many downstream modules encountered problems. In such case, if you require root node evaluation or error checking prior to calling `make_path` or `remove_tree`, you should take additional precautions.

If `make_path` or `remove_tree` encounters an error, a diagnostic message will be printed to `STDERR` via `carp` (for non-fatal errors) or via `croak` (for fatal errors).

If this behaviour is not desirable, the `error` attribute may be used to hold a reference to a variable, which will be used to store the diagnostics. The variable is made a reference to an array of hash references. Each hash contain a single key/value pair where the key is the name of the file, and the value is the error message (including the contents of `$!` when appropriate). If a general error is encountered the diagnostic key will be empty.

An example usage looks like:

```
remove_tree( 'foo/bar', 'bar/rat', {error => \my $err} );
if ($err && @$err) {
    for my $diag (@$err) {
        my ($file, $message) = %$diag;
```

(continues on next page)

(continued from previous page)

```
        if ($file eq '') {
            print "general error: $message\n";
        }
        else {
            print "problem unlinking $file: $message\n";
        }
    }
}
else {
    print "No error encountered\n";
}
```

Note that if no errors are encountered, `$err` will reference an empty array. This means that `$err` will always end up TRUE; so you need to test `@$err` to determine if errors occurred.

NOTES

`File::Path` blindly exports `mkpath` and `rmtree` into the current namespace. These days, this is considered bad style, but to change it now would break too much code. Nonetheless, you are invited to specify what it is you are expecting to use:

```
use File::Path 'rmtree';
```

The routines `make_path` and `remove_tree` are **not** exported by default. You must specify which ones you want to use.

```
use File::Path 'remove_tree';
```

Note that a side-effect of the above is that `mkpath` and `rmtree` are no longer exported at all. This is due to the way the `Exporter` module works. If you are migrating a codebase to use the new interface, you will have to list everything explicitly. But that's just good practice anyway.

```
use File::Path qw(remove_tree rmtree);
```

API CHANGES

The API was changed in the 2.0 branch. For a time, `mkpath` and `rmtree` tried, unsuccessfully, to deal with the two different calling mechanisms. This approach was considered a failure.

The new semantics are now only available with `make_path` and `remove_tree`. The old semantics are only available through `mkpath` and `rmtree`. Users are strongly encouraged to upgrade to at least 2.08 in order to avoid surprises.

SECURITY CONSIDERATIONS

There were race conditions in the 1.x implementations of File::Path's `rmtree` function (although sometimes patched depending on the OS distribution or platform). The 2.0 version contains code to avoid the problem mentioned in CVE-2002-0435.

See the following pages for more information:

```
http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=286905
http://www.nntp.perl.org/group/perl.perl5.porters/2005/01/msg97623.
↪html
http://www.debian.org/security/2005/dsa-696
```

Additionally, unless the `safe` parameter is set (or the third parameter in the traditional interface is TRUE), should a `remove_tree` be interrupted, files that were originally in read-only mode may now have their permissions set to a read-write (or "delete OK") mode.

The following CVE reports were previously filed against File-Path and are believed to have been addressed:

- http://cve.circl.lu/cve/CVE-2004-0452
- http://cve.circl.lu/cve/CVE-2005-0448

In February 2017 the cPanel Security Team reported an additional vulnerability in File-Path. The `chmod()` logic to make directories traversable can be abused to set the mode on an attacker-chosen file to an attacker-chosen value. This is due to the time-of-check-to-time-of-use (TOCTTOU) race condition (https://en.wikipedia.org/wiki/Time_of_check_to_time_of_use) between the `stat()` that decides the inode is a directory and the `chmod()` that tries to make it user-rwx. CPAN versions 2.13 and later incorporate a patch provided by John Lightsey to address this problem. This vulnerability has been reported as CVE-2017-6512.

DIAGNOSTICS

FATAL errors will cause the program to halt (`croak`), since the problem is so severe that it would be dangerous to continue. (This can always be trapped with `eval`, but it's not a good idea. Under the circumstances, dying is the best thing to do).

SEVERE errors may be trapped using the modern interface. If the they are not trapped, or if the old interface is used, such an error will cause the program will halt.

All other errors may be trapped using the modern interface, otherwise they will be `carp`ed about. Program execution will not be halted.

- mkdir [path]: [errmsg] (SEVERE)

  `make_path` was unable to create the path. Probably some sort of permissions error at the point of departure or insufficient resources (such as free inodes on Unix).

- No root path(s) specified

  `make_path` was not given any paths to create. This message is only emitted if the routine is called with the traditional interface. The modern interface will remain silent if given nothing to do.

- No such file or directory

  On Windows, if `make_path` gives you this warning, it may mean that you have exceeded your filesystem's maximum path length.

- cannot fetch initial working directory: [errmsg]

  `remove_tree` attempted to determine the initial directory by calling `Cwd::getcwd`, but the call failed for some reason. No attempt will be made to delete anything.

- cannot stat initial working directory: [errmsg]

  `remove_tree` attempted to stat the initial directory (after having successfully obtained its name via `getcwd`), however, the call failed for some reason. No attempt will be made to delete anything.

- cannot chdir to [dir]: [errmsg]

  `remove_tree` attempted to set the working directory in order to begin deleting the objects therein, but was unsuccessful. This is usually a permissions issue. The routine will continue to delete other things, but this directory will be left intact.

- directory [dir] changed before chdir, expected dev=[n] ino=[n], actual dev=[n] ino=[n], aborting. (FATAL)

  `remove_tree` recorded the device and inode of a directory, and then moved into it. It then performed a `stat` on the current directory and detected that the device and inode were no longer the same. As this is at the heart of the race condition problem, the program will die at this point.

- cannot make directory [dir] read+writeable: [errmsg]

  `remove_tree` attempted to change the permissions on the current directory to ensure that subsequent unlinkings would not run into problems, but was unable to do so. The permissions remain as they were, and the program will carry on, doing the best it can.

- cannot read [dir]: [errmsg]

  `remove_tree` tried to read the contents of the directory in order to acquire the names of the directory entries to be unlinked, but was unsuccessful. This is usually a permissions issue. The program will continue, but the files in this directory will remain after the call.

- cannot reset chmod [dir]: [errmsg]

  `remove_tree`, after having deleted everything in a directory, attempted to restore its permissions to the original state but failed. The directory may wind up being left behind.

- cannot remove [dir] when cwd is [dir]

  The current working directory of the program is `/some/path/to/here` and you are attempting to remove an ancestor, such as `/some/path`. The directory tree is left untouched.

  The solution is to `chdir` out of the child directory to a place outside the directory tree to be removed.

- cannot chdir to [parent-dir] from [child-dir]: [errmsg], aborting. (FATAL)

`remove_tree`, after having deleted everything and restored the permissions of a directory, was unable to chdir back to the parent. The program halts to avoid a race condition from occurring.

- cannot stat prior working directory [dir]: [errmsg], aborting. (FATAL)

  `remove_tree` was unable to stat the parent directory after having returned from the child. Since there is no way of knowing if we returned to where we think we should be (by comparing device and inode) the only way out is to `croak`.

- previous directory [parent-dir] changed before entering [child-dir], expected dev=[n] ino=[n], actual dev=[n] ino=[n], aborting. (FATAL)

  When `remove_tree` returned from deleting files in a child directory, a check revealed that the parent directory it returned to wasn't the one it started out from. This is considered a sign of malicious activity.

- cannot make directory [dir] writeable: [errmsg]

  Just before removing a directory (after having successfully removed everything it contained), `remove_tree` attempted to set the permissions on the directory to ensure it could be removed and failed. Program execution continues, but the directory may possibly not be deleted.

- cannot remove directory [dir]: [errmsg]

  `remove_tree` attempted to remove a directory, but failed. This may be because some objects that were unable to be removed remain in the directory, or it could be a permissions issue. The directory will be left behind.

- cannot restore permissions of [dir] to [0nnn]: [errmsg]

  After having failed to remove a directory, `remove_tree` was unable to restore its permissions from a permissive state back to a possibly more restrictive setting. (Permissions given in octal).

- cannot make file [file] writeable: [errmsg]

  `remove_tree` attempted to force the permissions of a file to ensure it could be deleted, but failed to do so. It will, however, still attempt to unlink the file.

- cannot unlink file [file]: [errmsg]

  `remove_tree` failed to remove a file. Probably a permissions issue.

- cannot restore permissions of [file] to [0nnn]: [errmsg]

  After having failed to remove a file, `remove_tree` was also unable to restore the permissions on the file to a possibly less permissive setting. (Permissions given in octal).

- unable to map [owner] to a uid, ownership not changed");

  `make_path` was instructed to give the ownership of created directories to the symbolic name [owner], but `getpwnam` did not return the corresponding numeric uid. The directory will be created, but ownership will not be changed.

- unable to map [group] to a gid, group ownership not changed

make_path was instructed to give the group ownership of created directories to the symbolic name [group], but getgrnam did not return the corresponding numeric gid. The directory will be created, but group ownership will not be changed.

SEE ALSO

- File::Remove

  Allows files and directories to be moved to the Trashcan/Recycle Bin (where they may later be restored if necessary) if the operating system supports such functionality. This feature may one day be made available directly in File::Path.

- File::Find::Rule

  When removing directory trees, if you want to examine each file to decide whether to delete it (and possibly leaving large swathes alone), File::Find::Rule offers a convenient and flexible approach to examining directory trees.

BUGS AND LIMITATIONS

The following describes File::Path limitations and how to report bugs.

MULTITHREADED APPLICATIONS

File::Path rmtree and remove_tree will not work with multithreaded applications due to its use of chdir. At this time, no warning or error is generated in this situation. You will certainly encounter unexpected results.

The implementation that surfaces this limitation will not be changed. See the File::Path::Tiny module for functionality similar to File::Path but which does not chdir.

NFS Mount Points

File::Path is not responsible for triggering the automounts, mirror mounts, and the contents of network mounted filesystems. If your NFS implementation requires an action to be performed on the filesystem in order for File::Path to perform operations, it is strongly suggested you assure filesystem availability by reading the root of the mounted filesystem.

REPORTING BUGS

Please report all bugs on the RT queue, either via the web interface:

http://rt.cpan.org/NoAuth/Bugs.html?Dist=File-Path

or by email:

```
bug-File-Path@rt.cpan.org
```

In either case, please **attach** patches to the bug report rather than including them inline in the web post or the body of the email.

You can also send pull requests to the Github repository:

https://github.com/rpcme/File-Path

ACKNOWLEDGEMENTS

Paul Szabo identified the race condition originally, and Brendan O'Dea wrote an implementation for Debian that addressed the problem. That code was used as a basis for the current code. Their efforts are greatly appreciated.

Gisle Aas made a number of improvements to the documentation for 2.07 and his advice and assistance is also greatly appreciated.

AUTHORS

Prior authors and maintainers: Tim Bunce, Charles Bailey, and David Landgren <david@landgren.net>.

Current maintainers are Richard Elberger <riche@cpan.org> and James (Jim) Keenan <jkeenan@cpan.org>.

CONTRIBUTORS

Contributors to File::Path, in alphabetical order by first name.

- <bulkdd@cpan.org>
- Charlie Gonzalez <itcharlie@cpan.org>
- Craig A. Berry <craigberry@mac.com>
- James E Keenan <jkeenan@cpan.org>
- John Lightsey <john@perlsec.org>
- Nigel Horne <njh@bandsman.co.uk>
- Richard Elberger <riche@cpan.org>
- Ryan Yee <ryee@cpan.org>
- Skye Shaw <shaw@cpan.org>
- Tom Lutz <tommylutz@gmail.com>
- Will Sheppard <willsheppard@github>

COPYRIGHT

This module is copyright (C) Charles Bailey, Tim Bunce, David Landgren, James Keenan and Richard Elberger 1995-2020. All rights reserved.

LICENSE

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

File::ShareDir

Locate per-dist and per-module shared files

SYNOPSIS

```
use File::ShareDir ':ALL';

# Where are distribution-level shared data files kept
$dir = dist_dir('File-ShareDir');

# Where are module-level shared data files kept
$dir = module_dir('File::ShareDir');

# Find a specific file in our dist/module shared dir
$file = dist_file(  'File-ShareDir',  'file/name.txt');
$file = module_file('File::ShareDir', 'file/name.txt');

# Like module_file, but search up the inheritance tree
$file = class_file( 'Foo::Bar', 'file/name.txt' );
```

DESCRIPTION

The intent of *File::ShareDir* is to provide a companion to *Class::Inspector* and File::HomeDir, modules that take a process that is well-known by advanced Perl developers but gets a little tricky, and make it more available to the larger Perl community.

Quite often you want or need your Perl module (CPAN or otherwise) to have access to a large amount of read-only data that is stored on the file-system at run-time.

On a linux-like system, this would be in a place such as /usr/share, however Perl runs on a wide variety of different systems, and so the use of any one location is unreliable.

Perl provides a little-known method for doing this, but almost nobody is aware that it exists. As a result, module authors often go through some very strange ways to make the data available to their code.

The most common of these is to dump the data out to an enormous Perl data structure and save it into the module itself. The result are enormous multi-megabyte .pm files that chew up a lot of memory needlessly.

Another method is to put the data "file" after the __DATA__ compiler tag and limit yourself to access as a filehandle.

The problem to solve is really quite simple.

```
1. Write the data files to the system at install time.

2. Know where you put them at run-time.
```

Perl's install system creates an "auto" directory for both every distribution and for every module file.

These are used by a couple of different auto-loading systems to store code fragments generated at install time, and various other modules written by the Perl "ancient masters".

But the same mechanism is available to any dist or module to store any sort of data.

Using Data in your Module

`File::ShareDir` forms one half of a two part solution.

Once the files have been installed to the correct directory, you can use `File::ShareDir` to find your files again after the installation.

For the installation half of the solution, see File::ShareDir::Install and its `install_share` directive.

Using File::ShareDir::Install together with *File::ShareDir* allows one to rely on the files in appropriate `dist_dir()` or `module_dir()` in development phase, too.

FUNCTIONS

`File::ShareDir` provides four functions for locating files and directories.

For greater maintainability, none of these are exported by default and you are expected to name the ones you want at use-time, or provide the `':ALL'` tag. All of the following are equivalent.

```
# Load but don't import, and then call directly
use File::ShareDir;
$dir = File::ShareDir::dist_dir('My-Dist');

# Import a single function
use File::ShareDir 'dist_dir';
dist_dir('My-Dist');

# Import all the functions
use File::ShareDir ':ALL';
dist_dir('My-Dist');
```

All of the functions will check for you that the dir/file actually exists, and that you have read permissions, or they will throw an exception.

dist_dir

```
# Get a distribution's shared files directory
my $dir = dist_dir('My-Distribution');
```

The `dist_dir` function takes a single parameter of the name of an installed (CPAN or otherwise) distribution, and locates the shared data directory created at install time for it.

Returns the directory path as a string, or dies if it cannot be located or is not readable.

module_dir

```
# Get a module's shared files directory
my $dir = module_dir('My::Module');
```

The `module_dir` function takes a single parameter of the name of an installed (CPAN or otherwise) module, and locates the shared data directory created at install time for it.

In order to find the directory, the module **must** be loaded when calling this function.

Returns the directory path as a string, or dies if it cannot be located or is not readable.

dist_file

```
# Find a file in our distribution shared dir
my $dir = dist_file('My-Distribution', 'file/name.txt');
```

The `dist_file` function takes two parameters of the distribution name and file name, locates the dist directory, and then finds the file within it, verifying that the file actually exists, and that it is readable.

The filename should be a relative path in the format of your local filesystem. It will simply added to the directory using *File::Spec*'s `catfile` method.

Returns the file path as a string, or dies if the file or the dist's directory cannot be located, or the file is not readable.

module_file

```
# Find a file in our module shared dir
my $dir = module_file('My::Module', 'file/name.txt');
```

The `module_file` function takes two parameters of the module name and file name. It locates the module directory, and then finds the file within it, verifying that the file actually exists, and that it is readable.

In order to find the directory, the module **must** be loaded when calling this function.

The filename should be a relative path in the format of your local filesystem. It will simply added to the directory using *File::Spec*'s `catfile` method.

Returns the file path as a string, or dies if the file or the dist's directory cannot be located, or the file is not readable.

class_file

```
# Find a file in our module shared dir, or in our parent class
my $dir = class_file('My::Module', 'file/name.txt');
```

The `module_file` function takes two parameters of the module name and file name. It locates the module directory, and then finds the file within it, verifying that the file actually exists, and that it is readable.

In order to find the directory, the module **must** be loaded when calling this function.

The filename should be a relative path in the format of your local filesystem. It will simply added to the directory using *File::Spec*'s `catfile` method.

If the file is NOT found for that module, `class_file` will scan up the module's @ISA tree, looking for the file in all of the parent classes.

This allows you to, in effect, "subclass" shared files.

Returns the file path as a string, or dies if the file or the dist's directory cannot be located, or the file is not readable.

EXTENDING

Overriding Directory Resolution

`File::ShareDir` has two convenience hashes for people who have advanced usage requirements of `File::ShareDir` such as using uninstalled `share` directories during development.

```
#
# Dist-Name => /absolute/path/for/DistName/share/dir
#
%File::ShareDir::DIST_SHARE

#
# Module::Name => /absolute/path/for/Module/Name/share/dir
#
%File::ShareDir::MODULE_SHARE
```

Setting these values any time before the corresponding calls

```
dist_dir('Dist-Name')
dist_file('Dist-Name','some/file');

module_dir('Module::Name');
module_file('Module::Name','some/file');
```

Will override the base directory for resolving those calls.

An example of where this would be useful is in a test for a module that depends on files installed into a share directory, to enable the tests to use the development copy without needing to install them first.

```
use File::ShareDir;
use Cwd qw( getcwd );
use File::Spec::Functions qw( rel2abs catdir );

$File::ShareDir::MODULE_SHARE{'Foo::Module'} =␣
↪rel2abs(catfile(getcwd,'share'));

use Foo::Module;

# internal calls in Foo::Module to module_file('Foo::Module','bar')␣
```

(continues on next page)

(continued from previous page)

```
 ↪now resolves to
# the source trees share/ directory instead of something in @INC
```

SUPPORT

Bugs should always be submitted via the CPAN request tracker, see below.

You can find documentation for this module with the perldoc command.

```
perldoc File::ShareDir
```

You can also look for information at:

- RT: CPAN's request tracker

  http://rt.cpan.org/NoAuth/Bugs.html?Dist=File-ShareDir

- AnnoCPAN: Annotated CPAN documentation

  http://annocpan.org/dist/File-ShareDir

- CPAN Ratings

  http://cpanratings.perl.org/s/File-ShareDir

- CPAN Search

  http://search.cpan.org/dist/File-ShareDir/

Where can I go for other help?

If you have a bug report, a patch or a suggestion, please open a new report ticket at CPAN (but please check previous reports first in case your issue has already been addressed).

Report tickets should contain a detailed description of the bug or enhancement request and at least an easily verifiable way of reproducing the issue or fix. Patches are always welcome, too.

Where can I go for help with a concrete version?

Bugs and feature requests are accepted against the latest version only. To get patches for earlier versions, you need to get an agreement with a developer of your choice - who may or not report the issue and a suggested fix upstream (depends on the license you have chosen).

Business support and maintenance

For business support you can contact the maintainer via his CPAN email address. Please keep in mind that business support is neither available for free nor are you eligible to receive any support based on the license distributed with this package.

AUTHOR

Adam Kennedy adamk@cpan.org

MAINTAINER

Jens Rehsack rehsack@cpan.org

SEE ALSO

File::ShareDir::Install, File::ConfigDir, File::HomeDir, Module::Install, Module::Install::Share, File::ShareDir::PAR, Dist::Zilla::Plugin::ShareDir

COPYRIGHT

Copyright 2005 - 2011 Adam Kennedy, Copyright 2014 - 2018 Jens Rehsack.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

The full text of the license can be found in the LICENSE file included with this module.

File::Spec

portably perform operations on file names

SYNOPSIS

```
use File::Spec;

my $x = File::Spec->catfile('a', 'b', 'c');
```

which returns 'a/b/c' under Unix. Or:

```
use File::Spec::Functions;

my $x = catfile('a', 'b', 'c');
```

DESCRIPTION

This module is designed to support operations commonly performed on file specifications (usually called "file names", but not to be confused with the contents of a file, or Perl's file handles), such as concatenating several directory and file names into a single path, or determining whether a path is rooted. It is based on code directly taken from MakeMaker 5.17, code written by Andreas König, Andy Dougherty, Charles Bailey, Ilya Zakharevich, Paul Schinder, and others.

Since these functions are different for most operating systems, each set of OS specific routines is available in a separate module, including:

```
File::Spec::Unix
File::Spec::Mac
File::Spec::OS2
File::Spec::Win32
File::Spec::VMS
```

The module appropriate for the current OS is automatically loaded by File::Spec. Since some modules (like VMS) make use of facilities available only under that OS, it may not be possible to load all modules under all operating systems.

Since File::Spec is object oriented, subroutines should not be called directly, as in:

```
File::Spec::catfile('a','b');
```

but rather as class methods:

```
File::Spec->catfile('a','b');
```

For simple uses, *File::Spec::Functions* provides convenient functional forms of these methods.

METHODS

- canonpath

  No physical check on the filesystem, but a logical cleanup of a path.

  ```
  $cpath = File::Spec->canonpath( $path ) ;
  ```

  Note that this does *not* collapse `x/../y` sections into `y`. This is by design. If `/foo` on your system is a symlink to `/bar/baz`, then `/foo/../quux` is actually `/bar/quux`, not `/quux` as a naive `../`-removal would give you. If you want to do this kind of processing, you probably want `Cwd`'s `realpath()` function to actually traverse the filesystem cleaning up paths like this.

- catdir

  Concatenate two or more directory names to form a complete path ending with a directory. But remove the trailing slash from the resulting string, because it doesn't look good, isn't necessary and confuses OS/2. Of course, if this is the root directory, don't cut off the trailing slash :-)

  ```
  $path = File::Spec->catdir( @directories );
  ```

- catfile

  Concatenate one or more directory names and a filename to form a complete path ending with a filename

  ```
  $path = File::Spec->catfile( @directories, $filename );
  ```

- curdir

  Returns a string representation of the current directory.

```
$curdir = File::Spec->curdir();
```

- devnull

  Returns a string representation of the null device.

```
$devnull = File::Spec->devnull();
```

- rootdir

  Returns a string representation of the root directory.

```
$rootdir = File::Spec->rootdir();
```

- tmpdir

  Returns a string representation of the first writable directory from a list of possible temporary directories. Returns the current directory if no writable temporary directories are found. The list of directories checked depends on the platform; e.g. File::Spec::Unix checks `$ENV{TMPDIR}` (unless taint is on) and `/tmp`.

```
$tmpdir = File::Spec->tmpdir();
```

- updir

  Returns a string representation of the parent directory.

```
$updir = File::Spec->updir();
```

- no_upwards

  Given a list of files in a directory (such as from `readdir()`), strip out `'.'` and `'..'`.

  SECURITY NOTE: This does NOT filter paths containing `'..'`, like `'../../../../etc/passwd'`, only literal matches to `'.'` and `'..'`.

```
@paths = File::Spec->no_upwards( readdir $dirhandle );
```

- case_tolerant

  Returns a true or false value indicating, respectively, that alphabetic case is not or is significant when comparing file specifications. Cygwin and Win32 accept an optional drive argument.

```
$is_case_tolerant = File::Spec->case_tolerant();
```

- file_name_is_absolute

  Takes as its argument a path, and returns true if it is an absolute path.

```
$is_absolute = File::Spec->file_name_is_absolute( $path );
```

  This does not consult the local filesystem on Unix, Win32, OS/2, or Mac OS (Classic). It does consult the working environment for VMS (see *"file_name_is_absolute" in File::Spec::VMS*).

- path

  Takes no argument. Returns the environment variable PATH (or the local platform's equivalent) as a list.

```
@PATH = File::Spec->path();
```

- join

  join is the same as catfile.

- splitpath

  Splits a path in to volume, directory, and filename portions. On systems with no concept of volume, returns '' for volume.

```
($volume,$directories,$file) =
                    File::Spec->splitpath( $path );
($volume,$directories,$file) =
                    File::Spec->splitpath( $path, $no_file );
```

  For systems with no syntax differentiating filenames from directories, assumes that the last file is a path unless `$no_file` is true or a trailing separator or `/.` or `/..` is present. On Unix, this means that `$no_file` true makes this return ('', $path, '').

  The directory portion may or may not be returned with a trailing '/'.

  The results can be passed to "catpath()" to get back a path equivalent to (usually identical to) the original path.

- splitdir

  The opposite of "catdir".

```
@dirs = File::Spec->splitdir( $directories );
```

  `$directories` must be only the directory portion of the path on systems that have the concept of a volume or that have path syntax that differentiates files from directories.

  Unlike just splitting the directories on the separator, empty directory names (`''`) can be returned, because these are significant on some OSes.

- catpath()

  Takes volume, directory and file portions and returns an entire path. Under Unix, `$volume` is ignored, and directory and file are concatenated. A '/' is inserted if need be. On other OSes, `$volume` is significant.

```
$full_path = File::Spec->catpath( $volume, $directory, $file );
```

- abs2rel

  Takes a destination path and an optional base path returns a relative path from the base path to the destination path:

```
$rel_path = File::Spec->abs2rel( $path ) ;
$rel_path = File::Spec->abs2rel( $path, $base ) ;
```

If $base is not present or '', then *Cwd::cwd()* is used. If $base is relative, then it is converted to absolute form using "rel2abs()". This means that it is taken to be relative to *Cwd::cwd()*.

On systems with the concept of volume, if $path and $base appear to be on two different volumes, we will not attempt to resolve the two paths, and we will instead simply return $path. Note that previous versions of this module ignored the volume of $base, which resulted in garbage results part of the time.

On systems that have a grammar that indicates filenames, this ignores the $base filename as well. Otherwise all path components are assumed to be directories.

If $path is relative, it is converted to absolute form using "rel2abs()". This means that it is taken to be relative to *Cwd::cwd()*.

No checks against the filesystem are made. On VMS, there is interaction with the working environment, as logicals and macros are expanded.

Based on code written by Shigio Yamaguchi.

- rel2abs()

  Converts a relative path to an absolute path.

```
$abs_path = File::Spec->rel2abs( $path ) ;
$abs_path = File::Spec->rel2abs( $path, $base ) ;
```

If $base is not present or '', then *Cwd::cwd()* is used. If $base is relative, then it is converted to absolute form using "rel2abs()". This means that it is taken to be relative to *Cwd::cwd()*.

On systems with the concept of volume, if $path and $base appear to be on two different volumes, we will not attempt to resolve the two paths, and we will instead simply return $path. Note that previous versions of this module ignored the volume of $base, which resulted in garbage results part of the time.

On systems that have a grammar that indicates filenames, this ignores the $base filename as well. Otherwise all path components are assumed to be directories.

If $path is absolute, it is cleaned up and returned using "canonpath".

No checks against the filesystem are made. On VMS, there is interaction with the working environment, as logicals and macros are expanded.

Based on code written by Shigio Yamaguchi.

For further information, please see *File::Spec::Unix*, *File::Spec::Mac*, *File::Spec::OS2*, *File::Spec::Win32*, or *File::Spec::VMS*.

SEE ALSO

File::Spec::Unix, *File::Spec::Mac*, *File::Spec::OS2*, *File::Spec::Win32*, *File::Spec::VMS*, *File::Spec::Functions*, *ExtUtils::MakeMaker*

AUTHOR

Maintained by perl5-porters <perl5-porters@perl.org>.

The vast majority of the code was written by Kenneth Albanowski <kjahds@kjahds.com>, Andy Dougherty <doughera@lafayette.edu>, Andreas König <A.Koenig@franz.ww.TU-Berlin.DE>, Tim Bunce <Tim.Bunce@ig.co.uk>. VMS support by Charles Bailey <bailey@newman.upenn.edu>. OS/2 support by Ilya Zakharevich <ilya@math.ohio-state.edu>. Mac support by Paul Schinder <schinder@pobox.com>, and Thomas Wegner <wegner_thomas@yahoo.com>. abs2rel() and rel2abs() written by Shigio Yamaguchi <shigio@tamacom.com>, modified by Barrie Slaymaker <barries@slaysys.com>. splitpath(), splitdir(), catpath() and catdir() by Barrie Slaymaker.

COPYRIGHT

Copyright (c) 2004-2013 by the Perl 5 Porters. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

Modules

- `File::Spec::AmigaOS` - File::Spec for AmigaOS
- `File::Spec::Cygwin` - methods for Cygwin file specs
- `File::Spec::Epoc` - methods for Epoc file specs
- `File::Spec::Functions` - portably perform operations on file names
- `File::Spec::Mac` - File::Spec for Mac OS (Classic)
- `File::Spec::OS2` - methods for OS/2 file specs
- `File::Spec::Unix` - File::Spec for Unix, base for other File::Spec modules
- `File::Spec::VMS` - methods for VMS file specs
- `File::Spec::Win32` - methods for Win32 file specs

File::Spec::AmigaOS

File::Spec for AmigaOS

SYNOPSIS

```
require File::Spec::AmigaOS; # Done automatically by File::Spec
                             # if needed
```

DESCRIPTION

Methods for manipulating file specifications.

METHODS

- tmpdir

  Returns $ENV{TMPDIR} or if that is unset, “/t”.

- file_name_is_absolute

  Returns true if there’s a colon in the file name, or if it begins with a slash.

All the other methods are from *File::Spec::Unix*.

File::Spec::Cygwin

methods for Cygwin file specs

SYNOPSIS

```
require File::Spec::Cygwin; # Done internally by File::Spec if␣
↪needed
```

DESCRIPTION

See *File::Spec* and *File::Spec::Unix*. This package overrides the implementation of these methods, not the semantics.

This module is still in beta. Cygwin-knowledgeable folks are invited to offer patches and suggestions.

- canonpath

  Any \ (backslashes) are converted to / (forward slashes), and then File::Spec::Unix canonpath() is called on the result.

- file_name_is_absolute

  True is returned if the file name begins with `drive_letter:`, and if not, File::Spec::Unix file_name_is_absolute() is called.

- tmpdir (override)

  Returns a string representation of the first existing directory from the following list:

  ```
  $ENV{TMPDIR}
  /tmp
  $ENV{'TMP'}
  $ENV{'TEMP'}
  C:/temp
  ```

  If running under taint mode, and if the environment variables are tainted, they are not used.

- case_tolerant

  Override Unix. Cygwin case-tolerance depends on managed mount settings and as with MsWin32 on GetVolumeInformation() $ouFsFlags == FS_CASE_SENSITIVE, indicating the case significance when comparing file specifications. Default: 1

COPYRIGHT

Copyright (c) 2004,2007 by the Perl 5 Porters. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

File::Spec::Epoc

methods for Epoc file specs

SYNOPSIS

```
require File::Spec::Epoc; # Done internally by File::Spec if needed
```

DESCRIPTION

See File::Spec::Unix for a documentation of the methods provided there. This package overrides the implementation of these methods, not the semantics.

This package is still a work in progress. ;-)

- canonpath()

  No physical check on the filesystem, but a logical cleanup of a path. On UNIX eliminated successive slashes and successive "/.".

AUTHOR

o.flebbe@gmx.de

COPYRIGHT

Copyright (c) 2004 by the Perl 5 Porters. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

SEE ALSO

See *File::Spec* and *File::Spec::Unix*. This package overrides the implementation of these methods, not the semantics.

File::Spec::Functions

portably perform operations on file names

SYNOPSIS

```
use File::Spec::Functions;
my $x = catfile('a', 'b');
```

DESCRIPTION

This module exports convenience functions for all of the class methods provided by File::Spec.

For a reference of available functions, please consult *File::Spec::Unix*, which contains the entire set, and which is inherited by the modules for other platforms. For further information, please see *File::Spec::Mac*, *File::Spec::OS2*, *File::Spec::Win32*, or *File::Spec::VMS*.

Exports

The following functions are exported by default.

```
canonpath
catdir
catfile
curdir
rootdir
updir
no_upwards
file_name_is_absolute
path
```

The following functions are exported only by request.

```
devnull
tmpdir
splitpath
splitdir
catpath
abs2rel
rel2abs
case_tolerant
```

All the functions may be imported using the `:ALL` tag.

COPYRIGHT

Copyright (c) 2004 by the Perl 5 Porters. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

SEE ALSO

File::Spec, File::Spec::Unix, File::Spec::Mac, File::Spec::OS2, File::Spec::Win32, File::Spec::VMS, ExtUtils::MakeMaker

File::Spec::Mac

File::Spec for Mac OS (Classic)

SYNOPSIS

```
require File::Spec::Mac; # Done internally by File::Spec if needed
```

DESCRIPTION

Methods for manipulating file specifications.

METHODS

- canonpath

  On Mac OS, there's nothing to be done. Returns what it's given.

- catdir()

  Concatenate two or more directory names to form a path separated by colons (":") ending with a directory. Resulting paths are **relative** by default, but can be forced to be absolute (but avoid this, see below). Automatically puts a trailing ":" on the end of the complete path, because that's what's done in MacPerl's environment and helps to distinguish a file path from a directory path.

  IMPORTANT NOTE: Beginning with version 1.3 of this module, the resulting path is relative by default and *not* absolute. This decision was made due to portability reasons. Since `File::Spec->catdir()` returns relative paths on all other operating systems, it will now also follow this convention on Mac OS. Note that this may break some existing scripts.

  The intended purpose of this routine is to concatenate *directory names*. But because of the nature of Macintosh paths, some additional possibilities are allowed to make using this routine give reasonable results for some common situations. In other words, you are also allowed to concatenate *paths* instead of directory names (strictly speaking, a string like ":a" is a path, but not a name, since it contains a punctuation character ":").

  So, beside calls like

  ```
  catdir("a") = ":a:"
  catdir("a","b") = ":a:b:"
  catdir() = ""                    (special case)
  ```

  calls like the following

```
catdir(":a:") = ":a:"
catdir(":a","b") = ":a:b:"
catdir(":a:","b") = ":a:b:"
catdir(":a:",":b:") = ":a:b:"
catdir(":") = ":"
```

are allowed.

Here are the rules that are used in `catdir()`; note that we try to be as compatible as possible to Unix:

1. The resulting path is relative by default, i.e. the resulting path will have a leading colon.
2. A trailing colon is added automatically to the resulting path, to denote a directory.
3. Generally, each argument has one leading ":" and one trailing ":" removed (if any). They are then joined together by a ":". Special treatment applies for arguments denoting updir paths like "::lib:", see (4), or arguments consisting solely of colons ("colon paths"), see (5).
4. When an updir path like ":::lib::" is passed as argument, the number of directories to climb up is handled correctly, not removing leading or trailing colons when necessary. E.g.

   ```
   catdir(":::a","::b","c")    = ":::a::b:c:"
   catdir(":::a::","::b","c")  = ":::a:::b:c:"
   ```

5. Adding a colon ":" or empty string "" to a path at *any* position doesn't alter the path, i.e. these arguments are ignored. (When a "" is passed as the first argument, it has a special meaning, see (6)). This way, a colon ":" is handled like a "." (curdir) on Unix, while an empty string "" is generally ignored (see *"canonpath()" in File::Spec::Unix*). Likewise, a "::" is handled like a ".." (updir), and a ":::" is handled like a "../.." etc. E.g.

   ```
   catdir("a",":",":","b")   = ":a:b:"
   catdir("a",":","::",":b") = ":a::b:"
   ```

6. If the first argument is an empty string "" or is a volume name, i.e. matches the pattern /^[^:]+:/, the resulting path is **absolute**.
7. Passing an empty string "" as the first argument to `catdir()` is like passing`File::Spec->rootdir()` as the first argument, i.e.

   ```
   catdir("","a","b")          is the same as

   catdir(rootdir(),"a","b").
   ```

   This is true on Unix, where `catdir("","a","b")` yields "/a/b" and `rootdir()` is "/". Note that `rootdir()` on Mac OS is the startup volume, which is the closest in concept to Unix' "/". This should help to run existing scripts originally written for Unix.

8. For absolute paths, some cleanup is done, to ensure that the volume name isn't immediately followed by updirs. This is invalid, because this would go beyond "root". Generally, these cases are handled like their Unix counterparts:

```
Unix:
   Unix->catdir("","")                    =  "/"
   Unix->catdir("",".")                   =  "/"
   Unix->catdir("","..")                  =  "/"         # can't␣
↪go
                                                         #␣
↪beyond root
   Unix->catdir("",".","..","..","a")     =  "/a"
Mac:
   Mac->catdir("","")                     =  rootdir()   # (e.g.
↪"HD:")
   Mac->catdir("",":")                    =  rootdir()
   Mac->catdir("","::")                   =  rootdir()   # can't␣
↪go
                                                         #␣
↪beyond root
   Mac->catdir("",":","::","::","a")      =  rootdir() . "a:"
                                                      # (e.g.
↪"HD:a:")
```

However, this approach is limited to the first arguments following "root" (again, see *"canonpath()" in File::Spec::Unix*. If there are more arguments that move up the directory tree, an invalid path going beyond root can be created.

As you've seen, you can force `catdir()` to create an absolute path by passing either an empty string or a path that begins with a volume name as the first argument. However, you are strongly encouraged not to do so, since this is done only for backward compatibility. Newer versions of File::Spec come with a method called `catpath()` (see below), that is designed to offer a portable solution for the creation of absolute paths. It takes volume, directory and file portions and returns an entire path. While `catdir()` is still suitable for the concatenation of *directory names*, you are encouraged to use `catpath()` to concatenate *volume names* and *directory paths*. E.g.

```
$dir      = File::Spec->catdir("tmp","sources");
$abs_path = File::Spec->catpath("MacintoshHD:", $dir,"");
```

yields

```
"MacintoshHD:tmp:sources:" .
```

- catfile

  Concatenate one or more directory names and a filename to form a complete path ending with a filename. Resulting paths are **relative** by default, but can be forced to be absolute (but avoid this).

  IMPORTANT NOTE: Beginning with version 1.3 of this module, the resulting path is relative by default and *not* absolute. This decision was made due to portability

reasons. Since `File::Spec->catfile()` returns relative paths on all other operating systems, it will now also follow this convention on Mac OS. Note that this may break some existing scripts.

The last argument is always considered to be the file portion. Since `catfile()` uses `catdir()` (see above) for the concatenation of the directory portions (if any), the following with regard to relative and absolute paths is true:

```
catfile("")     = ""
catfile("file") = "file"
```

but

```
catfile("","")        = rootdir()         # (e.g. "HD:")
catfile("","file")    = rootdir() . file  # (e.g. "HD:file")
catfile("HD:","file") = "HD:file"
```

This means that `catdir()` is called only when there are two or more arguments, as one might expect.

Note that the leading ":" is removed from the filename, so that

```
catfile("a","b","file")  = ":a:b:file"    and

catfile("a","b",":file") = ":a:b:file"
```

give the same answer.

To concatenate *volume names*, *directory paths* and *filenames*, you are encouraged to use `catpath()` (see below).

- curdir

  Returns a string representing the current directory. On Mac OS, this is ":".

- devnull

  Returns a string representing the null device. On Mac OS, this is "Dev:Null".

- rootdir

  Returns the empty string. Mac OS has no real root directory.

- tmpdir

  Returns the contents of $ENV{TMPDIR}, if that directory exits or the current working directory otherwise. Under MacPerl, $ENV{TMPDIR} will contain a path like "MacintoshHD:Temporary Items:", which is a hidden directory on your startup volume.

- updir

  Returns a string representing the parent directory. On Mac OS, this is "::".

- file_name_is_absolute

  Takes as argument a path and returns true, if it is an absolute path. If the path has a leading ":", it's a relative path. Otherwise, it's an absolute path, unless the path doesn't contain any colons, i.e. it's a name like "a". In this particular case,

the path is considered to be relative (i.e. it is considered to be a filename). Use ":" in the appropriate place in the path if you want to distinguish unambiguously. As a special case, the filename '' is always considered to be absolute. Note that with version 1.2 of File::Spec::Mac, this does no longer consult the local filesystem.

E.g.

```
File::Spec->file_name_is_absolute("a");         # false␣
↪(relative)
File::Spec->file_name_is_absolute(":a:b:");     # false␣
↪(relative)
File::Spec->file_name_is_absolute("MacintoshHD:");
                                                # true␣
↪(absolute)
File::Spec->file_name_is_absolute("");          # true␣
↪(absolute)
```

- path

  Returns the null list for the MacPerl application, since the concept is usually meaningless under Mac OS. But if you're using the MacPerl tool under MPW, it gives back $ENV{Commands} suitably split, as is done in

  lib
  ExtUtils:MM_Mac.pm.

- splitpath

```
($volume,$directories,$file) = File::Spec->splitpath( $path );
($volume,$directories,$file) = File::Spec->splitpath( $path,
                                                      $no_file␣
↪);
```

  Splits a path into volume, directory, and filename portions.

  On Mac OS, assumes that the last part of the path is a filename unless $no_file is true or a trailing separator ":" is present.

  The volume portion is always returned with a trailing ":". The directory portion is always returned with a leading (to denote a relative path) and a trailing ":" (to denote a directory). The file portion is always returned *without* a leading ":". Empty portions are returned as empty string ''.

  The results can be passed to `catpath()` to get back a path equivalent to (usually identical to) the original path.

- splitdir

  The opposite of `catdir()`.

```
@dirs = File::Spec->splitdir( $directories );
```

  $directories should be only the directory portion of the path on systems that have the concept of a volume or that have path syntax that differentiates files from directories. Consider using `splitpath()` otherwise.

Unlike just splitting the directories on the separator, empty directory names ("") can be returned. Since `catdir()` on Mac OS always appends a trailing colon to distinguish a directory path from a file path, a single trailing colon will be ignored, i.e. there's no empty directory name after it.

Hence, on Mac OS, both

```
File::Spec->splitdir( ":a:b::c:" );    and
File::Spec->splitdir( ":a:b::c" );
```

yield:

```
( "a", "b", "::", "c")
```

while

```
File::Spec->splitdir( ":a:b::c::" );
```

yields:

```
( "a", "b", "::", "c", "::")
```

- catpath

```
$path = File::Spec->catpath($volume,$directory,$file);
```

Takes volume, directory and file portions and returns an entire path. On Mac OS, $volume, $directory and $file are concatenated. A ':' is inserted if need be. You may pass an empty string for each portion. If all portions are empty, the empty string is returned. If $volume is empty, the result will be a relative path, beginning with a ':'. If $volume and $directory are empty, a leading ":" (if any) is removed form $file and the remainder is returned. If $file is empty, the resulting path will have a trailing ':'.

- abs2rel

Takes a destination path and an optional base path and returns a relative path from the base path to the destination path:

```
$rel_path = File::Spec->abs2rel( $path ) ;
$rel_path = File::Spec->abs2rel( $path, $base ) ;
```

Note that both paths are assumed to have a notation that distinguishes a directory path (with trailing ':') from a file path (without trailing ':').

If $base is not present or '', then the current working directory is used. If $base is relative, then it is converted to absolute form using `rel2abs()`. This means that it is taken to be relative to the current working directory.

If $path and $base appear to be on two different volumes, we will not attempt to resolve the two paths, and we will instead simply return $path. Note that previous versions of this module ignored the volume of $base, which resulted in garbage results part of the time.

If $base doesn't have a trailing colon, the last element of $base is assumed to be a filename. This filename is ignored. Otherwise all path components are assumed to

be directories.

If $path is relative, it is converted to absolute form using `rel2abs()`. This means that it is taken to be relative to the current working directory.

Based on code written by Shigio Yamaguchi.

- rel2abs

  Converts a relative path to an absolute path:

```
$abs_path = File::Spec->rel2abs( $path ) ;
$abs_path = File::Spec->rel2abs( $path, $base ) ;
```

  Note that both paths are assumed to have a notation that distinguishes a directory path (with trailing ':') from a file path (without trailing ':').

  If $base is not present or '', then $base is set to the current working directory. If $base is relative, then it is converted to absolute form using `rel2abs()`. This means that it is taken to be relative to the current working directory.

  If $base doesn't have a trailing colon, the last element of $base is assumed to be a filename. This filename is ignored. Otherwise all path components are assumed to be directories.

  If $path is already absolute, it is returned and $base is ignored.

  Based on code written by Shigio Yamaguchi.

AUTHORS

See the authors list in *File::Spec*. Mac OS support by Paul Schinder schinder@pobox.com and Thomas Wegner <wegner_thomas@yahoo.com>.

COPYRIGHT

Copyright (c) 2004 by the Perl 5 Porters. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

SEE ALSO

See *File::Spec* and *File::Spec::Unix*. This package overrides the implementation of these methods, not the semantics.

File::Spec::OS2

methods for OS/2 file specs

SYNOPSIS

```
require File::Spec::OS2; # Done internally by File::Spec if needed
```

DESCRIPTION

See *File::Spec* and *File::Spec::Unix*. This package overrides the implementation of these methods, not the semantics.

Amongst the changes made for OS/2 are...

- tmpdir

  Modifies the list of places temp directory information is looked for.

  ```
  $ENV{TMPDIR}
  $ENV{TEMP}
  $ENV{TMP}
  /tmp
  /
  ```

- splitpath

  Volumes can be drive letters or UNC sharenames (\server\share).

COPYRIGHT

Copyright (c) 2004 by the Perl 5 Porters. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

File::Spec::Unix

File::Spec for Unix, base for other File::Spec modules

SYNOPSIS

```
require File::Spec::Unix; # Done automatically by File::Spec
```

DESCRIPTION

Methods for manipulating file specifications. Other File::Spec modules, such as File::Spec::Mac, inherit from File::Spec::Unix and override specific methods.

METHODS

- canonpath()

  No physical check on the filesystem, but a logical cleanup of a path. On UNIX eliminates successive slashes and successive “/.”.

```
$cpath = File::Spec->canonpath( $path ) ;
```

Note that this does *not* collapse `x/../y` sections into `y`. This is by design. If `/foo` on your system is a symlink to `/bar/baz`, then `/foo/../quux` is actually `/bar/quux`, not `/quux` as a naive `../`-removal would give you. If you want to do this kind of processing, you probably want `Cwd`'s `realpath()` function to actually traverse the filesystem cleaning up paths like this.

- catdir()

  Concatenate two or more directory names to form a complete path ending with a directory. But remove the trailing slash from the resulting string, because it doesn't look good, isn't necessary and confuses OS2. Of course, if this is the root directory, don't cut off the trailing slash :-)

- catfile

  Concatenate one or more directory names and a filename to form a complete path ending with a filename

- curdir

  Returns a string representation of the current directory. "." on UNIX.

- devnull

  Returns a string representation of the null device. "/dev/null" on UNIX.

- rootdir

  Returns a string representation of the root directory. "/" on UNIX.

- tmpdir

  Returns a string representation of the first writable directory from the following list or the current directory if none from the list are writable:

```
$ENV{TMPDIR}
/tmp
```

  If running under taint mode, and if $ENV{TMPDIR} is tainted, it is not used.

- updir

  Returns a string representation of the parent directory. ".." on UNIX.

- no_upwards

  Given a list of file names, strip out those that refer to a parent directory. (Does not strip symlinks, only '.', '..', and equivalents.)

- case_tolerant

  Returns a true or false value indicating, respectively, that alphabetic is not or is significant when comparing file specifications.

- file_name_is_absolute

  Takes as argument a path and returns true if it is an absolute path.

This does not consult the local filesystem on Unix, Win32, OS/2 or Mac OS (Classic). It does consult the working environment for VMS (see *"file_name_is_absolute" in File::Spec::VMS*).

- path

  Takes no argument, returns the environment variable PATH as an array.

- join

  join is the same as catfile.

- splitpath

```
($volume,$directories,$file) = File::Spec->splitpath( $path );
($volume,$directories,$file) = File::Spec->splitpath( $path,
                                                      $no_file␣
↪);
```

  Splits a path into volume, directory, and filename portions. On systems with no concept of volume, returns '' for volume.

  For systems with no syntax differentiating filenames from directories, assumes that the last file is a path unless $no_file is true or a trailing separator or /. or /.. is present. On Unix this means that $no_file true makes this return ('', $path, '').

  The directory portion may or may not be returned with a trailing '/'.

  The results can be passed to "catpath()" to get back a path equivalent to (usually identical to) the original path.

- splitdir

  The opposite of "catdir()".

```
@dirs = File::Spec->splitdir( $directories );
```

  $directories must be only the directory portion of the path on systems that have the concept of a volume or that have path syntax that differentiates files from directories.

  Unlike just splitting the directories on the separator, empty directory names ('') can be returned, because these are significant on some OSs.

  On Unix,

```
File::Spec->splitdir( "/a/b//c/" );
```

  Yields:

```
( '', 'a', 'b', '', 'c', '' )
```

- catpath()

  Takes volume, directory and file portions and returns an entire path. Under Unix, $volume is ignored, and directory and file are concatenated. A '/' is inserted if needed (though if the directory portion doesn't start with '/' it is not added). On other OSs, $volume is significant.

- abs2rel

  Takes a destination path and an optional base path returns a relative path from the base path to the destination path:

```
$rel_path = File::Spec->abs2rel( $path ) ;
$rel_path = File::Spec->abs2rel( $path, $base ) ;
```

  If $base is not present or '', then *cwd()* is used. If $base is relative, then it is converted to absolute form using "rel2abs()". This means that it is taken to be relative to *cwd()*.

  On systems that have a grammar that indicates filenames, this ignores the $base filename. Otherwise all path components are assumed to be directories.

  If $path is relative, it is converted to absolute form using "rel2abs()". This means that it is taken to be relative to *cwd()*.

  No checks against the filesystem are made, so the result may not be correct if `$base` contains symbolic links. (Apply *Cwd::abs_path()* beforehand if that is a concern.) On VMS, there is interaction with the working environment, as logicals and macros are expanded.

  Based on code written by Shigio Yamaguchi.

- rel2abs()

  Converts a relative path to an absolute path.

```
$abs_path = File::Spec->rel2abs( $path ) ;
$abs_path = File::Spec->rel2abs( $path, $base ) ;
```

  If $base is not present or '', then *cwd()* is used. If $base is relative, then it is converted to absolute form using "rel2abs()". This means that it is taken to be relative to *cwd()*.

  On systems that have a grammar that indicates filenames, this ignores the $base filename. Otherwise all path components are assumed to be directories.

  If $path is absolute, it is cleaned up and returned using "canonpath()".

  No checks against the filesystem are made. On VMS, there is interaction with the working environment, as logicals and macros are expanded.

  Based on code written by Shigio Yamaguchi.

COPYRIGHT

Copyright (c) 2004 by the Perl 5 Porters. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

Please submit bug reports at https://github.com/Perl/perl5/issues.

SEE ALSO

File::Spec

Functions

File::Spec

canonpath

`File::Spec->canonpath($path)` - logical cleanup of a path.

catdir

`File::Spec->catdir(@dirs)` - join directory names into a path.

catfile

`File::Spec->catfile(@dirs, $file)` - join directories and a filename.

File::Spec::VMS

methods for VMS file specs

SYNOPSIS

```
require File::Spec::VMS; # Done internally by File::Spec if needed
```

DESCRIPTION

See File::Spec::Unix for a documentation of the methods provided there. This package overrides the implementation of these methods, not the semantics.

The default behavior is to allow either VMS or Unix syntax on input and to return VMS syntax on output unless Unix syntax has been explicitly requested via the `DECC$FILE_NAME_UNIX_REPORT` CRTL feature.

- canonpath (override)

  Removes redundant portions of file specifications and returns results in native syntax unless Unix filename reporting has been enabled.

- catdir (override)

  Concatenates a list of file specifications, and returns the result as a native directory specification unless the Unix filename reporting feature has been enabled. No check is made for "impossible" cases (e.g. elements other than the first being absolute filespecs).

- catfile (override)

  Concatenates a list of directory specifications with a filename specification to build a path.

- curdir (override)

  Returns a string representation of the current directory: '[]' or '.'

- devnull (override)

  Returns a string representation of the null device: '_NLA0:' or '/dev/null'

- rootdir (override)

  Returns a string representation of the root directory: 'SYS$DISK:[000000]' or '/'

- tmpdir (override)

  Returns a string representation of the first writable directory from the following list or '' if none are writable:

```
/tmp if C<DECC$FILENAME_UNIX_REPORT> is enabled.
sys$scratch:
$ENV{TMPDIR}
```

  If running under taint mode, and if $ENV{TMPDIR} is tainted, it is not used.

- updir (override)

  Returns a string representation of the parent directory: '[-]' or '..'

- case_tolerant (override)

  VMS file specification syntax is case-tolerant.

- path (override)

  Translate logical name DCL$PATH as a searchlist, rather than trying to `split` string value of `$ENV{'PATH'}`.

- file_name_is_absolute (override)

  Checks for VMS directory spec as well as Unix separators.

- splitpath (override)

```
($volume,$directories,$file) = File::Spec->splitpath( $path );
($volume,$directories,$file) = File::Spec->splitpath( $path,
                                                      $no_file␣
↪);
```

  Passing a true value for `$no_file` indicates that the path being split only contains directory components, even on systems where you can usually (when not supporting a foreign syntax) tell the difference between directories and files at a glance.

- splitdir (override)

  Split a directory specification into the components.

- catpath (override)

  Construct a complete filespec.

- abs2rel (override)

  Attempt to convert an absolute file specification to a relative specification.

- rel2abs (override)

  Return an absolute file specification from a relative one.

COPYRIGHT

Copyright (c) 2004-14 by the Perl 5 Porters. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

SEE ALSO

See *File::Spec* and *File::Spec::Unix*. This package overrides the implementation of these methods, not the semantics.

An explanation of VMS file specs can be found at http://h71000.www7.hp.com/doc/731FINAL/4506/4506pro_014.html#apps_locating_naming_files.

File::Spec::Win32

methods for Win32 file specs

SYNOPSIS

```
require File::Spec::Win32; # Done internally by File::Spec if needed
```

DESCRIPTION

See File::Spec::Unix for a documentation of the methods provided there. This package overrides the implementation of these methods, not the semantics.

- devnull

  Returns a string representation of the null device.

- tmpdir

  Returns a string representation of the first existing directory from the following list:

  ```
  $ENV{TMPDIR}
  $ENV{TEMP}
  $ENV{TMP}
  SYS:/temp
  C:\system\temp
  C:/temp
  /tmp
  /
  ```

  The SYS:/temp is preferred in Novell NetWare and the C:\system\temp for Symbian (the File::Spec::Win32 is used also for those platforms).

If running under taint mode, and if the environment variables are tainted, they are not used.

- case_tolerant

  MSWin32 case-tolerance depends on GetVolumeInformation() $ouFsFlags == FS_CASE_SENSITIVE, indicating the case significance when comparing file specifications. Since XP FS_CASE_SENSITIVE is effectively disabled for the NT subsubsystem. See http://cygwin.com/ml/cygwin/2007-07/msg00891.html Default: 1

- file_name_is_absolute

  As of right now, this returns 2 if the path is absolute with a volume, 1 if it's absolute with no volume, 0 otherwise.

- catfile

  Concatenate one or more directory names and a filename to form a complete path ending with a filename

- canonpath

  No physical check on the filesystem, but a logical cleanup of a path. On UNIX eliminated successive slashes and successive "/.". On Win32 makes

```
dir1\dir2\dir3\..\..\dir4 -> \dir\dir4 and even
dir1\dir2\dir3\...\dir4   -> \dir\dir4
```

- splitpath

```
($volume,$directories,$file) = File::Spec->splitpath( $path );
($volume,$directories,$file) = File::Spec->splitpath( $path,
                                                      $no_file␣
↪);
```

  Splits a path into volume, directory, and filename portions. Assumes that the last file is a path unless the path ends in '\', '\.', '\..' or $no_file is true. On Win32 this means that $no_file true makes this return ($volume, $path, '').

  Separators accepted are \ and /.

  Volumes can be drive letters or UNC sharenames (\server\share).

  The results can be passed to "catpath" to get back a path equivalent to (usually identical to) the original path.

- splitdir

  The opposite of *catdir()*.

```
@dirs = File::Spec->splitdir( $directories );
```

  $directories must be only the directory portion of the path on systems that have the concept of a volume or that have path syntax that differentiates files from directories.

  Unlike just splitting the directories on the separator, leading empty and trailing directory entries can be returned, because these are significant on some OSs. So,

```
File::Spec->splitdir( "/a/b//c/" );
```

Yields:

```
( '', 'a', 'b', '', 'c', '' )
```

- catpath

  Takes volume, directory and file portions and returns an entire path. Under Unix, $volume is ignored, and this is just like catfile(). On other OSs, the $volume become significant.

Note For File::Spec::Win32 Maintainers

Novell NetWare inherits its File::Spec behaviour from File::Spec::Win32.

COPYRIGHT

Copyright (c) 2004,2007 by the Perl 5 Porters. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

SEE ALSO

See *File::Spec* and *File::Spec::Unix*. This package overrides the implementation of these methods, not the semantics.

File::Temp

return name and handle of a temporary file safely

VERSION

version 0.2312

SYNOPSIS

```
use File::Temp qw/ tempfile tempdir /;

$fh = tempfile();
($fh, $filename) = tempfile();

($fh, $filename) = tempfile( $template, DIR => $dir);
($fh, $filename) = tempfile( $template, SUFFIX => '.dat');
($fh, $filename) = tempfile( $template, TMPDIR => 1 );

binmode( $fh, ":utf8" );

$dir = tempdir( CLEANUP => 1 );
($fh, $filename) = tempfile( DIR => $dir );
```

Object interface:

```
require File::Temp;
use File::Temp ();
use File::Temp qw/ :seekable /;

$fh = File::Temp->new();
$fname = $fh->filename;

$fh = File::Temp->new(TEMPLATE => $template);
$fname = $fh->filename;

$tmp = File::Temp->new( UNLINK => 0, SUFFIX => '.dat' );
print $tmp "Some data\n";
print "Filename is $tmp\n";
$tmp->seek( 0, SEEK_END );

$dir = File::Temp->newdir(); # CLEANUP => 1 by default
```

The following interfaces are provided for compatibility with existing APIs. They should not be used in new code.

MkTemp family:

```
use File::Temp qw/ :mktemp  /;

($fh, $file) = mkstemp( "tmpfileXXXXX" );
($fh, $file) = mkstemps( "tmpfileXXXXXX", $suffix);

$tmpdir = mkdtemp( $template );

$unopened_file = mktemp( $template );
```

POSIX functions:

```
use File::Temp qw/ :POSIX /;

$file = tmpnam();
$fh = tmpfile();

($fh, $file) = tmpnam();
```

Compatibility functions:

```
$unopened_file = File::Temp::tempnam( $dir, $pfx );
```

DESCRIPTION

`File::Temp` can be used to create and open temporary files in a safe way. There is both a function interface and an object-oriented interface. The File::Temp constructor or the tempfile() function can be used to return the name and the open filehandle of a temporary file. The tempdir() function can be used to create a temporary directory.

The security aspect of temporary file creation is emphasized such that a filehandle and filename are returned together. This helps guarantee that a race condition can not occur where the temporary file is created by another process between checking for the existence of the file and its opening. Additional security levels are provided to check, for example, that the sticky bit is set on world writable directories. See “safe_level” for more information.

For compatibility with popular C library functions, Perl implementations of the mkstemp() family of functions are provided. These are, mkstemp(), mkstemps(), mkdtemp() and mktemp().

Additionally, implementations of the standard *POSIX* tmpnam() and tmpfile() functions are provided if required.

Implementations of mktemp(), tmpnam(), and tempnam() are provided, but should be used with caution since they return only a filename that was valid when function was called, so cannot guarantee that the file will not exist by the time the caller opens the filename.

Filehandles returned by these functions support the seekable methods.

OBJECT-ORIENTED INTERFACE

This is the primary interface for interacting with `File::Temp`. Using the OO interface a temporary file can be created when the object is constructed and the file can be removed when the object is no longer required.

Note that there is no method to obtain the filehandle from the `File::Temp` object. The object itself acts as a filehandle. The object isa `IO::Handle` and isa `IO::Seekable` so all those methods are available.

Also, the object is configured such that it stringifies to the name of the temporary file and so can be compared to a filename directly. It numifies to the `refaddr` the same as other handles and so can be compared to other handles with ==.

```
$fh eq $filename        # as a string
$fh != \*STDOUT         # as a number
```

Available since 0.14.

- **new**

  Create a temporary file object.

  ```
  my $tmp = File::Temp->new();
  ```

  by default the object is constructed as if `tempfile` was called without options, but with the additional behaviour that the temporary file is removed by the object destructor if UNLINK is set to true (the default).

  Supported arguments are the same as for `tempfile`: UNLINK (defaulting to true), DIR, EXLOCK, PERMS and SUFFIX. Additionally, the filename template is specified using the TEMPLATE option. The OPEN option is not supported (the file is always opened).

```
$tmp = File::Temp->new( TEMPLATE => 'tempXXXXX',
                        DIR => 'mydir',
                        SUFFIX => '.dat');
```

Arguments are case insensitive.

Can call croak() if an error occurs.

Available since 0.14.

TEMPLATE available since 0.23

- **newdir**

  Create a temporary directory using an object oriented interface.

```
$dir = File::Temp->newdir();
```

  By default the directory is deleted when the object goes out of scope.

  Supports the same options as the `tempdir` function. Note that directories created with this method default to CLEANUP => 1.

```
$dir = File::Temp->newdir( $template, %options );
```

  A template may be specified either with a leading template or with a TEMPLATE argument.

  Available since 0.19.

  TEMPLATE available since 0.23.

- **filename**

  Return the name of the temporary file associated with this object (if the object was created using the "new" constructor).

```
$filename = $tmp->filename;
```

  This method is called automatically when the object is used as a string.

  Current API available since 0.14

- **dirname**

  Return the name of the temporary directory associated with this object (if the object was created using the "newdir" constructor).

```
$dirname = $tmpdir->dirname;
```

  This method is called automatically when the object is used in string context.

- **unlink_on_destroy**

  Control whether the file is unlinked when the object goes out of scope. The file is removed if this value is true and $KEEP_ALL is not.

```
$fh->unlink_on_destroy( 1 );
```

Default is for the file to be removed.

Current API available since 0.15

- **DESTROY**

  When the object goes out of scope, the destructor is called. This destructor will attempt to unlink the file (using unlink1) if the constructor was called with UNLINK set to 1 (the default state if UNLINK is not specified).

  No error is given if the unlink fails.

  If the object has been passed to a child process during a fork, the file will be deleted when the object goes out of scope in the parent.

  For a temporary directory object the directory will be removed unless the CLEANUP argument was used in the constructor (and set to false) or `unlink_on_destroy` was modified after creation. Note that if a temp directory is your current directory, it cannot be removed - a warning will be given in this case. `chdir()` out of the directory before letting the object go out of scope.

  If the global variable $KEEP_ALL is true, the file or directory will not be removed.

FUNCTIONS

This section describes the recommended interface for generating temporary files and directories.

- **tempfile**

  This is the basic function to generate temporary files. The behaviour of the file can be changed using various options:

```
$fh = tempfile();
($fh, $filename) = tempfile();
```

  Create a temporary file in the directory specified for temporary files, as specified by the tmpdir() function in *File::Spec*.

```
($fh, $filename) = tempfile($template);
```

  Create a temporary file in the current directory using the supplied template. Trailing `X' characters are replaced with random letters to generate the filename. At least four `X' characters must be present at the end of the template.

```
($fh, $filename) = tempfile($template, SUFFIX => $suffix)
```

  Same as previously, except that a suffix is added to the template after the `X' translation. Useful for ensuring that a temporary filename has a particular extension when needed by other applications. But see the WARNING at the end.

```
($fh, $filename) = tempfile($template, DIR => $dir);
```

  Translates the template as before except that a directory name is specified.

```
($fh, $filename) = tempfile($template, TMPDIR => 1);
```

Equivalent to specifying a DIR of "File::Spec->tmpdir", writing the file into the same temporary directory as would be used if no template was specified at all.

```
($fh, $filename) = tempfile($template, UNLINK => 1);
```

Return the filename and filehandle as before except that the file is automatically removed when the program exits (dependent on $KEEP_ALL). Default is for the file to be removed if a file handle is requested and to be kept if the filename is requested. In a scalar context (where no filename is returned) the file is always deleted either (depending on the operating system) on exit or when it is closed (unless $KEEP_ALL is true when the temp file is created).

Use the object-oriented interface if fine-grained control of when a file is removed is required.

If the template is not specified, a template is always automatically generated. This temporary file is placed in tmpdir() (*File::Spec*) unless a directory is specified explicitly with the DIR option.

```
$fh = tempfile( DIR => $dir );
```

If called in scalar context, only the filehandle is returned and the file will automatically be deleted when closed on operating systems that support this (see the description of tmpfile() elsewhere in this document). This is the preferred mode of operation, as if you only have a filehandle, you can never create a race condition by fumbling with the filename. On systems that can not unlink an open file or can not mark a file as temporary when it is opened (for example, Windows NT uses the `O_TEMPORARY` flag) the file is marked for deletion when the program ends (equivalent to setting UNLINK to 1). The `UNLINK` flag is ignored if present.

```
(undef, $filename) = tempfile($template, OPEN => 0);
```

This will return the filename based on the template but will not open this file. Cannot be used in conjunction with UNLINK set to true. Default is to always open the file to protect from possible race conditions. A warning is issued if warnings are turned on. Consider using the tmpnam() and mktemp() functions described elsewhere in this document if opening the file is not required.

To open the temporary filehandle with O_EXLOCK (open with exclusive file lock) use `EXLOCK=>1`. This is supported only by some operating systems (most notably BSD derived systems). By default EXLOCK will be false. Former `File::Temp` versions set EXLOCK to true, so to be sure to get an unlocked filehandle also with older versions, explicitly set `EXLOCK=>0`.

```
($fh, $filename) = tempfile($template, EXLOCK => 1);
```

By default, the temp file is created with 0600 file permissions. Use `PERMS` to change this:

```
($fh, $filename) = tempfile($template, PERMS => 0666);
```

Options can be combined as required.

Will croak() if there is an error.

Available since 0.05.

UNLINK flag available since 0.10.

TMPDIR flag available since 0.19.

EXLOCK flag available since 0.19.

PERMS flag available since 0.2310.

- **tempdir**

  This is the recommended interface for creation of temporary directories. By default the directory will not be removed on exit (that is, it won't be temporary; this behaviour can not be changed because of issues with backwards compatibility). To enable removal either use the CLEANUP option which will trigger removal on program exit, or consider using the "newdir" method in the object interface which will allow the directory to be cleaned up when the object goes out of scope.

  The behaviour of the function depends on the arguments:

```
$tempdir = tempdir();
```

  Create a directory in tmpdir() (see *File::Spec*).

```
$tempdir = tempdir( $template );
```

  Create a directory from the supplied template. This template is similar to that described for tempfile(). `X' characters at the end of the template are replaced with random letters to construct the directory name. At least four `X' characters must be in the template.

```
$tempdir = tempdir ( DIR => $dir );
```

  Specifies the directory to use for the temporary directory. The temporary directory name is derived from an internal template.

```
$tempdir = tempdir ( $template, DIR => $dir );
```

  Prepend the supplied directory name to the template. The template should not include parent directory specifications itself. Any parent directory specifications are removed from the template before prepending the supplied directory.

```
$tempdir = tempdir ( $template, TMPDIR => 1 );
```

  Using the supplied template, create the temporary directory in a standard location for temporary files. Equivalent to doing

```
$tempdir = tempdir ( $template, DIR => File::Spec->tmpdir);
```

  but shorter. Parent directory specifications are stripped from the template itself. The `TMPDIR` option is ignored if `DIR` is set explicitly. Additionally, `TMPDIR` is implied if neither a template nor a directory are supplied.

```
$tempdir = tempdir( $template, CLEANUP => 1);
```

Create a temporary directory using the supplied template, but attempt to remove it (and all files inside it) when the program exits. Note that an attempt will be made to remove all files from the directory even if they were not created by this module (otherwise why ask to clean it up?). The directory removal is made with the rmtree() function from the *File::Path* module. Of course, if the template is not specified, the temporary directory will be created in tmpdir() and will also be removed at program exit.

Will croak() if there is an error.

Current API available since 0.05.

MKTEMP FUNCTIONS

The following functions are Perl implementations of the mktemp() family of temp file generation system calls.

- **mkstemp**

  Given a template, returns a filehandle to the temporary file and the name of the file.

```
($fh, $name) = mkstemp( $template );
```

  In scalar context, just the filehandle is returned.

  The template may be any filename with some number of X's appended to it, for example `/tmp/temp.XXXX`. The trailing X's are replaced with unique alphanumeric combinations.

  Will croak() if there is an error.

  Current API available since 0.05.

- **mkstemps**

  Similar to mkstemp(), except that an extra argument can be supplied with a suffix to be appended to the template.

```
($fh, $name) = mkstemps( $template, $suffix );
```

  For example a template of `testXXXXXX` and suffix of `.dat` would generate a file similar to `testhGji_w.dat`.

  Returns just the filehandle alone when called in scalar context.

  Will croak() if there is an error.

  Current API available since 0.05.

- **mkdtemp**

  Create a directory from a template. The template must end in X's that are replaced by the routine.

```
$tmpdir_name = mkdtemp($template);
```

  Returns the name of the temporary directory created.

Directory must be removed by the caller.

Will croak() if there is an error.

Current API available since 0.05.

- **mktemp**

  Returns a valid temporary filename but does not guarantee that the file will not be opened by someone else.

```
$unopened_file = mktemp($template);
```

  Template is the same as that required by mkstemp().

  Will croak() if there is an error.

  Current API available since 0.05.

POSIX FUNCTIONS

This section describes the re-implementation of the tmpnam() and tmpfile() functions described in *POSIX* using the mkstemp() from this module.

Unlike the *POSIX* implementations, the directory used for the temporary file is not specified in a system include file (`P_tmpdir`) but simply depends on the choice of tmpdir() returned by *File::Spec*. On some implementations this location can be set using the `TMPDIR` environment variable, which may not be secure. If this is a problem, simply use mkstemp() and specify a template.

- **tmpnam**

  When called in scalar context, returns the full name (including path) of a temporary file (uses mktemp()). The only check is that the file does not already exist, but there is no guarantee that that condition will continue to apply.

```
$file = tmpnam();
```

  When called in list context, a filehandle to the open file and a filename are returned. This is achieved by calling mkstemp() after constructing a suitable template.

```
($fh, $file) = tmpnam();
```

  If possible, this form should be used to prevent possible race conditions.

  See *"tmpdir" in File::Spec* for information on the choice of temporary directory for a particular operating system.

  Will croak() if there is an error.

  Current API available since 0.05.

- **tmpfile**

  Returns the filehandle of a temporary file.

```
$fh = tmpfile();
```

The file is removed when the filehandle is closed or when the program exits. No access to the filename is provided.

If the temporary file can not be created undef is returned. Currently this command will probably not work when the temporary directory is on an NFS file system.

Will croak() if there is an error.

Available since 0.05.

Returning undef if unable to create file added in 0.12.

ADDITIONAL FUNCTIONS

These functions are provided for backwards compatibility with common tempfile generation C library functions.

They are not exported and must be addressed using the full package name.

- **tempnam**

  Return the name of a temporary file in the specified directory using a prefix. The file is guaranteed not to exist at the time the function was called, but such guarantees are good for one clock tick only. Always use the proper form of `sysopen` with `O_CREAT | O_EXCL` if you must open such a filename.

```
$filename = File::Temp::tempnam( $dir, $prefix );
```

  Equivalent to running mktemp() with $dir/$prefixXXXXXXXX (using unix file convention as an example)

  Because this function uses mktemp(), it can suffer from race conditions.

  Will croak() if there is an error.

  Current API available since 0.05.

UTILITY FUNCTIONS

Useful functions for dealing with the filehandle and filename.

- **unlink0**

  Given an open filehandle and the associated filename, make a safe unlink. This is achieved by first checking that the filename and filehandle initially point to the same file and that the number of links to the file is 1 (all fields returned by stat() are compared). Then the filename is unlinked and the filehandle checked once again to verify that the number of links on that file is now 0. This is the closest you can come to making sure that the filename unlinked was the same as the file whose descriptor you hold.

```
unlink0($fh, $path)
   or die "Error unlinking file $path safely";
```

  Returns false on error but croaks() if there is a security anomaly. The filehandle is not closed since on some occasions this is not required.

On some platforms, for example Windows NT, it is not possible to unlink an open file (the file must be closed first). On those platforms, the actual unlinking is deferred until the program ends and good status is returned. A check is still performed to make sure that the filehandle and filename are pointing to the same thing (but not at the time the end block is executed since the deferred removal may not have access to the filehandle).

Additionally, on Windows NT not all the fields returned by stat() can be compared. For example, the `dev` and `rdev` fields seem to be different. Also, it seems that the size of the file returned by stat() does not always agree, with `stat(FH)` being more accurate than `stat(filename)`, presumably because of caching issues even when using autoflush (this is usually overcome by waiting a while after writing to the tempfile before attempting to `unlink0` it).

Finally, on NFS file systems the link count of the file handle does not always go to zero immediately after unlinking. Currently, this command is expected to fail on NFS disks.

This function is disabled if the global variable $KEEP_ALL is true and an unlink on open file is supported. If the unlink is to be deferred to the END block, the file is still registered for removal.

This function should not be called if you are using the object oriented interface since the it will interfere with the object destructor deleting the file.

Available Since 0.05.

If can not unlink open file, defer removal until later available since 0.06.

- **cmpstat**

  Compare `stat` of filehandle with `stat` of provided filename. This can be used to check that the filename and filehandle initially point to the same file and that the number of links to the file is 1 (all fields returned by stat() are compared).

```
cmpstat($fh, $path)
   or die "Error comparing handle with file";
```

  Returns false if the stat information differs or if the link count is greater than 1. Calls croak if there is a security anomaly.

  On certain platforms, for example Windows, not all the fields returned by stat() can be compared. For example, the `dev` and `rdev` fields seem to be different in Windows. Also, it seems that the size of the file returned by stat() does not always agree, with `stat(FH)` being more accurate than `stat(filename)`, presumably because of caching issues even when using autoflush (this is usually overcome by waiting a while after writing to the tempfile before attempting to `unlink0` it).

  Not exported by default.

  Current API available since 0.14.

- **unlink1**

  Similar to `unlink0` except after file comparison using cmpstat, the filehandle is closed prior to attempting to unlink the file. This allows the file to be removed without using an END block, but does mean that the post-unlink comparison of the filehandle state provided by `unlink0` is not available.

```
unlink1($fh, $path)
   or die "Error closing and unlinking file";
```

Usually called from the object destructor when using the OO interface.

Not exported by default.

This function is disabled if the global variable $KEEP_ALL is true.

Can call croak() if there is a security anomaly during the stat() comparison.

Current API available since 0.14.

- **cleanup**

  Calling this function will cause any temp files or temp directories that are registered for removal to be removed. This happens automatically when the process exits but can be triggered manually if the caller is sure that none of the temp files are required. This method can be registered as an Apache callback.

  Note that if a temp directory is your current directory, it cannot be removed. `chdir()` out of the directory first before calling `cleanup()`. (For the cleanup at program exit when the CLEANUP flag is set, this happens automatically.)

  On OSes where temp files are automatically removed when the temp file is closed, calling this function will have no effect other than to remove temporary directories (which may include temporary files).

```
File::Temp::cleanup();
```

  Not exported by default.

  Current API available since 0.15.

PACKAGE VARIABLES

These functions control the global state of the package.

- **safe_level**

  Controls the lengths to which the module will go to check the safety of the temporary file or directory before proceeding. Options are:

  – STANDARD

    Do the basic security measures to ensure the directory exists and is writable, that temporary files are opened only if they do not already exist, and that possible race conditions are avoided. Finally the unlink0 function is used to remove files safely.

  – MEDIUM

    In addition to the STANDARD security, the output directory is checked to make sure that it is owned either by root or the user running the program. If the directory is writable by group or by other, it is then checked to make sure that the sticky bit is set.

    Will not work on platforms that do not support the `-k` test for sticky bit.

– HIGH

In addition to the MEDIUM security checks, also check for the possibility of ``chown() giveaway'' using the *POSIX* sysconf() function. If this is a possibility, each directory in the path is checked in turn for safeness, recursively walking back to the root directory.

For platforms that do not support the *POSIX* `_PC_CHOWN_RESTRICTED` symbol (for example, Windows NT) it is assumed that ``chown() giveaway'' is possible and the recursive test is performed.

The level can be changed as follows:

```
File::Temp->safe_level( File::Temp::HIGH );
```

The level constants are not exported by the module.

Currently, you must be running at least perl v5.6.0 in order to run with MEDIUM or HIGH security. This is simply because the safety tests use functions from *Fcntl* that are not available in older versions of perl. The problem is that the version number for Fcntl is the same in perl 5.6.0 and in 5.005_03 even though they are different versions.

On systems that do not support the HIGH or MEDIUM safety levels (for example Win NT or OS/2) any attempt to change the level will be ignored. The decision to ignore rather than raise an exception allows portable programs to be written with high security in mind for the systems that can support this without those programs failing on systems where the extra tests are irrelevant.

If you really need to see whether the change has been accepted simply examine the return value of `safe_level`.

```
$newlevel = File::Temp->safe_level( File::Temp::HIGH );
die "Could not change to high security"
    if $newlevel != File::Temp::HIGH;
```

Available since 0.05.

- TopSystemUID

This is the highest UID on the current system that refers to a root UID. This is used to make sure that the temporary directory is owned by a system UID (`root`, `bin`, `sys` etc) rather than simply by root.

This is required since on many unix systems `/tmp` is not owned by root.

Default is to assume that any UID less than or equal to 10 is a root UID.

```
File::Temp->top_system_uid(10);
my $topid = File::Temp->top_system_uid;
```

This value can be adjusted to reduce security checking if required. The value is only relevant when `safe_level` is set to MEDIUM or higher.

Available since 0.05.

- **$KEEP_ALL**

Controls whether temporary files and directories should be retained regardless of any instructions in the program to remove them automatically. This is useful for debugging but should not be used in production code.

```
$File::Temp::KEEP_ALL = 1;
```

Default is for files to be removed as requested by the caller.

In some cases, files will only be retained if this variable is true when the file is created. This means that you can not create a temporary file, set this variable and expect the temp file to still be around when the program exits.

- **$DEBUG**

  Controls whether debugging messages should be enabled.

```
$File::Temp::DEBUG = 1;
```

  Default is for debugging mode to be disabled.

  Available since 0.15.

WARNING

For maximum security, endeavour always to avoid ever looking at, touching, or even imputing the existence of the filename. You do not know that that filename is connected to the same file as the handle you have, and attempts to check this can only trigger more race conditions. It's far more secure to use the filehandle alone and dispense with the filename altogether.

If you need to pass the handle to something that expects a filename then on a unix system you can use `"/dev/fd/" . fileno($fh)` for arbitrary programs. Perl code that uses the 2-argument version of open can be passed `"+<=&" . fileno($fh)`. Otherwise you will need to pass the filename. You will have to clear the close-on-exec bit on that file descriptor before passing it to another process.

```
use Fcntl qw/F_SETFD F_GETFD/;
fcntl($tmpfh, F_SETFD, 0)
    or die "Can't clear close-on-exec flag on temp fh: $!\n";
```

Temporary files and NFS

Some problems are associated with using temporary files that reside on NFS file systems and it is recommended that a local filesystem is used whenever possible. Some of the security tests will most probably fail when the temp file is not local. Additionally, be aware that the performance of I/O operations over NFS will not be as good as for a local disk.

Forking

In some cases files created by File::Temp are removed from within an END block. Since END blocks are triggered when a child process exits (unless `POSIX::_exit()` is used by the child) File::Temp takes care to only remove those temp files created by a particular

process ID. This means that a child will not attempt to remove temp files created by the parent process.

If you are forking many processes in parallel that are all creating temporary files, you may need to reset the random number seed using srand(EXPR) in each child else all the children will attempt to walk through the same set of random file names and may well cause themselves to give up if they exceed the number of retry attempts.

Directory removal

Note that if you have chdir'ed into the temporary directory and it is subsequently cleaned up (either in the END block or as part of object destruction), then you will get a warning from File::Path::rmtree().

Taint mode

If you need to run code under taint mode, updating to the latest *File::Spec* is highly recommended. On Windows, if the directory given by File::Spec::tmpdir isn't writable, File::Temp will attempt to fallback to the user's local application data directory or croak with an error.

BINMODE

The file returned by File::Temp will have been opened in binary mode if such a mode is available. If that is not correct, use the `binmode()` function to change the mode of the filehandle.

Note that you can modify the encoding of a file opened by File::Temp also by using `binmode()`.

HISTORY

In March 2000, the OpenBSD mkstemp() code was translated to Perl for total control of the code's security checking, to ensure the presence of the function regardless of operating system and to help with portability. The module was shipped as a standard part of perl from v5.6.1.

Thanks to Tom Christiansen for suggesting that this module should be written and providing ideas for code improvements and security enhancements.

SEE ALSO

"tmpnam" in POSIX, *"tmpfile" in POSIX*, *File::Spec*, *File::Path*

See *IO::File* and File::MkTemp, Apache::TempFile for different implementations of temporary file handling.

See File::Tempdir for an alternative object-oriented wrapper for the `tempdir` function.

SUPPORT

Bugs may be submitted through the RT bug tracker (or bug-File-Temp@rt.cpan.org).

There is also a mailing list available for users of this distribution, at http://lists.perl.org/list/cpan-workers.html.

There is also an irc channel available for users of this distribution, at `#toolchain` on `irc.perl.org`.

AUTHOR

Tim Jenness tjenness@cpan.org

CONTRIBUTORS

- Karen Etheridge ether@cpan.org
- David Golden dagolden@cpan.org
- Slaven Rezic srezic@cpan.org
- mohawk2 mohawk2@users.noreply.github.com
- Roy Ivy III rivy.dev@gmail.com
- Craig A. Berry craigberry@mac.com
- Olivier Mengué dolmen@cpan.org
- Peter Rabbitson ribasushi@cpan.org
- Ben Tilly btilly@gmail.com
- Brian Mowrey brian@drlabs.org
- Dagfinn Ilmari Mannsåker ilmari@ilmari.org
- David Steinbrunner dsteinbrunner@pobox.com
- Ed Avis eda@linux01.wcl.local
- Guillem Jover guillem@hadrons.org
- James E. Keenan jkeen@verizon.net
- Kevin Ryde user42@zip.com.au
- mauke lukasmai.403@gmail.com
- Nicolas R atoomic@cpan.org
- Peter John Acklam pjacklam@online.no
- Tim Gim Yee tim.gim.yee@gmail.com

COPYRIGHT AND LICENSE

This software is copyright (c) 2025 by Tim Jenness and the UK Particle Physics and Astronomy Research Council.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

File::Which

Perl implementation of the which utility as an API

VERSION

version 1.27

SYNOPSIS

```
use File::Which;                    # exports which()
use File::Which qw(which where);  # exports which() and where()

my $exe_path = which 'perldoc';

my @paths = where 'perl';
# Or
my @paths = which 'perl'; # an array forces search for all of them
```

DESCRIPTION

File::Which finds the full or relative paths to executable programs on the system. This is normally the function of `which` utility. `which` is typically implemented as either a program or a built in shell command. On some platforms, such as Microsoft Windows it is not provided as part of the core operating system. This module provides a consistent API to this functionality regardless of the underlying platform.

The focus of this module is correctness and portability. As a consequence platforms where the current directory is implicitly part of the search path such as Microsoft Windows will find executables in the current directory, whereas on platforms such as UNIX where this is not the case executables in the current directory will only be found if the current directory is explicitly added to the path.

If you need a portable `which` on the command line in an environment that does not provide it, install App::pwhich which provides a command line interface to this API.

Implementations

File::Which searches the directories of the user's `PATH` (the current implementation uses *File::Spec#path* to determine the correct `PATH`), looking for executable files having the name specified as a parameter to "which". Under Win32 systems, which do not have a notion of directly executable files, but uses special extensions such as `.exe` and `.bat` to identify them, `File::Which` takes extra steps to assure that you will find the correct file (so for example, you might be searching for `perl`, it'll try `perl.exe`, `perl.bat`, etc.)

Linux, *BSD and other UNIXes

There should not be any surprises here. The current directory will not be searched unless it is explicitly added to the path.

Modern Windows (including NT, XP, Vista, 7, 8, 10 etc)

Windows NT has a special environment variable called `PATHEXT`, which is used by the shell to look for executable files. Usually, it will contain a list in the form `.EXE;.BAT;.COM;.JS;.VBS` etc. If `File::Which` finds such an environment variable, it parses the list and uses it as the different extensions.

Cygwin

Cygwin provides a Unix-like environment for Microsoft Windows users. In most ways it works like other Unix and Unix-like environments, but in a few key aspects it works like Windows. As with other Unix environments, the current directory is not included in the search unless it is explicitly included in the search path. Like on Windows, files with `.EXE` or <.BAT> extensions will be discovered even if they are not part of the query. `.COM` or extensions specified using the `PATHEXT` environment variable will NOT be discovered without the fully qualified name, however.

Windows ME, 98, 95, MS-DOS, OS/2

This set of operating systems don't have the `PATHEXT` variable, and usually you will find executable files there with the extensions `.exe`, `.bat` and (less likely) `.com`. `File::Which` uses this hardcoded list if it's running under Win32 but does not find a `PATHEXT` variable.

As of 2015 none of these platforms are tested frequently (or perhaps ever), but the current maintainer is determined not to intentionally remove support for older operating systems.

VMS

Same case as Windows 9x: uses `.exe` and `.com` (in that order).

As of 2015 the current maintainer does not test on VMS, and is in fact not certain it has ever been tested on VMS. If this platform is important to you and you can help me verify and or support it on that platform please contact me.

FUNCTIONS

which

```
my $path = which $short_exe_name;
my @paths = which $short_exe_name;
```

Exported by default.

`$short_exe_name` is the name used in the shell to call the program (for example, `perl`).

If it finds an executable with the name you specified, `which()` will return the absolute path leading to this executable (for example, `/usr/bin/perl` or `C:\Perl\Bin\perl.exe`).

If it does *not* find the executable, it returns `undef`.

If `which()` is called in list context, it will return *all* the matches.

where

```
my @paths = where $short_exe_name;
```

Not exported by default.

Same as "which" in array context. Similar to the `where` csh built-in command or `which -a` command for platforms that support the `-a` option. Will return an array containing all the path names matching `$short_exe_name`.

GLOBALS

$IMPLICIT_CURRENT_DIR

True if the current directory is included in the search implicitly on whatever platform you are using. Normally the default is reasonable, but on Windows the current directory is included implicitly for older shells like `cmd.exe` and `command.com`, but not for newer shells like PowerShell. If you overrule this default, you should ALWAYS localize the variable to the tightest scope possible, since setting this variable from a module can affect other modules. Thus on Windows you can get the correct result if the user is running either `cmd.exe` or PowerShell on Windows you can do this:

```
use File::Which qw( which );
use Shell::Guess;

my $path = do {
  my $is_power = Shell::Guess->running_shell->is_power;
  local $File::Which::IMPLICIT_CURRENT_DIR = !$is_power;
  which 'foo';
};
```

For a variety of reasons it is difficult to accurately compute the shell that a user is using, but Shell::Guess makes a reasonable effort.

CAVEATS

This module has no non-core requirements for Perl 5.6.2 and better.

This module is fully supported back to Perl 5.8.1. It may work on 5.8.0. It should work on Perl 5.6.x and I may even test on 5.6.2. I will accept patches to maintain compatibility for such older Perls, but you may need to fix it on 5.6.x / 5.8.0 and send me a patch.

Not tested on VMS although there is platform specific code for those. Anyone who haves a second would be very kind to send me a report of how it went.

SUPPORT

Bugs should be reported via the GitHub issue tracker

https://github.com/uperl/File-Which/issues

For other issues, contact the maintainer.

SEE ALSO

- pwhich, App::pwhich

  Command line interface to this module.

- *IPC::Cmd*

  Requires Perl 5.8.3. Included as part of the Perl core as of 5.9.5.

  This module provides (among other things) a `can_run` function, which is similar to `which`. It is a much heavier module since it does a lot more, and if you use `can_run` it pulls in *ExtUtils::MakeMaker*. This combination may be overkill for applications which do not need *IPC::Cmd*'s complicated interface for running programs, or do not need the memory overhead required for installing Perl modules.

  At least some older versions will find executables in the current directory, even if the current directory is not in the search path (which is the default on modern Unix).

  `can_run` converts directory path name to the 8.3 version on Windows using `Win32::GetShortPathName` in some cases. This is frequently useful for tools that just need to run something using `system` in scalar mode, but may be inconvenient for tools like App::pwhich where user readability is a premium. Relying on `Win32::GetShortPathName` to produce filenames without spaces is problematic, as 8.3 filenames can be turned off with tweaks to the registry (see https://technet.microsoft.com/en-us/library/cc959352.aspx).

- Devel::CheckBin

  Requires Perl 5.8.1.

  This module purports to "check that a command is available", but does not provide any documentation on how you might use it.

  This module also relies on *ExtUtils::MakeMaker* so has the same overhead burdens as *IPC::Cmd*.

AUTHORS

- Per Einar Ellefsen pereinar@cpan.org
- Adam Kennedy adamk@cpan.org
- Graham Ollis plicease@cpan.org

COPYRIGHT AND LICENSE

This software is copyright (c) 2002 by Per Einar Ellefsen pereinar@cpan.org.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

File::stat

by-name interface to Perl's built-in stat() functions

SYNOPSIS

```
use File::stat;
my $st = stat($file) or die "No $file: $!";
if ( ($st->mode & 0111) && ($st->nlink > 1) ) {
    print "$file is executable with lotsa links\n";
}

if ( -x $st ) {
    print "$file is executable\n";
}

use Fcntl "S_IRUSR";
if ( $st->cando(S_IRUSR, 1) ) {
    print "My effective uid can read $file\n";
}

use File::stat qw(:FIELDS);
stat($file) or die "No $file: $!";
if ( ($st_mode & 0111) && ($st_nlink > 1) ) {
    print "$file is executable with lotsa links\n";
}
```

DESCRIPTION

This module's default exports override the core stat() and lstat() functions, replacing them with versions that return "File::stat" objects. This object has methods that return the similarly named structure field name from the stat(2) function; namely, dev, ino, mode, nlink, uid, gid, rdev, size, atime, mtime, ctime, blksize, and blocks.

As of version 1.02 (provided with perl 5.12) the object provides `"-X"` overloading, so you can call filetest operators (`-f`, `-x`, and so on) on it. It also provides a `->cando` method, called like

```
$st->cando( ACCESS, EFFECTIVE )
```

where *ACCESS* is one of `S_IRUSR`, `S_IWUSR` or `S_IXUSR` from the *Fcntl* module, and *EFFECTIVE* indicates whether to use effective (true) or real (false) ids. The method interprets the `mode`, `uid` and `gid` fields, and returns whether or not the current process would be allowed the specified access.

If you don't want to use the objects, you may import the `->cando` method into your namespace as a regular function called `stat_cando`. This takes an arrayref containing the return values of `stat` or `lstat` as its first argument, and interprets it for you.

You may also import all the structure fields directly into your namespace as regular variables using the :FIELDS import tag. (Note that this still overrides your stat() and lstat() functions.) Access these fields as variables named with a preceding `st_` in front their method names. Thus, `$stat_obj->dev()` corresponds to $st_dev if you import the fields.

To access this functionality without the core overrides, pass the `use` an empty import list, and then access functions with their full qualified names:

```
use File::stat ();
my $st = File::stat::stat($file);
```

On the other hand, the built-ins are still available via the `CORE::` pseudo-package even if you let File::stat override them:

```
use File::stat;
my ($dev, $ino, $mode) = CORE::stat($file);
```

As of version 1.15 (provided with perl 5.44) `stat` and `lstat` can be called without arguments, in which case $_ is used (just like the built-in `stat`/`lstat` functions):

```
my $st_1 = stat;   # stat($_)
my $st_2 = lstat;  # lstat($_)
```

BUGS

The built-in `stat` and `lstat` functions recognize the special filehandle _ (underscore) to indicate that no actual `stat` be done; instead the results of the last `stat` or `lstat` or filetest operation should be returned. This syntax does not work with File::stat, but the same result can be achieved by passing `stat` a reference to the *_ typeglob:

```
use File::stat;
my $stat_obj = stat \*_;  # reuse results of last stat operation
```

Alternatively, another workaround is to explicitly populate the object using the unexported and undocumented populate() function with CORE::stat():

```
my $stat_obj = File::stat::populate(CORE::stat(_));
```

ERRORS

- -%s is not implemented on a File::stat object

  The filetest operators `-t`, `-T` and `-B` are not implemented, as they require more information than just a stat buffer.

WARNINGS

These can all be disabled with

```
no warnings "File::stat";
```

- File::stat ignores use filetest 'access'

  You have tried to use one of the `-rwxRWX` filetests with `use filetest 'access'` in effect. `File::stat` will ignore the pragma, and just use the information in the `mode` member as usual.

- File::stat ignores VMS ACLs

  VMS systems have a permissions structure that cannot be completely represented in a stat buffer, and unlike on other systems the builtin filetest operators respect this. The `File::stat` overloads, however, do not, since the information required is not available.

NOTE

While this class is currently implemented using the *Class::Struct* module to build a struct-like class, you shouldn't rely upon this.

AUTHOR

Tom Christiansen

5.2.45 FileCache

keep more files open than the system permits

SYNOPSIS

```
no strict 'refs';

use FileCache;
# or
use FileCache maxopen => 16;

cacheout $mode, $path;
# or
cacheout $path;
print $path @data;

$fh = cacheout $mode, $path;
# or
$fh = cacheout $path;
print $fh @data;
```

DESCRIPTION

The `cacheout` function will make sure that there's a filehandle open for reading or writing available as the pathname you give it. It automatically closes and re-opens files if you exceed your system's maximum number of file descriptors, or the suggested maximum *maxopen*.

- cacheout EXPR

  The 1-argument form of cacheout will open a file for writing (`'>'`) on it's first use, and appending (`'>>'`) thereafter.

  Returns EXPR on success for convenience. You may neglect the return value and manipulate EXPR as the filehandle directly if you prefer.

- cacheout MODE, EXPR

  The 2-argument form of cacheout will use the supplied mode for the initial and subsequent openings. Most valid modes for 3-argument open are supported namely; `'>'`, `'+>'`, `'<'`, `'<+'`, `'>>'`, `'|-'` and `'-|'`

  To pass supplemental arguments to a program opened with `'|-'` or `'-|'` append them to the command string as you would system EXPR.

  Returns EXPR on success for convenience. You may neglect the return value and manipulate EXPR as the filehandle directly if you prefer.

CAVEATS

While it is permissible to `close` a FileCache managed file, do not do so if you are calling `FileCache::cacheout` from a package other than which it was imported, or with another module which overrides `close`. If you must, use `FileCache::cacheout_close`.

Although FileCache can be used with piped opens ('-|' or '|-') doing so is strongly discouraged. If FileCache finds it necessary to close and then reopen a pipe, the command at the far end of the pipe will be reexecuted - the results of performing IO on FileCache'd pipes is unlikely to be what you expect. The ability to use FileCache on pipes may be removed in a future release.

FileCache does not store the current file offset if it finds it necessary to close a file. When the file is reopened, the offset will be as specified by the original open file mode. This could be construed to be a bug.

The module functionality relies on symbolic references, so things will break under 'use strict' unless 'no strict "refs"' is also specified.

BUGS

`sys/param.h` lies with its `NOFILE` define on some systems, so you may have to set *maxopen* yourself.

5.2.46 FileHandle

supply object methods for filehandles

SYNOPSIS

```
use FileHandle;

my $fh = FileHandle->new;
if ($fh->open("< file")) {
    print <$fh>;
    $fh->close;
}

my $fh = FileHandle->new("> FOO");
if (defined $fh) {
    print $fh "bar\n";
    $fh->close;
}

my $fh = FileHandle->new("file", "r");
if (defined $fh) {
    print <$fh>;
    undef $fh;       # automatically closes the file
}

my $fh = FileHandle->new("file", O_WRONLY|O_APPEND);
if (defined $fh) {
    print $fh "corge\n";
    undef $fh;       # automatically closes the file
}

my $pos = $fh->getpos;
$fh->setpos($pos);

$fh->setvbuf(my $buffer_var, _IOLBF, 1024);

my ($readfh, $writefh) = FileHandle::pipe;

autoflush STDOUT 1;
```

DESCRIPTION

NOTE: This class is now a front-end to the IO::* classes.

`FileHandle::new` creates a `FileHandle`, which is a reference to a newly created symbol (see the *Symbol* package). If it receives any parameters, they are passed to `FileHandle::open`; if the open fails, the `FileHandle` object is destroyed. Otherwise, it is returned to the caller.

`FileHandle::new_from_fd` creates a `FileHandle` like `new` does. It requires two parameters, which are passed to `FileHandle::fdopen`; if the fdopen fails, the `FileHandle` object is destroyed. Otherwise, it is returned to the caller.

`FileHandle::open` accepts one parameter or two. With one parameter, it is just a front end for the built-in open function. With two parameters, the first parameter is a file-

name that may include whitespace or other special characters, and the second parameter is the open mode, optionally followed by a file permission value.

If `FileHandle::open` receives a Perl mode string (">", "+<", etc.) or a POSIX fopen() mode string ("w", "r+", etc.), it uses the basic Perl open operator.

If `FileHandle::open` is given a numeric mode, it passes that mode and the optional permissions value to the Perl `sysopen` operator. For convenience, `FileHandle::import` tries to import the O_XXX constants from the Fcntl module. If dynamic loading is not available, this may fail, but the rest of FileHandle will still work.

`FileHandle::fdopen` is like open except that its first parameter is not a filename but rather a file handle name, a FileHandle object, or a file descriptor number.

If the C functions fgetpos() and fsetpos() are available, then `FileHandle::getpos` returns an opaque value that represents the current position of the FileHandle, and `FileHandle::setpos` uses that value to return to a previously visited position.

If the C function setvbuf() is available, then `FileHandle::setvbuf` sets the buffering policy for the FileHandle. The calling sequence for the Perl function is the same as its C counterpart, including the macros `_IOFBF`, `_IOLBF`, and `_IONBF`, except that the buffer parameter specifies a scalar variable to use as a buffer. WARNING: A variable used as a buffer by `FileHandle::setvbuf` must not be modified in any way until the FileHandle is closed or until `FileHandle::setvbuf` is called again, or memory corruption may result!

See *perlfunc* for complete descriptions of each of the following supported `FileHandle` methods, which are just front ends for the corresponding built-in functions:

```
close
fileno
getc
gets
eof
clearerr
seek
tell
```

See *perlvar* for complete descriptions of each of the following supported `FileHandle` methods:

```
autoflush
output_field_separator
output_record_separator
input_record_separator
input_line_number
format_page_number
format_lines_per_page
format_lines_left
format_name
format_top_name
format_line_break_characters
format_formfeed
```

Furthermore, for doing normal I/O you might need these:

- $fh->print

  See *"print" in perlfunc*.

- $fh->printf

  See *"printf" in perlfunc*.

- $fh->getline

  This works like <$fh> described in *"I/O Operators" in perlop* except that it's more readable and can be safely called in a list context but still returns just one line.

- $fh->getlines

  This works like <$fh> when called in a list context to read all the remaining lines in a file, except that it's more readable. It will also croak() if accidentally called in a scalar context.

There are many other functions available since FileHandle is descended from IO::File, IO::Seekable, and IO::Handle. Please see those respective pages for documentation on more functions.

SEE ALSO

The **IO** extension, *perlfunc*, *"I/O Operators" in perlop*.

5.2.47 Filter

Namespace containing the following modules:

- `Filter::Simple` - Simplified source filtering
- `Filter::Util`
- `Filter::cpp` - cpp source filter
- `Filter::decrypt` - A sample decrypting source filter: the source after `use Filter::decrypt` is XOR-ed with "Perl" as the compiler reads it.
- `Filter::exec` - exec source filter
- `Filter::m4` - M4 source filter
- `Filter::sh` - sh source filter
- `Filter::tee` - Copies the source that follows `use Filter::tee` to a file, as the compiler reads it.

Filter::Simple

Simplified source filtering

SYNOPSIS

```
# in MyFilter.pm:

    package MyFilter;
```

(continues on next page)

(continued from previous page)

```
    use Filter::Simple;

    FILTER { ... };

    # or just:
    #
    # use Filter::Simple sub { ... };

# in user's code:

    use MyFilter;

    # this code is filtered

    no MyFilter;

    # this code is not
```

DESCRIPTION

The Problem

Source filtering is an immensely powerful feature of recent versions of Perl. It allows one to extend the language itself (e.g. the Switch module), to simplify the language (e.g. Language::Pythonesque), or to completely recast the language (e.g. Lingua::Romana::Perligata). Effectively, it allows one to use the full power of Perl as its own, recursively applied, macro language.

The excellent Filter::Util::Call module (by Paul Marquess) provides a usable Perl interface to source filtering, but it is often too powerful and not nearly as simple as it could be.

To use the module it is necessary to do the following:

1. Download, build, and install the Filter::Util::Call module. (If you have Perl 5.7.1 or later, this is already done for you.)
2. Set up a module that does a `use Filter::Util::Call`.
3. Within that module, create an `import` subroutine.
4. Within the `import` subroutine do a call to `filter_add`, passing it either a subroutine reference.
5. Within the subroutine reference, call `filter_read` or `filter_read_exact` to "prime" $_ with source code data from the source file that will `use` your module. Check the status value returned to see if any source code was actually read in.
6. Process the contents of $_ to change the source code in the desired manner.
7. Return the status value.
8. If the act of unimporting your module (via a no) should cause source code filtering to cease, create an `unimport` subroutine, and have it call `filter_del`. Make sure

that the call to `filter_read` or `filter_read_exact` in step 5 will not accidentally read past the no. Effectively this limits source code filters to line-by-line operation, unless the `import` subroutine does some fancy pre-pre-parsing of the source code it's filtering.

For example, here is a minimal source code filter in a module named BANG.pm. It simply converts every occurrence of the sequence `BANG\s+BANG` to the sequence `die 'BANG' if $BANG` in any piece of code following a `use BANG;` statement (until the next `no BANG;` statement, if any):

```
package BANG;

use Filter::Util::Call ;

sub import {
    filter_add( sub {
    my $caller = caller;
    my ($status, $no_seen, $data);
    while ($status = filter_read()) {
        if (/^\s*no\s+$caller\s*;\s*?$/) {
            $no_seen=1;
            last;
        }
        $data .= $_;
        $_ = "";
    }
    $_ = $data;
    s/BANG\s+BANG/die 'BANG' if \$BANG/g
        unless $status < 0;
    $_ .= "no $class;\n" if $no_seen;
    return 1;
    })
}

sub unimport {
    filter_del();
}

1 ;
```

This level of sophistication puts filtering out of the reach of many programmers.

A Solution

The Filter::Simple module provides a simplified interface to Filter::Util::Call; one that is sufficient for most common cases.

Instead of the above process, with Filter::Simple the task of setting up a source code filter is reduced to:

1. Download and install the Filter::Simple module. (If you have Perl 5.7.1 or later, this is already done for you.)

2. Set up a module that does a `use Filter::Simple` and then calls `FILTER { ... }`.
3. Within the anonymous subroutine or block that is passed to `FILTER`, process the contents of `$_` to change the source code in the desired manner.

In other words, the previous example, would become:

```
package BANG;
use Filter::Simple;

FILTER {
    s/BANG\s+BANG/die 'BANG' if \$BANG/g;
};

1 ;
```

Note that the source code is passed as a single string, so any regex that uses `^` or `$` to detect line boundaries will need the `/m` flag.

Disabling or changing <no> behaviour

By default, the installed filter only filters up to a line consisting of one of the three standard source "terminators":

```
no ModuleName;  # optional comment
```

or:

```
__END__
```

or:

```
__DATA__
```

but this can be altered by passing a second argument to `use Filter::Simple` or `FILTER` (just remember: there's *no* comma after the initial block when you use `FILTER`).

That second argument may be either a `qr`'d regular expression (which is then used to match the terminator line), or a defined false value (which indicates that no terminator line should be looked for), or a reference to a hash (in which case the terminator is the value associated with the key `'terminator'`.

For example, to cause the previous filter to filter only up to a line of the form:

```
GNAB esu;
```

you would write:

```
package BANG;
use Filter::Simple;

FILTER {
    s/BANG\s+BANG/die 'BANG' if \$BANG/g;
```

(continues on next page)

(continued from previous page)

```
}
qr/^\s*GNAB\s+esu\s*;\s*?$/;
```

or:

```
FILTER {
    s/BANG\s+BANG/die 'BANG' if \$BANG/g;
}
{ terminator => qr/^\s*GNAB\s+esu\s*;\s*?$/ };
```

and to prevent the filter's being turned off in any way:

```
package BANG;
use Filter::Simple;

FILTER {
    s/BANG\s+BANG/die 'BANG' if \$BANG/g;
}
"";    # or: 0
```

or:

```
FILTER {
    s/BANG\s+BANG/die 'BANG' if \$BANG/g;
}
{ terminator => "" };
```

Note that, no matter what you set the terminator pattern to, the actual terminator itself *must* be contained on a single source line.

All-in-one interface

Separating the loading of Filter::Simple:

```
use Filter::Simple;
```

from the setting up of the filtering:

```
FILTER { ... };
```

is useful because it allows other code (typically parser support code or caching variables) to be defined before the filter is invoked. However, there is often no need for such a separation.

In those cases, it is easier to just append the filtering subroutine and any terminator specification directly to the `use` statement that loads Filter::Simple, like so:

```
use Filter::Simple sub {
    s/BANG\s+BANG/die 'BANG' if \$BANG/g;
};
```

This is exactly the same as:

```
use Filter::Simple;
BEGIN {
    Filter::Simple::FILTER {
        s/BANG\s+BANG/die 'BANG' if \$BANG/g;
    };
}
```

except that the FILTER subroutine is not exported by Filter::Simple.

Filtering only specific components of source code

One of the problems with a filter like:

```
use Filter::Simple;

FILTER { s/BANG\s+BANG/die 'BANG' if \$BANG/g };
```

is that it indiscriminately applies the specified transformation to the entire text of your source program. So something like:

```
warn 'BANG BANG, YOU'RE DEAD';
BANG BANG;
```

will become:

```
warn 'die 'BANG' if $BANG, YOU'RE DEAD';
die 'BANG' if $BANG;
```

It is very common when filtering source to only want to apply the filter to the non-character-string parts of the code, or alternatively to *only* the character strings.

Filter::Simple supports this type of filtering by automatically exporting the FILTER_ONLY subroutine.

FILTER_ONLY takes a sequence of specifiers that install separate (and possibly multiple) filters that act on only parts of the source code. For example:

```
use Filter::Simple;

FILTER_ONLY
    code      => sub { s/BANG\s+BANG/die 'BANG' if \$BANG/g },
    quotelike => sub { s/BANG\s+BANG/CHITTY CHITTY/g };
```

The "code" subroutine will only be used to filter parts of the source code that are not quotelikes, POD, or __DATA__. The quotelike subroutine only filters Perl quotelikes (including here documents).

The full list of alternatives is:

- "code"

  Filters only those sections of the source code that are not quotelikes, POD, or __DATA__.

- `"code_no_comments"`

  Filters only those sections of the source code that are not quotelikes, POD, comments, or `__DATA__`.

- `"executable"`

  Filters only those sections of the source code that are not POD or `__DATA__`.

- `"executable_no_comments"`

  Filters only those sections of the source code that are not POD, comments, or `__DATA__`.

- `"quotelike"`

  Filters only Perl quotelikes (as interpreted by `&Text::Balanced::extract_quotelike`).

- `"string"`

  Filters only the string literal parts of a Perl quotelike (i.e. the contents of a string literal, either half of a `tr///`, the second half of an `s///`).

- `"regex"`

  Filters only the pattern literal parts of a Perl quotelike (i.e. the contents of a `qr//` or an `m//`, the first half of an `s///`).

- `"all"`

  Filters everything. Identical in effect to `FILTER`.

Except for `FILTER_ONLY code => sub {...}`, each of the component filters is called repeatedly, once for each component found in the source code.

Note that you can also apply two or more of the same type of filter in a single `FILTER_ONLY`. For example, here's a simple macro-preprocessor that is only applied within regexes, with a final debugging pass that prints the resulting source code:

```
use Regexp::Common;
FILTER_ONLY
    regex => sub { s/!\[/[^/g },
    regex => sub { s/%d/$RE{num}{int}/g },
    regex => sub { s/%f/$RE{num}{real}/g },
    all   => sub { print if $::DEBUG };
```

Filtering only the code parts of source code

Most source code ceases to be grammatically correct when it is broken up into the pieces between string literals and regexes. So the `'code'` and `'code_no_comments'` component filter behave slightly differently from the other partial filters described in the previous section.

Rather than calling the specified processor on each individual piece of code (i.e. on the bits between quotelikes), the `'code...'` partial filters operate on the entire source code, but with the quotelike bits (and, in the case of `'code_no_comments'`, the comments) "blanked out".

That is, a `'code...'` filter *replaces* each quoted string, quotelike, regex, POD, and __DATA__ section with a placeholder. The delimiters of this placeholder are the contents of the `$;` variable at the time the filter is applied (normally `"\034"`). The remaining four bytes are a unique identifier for the component being replaced.

This approach makes it comparatively easy to write code preprocessors without worrying about the form or contents of strings, regexes, etc.

For convenience, during a `'code...'` filtering operation, Filter::Simple provides a package variable (`$Filter::Simple::placeholder`) that contains a pre-compiled regex that matches any placeholder...and captures the identifier within the placeholder. Placeholders can be moved and re-ordered within the source code as needed.

In addition, a second package variable (`@Filter::Simple::components`) contains a list of the various pieces of `$_`, as they were originally split up to allow placeholders to be inserted.

Once the filtering has been applied, the original strings, regexes, POD, etc. are re-inserted into the code, by replacing each placeholder with the corresponding original component (from `@components`). Note that this means that the `@components` variable must be treated with extreme care within the filter. The `@components` array stores the "back- translations" of each placeholder inserted into `$_`, as well as the interstitial source code between placeholders. If the placeholder backtranslations are altered in `@components`, they will be similarly changed when the placeholders are removed from `$_` after the filter is complete.

For example, the following filter detects concatenated pairs of strings/quotelikes and reverses the order in which they are concatenated:

```
package DemoRevCat;
use Filter::Simple;

FILTER_ONLY code => sub {
    my $ph = $Filter::Simple::placeholder;
    s{ ($ph) \s* [.] \s* ($ph) }{ $2.$1 }gx
};
```

Thus, the following code:

```
use DemoRevCat;

my $str = "abc" . q(def);

print "$str\n";
```

would become:

```
my $str = q(def)."abc";

print "$str\n";
```

and hence print:

```
defabc
```

Using Filter::Simple with an explicit `import` subroutine

Filter::Simple generates a special `import` subroutine for your module (see "How it works") which would normally replace any `import` subroutine you might have explicitly declared.

However, Filter::Simple is smart enough to notice your existing `import` and Do The Right Thing with it. That is, if you explicitly define an `import` subroutine in a package that's using Filter::Simple, that `import` subroutine will still be invoked immediately after any filter you install.

The only thing you have to remember is that the `import` subroutine *must* be declared *before* the filter is installed. If you use `FILTER` to install the filter:

```
package Filter::TurnItUpTo11;

use Filter::Simple;

FILTER { s/(\w+)/\U$1/ };
```

that will almost never be a problem, but if you install a filtering subroutine by passing it directly to the `use Filter::Simple` statement:

```
package Filter::TurnItUpTo11;

use Filter::Simple sub{ s/(\w+)/\U$1/ };
```

then you must make sure that your `import` subroutine appears before that `use` statement.

Using Filter::Simple and Exporter together

Likewise, Filter::Simple is also smart enough to Do The Right Thing if you use Exporter:

```
package Switch;
use base Exporter;
use Filter::Simple;

@EXPORT    = qw(switch case);
@EXPORT_OK = qw(given  when);

FILTER { $_ = magic_Perl_filter($_) }
```

Immediately after the filter has been applied to the source, Filter::Simple will pass control to Exporter, so it can do its magic too.

Of course, here too, Filter::Simple has to know you're using Exporter before it applies the filter. That's almost never a problem, but if you're nervous about it, you can guarantee that things will work correctly by ensuring that your `use base Exporter` always precedes your `use Filter::Simple`.

How it works

The Filter::Simple module exports into the package that calls `FILTER` (or `use`s it directly) – such as package "BANG" in the above example – two automagically constructed subroutines – `import` and `unimport` – which take care of all the nasty details.

In addition, the generated `import` subroutine passes its own argument list to the filtering subroutine, so the BANG.pm filter could easily be made parametric:

```
package BANG;

use Filter::Simple;

FILTER {
    my ($die_msg, $var_name) = @_;
    s/BANG\s+BANG/die '$die_msg' if \${$var_name}/g;
};

# and in some user code:

use BANG "BOOM", "BAM";  # "BANG BANG" becomes: die 'BOOM' if $BAM
```

The specified filtering subroutine is called every time a `use BANG` is encountered, and passed all the source code following that call, up to either the next `no BANG;` (or whatever terminator you've set) or the end of the source file, whichever occurs first. By default, any `no BANG;` call must appear by itself on a separate line, or it is ignored.

AUTHOR

Damian Conway

CONTACT

Filter::Simple is now maintained by the Perl5-Porters. Please submit bug via the `perlbug` tool that comes with your perl. For usage instructions, read `perldoc perlbug` or possibly `man perlbug`. For mostly anything else, please contact perl5-porters@perl.org.

Maintainer of the CPAN release is Steffen Mueller smueller@cpan.org. Contact him with technical difficulties with respect to the packaging of the CPAN module.

Praise of the module, flowers, and presents still go to the author, Damian Conway damian@conway.org.

COPYRIGHT AND LICENSE


```
Copyright (c) 2000-2014, Damian Conway. All Rights Reserved.
This module is free software. It may be used, redistributed
and/or modified under the same terms as Perl itself.
```

Filter::Util

Namespace containing the following modules:

- `Filter::Util::Call` - Framework for writing Perl *source filters* - modules that rewrite program text on the fly as the tokenizer reads it.
- `Filter::Util::Exec` - Source filters that run the source through a command: the text after the `use` line goes to the command's standard input, and the compiler reads what the command writes.

Filter::Util::Call

Framework for writing Perl *source filters* - modules that rewrite program text on the fly as the tokenizer reads it.

A source filter is a normal Perl module that registers a callback with the compiler. Each time the tokenizer needs more source, it calls back into the filter, which pulls raw text from the next filter in the chain (or the file), mangles it, and hands the result back. The compiler never sees the original bytes - it sees whatever the filter produces. This is the mechanism behind `Switch`, `Filter::Simple`, `PDL::NiceSlice`, `Smart::Comments`, and similar compile-time rewriters.

The filter lifecycle

A filter goes through three phases. Each phase corresponds to one of the primary functions exported by this module:

- **Install** - the filter's `import` is called when Perl processes `use MyFilter`. `import` calls `filter_add` to register itself with the compiler. From this point on the compiler will route every read through the filter.
- **Read** - each time the compiler wants more source, it invokes the filter's `filter` method (or anonymous sub). The filter calls `filter_read` (line mode) or `filter_read_exact` (block mode) to pull raw text from the chain below, transforms it in `$_`, and returns a status: positive for "more data", `0` for EOF, negative for error.
- **Remove** - when the filter decides it is done (for example on seeing an end-marker in the source), it calls `filter_del`. The compiler then reads directly from whatever filter - or file - sat below it in the chain.

The two filter shapes

A filter is either a **method filter** (a blessed object whose class provides a `filter` method) or a **closure filter** (an anonymous sub passed directly to `filter_add`). The two shapes carry context differently - method filters stash state on the object, closure filters close over lexicals - but the read/return contract is identical in both cases.

Shared conventions

- `$_` is the filter's scratch buffer. It is cleared before the filter's `filter` is called; `filter_read` appends to it; the transformed contents of `$_` on return are what the compiler sees.

- The status code is three-valued: `> 0` success, `= 0` EOF, `< 0` error. Check it after every read.
- Filters only run at compile time. Calling `filter_add` while nothing is being compiled (at run time, outside a `BEGIN` or a string `eval`) installs nothing, because there is no parser to register with.

Skeleton

```
package MyFilter;
use Filter::Util::Call;

sub import {
    filter_add(bless []);         # method filter
}

sub filter {
    my ($self) = @_;
    my $status = filter_read();
    s/old/new/g if $status > 0;   # rewrite $_
    $status;
}
1;
```

filter_add and filter_read_exact

These two are Call.pm's own Perl, unchanged:

- `filter_add($obj)` installs a filter for the calling package. A `CODE` reference is a closure filter. Anything else is a method filter: a plain scalar, or a reference whose `ref` is exactly `ARRAY` or `HASH`, is wrapped as a scalar reference blessed into the caller's package; a blessed reference - an object built on an array or a hash included - is used as it is.
- `filter_read_exact($size)` calls `filter_read` until `$_` holds `$size` bytes or the chain reports EOF or an error. It returns 1 for EOF with data still in `$_`, and croaks `filter_read_exact: size parameter must be > 0` for a size that is not positive.

This is a native implementation of the XS half of Filter::Util::Call 1.65; its `Filter/Util/Call.pm` is used unchanged.

Differences from upstream

One, where perl 5.44 with Call.xs crashes (t/81-xs-native/Filter-Util-Call/150-or-better.t):

- `filter_add` while nothing is being compiled installs nothing and leaves memory alone. Call.xs stores the filter's state in the slots of a handle the tokenizer never made, which are past the end of the string it allocated: it overwrites neighbouring variables, and perl5 can segfault.

Functions

Filter lifecycle

`filter_del`

Remove the calling filter from the active chain. Subsequent reads bypass it and hit the next filter down (or the source file).

`filter_add`

Install a source filter for the current package. Called from the filter module's `import`.

Input handling

`filter_read`

Pull the next line of source - or up to a given number of bytes - from the filter chain into `$_`.

`filter_read_exact`

Pull exactly `$size` bytes from the filter chain into `$_`, looping over short reads until the request is satisfied or the stream ends.

Internal dispatch

`real_import`

Low-level installer that wires a pre-blessed object or code ref into the tokenizer. Called by `filter_add` after it has determined the filter shape; not intended for direct user calls.

filter_read

Pull the next line of source - or up to a given number of bytes - from the filter chain into `$_`.

Synopsis

```
my $status = filter_read();         # line mode
my $status = filter_read($size);    # block mode, up to $size bytes
```

What you get back

An integer status, three-valued:

- `> 0` - more source was appended to `$_`.
- `= 0` - EOF: no more source is available from anything below this filter in the chain.
- `< 0` - a read error occurred.

The data is appended to `$_`, not returned. The caller's filter body then transforms `$_` and hands it back to the compiler by returning the same status.

In block mode, `filter_read($size)` may return fewer than `$size` bytes - short reads are normal. Use `filter_read_exact` when the filter needs a guaranteed block size.

Examples

A minimal filter that rewrites `Joe` to `Jim` line by line:

```
sub filter {
    my $status = filter_read();
    s/Joe/Jim/g if $status > 0;
    $status;
}
```

Block mode, processing a chunk at a time:

```
sub filter {
    my $status = filter_read(4096);
    s/secret/*****/g if $status > 0;
    $status;
}
```

Edge cases

- Calling `filter_read` from outside a filter's body (for example at module top level) has no useful effect - no parser is active to read from.
- On EOF, `$_` may still contain buffered data from an earlier partial read; the caller decides whether to flush it.

See also

- `filter_read_exact` - same, but insists on a full block.
- `filter_add` - installs the filter that `filter_read` reads under.
- `filter_del` - stops the current filter so subsequent reads go straight to the chain below.

real_import

Low-level installer that wires a pre-blessed object or code ref into the tokenizer. Called by `filter_add` after it has determined the filter shape; not intended for direct user calls.

Synopsis

```
Filter::Util::Call::real_import($object_or_coderef, $package, $is_
↪coderef);
```

What you get back

Nothing. `real_import` installs the filter as a side effect: the tokenizer now routes every read through it until `filter_del` is called or compilation of the enclosing file finishes.

The three arguments carry the state that the tokenizer will hand back to every `filter` invocation:

- `$object_or_coderef` - the blessed filter object (method filter) or the anonymous sub (closure filter).
- `$package` - the module name, used in diagnostic messages such as `"Filter::Util::Call - MyFilter::filter returned N values, 1 was expected"`.
- `$is_coderef` - true if the first argument is a code ref (the closure-filter shape), false if it is an object (the method-filter shape).

Edge cases

- If `real_import` runs while nothing is being compiled (for example at run time), it installs nothing: there is no parser to drive the filter.
- The module-name argument is copied into a C-string buffer that lives for the lifetime of the process; filter registrations are intentionally not reclaimed.

See also

- `filter_add` - the high-level entry point that prepares arguments and calls `real_import`.
- `unimport` - the inverse operation, exposed for `no MyFilter` idioms.

filter_del

Remove the calling filter from the active chain. Subsequent reads bypass it and hit the next filter down (or the source file).

Synopsis

```
filter_del();
```

What you get back

Nothing useful - `filter_del` is a statement, not an expression.

It does not stop the current `filter` invocation: the caller finishes its current call and returns a status to the compiler normally. Starting with the *next* read, the compiler skips this filter and reads from whatever sat below it in the chain.

Examples

Disable the filter as soon as a sentinel line is seen, so the compiler processes the rest of the file untouched:

```
sub filter {
    my $status = filter_read();
    if ($status > 0 && /^__END_FILTER__$/) {
        filter_del();
        $_ = '';     # drop the sentinel line itself
    }
    $status;
}
```

Edge cases

- Calling `filter_del` when no filter is active is a no-op.
- `filter_del` marks the filter inactive but does not free its per-filter state; the tokenizer releases that state when it tears down the parser at end of compilation.
- It takes no arguments.

See also

- `filter_add` - installs the filter that `filter_del` later removes.
- `filter_read` - normally called right before `filter_del` in the sentinel branch.

filter_read_exact

Pull exactly `$size` bytes from the filter chain into `$_`, looping over short reads until the request is satisfied or the stream ends.

Synopsis

```
my $status = filter_read_exact($size);
```

What you get back

An integer status using the same convention as `filter_read`:

- `> 0` - a full `$size`-byte block was read into `$_`.
- `= 0` - EOF reached with no pending data, or EOF reached with a partial block (the partial block is still in `$_`, and the status returned to the caller is 1 in that specific case to signal "there is data to process"). See the note below.
- `< 0` - a read error occurred.

`filter_read_exact` keeps calling `filter_read` internally until it has accumulated `$size` bytes or the chain reports EOF / error. It is the right choice for filters that operate on fixed-size frames (for example, a binary decoder).

Examples

Read 16-byte records until end of file:

```
sub filter {
    my $status = filter_read_exact(16);
    decode_record($_) if $status > 0;
    $status;
}
```

Edge cases

- `$size` must be strictly positive. Calling `filter_read_exact(0)` or with a negative argument croaks with "`filter_read_exact: size parameter must be > 0`".
- On EOF with a partially-filled `$_`, the returned status is 1, not `0` - this lets the filter process the tail of the stream before the next call reports true EOF.

Differences from upstream

Fully compatible with upstream `Filter::Util::Call` 1.65.

See also

- `filter_read` - the short-read variant; cheaper when the filter can tolerate partial reads.
- `filter_add` - every filter that calls `filter_read_exact` must first register itself with `filter_add`.

filter_add

Install a source filter for the current package. Called from the filter module's `import`.

Synopsis

```
filter_add($object);     # method filter: $object is blessed state
filter_add(sub { ... }); # closure filter: anonymous sub
```

What you get back

Nothing. `filter_add` registers the filter as a side effect; the compiler now routes every subsequent read through it until the filter calls `filter_del` or compilation of the enclosing file finishes.

The shape of the argument picks the filter style:

- A `CODE` reference - closure filter. The sub is called for every compiler read; state lives in lexicals captured by the closure.

- Anything else - method filter. If the argument is a plain scalar or an unblessed ARRAY / HASH reference, `filter_add` blesses a reference to it into the caller's package, so the caller's `filter` method receives it as `$self`. An already-blessed reference is used as-is.

Examples

A closure filter installed from `import`:

```
sub import {
    filter_add(sub {
        my $status = filter_read();
        tr/a-z/A-Z/ if $status > 0;
        $status;
    });
}
```

A method filter that carries a substitution count as its state:

```
sub import {
    my $count = 0;
    filter_add(bless \$count); # becomes $self in filter()
}
sub filter {
    my ($self) = @_;
    my $status = filter_read();
    $$self++ if s/Joe/Jim/g && $status > 0;
    $status;
}
```

Edge cases

- `filter_add` looks up the caller's package through `caller`, so it must be called from code in the filter module's package, normally its `import`. Invoking it from a helper in another package will bless the state object into that package.
- Source filters only act during compilation. Calling `filter_add` at runtime (outside any `use`/`BEGIN` chain) has no effect.
- Passing a plain scalar or an unblessed ARRAY / HASH reference is legal: `filter_add` wraps and blesses the value for you. Any other unblessed reference, such as a SCALAR reference, is installed unblessed, and the first read dies with `Can't call method "filter" on unblessed reference`.

Differences from upstream

Fully compatible with upstream `Filter::Util::Call` 1.65.

See also

- `filter_read` - the read primitive every filter body calls.
- `filter_del` - stop the filter from inside its own body.
- `Filter::Simple` - a higher-level wrapper around this module, preferred for new filters that do not need the fine-grained read control.

Filter::Util::Exec

Source filters that run the source through a command: the text after the `use` line goes to the command's standard input, and the compiler reads what the command writes.

Synopsis

```
use Filter::exec qw(tr a-z A-Z);     # the command and its arguments
use Filter::sh 'tr a-z A-Z';         # a command line for sh -c
use Filter::cpp;                     # the C preprocessor
use Filter::m4;                      # m4
```

How it works

`Filter::Util::Exec::filter_add($module, $command, @args)`, which those four modules call from their `import`, forks and runs the command with a pipe on each side and installs a source filter. Each time the compiler wants more source, the filter writes the next lines of the file into the command's input and reads what it has written back; when the file ends it closes the command's input, and when the command's output ends the filter removes itself. A command that cannot be run dies "execvp failed for command 'NAME': REASON".

This is a native implementation of `Exec.xs` of Filter 1.65 (its Unix branch); its `Filter/Util/Exec.pm` is used unchanged, and so are the `Filter::exec`, `Filter::sh`, `Filter::cpp` and `Filter::m4` built on it.

Differences from upstream

Two, each where perl 5.44 with Exec.xs crashes or loses the program (t/81-xs-native/Filter/150-or-better.t):

- `filter_add` while nothing is being compiled installs nothing and runs no command; perl5 segfaults.
- A command that stops reading before the end of the source (`head -n 1` on a file longer than the pipe holds) is read to the end of its output. perl5 dies of SIGPIPE without a word; with SIGPIPE ignored, it mostly drops what the command wrote and compiles an empty program.

Functions

Source filters

`filter_add`

Run a command as a coprocess and filter the rest of the source through it.

filter_add

Run a command as a coprocess and filter the rest of the source through it.

Synopsis

```
sub import {
    my ($self, @args) = @_;
    Filter::Util::Exec::filter_add($self, 'sh', '-c', "@args");
}
```

What you get back

Nothing. The first argument is the calling module (unused); the command is looked up in PATH and gets the rest as its arguments. The filter reads the source from the line after the `use` that called the `import`.

Filter::cpp

cpp source filter

SYNOPSIS

```
use Filter::cpp ;
```

DESCRIPTION

This source filter pipes the current source file through the C pre-processor (cpp) if it is available.

As with all source filters its scope is limited to the current source file only. Every file you want to be processed by the filter must have a

```
use Filter::cpp ;
```

near the top.

Here is an example script which uses the filter:

```
use Filter::cpp ;

#define FRED 1
```

(continues on next page)

(continued from previous page)

```
$a = 2 + FRED ;
print "a = $a\n" ;
#ifdef FRED
print "Hello FRED\n" ;
#else
print "Where is FRED\n" ;
#endif
```

And here is what it will output:

```
a = 3
Hello FRED
```

This example below, provided by Michael G Schwern, shows a clever way to get Perl to use a C pre-processor macro when the Filter::cpp module is available, or to use a Perl sub when it is not.

```
# use Filter::cpp if we can.
BEGIN { eval 'use Filter::cpp' }

sub PRINT {
    my($string) = shift;

#define PRINT($string) \
    (print $string."\n")
}

PRINT("Mu");
```

Look at Michael's Tie::VecArray module for a practical use.

AUTHOR

Paul Marquess

DATE

11th December 1995.

Filter::decrypt

A sample decrypting source filter: the source after `use Filter::decrypt` is XOR-ed with "Perl" as the compiler reads it.

Synopsis

```
use Filter::decrypt;
```

The rest of the file is the output of the dist's `decrypt/encrypt` script: the two bytes 0xff 0x00, then the source, each block of four bytes XOR-ed with "Perl".

How it works

`import` installs a source filter. On its first read it checks the two-byte header ("truncated file", "bad encryption format") and that no filter was added on the `use` line after it ("too many filters"); then it reads four bytes at a time and serves the decrypted text by the line or the block, as the compiler asks. `import` dies "debugger disabled" under `perl -d`, and loading the module dies "Aborting, Compiler detected" once the compiler backend (`B`) is loaded.

The module is a template for a real decryption filter, not a protection: the decrypted source passes through memory, and `B::Deparse` can print what it compiled to.

This is a native implementation of `decrypt.xs` of Filter 1.65 as the dist builds it (`-DBYPASS`: no refusal of a dynamically linked module or of a perl built with -DDEBUGGING); its `Filter/decrypt.pm` is used unchanged.

Differences from upstream

One, where perl 5.44 with decrypt.xs crashes (t/81-xs-native/Filter/150-or-better.t):

- `Filter::decrypt->import` while nothing is being compiled installs nothing and leaves memory alone. decrypt.xs stores the filter's state in the slots of a handle the tokenizer never made, past the end of the string it allocated: it overwrites neighbouring variables, and perl5 can segfault.

Filter::exec

exec source filter

SYNOPSIS

```
use Filter::exec qw(command parameters) ;
```

DESCRIPTION

This filter pipes the current source file through the program which corresponds to the `command` parameter.

As with all source filters its scope is limited to the current source file only. Every file you want to be processed by the filter must have a

```
use Filter::exec qw(command ) ;
```

near the top.

Here is an example script which uses the filter:

```
use Filter::exec qw(tr XYZ PQR) ;
$a = 1 ;
print "XYZ a = $a\n" ;
```

And here is what it will output:

```
PQR = 1
```

WARNING

You should be *very* careful when using this filter. Because of the way the filter is implemented it is possible to end up with deadlock.

Be especially careful when stacking multiple instances of the filter in a single source file.

AUTHOR

Paul Marquess

DATE

11th December 1995.

Filter::m4

M4 source filter

SYNOPSIS

```
use Filter::m4;
use Filter::m4 'prefix';
```

DESCRIPTION

This source filter pipes the current source file through the M4 macro processor (`m4`) if it is available.

As with all source filters its scope is limited to the current source file only. Every file you want to be processed by the filter must have the following line near the top.

```
use Filter::m4;
```

EXAMPLE

Here is a small example that shows how to define and use an M4 macro:

```
use Filter::m4;

define(`foo', `$1 =~ s/bar/baz/r')

$a = "foobar";
print "a = " . foo(`$a') . "\n";
```

The output of the above example:

```
a = foobaz
```

NOTES

By default, M4 uses ` and ‘ as quotes; however, this is configurable using M4's `changequote` builtin.

M4 uses `$1`, `$2`, etc., to indicate arguments in macros. To avoid clashes with Perl regex syntax it is recommended to use Perl's alternative forms `${1}`, `${1}`, etc.

The following keywords in M4 and Perl are identical:

```
eval
format
index
mkstemp
shift
substr
```

If you need such keywords in your Perl code you have to use one of the following three solutions.

- Protect the keyword with M4 quotes, for example `` `shift' ``.
- Redefine the problematic M4 builtin using `defn`, as outlined in section *Renaming macros* of the M4 info manual.
- Use the `prefix` option. This adds the prefix `m4_` to all M4 builtins (but not to user-defined macros). For example, you will have to use `m4_shift` instead of `shift`.

AUTHOR

Werner Lemberg

DATE

17th March 2018.

Filter::sh

sh source filter

SYNOPSIS

```
use Filter::sh 'command' ;
```

DESCRIPTION

This filter pipes the current source file through the program which corresponds to the `command` parameter using the Bourne shell.

As with all source filters its scope is limited to the current source file only. Every file you want to be processed by the filter must have a

```
use Filter::sh 'command' ;
```

near the top.

Here is an example script which uses the filter:

```
use Filter::sh 'tr XYZ PQR' ;
$a = 1 ;
print "XYZ a = $a\n" ;
```

And here is what it will output:

```
PQR = 1
```

WARNING

You should be *very* careful when using this filter. Because of the way the filter is implemented it is possible to end up with deadlock.

Be especially careful when stacking multiple instances of the filter in a single source file.

AUTHOR

Paul Marquess

DATE

11th December 1995.

Filter::tee

Copies the source that follows `use Filter::tee` to a file, as the compiler reads it.

Synopsis

```
use Filter::tee 'filename';     # empty the file first, then write
use Filter::tee '>filename';    # the same
use Filter::tee '>>filename';   # append
```

How it works

`import` opens the file and installs a source filter. Each time the compiler reads more of the file being compiled, the filter passes the text on unchanged and writes a copy of it to the file; at the end of the source it closes the file and removes itself. Used after another filter (`use Filter::decrypt; use Filter::tee ...`), it copies what that filter produced, which is what makes it a debugging aid for filters.

This is a native implementation of `tee.xs` of Filter 1.65; its `Filter/tee.pm` is used unchanged.

Differences from upstream

None.

5.2.48 FindBin

Locate directory of original Perl script

SYNOPSIS

```
use FindBin;
use lib "$FindBin::Bin/../lib";

use FindBin qw($Bin);
use lib "$Bin/../lib";
```

DESCRIPTION

Locates the full path to the script bin directory to allow the use of paths relative to the bin directory.

This allows a user to setup a directory tree for some software with directories `<root>/bin` and `<root>/lib`, and then the above example will allow the use of modules in the lib directory without knowing where the software tree is installed.

If `perl` is invoked using the `-e` option or the Perl script is read from `STDIN`, then `FindBin` sets both `$Bin` and `$RealBin` to the current directory.

EXPORTABLE VARIABLES

- `$Bin` or `$Dir`

  Path to the bin **directory** from where script was invoked

- `$Script`

  Basename of the script from which `perl` was invoked

- `$RealBin` or `$RealDir`

  `$Bin` with all links resolved

- `$RealScript`

  `$Script` with all links resolved

You can also use the ALL tag to export all of the above variables together:

```
use FindBin ':ALL';
```

KNOWN ISSUES

If there are two modules using FindBin from different directories under the same interpreter, this won't work. Since FindBin uses a BEGIN block, it'll be executed only once, and only the first caller will get it right. This is a problem under mod_perl and other persistent Perl environments, where you shouldn't use this module. Which also means that you should avoid using FindBin in modules that you plan to put on CPAN. Call the again function to make sure that FindBin will work:

```
use FindBin;
FindBin::again(); # or FindBin->again;
```

In former versions of FindBin there was no again function. The workaround was to force the BEGIN block to be executed again:

```
delete $INC{'FindBin.pm'};
require FindBin;
```

AUTHORS

FindBin is supported as part of the core perl distribution. Please submit bug reports at https://github.com/Perl/perl5/issues.

Graham Barr <gbarr@pobox.com> Nick Ing-Simmons <nik@tiuk.ti.com>

COPYRIGHT

Copyright (c) 1995 Graham Barr & Nick Ing-Simmons. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

5.2.49 GD

Perl bindings for libgd, the GD graphics library.

This is the GD module, version 2.91 of the CPAN API: the same packages, methods, arguments and return values that Perl code written against GD already expects. Drawing goes to the libgd.so.3 installed on the machine, which is located when the module is loaded; if that library is absent, use GD dies with a message naming the distribution package to install, as it does under a perl whose GD was never built.

```
use GD;

my $im    = GD::Image->new(200, 200, 1);   # 1 = truecolor
my $white = $im->colorAllocate(255, 255, 255);
my $blue  = $im->colorAllocate(0, 0, 255);
```

(continues on next page)

(continued from previous page)

```
$im->filledRectangle(0, 0, 199, 199, $white);
$im->line(0, 0, 199, 199, $blue);

open my $fh, '>', 'out.png' or die $!;
binmode $fh;
print $fh $im->png;
```

The colour index

GD's defining quirk is that you do not draw with a colour, you draw with the integer `colorAllocate` gives back. On a palette image that integer really is an index into a 256-entry table, and running out of entries makes `colorAllocate` return -1. On a true-color image - what you get by passing a true third argument to `new` - the integer is the packed colour itself and there is no limit, but the API keeps the same shape.

GD::Image

An image is either **palette** or **truecolor**, chosen when you make it, and the difference runs through the whole API:

```
my $palette   = GD::Image->new(200, 200);      # 256 colours, indexed
my $truecolor = GD::Image->new(200, 200, 1);   # 24-bit + alpha
```

Drawing

Note the shapes of the argument lists, which are not uniform and are the usual source of surprise. Rectangles take two corners; arcs and ellipses take a centre plus **width and height** (not a radius); `filledArc` adds start and end angles in degrees.

```
$im->line(0, 0, 199, 199, $blue);
$im->rectangle(10, 10, 60, 60, $blue);          # outline
$im->filledRectangle(70, 10, 120, 60, $red);    # solid
$im->arc(100, 100, 80, 80, 0, 360, $blue);      # centre, w, h, start, end
$im->filledArc(100, 100, 120, 60, 0, 180, $red);
$im->setPixel(5, 5, $blue);
$im->fill(30, 30, $red);                        # flood fill
```

Polygons are drawn from a `GD::Polygon`, and the special colour values `gdBrushed`, `gdStyled`, `gdTiled` and `gdAntiAliased` select the brush, style or tile you set beforehand instead of a plain colour.

Reading and writing

Output methods return the encoded bytes rather than writing a file, so you print them yourself:

```
open my $fh, '>', 'out.png' or die $!;
binmode $fh;
print $fh $im->png;             # or ->jpeg($quality), ->gif, ->webp
```

Reading goes through `newFrom<Format>` for a filehandle or `newFrom<Format>Data` for bytes you already have. Which formats work depends on how the installed libgd was built. `GD::Image->supported` lists them, `GD::supportsFileType('photo.webp')` answers for one file name, and the methods of a format this libgd lacks do not exist, so `GD::Image->can('newFromTiff')` is a test too. A GD built from source settles that when it is compiled; here it is settled when the module is loaded, by the libgd found then.

A reader dies when libgd cannot decode what it was given. The PNG, JPEG and TIFF readers, and `newFromBmpData` and `newFromWBMPData`, return palette images unless you pass a true final argument - `GD::Image->newFromPngData($bytes, 1)` - or have made truecolor the default with `GD::Image->trueColor(1)`.

GD::Font - the built-in bitmap fonts

libgd ships five fixed-size bitmap fonts, reached as class methods:

```
my $font = GD::Font->Small;
$image->string($font, 10, 10, 'hello', $black);
```

The five are `Tiny`, `Small`, `MediumBold`, `Large` and `Giant`, and `GD` also exports them as the functions `gdTinyFont`, `gdSmallFont`, `gdMediumBoldFont`, `gdLargeFont` and `gdGiantFont`, which is the spelling most existing code uses.

A font knows its own metrics: `$font->width` and `$font->height` are the size of one character cell, which is what you need to work out where a string will land, and `$font->nchars` and `$font->offset` describe which characters it covers.

These fonts are fixed-width bitmaps and cannot be scaled or rotated. For scalable text, use `$image->stringFT`, which draws with FreeType from a TrueType file at any size and angle.

What is native and what is Perl

GD is a large module whose pure-Perl half does real work: `GD::Polygon`, `GD::Polyline` and `GD::Simple` are Perl, and so is the filehandle plumbing behind `newFromPng` and friends. That half runs here exactly as it does on perl5, from the module's own source; only the parts that were C are native. So `GD::Simple->new(...)->moveTo(...)` behaves as its own documentation says, because it IS its own code.

Differences from upstream

Each where XS GD crashes, reads memory it does not own, or refuses what libgd can do (t/64-auto-ffi/332-gd-or-better.t):

- `cloneImage` after `filledPolygon` copies the image, where libgd's `gdImageClone` crashes on such an image.
- A `newFrom*` reader or an `affine*` constructor called with an argument short (through &, past the prototype) croaks its usage; GD.xs reads the missing argument from beyond the stack.

- `scatterColor` treats a colour that is not an integer as 0, and an empty colour list as no colours; GD.xs passes uninitialised memory for the one and a negative count for the other.
- The formats are the ones libgd reads, not the ones the module's build guessed from pkg-config: an XS GD may lack `newFromXpm` on a libgd that reads XPM.
- A method on an image whose `DESTROY` has run, or on the empty copy `Storable::dclone` makes of an image, croaks "not a live GD::Image"; GD.xs passes the NULL pointer to libgd.
- `GD::Font->load` refuses a font file whose header gives no characters or a negative size, as it refuses a file it cannot read; GD.xs trusts the header.

Modules

- `GD::Group` - Simple object for recursive grouping
- `GD::Image` - Image class for the GD image library
- `GD::Polygon` - Polygon class for the GD image library
- `GD::Polyline` - Polyline object and Polygon utilities (including splines) for use with GD
- `GD::Simple` - Simplified interface to GD library

Functions

Library information

`VERSION_STRING`

The version of the libgd actually loaded, as a string such as `"2.3.3"` (gd.h's `GD_VERSION_STRING`). This is a run-time answer: it describes the library on this machine, not the one anything was built against. The version of the Perl module is `$GD::VERSION`.

`LIBGD_VERSION`

The version of the libgd actually loaded, as one number: major, then minor and release as two decimal places each, so that versions compare numerically. libgd 2.3.3 is `2.0303`.

`supportsFileType`

True when the loaded libgd can read, or with a true second argument write, the image file named by its first argument.

GD::Group

Simple object for recursive grouping

DESCRIPTION

Does absolutely nothing with GD, but works nicely with GD::SVG.

GD::Image

Image class for the GD image library

SYNOPSIS

See *GD*

DESCRIPTION

Supported Image formats, also returned by `supported()`, as lowercase strings.

- Png
- Gif
- Gd
- Gd2
- Jpeg
- Tiff
- Xpm
- Xbm
- WBMP
- BMP
- GifAnim
- Webp
- Heif
- Avif
- Jxl

Unsupported Image formats:

See *GD*

AUTHOR

The GD.pm interface is copyright 1995-2005, Lincoln D. Stein. It is distributed under the same terms as Perl itself. See the “Artistic License” in the Perl source code distribution for licensing terms.

The latest versions of GD.pm are available on CPAN:

```
http://www.cpan.org
```

SEE ALSO

GD GD::Polyline, GD::SVG, *GD::Simple*, Image::Magick

GD::Polygon

Polygon class for the GD image library

SYNOPSIS

See *GD*

DESCRIPTION

See *GD*

AUTHOR

The GD.pm interface is copyright 1995-2005, Lincoln D. Stein. It is distributed under the same terms as Perl itself. See the "Artistic License" in the Perl source code distribution for licensing terms.

The latest versions of GD.pm are available on CPAN:

```
http://www.cpan.org
```

SEE ALSO

GD GD::Polyline, GD::SVG, *GD::Simple*, Image::Magick

GD::Polyline

Polyline object and Polygon utilities (including splines) for use with GD

SYNOPSIS

```
use GD;
use GD::Polyline;

# create an image
$image = GD::Image->new (500,300);
$white  = $image->colorAllocate(255,255,255);
$black  = $image->colorAllocate(  0,  0,  0);
$red    = $image->colorAllocate(255,  0,  0);

# create a new polyline
$polyline = GD::Polyline->new;

# add some points
```

(continues on next page)

(continued from previous page)

```
$polyline->addPt(  0,  0);
$polyline->addPt(  0,100);
$polyline->addPt( 50,125);
$polyline->addPt(100,  0);

# polylines can use polygon methods (and vice versa)
$polyline->offset(200,100);

# rotate 60 degrees, about the centroid
$polyline->rotate(3.14159/3, $polyline->centroid());

# scale about the centroid
$polyline->scale(1.5, 2, $polyline->centroid());

# draw the polyline
$image->polydraw($polyline,$black);

# create a spline, which is also a polyine
$spline = $polyline->addControlPoints->toSpline;
$image->polydraw($spline,$red);

# output the png
binmode STDOUT;
print $image->png;
```

DESCRIPTION

Polyline.pm extends the GD module by allowing you to create polylines. Think of a polyline as "an open polygon", that is, the last vertex is not connected to the first vertex (unless you expressly add the same value as both points).

For the remainder of this doc, "polyline" will refer to a GD::Polyline, "polygon" will refer to a GD::Polygon that is not a polyline, and "polything" and "$poly" may be either.

The big feature added to GD by this module is the means to create splines, which are approximations to curves.

The Polyline Object

GD::Polyline defines the following class:

- `GD::Polyline`

  A polyline object, used for storing lists of vertices prior to rendering a polyline into an image.

- `new`

  `GD::Polyline->new` *class method*

  Create an empty polyline with no vertices.

```
$polyline = GD::Polyline->new;

$polyline->addPt(  0,  0);
$polyline->addPt(  0,100);
$polyline->addPt( 50,100);
$polyline->addPt(100,  0);

$image->polydraw($polyline,$black);
```

In fact GD::Polyline is a subclass of GD::Polygon, so all polygon methods (such as **offset** and **transform**) may be used on polylines. Some new methods have thus been added to GD::Polygon (such as **rotate**) and a few updated/modified/enhanced (such as **scale**) *in this module*. See section "New or Updated GD::Polygon Methods" for more info.

Note that this module is very "young" and should be considered subject to change in future releases, and/or possibly folded in to the existing polygon object and/or GD module.

Updated Polygon Methods

The following methods (defined in GD.pm) are OVERRIDDEN if you use this module.

All effort has been made to provide 100% backward compatibility, but if you can confirm that has not been achieved, please consider that a bug and let the the author of Polyline.pm know.

- `scale`

  `$poly->scale($sx, $sy, $cx, $cy)` *object method – UPDATE to GD::Polygon::scale*

  Scale a polything in along x-axis by $sx and along the y-axis by $sy, about centery point ($cx, $cy).

  Center point ($cx, $cy) is optional – if these are omitted, the function will scale about the origin.

  To flip a polything, use a scale factor of -1. For example, to flip the polything top to bottom about line y = 100, use:

```
$poly->scale(1, -1, 0, 100);
```

New Polygon Methods

The following methods are added to GD::Polygon, and thus can be used by polygons and polylines.

Don't forget: a polyline is a GD::Polygon, so GD::Polygon methods like offset() can be used, and they can be used in GD::Image methods like filledPolygon().

- `rotate`

  `$poly->rotate($angle, $cx, $cy)` *object method*

  Rotate a polything through $angle (clockwise, in radians) about center point ($cx, $cy).

Center point ($cx, $cy) is optional – if these are omitted, the function will rotate about the origin

In this function and other angle-oriented functions in GD::Polyline, positive $angle corresponds to clockwise rotation. This is opposite of the usual Cartesian sense, but that is because the raster is opposite of the usual Cartesian sense in that the y-axis goes "down".

- `centroid`

  `($cx, $cy) = $poly->centroid($scale)` *object method*

  Calculate and return ($cx, $cy), the centroid of the vertices of the polything. For example, to rotate something 180 degrees about it's centroid:

```
$poly->rotate(3.14159, $poly->centroid());
```

  $scale is optional; if supplied, $cx and $cy are multiplied by $scale before returning. The main use of this is to shift an polything to the origin like this:

```
$poly->offset($poly->centroid(-1));
```

- `segLength`

  `@segLengths = $poly->segLength()` *object method*

  In array context, returns an array the lengths of the segments in the polything. Segment n is the segment from vertex n to vertex n+1. Polygons have as many segments as vertices; polylines have one fewer.

  In a scalar context, returns the sum of the array that would have been returned in the array context.

- `segAngle`

  `@segAngles = $poly->segAngle()` *object method*

  Returns an array the angles of each segment from the x-axis. Segment n is the segment from vertex n to vertex n+1. Polygons have as many segments as vertices; polylines have one fewer.

  Returned angles will be on the interval 0 <= $angle < 2 * pi and angles increase in a clockwise direction.

- `vertexAngle`

  `@vertexAngles = $poly->vertexAngle()` *object method*

  Returns an array of the angles between the segment into and out of each vertex. For polylines, the vertex angle at vertex 0 and the last vertex are not defined; however $vertexAngle[0] will be undef so that $vertexAngle[1] will correspond to vertex 1.

  Returned angles will be on the interval 0 <= $angle < 2 * pi and angles increase in a clockwise direction.

  Note that this calculation does not attempt to figure out the "interior" angle with respect to "inside" or "outside" the polygon, but rather, just the angle between the adjacent segments in a clockwise sense. Thus a polygon with all right angles will have vertex angles of either pi/2 or 3*pi/2, depending on the way the polygon was "wound".

- `toSpline`

  `$poly->toSpline()` *object method & factory method*

  Create a new polything which is a reasonably smooth curve using cubic spline algorithms, often referred to as Bezier curves. The "source" polything is called the "control polything". If it is a polyline, the control polyline must have 4, 7, 10, or some number of vertices of equal to 3n+1. If it is a polygon, the control polygon must have 3, 6, 9, or some number of vertices of equal to 3n.

```
$spline = $poly->toSpline();
$image->polydraw($spline,$red);
```

  In brief, groups of four points from the control polyline are considered "control points" for a given portion of the spline: the first and fourth are "anchor points", and the spline passes through them; the second and third are "director points". The spline does not pass through director points, however the spline is tangent to the line segment from anchor point to adjacent director point.

  The next portion of the spline reuses the previous portion's last anchor point. The spline will have a cusp (non-continuous slope) at an anchor point, unless the anchor points and its adjacent director point are colinear.

  In the current implementation, toSpline() return a fixed number of segments in the returned polyline per set-of-four control points. In the future, this and other parameters of the algorithm may be configurable.

- `addControlPoints`

  `$polyline->addControlPoints()` *object method & factory method*

  So you say: "OK. Splines sound cool. But how can I get my anchor points and its adjacent director point to be colinear so that I have a nice smooth curves from my polyline?" Relax! For The Lazy: addControlPoints() to the rescue.

  addControlPoints() returns a polyline that can serve as the control polyline for toSpline(), which returns another polyline which is the spline. Is your head spinning yet? Think of it this way:

  - *

    If you have a polyline, and you have already put your control points where you want them, call toSpline() directly. Remember, only every third vertex will be "on" the spline.

    You get something that looks like the spline "inscribed" inside the control polyline.

  - *

    If you have a polyline, and you want all of its vertices on the resulting spline, call addControlPoints() and then toSpline():

```
$control = $polyline->addControlPoints();
$spline  = $control->toSpline();
$image->polyline($spline,$red);
```

You get something that looks like the control polyline "inscribed" inside the spline.

Adding "good" control points is subjective; this particular algorithm reveals its author's tastes. In the future, you may be able to alter the taste slightly via parameters to the algorithm. For The Hubristic: please build a better one!

And for The Impatient: note that addControlPoints() returns a polyline, so you can pile up the call like this, if you'd like:

```
$image->polyline($polyline->addControlPoints()->toSpline(),
 ↪$mauve);
```

New GD::Image Methods

- `polyline`

  `$image->polyline(polyline,color)` *object method*

  ```
  $image->polyline($polyline,$black)
  ```

  This draws a polyline with the specified color. Both real color indexes and the special colors gdBrushed, gdStyled and gdStyledBrushed can be specified.

  Neither the polyline() method or the polygon() method are very picky: you can call either method with either a GD::Polygon or a GD::Polyline. The *method* determines if the shape is "closed" or "open" as drawn, *not* the object type.

- `polydraw`

  `$image->polydraw(polything,color)` *object method*

  ```
  $image->polydraw($poly,$black)
  ```

  This method draws the polything as expected (polygons are closed, polylines are open) by simply checking the object type and calling either $image->polygon() or $image->polyline().

Examples

Please see file "polyline-examples.pl" that is included with the distribution.

See Also

For more info on Bezier splines, see http://www.webreference.com/dlab/9902/bezier.html.

Future Features

On the drawing board are additional features such as:

```
- polygon winding algorithms (to determine if a point is "inside" or␣
↪"outside" the polygon)

- new polygon from bounding box

- find bounding polygon (tightest fitting simple convex polygon for␣
↪a given set of vertices)

- addPts() method to add many points at once

- clone() method for polygon

- functions to interwork GD with SVG
```

Please provide input on other possible features you'd like to see.

Author

This module has been written by Daniel J. Harasty. Please send questions, comments, complaints, and kudos to him at harasty@cpan.org.

Thanks to Lincoln Stein for input and patience with me and this, my first CPAN contribution.

Copyright Information

The Polyline.pm module is copyright 2002, Daniel J. Harasty. It is distributed under the same terms as Perl itself. See the "Artistic License" in the Perl source code distribution for licensing terms.

The latest version of Polyline.pm is available at your favorite CPAN repository and/or along with GD.pm by Lincoln D. Stein at http://stein.cshl.org/WWW/software/GD.

GD::Simple

Simplified interface to GD library

SYNOPSIS

For a nice tutorial on using this module, see Gabor Szabo's article at http://perl-maven.com/drawing-images-using-gd-simple.

```
use GD::Simple;

# create a new image
$img = GD::Simple->new(400,250);

# draw a red rectangle with blue borders
$img->bgcolor('red');
$img->fgcolor('blue');
```

(continues on next page)

(continued from previous page)

```
$img->rectangle(10,10,50,50);

# draw an empty rectangle with green borders
$img->bgcolor(undef);
$img->fgcolor('green');
$img->rectangle(30,30,100,100);

# move to (80,80) and draw a green line to (100,190)
$img->moveTo(80,80);
$img->lineTo(100,190);

# draw a solid orange ellipse
$img->moveTo(110,100);
$img->bgcolor('orange');
$img->fgcolor('orange');
$img->ellipse(40,40);

# draw a black filled arc
$img->moveTo(150,150);
$img->fgcolor('black');
$img->arc(50,50,0,100,gdNoFill|gdEdged);

# draw a string at (10,180) using the default
# built-in font
$img->moveTo(10,180);
$img->string('This is very simple');

# draw a string at (280,210) using 20 point
# times italic, angled upward 90 degrees
$img->moveTo(280,210);
$img->font('Times:italic');
$img->fontsize(20);
$img->angle(-90);
$img->string('This is very fancy');

# some turtle graphics
$img->moveTo(300,100);
$img->penSize(3,3);
$img->angle(0);
$img->line(20);   # 20 pixels going to the right
$img->turn(30);   # set turning angle to 30 degrees
$img->line(20);   # 20 pixel line
$img->line(20);
$img->line(20);
$img->turn(-90); # set turning angle to -90 degrees
$img->line(50);  # 50 pixel line

# draw a cyan polygon edged in blue
my $poly = GD::Polygon->new;
```

(continues on next page)

(continued from previous page)

```
 $poly->addPt(150,100);
 $poly->addPt(199,199);
 $poly->addPt(100,199);
 $img->bgcolor('cyan');
 $img->fgcolor('blue');
 $img->penSize(1,1);
 $img->polygon($poly);

# convert into png data
print $img->png;
```

DESCRIPTION

GD::Simple is a subclass of the GD library that shortens many of the long GD method calls by storing information about the pen color, size and position in the GD object itself. It also adds a small number of "turtle graphics" style calls for those who prefer to work in polar coordinates. In addition, the library allows you to use symbolic names for colors, such as "chartreuse", and will manage the colors for you.

The Pen

GD::Simple maintains a "pen" whose settings are used for line- and shape-drawing operations. The pen has the following properties:

- fgcolor

  The pen foreground color is the color of lines and the borders of filled and unfilled shapes.

- bgcolor

  The pen background color is the color of the contents of filled shapes.

- pensize

  The pen size is the width of the pen. Larger sizes draw thicker lines.

- position

  The pen position is its current position on the canvas in (X,Y) coordinates.

- angle

  When drawing in turtle mode, the pen angle determines the current direction of lines of relative length.

- turn

  When drawing in turtle mode, the turn determines the clockwise or counterclockwise angle that the pen will turn before drawing the next line.

- font

  The font to use when drawing text. Both built-in bitmapped fonts and TrueType fonts are supported.

- fontsize

  The size of the font to use when drawing with TrueType fonts.

One sets the position and properties of the pen and then draws. As the drawing progresses, the position of the pen is updated.

Methods

GD::Simple introduces a number of new methods, a few of which have the same name as GD::Image methods, and hence change their behavior. In addition to these new methods, GD::Simple objects support all of the GD::Image methods. If you make a method call that isn't directly supported by GD::Simple, it refers the request to the underlying GD::Image object. Hence one can load a JPEG image into GD::Simple and declare it to be TrueColor by using this call, which is effectively inherited from GD::Image:

```
my $img = GD::Simple->newFromJpeg('./myimage.jpg',1);
```

The rest of this section describes GD::Simple-specific methods.

- $img = GD::Simple->new($x,$y [,$truecolor])
- $img = GD::Simple->new($gd)

  Create a new GD::Simple object. There are two forms of new(). In the first form, pass the width and height of the desired canvas, and optionally a boolean flag to request a truecolor image. In the second form, pass a previously-created GD::Image object.

- GD::Simple->class('GD');
- GD::Simple->class('GD::SVG');

  Select whether new() should use GD or GD::SVG internally. Call GD::Simple->class('GD::SVG') before calling new() if you wish to generate SVG images.

  If future GD subclasses are created, this method will subport them.

- $img->moveTo($x,$y)

  This call changes the position of the pen without drawing. It moves the pen to position ($x,$y) on the drawing canvas.

- $img->move($dx,$dy)
- $img->move($dr)

  This call changes the position of the pen without drawing. When called with two arguments it moves the pen $dx pixels to the right and $dy pixels downward. When called with one argument it moves the pen $dr pixels along the vector described by the current pen angle.

- $img->lineTo($x,$y)

  The lineTo() call simultaneously draws and moves the pen. It draws a line from the current pen position to the position defined by ($x,$y) using the current pen size and color. After drawing, the position of the pen is updated to the new position.

- $img->line($x1,$y1,$x2,$y2 [,$color])

- $img->line($dx,$dy)
- $img->line($dr)

  The line() call simultaneously draws and moves the pen. When called with two arguments it draws a line from the current position of the pen to the position $dx pixels to the right and $dy pixels down. When called with one argument, it draws a line $dr pixels long along the angle defined by the current pen angle.

  When called with four or five arguments, line() behaves like GD::Image->line().

- $img->clear

  This method clears the canvas by painting over it with the current background color.

- $img->rectangle($x1,$y1,$x2,$y2)

  This method draws the rectangle defined by corners ($x1,$y1), ($x2,$y2). The rectangle's edges are drawn in the foreground color and its contents are filled with the background color. To draw a solid rectangle set bgcolor equal to fgcolor. To draw an unfilled rectangle (transparent inside), set bgcolor to undef.

- $img->ellipse($width,$height)

  This method draws the ellipse centered at the current location with width $width and height $height. The ellipse's border is drawn in the foreground color and its contents are filled with the background color. To draw a solid ellipse set bgcolor equal to fgcolor. To draw an unfilled ellipse (transparent inside), set bgcolor to undef.

- $img->arc([$cx,$cy,] $width,$height,$start,$end [,$style])

  This method draws filled and unfilled arcs, at the current position, with the current fore- and background colors. See *GD* for a description of the arguments. To draw a solid arc (such as a pie wedge) set bgcolor equal to fgcolor. To draw an unfilled arc, set bgcolor to undef.

- $img->polygon($poly)

  This method draws filled and unfilled polygon using the current settings of fgcolor for the polygon border and bgcolor for the polygon fill color. See *GD* for a description of creating polygons. To draw a solid polygon set bgcolor equal to fgcolor. To draw an unfilled polygon, set bgcolor to undef.

- $img->polyline($poly)

  This method draws polygons without closing the first and last vertices (similar to GD::Image->unclosedPolygon()). It uses the fgcolor to draw the line.

- $img->string($string)

  This method draws the indicated string starting at the current position of the pen. The pen is moved to the end of the drawn string. Depending on the font selected with the font() method, this will use either a bitmapped GD font or a TrueType font. The angle of the pen will be consulted when drawing the text. For TrueType fonts, any angle is accepted. For GD bitmapped fonts, the angle can be either 0 (draw horizontal) or -90 (draw upwards).

For consistency between the TrueType and GD font behavior, the string is always drawn so that the current position of the pen corresponds to the bottom left of the first character of the text. This is different from the GD behavior, in which the first character of bitmapped fonts hangs down from the pen point.

This method returns a polygon indicating the bounding box of the rendered text. If an error occurred (such as invalid font specification) it returns undef and an error message in $@.

- $metrics = $img->fontMetrics
- ($metrics,$width,$height) = GD::Simple->fontMetrics($font,$fontsize,$string)

  This method returns information about the current font, most commonly a TrueType font. It can be invoked as an instance method (on a previously-created GD::Simple object) or as a class method (on the 'GD::Simple' class).

  When called as an instance method, fontMetrics() takes no arguments and returns a single hash reference containing the metrics that describe the currently selected font and size. The hash reference contains the following information:

```
xheight      the base height of the font from the bottom to the
↪top of
             a lowercase 'm'

ascent       the length of the upper stem of the lowercase 'd'

descent      the length of the lower step of the lowercase 'j'

lineheight   the distance from the bottom of the 'j' to the top
↪of
             the 'd'

leading      the distance between two adjacent lines
```

  This description and code was changed with 2.75.

- ($delta_x,$delta_y)= $img->stringBounds($string)

  This method indicates the X and Y offsets (which may be negative) that will occur when the given string is drawn using the current font, fontsize and angle. When the string is drawn horizontally, it gives the width and height of the string's bounding box.

- $delta_x = $img->stringWidth($string)

  This method indicates the width of the string given the current font, fontsize and angle. It is the same as ($img->stringBounds($string))[0]

- ($x,$y) = $img->curPos

  Return the current position of the pen. Set the current position using moveTo().

- $font = $img->font([$newfont] [,$newsize])

  Get or set the current font. Fonts can be GD::Font objects, TrueType font file paths, or fontconfig font patterns like "Times:italic" (see fontconfig). The latter feature

requires that you have the fontconfig library installed and are using libgd version 2.0.33 or higher.

As a shortcut, you may pass two arguments to set the font and the fontsize simultaneously. The fontsize is only valid when drawing with TrueType fonts.

- $size = $img->fontsize([$newfontsize])

  Get or set the current font size. This is only valid for TrueType fonts.

- $size = $img->penSize([$newpensize])

  Get or set the current pen width for use during line drawing operations.

- $angle = $img->angle([$newangle])

  Set the current angle for use when calling line() or move() with a single argument.

  Here is an example of using turn() and angle() together to draw an octagon. The first line drawn is the downward-slanting top right edge. The last line drawn is the horizontal top of the octagon.

```
$img->moveTo(200,50);
$img->angle(0);
$img->turn(360/8);
for (1..8) { $img->line(50) }
```

- $angle = $img->turn([$newangle])

  Get or set the current angle to turn prior to drawing lines. This value is only used when calling line() or move() with a single argument. The turning angle will be applied to each call to line() or move() just before the actual drawing occurs.

  Angles are in degrees. Positive values turn the angle clockwise.

- $color = $img->fgcolor([$newcolor])

  Get or set the pen's foreground color. The current pen color can be set by (1) using an (r,g,b) triple; (2) using a previously-allocated color from the GD palette; or (3) by using a symbolic color name such as "chartreuse." The list of color names can be obtained using color_names(). The special color name 'transparent' will create a completely transparent color.

- $color = $img->bgcolor([$newcolor])

  Get or set the pen's background color. The current pen color can be set by (1) using an (r,g,b) triple; (2) using a previously-allocated color from the GD palette; or (3) by using a symbolic color name such as "chartreuse." The list of color names can be obtained using color_names(). The special color name 'transparent' will create a completely transparent color.

- $index = $img->translate_color(@args)

  Translates a color into a GD palette or TrueColor index. You may pass either an (r,g,b) triple or a symbolic color name. If you pass a previously-allocated index, the method will return it unchanged.

- $index = $img->alphaColor(@args,$alpha)

Creates an alpha color. You may pass either an (r,g,b) triple or a symbolic color name, followed by an integer indicating its opacity. The opacity value ranges from 0 (fully opaque) to 127 (fully transparent).

- @names = GD::Simple->color_names
- $translate_table = GD::Simple->color_names

  Called in a list context, color_names() returns the list of symbolic color names recognized by this module. Called in a scalar context, the method returns a hash reference in which the keys are the color names and the values are array references containing [r,g,b] triples.

- $gd = $img->gd

  Return the internal GD::Image object. Usually you will not need to call this since all GD methods are automatically referred to this object.

- ($red,$green,$blue) = GD::Simple->HSVtoRGB($hue,$saturation,$value)

  Convert a Hue/Saturation/Value (HSV) color into an RGB triple. The hue, saturation and value are integers from 0 to 255.

- ($hue,$saturation,$value) = GD::Simple->RGBtoHSV($red,$green,$blue)

  Convert a Red/Green/Blue (RGB) value into a Hue/Saturation/Value (HSV) triple. The hue, saturation and value are integers from 0 to 255.

COLORS

This script will create an image showing all the symbolic colors.

```
#!/usr/bin/perl

use strict;
use GD::Simple;

my @color_names = GD::Simple->color_names;
my $cols = int(sqrt(@color_names));
my $rows = int(@color_names/$cols)+1;

my $cell_width    = 100;
my $cell_height   = 50;
my $legend_height = 16;
my $width       = $cols * $cell_width;
my $height      = $rows * $cell_height;

my $img = GD::Simple->new($width,$height);
$img->font(gdSmallFont);

for (my $c=0; $c<$cols; $c++) {
  for (my $r=0; $r<$rows; $r++) {
    my $color = $color_names[$c*$rows + $r] or next;
    my @topleft  = ($c*$cell_width,$r*$cell_height);
    my @botright = ($topleft[0]+$cell_width,$topleft[1]+$cell_
```

(continues on next page)

(continued from previous page)

```
↪height-$legend_height);
    $img->bgcolor($color);
    $img->fgcolor($color);
    $img->rectangle(@topleft,@botright);
    $img->moveTo($topleft[0]+2,$botright[1]+$legend_height-2);
    $img->fgcolor('black');
    $img->string($color);
  }
}

print $img->png;
```

AUTHOR

The GD::Simple module is copyright 2004, Lincoln D. Stein. It is distributed under the same terms as Perl itself. See the "Artistic License" in the Perl source code distribution for licensing terms.

The latest versions of GD.pm are available at https://github.com/lstein/Perl-GD

SEE ALSO

GD, *GD::Polyline*, GD::SVG, Image::Magick

supportsFileType

True when the loaded libgd can read, or with a true second argument write, the image file named by its first argument.

```
print "can do webp\n" if GD::supportsFileType('photo.webp');
print "can write tga\n" if GD::supportsFileType('out.tga', 1);
```

libgd judges by the file name's extension, case-insensitively, so pass a file name: a bare `'webp'` has no extension and is always false. libgd is routinely packaged without WebP, AVIF or JXL, and some formats (TGA, XPM) can be read but not written, so asking is worth doing before offering a format to a user.

5.2.50 GDBM_File

Tie a Perl hash to a database file with GNU dbm's interface: its open modes, error codes, options, dumps and recovery calls.

`GDBM_File` binds a hash to a file with `tie`; from then on `$h{key}`, `delete`, `exists`, `keys` and `each` read and write the file. The object `tie` returns has gdbm's own methods besides: `count` the keys, `sync`, `reorganize`, `dump` and `load` the data in gdbm's dump formats, read the last error, and read or set options.

Synopsis

```
use GDBM_File;

my $db = tie my %h, 'GDBM_File', 'data.gdbm', GDBM_WRCREAT, 0640
    or die "$GDBM_File::gdbm_errno";
$h{greeting} = 'hello';
print $db->count, "\n";          # 1
$db->dump('data.dump');
undef $db;
untie %h;
```

Opening a database

```
my $db = tie %h, 'GDBM_File', $filename, $flags, $mode;
```

`$flags` is one access mode, OR'd with modifiers:

- `GDBM_READER` - open an existing file to read.
- `GDBM_WRITER` - open an existing file to read and write.
- `GDBM_WRCREAT` - the same, creating the file if it does not exist.
- `GDBM_NEWDB` - create the file, emptying it if it exists.
- `GDBM_SYNC` - commit every write to disk as it happens (see `sync`).
- `GDBM_FAST`, `GDBM_NOLOCK`, `GDBM_NOMMAP`, `GDBM_CLOEXEC`, `GDBM_BSEXACT`, `GDBM_XVERIFY`, `GDBM_PREREAD`, `GDBM_NUMSYNC` - accepted and without effect.

`$mode` is the permission of a file it creates. `tie` returns the database object, or undef when the file cannot be opened: `$GDBM_File::gdbm_errno` then says why. Keep the object to call methods on it, and `undef` it (or call `close`) before `untie`: the file stays open as long as the object lives.

Errors

`$GDBM_File::gdbm_errno` holds the code of the last failure: compare it with the `GDBM_*_ERROR` constants as a number, or print it for gdbm's message (with `$!` appended when the system's error says more). A failed method croaks with that message; `errno`, `syserrno` and `strerror` give the last error of one database, and `clear_error` clears it.

Options

These methods read an option and, given a value, set it:

- `$db->sync_mode`, `$db->sync_mode($bool)` - whether every write is committed to disk as it happens. Off unless the database was tied with `GDBM_SYNC`.
- `$db->cache_size`, `$db->cache_size($n)` - the cache size. It reads only once set: before, it croaks `GDBM_File::cache_size not implemented`.
- `$db->centfree`, `$db->centfree($bool)` and `$db->coalesce`, `$db->coalesce($bool)` - gdbm's free-space settings, both off until set. As with gdbm 1.26, `centfree` reads back inverted: 1 while off, 0 once set on.

- `$db->mmapsize`, `$db->mmapsize($n)` - the largest memory map.
- `$db->mmap` - true: the file is always memory-mapped.
- `$db->block_size` - the file's page size, typically 4096.
- `$db->dbname` - the file name the database was tied with.
- `$db->flags` - the `$flags` it was tied with, `GDBM_SYNC` included while the sync mode is on.

Only `sync_mode` changes how the database works. The cache size, free space settings and memory map size keep the value they are given and change nothing: they tune gdbm's own file layout, which pperl's files do not have. `setopt` reaches the same options by their `GDBM_SET*` codes.

DBM filters

`$db->filter_store_key(sub {...})`, `filter_store_value`, `filter_fetch_key` and `filter_fetch_value` install a sub that rewrites every key or value on its way into, or out of, the database. The sub changes `$_` in place; a store filter works on a copy, so your own variable keeps its value. Each method returns the filter it replaces (or undef), and `undef` as the argument removes the filter. The `DBM_Filter` module offers ready-made ones (`utf8`, `null`, `int32` and others). A filter that uses the same database croaks `recursion detected in filter_store_key` (or the filter's own name).

Files

pperl keeps a GDBM_File database in an LMDB file: the file named, with LMDB's lock file, `NAME-lock`, beside it. The gdbm library is not used, and its files cannot be read: tying a real gdbm file (or a Berkeley DB file) croaks, naming what the file is, and the file is left as it was, even with `GDBM_NEWDB`. To bring a gdbm database over, dump it with gdbm's own tools (`gdbm_dump`, or `dump` under perl 5) and `load` the dump. A key may be at most 511 bytes, 510 if it starts with a NUL byte; a longer one croaks.

Writes reach the file at `sync` and when the database is untied: until then the tie holds the file's one writer slot, and another process that writes the file waits for it (readers never wait), as with gdbm's own writer lock. `GDBM_SYNC`, `sync_mode(1)` or `$Peta::FFI::LMDB::DBM_WRITE = 'through'` commit every write as it happens; see `Peta::FFI::LMDB`.

Constants

GDBM_File exports gdbm's constants: the open modes and modifiers above, `GDBM_INSERT` and `GDBM_REPLACE` (the flag of `STORE`), the `GDBM_*_ERROR` and other error codes `$GDBM_File::gdbm_errno` holds, the `GDBM_SET*` option codes of `setopt`, and the `GDBM_SNAPSHOT_*` codes of `latest_snapshot`.

Differences from upstream

- The files are LMDB files, not gdbm files: real gdbm files are refused, and pperl's files cannot be read by gdbm. Data moves between them through `dump` and `load`, in either direction. Covered by `t/82-native/Peta-FFI-LMDB/090-gdbm.t` and `t/81-xs-native/GDBM_File/010-phenotype.t`.

- Keys are limited to 511 bytes; gdbm has no limit. Covered by `t/82-native/Peta-FFI-LMDB/090-gdbm.t`.
- Writes reach the file at `sync` and `untie` unless the sync mode is on, as with gdbm, and a writer holds the file until then. Covered by `t/82-native/Peta-FFI-LMDB/090-gdbm.t`.
- `load` reads dumps with empty keys or values, which gdbm itself writes but cannot load. Covered by `t/82-native/Peta-FFI-LMDB/090-gdbm.t`.
- Deleting the key `each` just returned is safe: the walk goes on with the next key, where gdbm loses its place and ends the walk early. Covered by `t/82-native/Peta-FFI-LMDB/090-gdbm.t`.
- Crash tolerance (`failure_atomic`, `latest_snapshot`) is reported unsupported, as by a gdbm built without it: pperl's files survive a crash by themselves. Covered by `t/81-xs-native/GDBM_File/150-upstream-snapshot.t`.
- The cache size, free-space and memory-map options have no effect. `cache_size` reads back a size once set, where perl 5 with gdbm 1.26 croaks `not implemented` on every read. Covered by `t/82-native/Peta-FFI-LMDB/090-gdbm.t`.

Everything else - error codes and messages, options, dump formats - behaves as with gdbm. Covered by `t/81-xs-native/GDBM_File/010-phenotype.t` and GDBM_File's own test suite, `t/81-xs-native/GDBM_File/100-upstream-count.t` to `150-upstream-snapshot.t`.

Functions

Errors

gdbm_check_syserr

True when an error code is one that `$!` explains further - a failed open, read or write - as in `gdbm_check_syserr($GDBM_File::gdbm_errno)`.

errno

The last error of this database: as a number one of the `GDBM_*` error codes (`GDBM_NO_ERROR`, 0, when none), as a string gdbm's message for it, such as `Item not found`.

syserrno

The system error number (the `$!` value) behind this database's last error, when that error is one the system's error explains; 0 otherwise.

strerror

gdbm's message for this database's last error, with the system's message appended when that says more.

clear_error

Forget this database's last error: `errno` reads `GDBM_NO_ERROR` again.

Maintenance

close

Close the database now, writing what is pending, even while copies of the database object exist. Needed when the object was kept from `tie`: `untie` alone leaves the file open as long as the object lives. Returns 0. Methods called afterwards croak `database was closed`.

needs_recovery

Whether the database needs `recover`: always 0, as an interrupted write leaves pperl's file as it was before the write.

reorganize

Shrink the file after many deletes: the data is copied, without the free space, into a new file that takes the old one's place and mode.

recover

Recover a damaged database. pperl's files are never left damaged, so recovery finds every key intact; it still makes the backup copy and fills in the statistics asked for.

sync

Write everything stored so far to the file and to disk, so that other processes - and the file after a crash - have it. Returns 0; a failure croaks. Without it, writes reach the file at `untie`, or as they happen in sync mode (`GDBM_SYNC`, `sync_mode(1)`).

count

The number of keys in the database, without walking them.

dump

Write the database's keys and values to a dump file in gdbm's format, which `load` - here or in perl 5 with gdbm - reads back.

load

Read a dump file - ASCII or binary, from `dump` here or from gdbm's own tools - into the database.

Options

block_size

The `$flags` the database was tied with, `GDBM_SYNC` included while the sync mode is on. The same method, under the option's name, reads and sets the options listed under *Options* above (`sync_mode`, `cache_size`, `block_size` and the others); with a value too many it croaks `NAME: too many arguments`.

Also available as `GDBM_File::cache_size`, `GDBM_File::centfree`, `GDBM_File::coalesce`, `GDBM_File::dbname`, `GDBM_File::flags`, `GDBM_File::mmap`, `GDBM_File::mmapsize`, `GDBM_File::sync_mode`.

setopt

`$db->setopt($code, $value, $length)`: gdbm's raw option call. `$code` is a `GDBM_SET*` (or `GDBM_GET*`) constant, `$value` an integer and `$length` its size in bytes, 4. Returns 0; an unknown option or a bad value croaks. The named methods under *Options* are easier to use.

Crash tolerance

convert

`$db->convert($format)`: change gdbm's file format, `GDBM_NUMSYNC` for its extended format or 0 for its standard one. pperl's files have one format, so it does nothing and returns 0.

failure_atomic

`$db->failure_atomic($even, $odd)`: gdbm's crash tolerance, which keeps two snapshot files of the database. Not supported: it croaks `gdbm_failure_atomic: Function usage error: Function not implemented`. pperl's files survive a crash without snapshots, keeping the writes made up to the last `sync`.

latest_snapshot

`GDBM_File->latest_snapshot($even, $odd)`: which of crash tolerance's two snapshot files to recover from. Not supported: it returns undef, in list context `(undef, GDBM_SNAPSHOT_ERR, ENOSYS)`.

crash_tolerance_status

`GDBM_File->crash_tolerance_status`: whether `failure_atomic` is supported - 0.

Library version

GDBM_version

The version of gdbm whose interface the module follows: in scalar context a string such as `1.26`, in list context its numbers, such as `(1, 26, 0)`. Call it on the class: `GDBM_File->GDBM_version`.

reorganize

Shrink the file after many deletes: the data is copied, without the free space, into a new file that takes the old one's place and mode.

Synopsis

```
delete @h{@old_keys};
$db->reorganize;
```

What you get back

0, or -1 with `errno` set: `GDBM_READER_CANT_REORGANIZE` for a database tied `GDBM_READER`, `GDBM_REORGANIZE_FAILED` when it cannot be done.

Edge cases

- It fails while another tie in the same process has the file open, and should not be run while other processes use the file.
- Writes pending are written first.

Differences from upstream

The work is a copy of the file without its free pages; gdbm rebuilds its own file instead. The result is the same: a smaller file with the same data. Covered by `t/82-native/Peta-FFI-LMDB/090-gdbm.t`.

See also

- `count` - how many keys there are
- `sync` - writes what is pending without shrinking

recover

Recover a damaged database. pperl's files are never left damaged, so recovery finds every key intact; it still makes the backup copy and fills in the statistics asked for.

Synopsis

```
my ($backup, %stat);
$db->recover(backup => \$backup, stat => \%stat);
```

Options

- `backup => \$name` - copy the file first, to `NAME.~N~` with the first free number `N`, and put that name in `$name`.

- `stat => \%stat` - fill in `recovered_keys` (every key), `recovered_buckets`, `failed_keys` and `failed_buckets` (both 0).
- `err => sub {...}`, `max_failed_keys`, `max_failed_buckets`, `max_failures` - accepted; with nothing failing they never matter.

What you get back

Nothing; a failure (the backup copy cannot be made) croaks. An unknown option croaks `NAME: unrecognized argument`, and an odd number of arguments with a usage message.

Differences from upstream

gdbm rebuilds a damaged file and reports what it lost; pperl's file has nothing to rebuild. Covered by `t/81-xs-native/GDBM_File/010-phenotype.t`.

See also

- `needs_recovery` - always 0

dump

Write the database's keys and values to a dump file in gdbm's format, which `load` - here or in perl 5 with gdbm - reads back.

Synopsis

```
$db->dump('data.dump');
$db->dump('data.bin', binary => 1, mode => 0600, overwrite => 1);
```

Options

- `binary => 1` - gdbm's old binary format instead of its ASCII one. The ASCII dump also records the file's name, owner and mode.
- `mode => $mode` - the dump file's permissions (default `0666`, less the umask).
- `overwrite => 1` - replace an existing dump file; without it, an existing file croaks.

What you get back

Nothing; a failure croaks with gdbm's message. An unknown option croaks `unrecognized keyword: NAME`.

Examples

Move a real gdbm database to pperl: dump it under perl 5 with gdbm, then load it here.

```
# perl 5:  (tied %old)->dump('move.dump');
my $db = tie my %h, 'GDBM_File', 'new.gdbm', GDBM_NEWDB, 0640 or␣
↪die;
$db->load('move.dump');
```

Differences from upstream

Fully compatible with upstream: gdbm loads pperl's dumps and pperl gdbm's. Covered by `t/81-xs-native/GDBM_File/010-phenotype.t`.

See also

- `load` - reads a dump back

load

Read a dump file - ASCII or binary, from dump here or from gdbm's own tools - into the database.

Synopsis

```
my $db = tie my %h, 'GDBM_File', 'a.gdbm', GDBM_NEWDB, 0640 or die;
$db->load('a.dump');
$db->load('more.dump', replace => 1);
```

Options

- `replace => 1` - a key already in the database takes the dump's value; without it such a key croaks.
- `restore_mode => 0`, `restore_owner => 0` - do not give the database file the mode, or the owner, an ASCII dump records. By default both are restored, and failing to restore them is not an error.
- `strict_errors => 1` - croak when the mode or owner cannot be restored.

What you get back

Nothing; a failure croaks `FILE:LINE: database load error: MESSAGE` (or without the line). An unknown option croaks `unrecognized keyword: NAME`.

Differences from upstream

A dump holding an empty key or an empty value loads; gdbm 1.26 writes such dumps but cannot load them. Covered by `t/82-native/Peta-FFI-LMDB/090-gdbm.t`.

See also

- dump - writes a dump

5.2.51 Getopt

Namespace containing the following modules:

- `Getopt::Long` - Extended processing of command line options
- `Getopt::Std` - Process single-character switches with switch clustering

Getopt::Long

Extended processing of command line options

SYNOPSIS

```
use Getopt::Long;
my $data   = "file.dat";
my $length = 24;
my $verbose;
GetOptions ("length=i" => \$length,    # numeric
            "file=s"   => \$data,      # string
            "verbose"  => \$verbose)   # flag
or die("Error in command line arguments\n");
```

DESCRIPTION

The Getopt::Long module implements an extended getopt function called GetOptions(). It parses the command line from `@ARGV`, recognizing and removing specified options and their possible values.

This function adheres to the POSIX syntax for command line options, with GNU extensions. In general, this means that options have long names instead of single letters, and are introduced with a double dash "–". Support for bundling of command line options, as was the case with the more traditional single-letter approach, is provided but not enabled by default.

Command Line Options, an Introduction

Command line operated programs traditionally take their arguments from the command line, for example filenames or other information that the program needs to know. Besides arguments, these programs often take command line *options* as well. Options are not necessary for the program to work, hence the name 'option', but are used to modify its default behaviour. For example, a program could do its job quietly, but with a suitable option it could provide verbose information about what it did.

Command line options come in several flavours. Historically, they are preceded by a single dash -, and consist of a single letter.

```
-l -a -c
```

Usually, these single-character options can be bundled:

```
-lac
```

Options can have values, the value is placed after the option character. Sometimes with whitespace in between, sometimes not:

```
-s 24 -s24
```

Due to the very cryptic nature of these options, another style was developed that used long names. So instead of a cryptic `-l` one could use the more descriptive `--long`. To distinguish between a bundle of single-character options and a long one, two dashes are used to precede the option name. Early implementations of long options used a plus + instead. Also, option values could be specified either like

```
--size=24
```

or

```
--size 24
```

The + form is now obsolete and strongly deprecated.

Getting Started with Getopt::Long

Getopt::Long is the Perl5 successor of `newgetopt.pl`. This was the first Perl module that provided support for handling the new style of command line options, in particular long option names, hence the Perl5 name Getopt::Long. This module also supports single-character options and bundling.

To use Getopt::Long from a Perl program, you must include the following line in your Perl program:

```
use Getopt::Long;
```

This will load the core of the Getopt::Long module and prepare your program for using it. Most of the actual Getopt::Long code is not loaded until you really call one of its functions.

In the default configuration, options names may be abbreviated to uniqueness, case does not matter, and a single dash is sufficient, even for long option names. Also, options may be placed between non-option arguments. See “Configuring Getopt::Long” for more details on how to configure Getopt::Long.

Simple options

The most simple options are the ones that take no values. Their mere presence on the command line enables the option. Popular examples are:

```
--all --verbose --quiet --debug
```

Handling simple options is straightforward:

```
my $verbose = '';   # option variable with default value (false)
my $all = '';       # option variable with default value (false)
GetOptions ('verbose' => \$verbose, 'all' => \$all);
```

The call to GetOptions() parses the command line arguments that are present in `@ARGV` and sets the option variable to the value 1 if the option did occur on the command line. Otherwise, the option variable is not touched. Setting the option value to true is often called *enabling* the option.

The option name as specified to the GetOptions() function is called the option *specification*. Later we'll see that this specification can contain more than just the option name. The reference to the variable is called the option *destination*.

GetOptions() will return a true value if the command line could be processed successfully. Otherwise, it will write error messages using die() and warn(), and return a false result.

A little bit less simple options

Getopt::Long supports two useful variants of simple options: *negatable* options and *incremental* options.

A negatable option is specified with an exclamation mark `!` after the option name:

```
my $verbose = '';   # option variable with default value (false)
GetOptions ('verbose!' => \$verbose);
```

Now, using `--verbose` on the command line will enable `$verbose`, as expected. But it is also allowed to use `--noverbose`, which will disable `$verbose` by setting its value to `0`. Using a suitable default value, the program can find out whether `$verbose` is false by default, or disabled by using `--noverbose`.

(If both `--verbose` and `--noverbose` are given, whichever is given last takes precedence.)

An incremental option is specified with a plus `+` after the option name:

```
my $verbose = '';   # option variable with default value (false)
GetOptions ('verbose+' => \$verbose);
```

Using `--verbose` on the command line will increment the value of `$verbose`. This way the program can keep track of how many times the option occurred on the command line. For example, each occurrence of `--verbose` could increase the verbosity level of the program.

Mixing command line option with other arguments

Usually programs take command line options as well as other arguments, for example, file names. It is good practice to always specify the options first, and the other arguments last. Getopt::Long will, however, allow the options and arguments to be mixed and 'filter out' all the options before passing the rest of the arguments to the program.

To stop Getopt::Long from processing further arguments, insert a double dash `--` on the command line:

```
--size 24 -- --all
```

In this example, `--all` will *not* be treated as an option, but passed to the program unharmed, in `@ARGV`.

Options with values

For options that take values it must be specified whether the option value is required or not, and what kind of value the option expects.

Three kinds of values are supported: integer numbers, floating point numbers, and strings.

If the option value is required, Getopt::Long will take the command line argument that follows the option and assign this to the option variable. If, however, the option value is specified as optional, this will only be done if that value does not look like a valid command line option itself.

```
my $tag = '';       # option variable with default value
GetOptions ('tag=s' => \$tag);
```

In the option specification, the option name is followed by an equals sign `=` and the letter `s`. The equals sign indicates that this option requires a value. The letter `s` indicates that this value is an arbitrary string. Other possible value types are `i` for integer values, and `f` for floating point values. Using a colon `:` instead of the equals sign indicates that the option value is optional. In this case, if no suitable value is supplied, string valued options get an empty string `''` assigned, while numeric options are set to `0`.

(If the same option appears more than once on the command line, the last given value is used. If you want to take all the values, see below.)

Options with multiple values

Options sometimes take several values. For example, a program could use multiple directories to search for library files:

```
--library lib/stdlib --library lib/extlib
```

To accomplish this behaviour, simply specify an array reference as the destination for the option:

```
GetOptions ("library=s" => \@libfiles);
```

Alternatively, you can specify that the option can have multiple values by adding a “@”, and pass a reference to a scalar as the destination:

```
GetOptions ("library=s@" => \$libfiles);
```

Used with the example above, `@libfiles` c.q. `@$libfiles` would contain two strings upon completion: `"lib/stdlib"` and `"lib/extlib"`, in that order. It is also possible to specify that only integer or floating point numbers are acceptable values.

Often it is useful to allow comma-separated lists of values as well as multiple occurrences of the options. This is easy using Perl's split() and join() operators:

```
GetOptions ("library=s" => \@libfiles);
@libfiles = split(/,/,join(',',@libfiles));
```

Of course, it is important to choose the right separator string for each purpose.

Warning: What follows is an experimental feature.

Options can take multiple values at once, for example

```
--coordinates 52.2 16.4 --rgbcolor 255 255 149
```

This can be accomplished by adding a repeat specifier to the option specification. Repeat specifiers are very similar to the `{...}` repeat specifiers that can be used with regular expression patterns. For example, the above command line would be handled as follows:

```
GetOptions('coordinates=f{2}' => \@coor, 'rgbcolor=i{3}' => \
 ↪@color);
```

The destination for the option must be an array or array reference.

It is also possible to specify the minimal and maximal number of arguments an option takes. `foo=s{2,4}` indicates an option that takes at least two and at most 4 arguments. `foo=s{1,}` indicates one or more values; `foo:s{,}` indicates zero or more option values.

Options with hash values

If the option destination is a reference to a hash, the option will take, as value, strings of the form *key=value*. The value will be stored with the specified key in the hash.

```
GetOptions ("define=s" => \%defines);
```

Alternatively you can use:

```
GetOptions ("define=s%" => \$defines);
```

When used with command line options:

```
--define os=linux --define vendor=redhat
```

the hash `%defines` (or `%$defines`) will contain two keys, `"os"` with value `"linux"` and `"vendor"` with value `"redhat"`. It is also possible to specify that only integer or floating point numbers are acceptable values. The keys are always taken to be strings.

User-defined subroutines to handle options

Ultimate control over what should be done when (actually: each time) an option is encountered on the command line can be achieved by designating a reference to a subroutine (or an anonymous subroutine) as the option destination. When GetOptions() encounters the option, it will call the subroutine with two or three arguments. The first

argument is the name of the option. (Actually, it is an object that stringifies to the name of the option.) For a scalar or array destination, the second argument is the value to be stored. For a hash destination, the second argument is the key to the hash, and the third argument the value to be stored. It is up to the subroutine to store the value, or do whatever it thinks is appropriate.

A trivial application of this mechanism is to implement options that are related to each other. For example:

```
my $verbose = '';   # option variable with default value (false)
GetOptions ('verbose' => \$verbose,
            'quiet'   => sub { $verbose = 0 });
```

Here `--verbose` and `--quiet` control the same variable `$verbose`, but with opposite values.

If the subroutine needs to signal an error, it should call die() with the desired error message as its argument. GetOptions() will catch the die(), issue the error message, and record that an error result must be returned upon completion.

If the text of the error message starts with an exclamation mark `!` it is interpreted specially by GetOptions(). There is currently one special command implemented: `die("!FINISH")` will cause GetOptions() to stop processing options, as if it encountered a double dash `--`.

Here is an example of how to access the option name and value from within a subroutine:

```
GetOptions ('opt=i' => \&handler);
sub handler {
    my ($opt_name, $opt_value) = @_;
    print("Option name is $opt_name and value is $opt_value\n");
}
```

Options with multiple names

Often it is user friendly to supply alternate mnemonic names for options. For example `--height` could be an alternate name for `--length`. Alternate names can be included in the option specification, separated by vertical bar `|` characters. To implement the above example:

```
GetOptions ('length|height=f' => \$length);
```

The first name is called the *primary* name, the other names are called *aliases*. When using a hash to store options, the key will always be the primary name.

Multiple alternate names are possible.

Case and abbreviations

Without additional configuration, GetOptions() will ignore the case of option names, and allow the options to be abbreviated to uniqueness.

```
GetOptions ('length|height=f' => \$length, "head" => \$head);
```

This call will allow `--l` and `--L` for the length option, but requires a least `--hea` and `--hei` for the head and height options.

Summary of Option Specifications

Each option specifier consists of two parts: the name specification and the argument specification.

The name specification contains the name of the option, optionally followed by a list of alternative names separated by vertical bar characters. The name is made up of alphanumeric characters, hyphens, underscores. If `pass_through` is disabled, a period is also allowed in option names.

```
length            option name is "length"
length|size|l     name is "length", aliases are "size" and "l"
```

The argument specification is optional. If omitted, the option is considered boolean, a value of 1 will be assigned when the option is used on the command line.

The argument specification can be

- !

  The option does not take an argument and may be negated by prefixing it with "no" or "no-". E.g. `"foo!"` will allow `--foo` (a value of 1 will be assigned) as well as `--nofoo` and `--no-foo` (a value of 0 will be assigned). If the option has aliases, this applies to the aliases as well.

  Using negation on a single letter option when bundling is in effect is pointless and will result in a warning.

- +

  The option does not take an argument and will be incremented by 1 every time it appears on the command line. E.g. `"more+"`, when used with `--more --more --more`, will increment the value three times, resulting in a value of 3 (provided it was 0 or undefined at first).

  The + specifier is ignored if the option destination is not a scalar.

- = *type* [*desttype*] [*repeat*]

  The option requires an argument of the given type. Supported types are:

  - s

    String. An arbitrary sequence of characters. It is valid for the argument to start with - or --.

  - i

    Integer. An optional leading plus or minus sign, followed by a sequence of digits.

  - o

Extended integer, Perl style. This can be either an optional leading plus or minus sign, followed by a sequence of digits, or an octal string (a zero, optionally followed by '0', '1', .. '7'), or a hexadecimal string (`0x` followed by '0' .. '9', 'a' .. 'f', case insensitive), or a binary string (`0b` followed by a series of '0' and '1').

– f

Real number. For example `3.14`, `-6.23E24` and so on.

The *desttype* can be `@` or `%` to specify that the option is list or a hash valued. This is only needed when the destination for the option value is not otherwise specified. It should be omitted when not needed.

The *repeat* specifies the number of values this option takes per occurrence on the command line. It has the format `{` [*min*] [`,` [*max*]] `}`.

min denotes the minimal number of arguments. It defaults to 1 for options with `=` and to 0 for options with `:`, see below. Note that *min* overrules the `=` / `:` semantics.

max denotes the maximum number of arguments. It must be at least *min*. If *max* is omitted, *but the comma is not*, there is no upper bound to the number of argument values taken.

- : *type* [*desttype*]

  Like `=`, but designates the argument as optional. If omitted, an empty string will be assigned to string values options, and the value zero to numeric options.

  Note that if a string argument starts with `-` or `--`, it will be considered an option on itself.

- : *number* [*desttype*]

  Like `:i`, but if the value is omitted, the *number* will be assigned.

  If the *number* is octal, hexadecimal or binary, behaves like `:o`.

- : `+` [*desttype*]

  Like `:i`, but if the value is omitted, the current value for the option will be incremented.

Advanced Possibilities

Object oriented interface

See *Getopt::Long::Parser*.

Callback object

In version 2.37 the first argument to the callback function was changed from string to object. This was done to make room for extensions and more detailed control. The object stringifies to the option name so this change should not introduce compatibility problems.

The callback object has the following methods:

- name

The name of the option, unabbreviated. For an option with multiple names it return the first (canonical) name.

- given

  The name of the option as actually used, unabbreveated.

Thread Safety

Getopt::Long is thread safe when using ithreads as of Perl 5.8. It is *not* thread safe when using the older (experimental and now obsolete) threads implementation that was added to Perl 5.005.

Documentation and help texts

Getopt::Long encourages the use of Pod::Usage to produce help messages. For example:

```
use Getopt::Long;
use Pod::Usage;

my $man = 0;
my $help = 0;

GetOptions('help|?' => \$help, man => \$man) or pod2usage(2);
pod2usage(1) if $help;
pod2usage(-exitval => 0, -verbose => 2) if $man;

__END__

=head1 NAME

sample - Using Getopt::Long and Pod::Usage

=head1 SYNOPSIS

sample [options] [file ...]

 Options:
   -help            brief help message
   -man             full documentation

=head1 OPTIONS

=over 8

=item B<-help>

Print a brief help message and exits.

=item B<-man>
```

(continues on next page)

(continued from previous page)

```
Prints the manual page and exits.

=back

=head1 DESCRIPTION

B<This program> will read the given input file(s) and do something
useful with the contents thereof.

=cut
```

See *Pod::Usage* for details.

Parsing options from an arbitrary array

By default, GetOptions parses the options that are present in the global array `@ARGV`. A special entry `GetOptionsFromArray` can be used to parse options from an arbitrary array.

```
use Getopt::Long qw(GetOptionsFromArray);
$ret = GetOptionsFromArray(\@myopts, ...);
```

When used like this, options and their possible values are removed from `@myopts`, the global `@ARGV` is not touched at all.

The following two calls behave identically:

```
$ret = GetOptions( ... );
$ret = GetOptionsFromArray(\@ARGV, ... );
```

This also means that a first argument hash reference now becomes the second argument:

```
$ret = GetOptions(\%opts, ... );
$ret = GetOptionsFromArray(\@ARGV, \%opts, ... );
```

Parsing options from an arbitrary string

A special entry `GetOptionsFromString` can be used to parse options from an arbitrary string.

```
use Getopt::Long qw(GetOptionsFromString);
$ret = GetOptionsFromString($string, ...);
```

The contents of the string are split into arguments using a call to `Text::ParseWords::shellwords`. As with `GetOptionsFromArray`, the global `@ARGV` is not touched.

It is possible that, upon completion, not all arguments in the string have been processed. `GetOptionsFromString` will, when called in list context, return both the return status and an array reference to any remaining arguments:

```
($ret, $args) = GetOptionsFromString($string, ... );
```

If any arguments remain, and `GetOptionsFromString` was not called in list context, a message will be given and `GetOptionsFromString` will return failure.

As with GetOptionsFromArray, a first argument hash reference now becomes the second argument. See the next section.

Storing options values in a hash

Sometimes, for example when there are a lot of options, having a separate variable for each of them can be cumbersome. GetOptions() supports, as an alternative mechanism, storing options values in a hash.

To obtain this, a reference to a hash must be passed *as the first argument* to GetOptions(). For each option that is specified on the command line, the option value will be stored in the hash with the option name as key. Options that are not actually used on the command line will not be put in the hash, on other words, `exists($h{option})` (or defined()) can be used to test if an option was used. The drawback is that warnings will be issued if the program runs under `use strict` and uses `$h{option}` without testing with exists() or defined() first.

```
my %h = ();
GetOptions (\%h, 'length=i');       # will store in $h{length}
```

For options that take list or hash values, it is necessary to indicate this by appending an @ or % sign after the type:

```
GetOptions (\%h, 'colours=s@');     # will push to @{$h{colours}}
```

To make things more complicated, the hash may contain references to the actual destinations, for example:

```
my $len = 0;
my %h = ('length' => \$len);
GetOptions (\%h, 'length=i');       # will store in $len
```

This example is fully equivalent with:

```
my $len = 0;
GetOptions ('length=i' => \$len);   # will store in $len
```

Any mixture is possible. For example, the most frequently used options could be stored in variables while all other options get stored in the hash:

```
my $verbose = 0;                    # frequently referred
my $debug = 0;                      # frequently referred
my %h = ('verbose' => \$verbose, 'debug' => \$debug);
GetOptions (\%h, 'verbose', 'debug', 'filter', 'size=i');
if ( $verbose ) { ... }
if ( exists $h{filter} ) { ... option 'filter' was specified ... }
```

Bundling

With bundling it is possible to set several single-character options at once. For example if a, v and x are all valid options,

```
-vax
```

will set all three.

Getopt::Long supports three styles of bundling. To enable bundling, a call to Getopt::Long::Configure is required.

The simplest style of bundling can be enabled with:

```
Getopt::Long::Configure ("bundling");
```

Configured this way, single-character options can be bundled but long options (and any of their auto-abbreviated shortened forms) **must** always start with a double dash `--` to avoid ambiguity. For example, when `vax`, `a`, `v` and `x` are all valid options,

```
-vax
```

will set `a`, `v` and `x`, but

```
--vax
```

will set `vax`.

The second style of bundling lifts this restriction. It can be enabled with:

```
Getopt::Long::Configure ("bundling_override");
```

Now, `-vax` will set the option `vax`.

In all of the above cases, option values may be inserted in the bundle. For example:

```
-h24w80
```

is equivalent to

```
-h 24 -w 80
```

A third style of bundling allows only values to be bundled with options. It can be enabled with:

```
Getopt::Long::Configure ("bundling_values");
```

Now, `-h24` will set the option `h` to `24`, but option bundles like `-vxa` and `-h24w80` are flagged as errors.

Enabling `bundling_values` will disable the other two styles of bundling.

When configured for bundling, single-character options are matched case sensitive while long options are matched case insensitive. To have the single-character options matched case insensitive as well, use:

```
Getopt::Long::Configure ("bundling", "ignorecase_always");
```

It goes without saying that bundling can be quite confusing.

The lonesome dash

Normally, a lone dash - on the command line will not be considered an option. Option processing will terminate (unless "permute" is configured) and the dash will be left in @ARGV.

It is possible to get special treatment for a lone dash. This can be achieved by adding an option specification with an empty name, for example:

```
GetOptions ('' => \$stdio);
```

A lone dash on the command line will now be a legal option, and using it will set variable $stdio.

Argument callback

A special option 'name' <> can be used to designate a subroutine to handle non-option arguments. When GetOptions() encounters an argument that does not look like an option, it will immediately call this subroutine and passes it one parameter: the argument name.

For example:

```
my $width = 80;
sub process { ... }
GetOptions ('width=i' => \$width, '<>' => \&process);
```

When applied to the following command line:

```
arg1 --width=72 arg2 --width=60 arg3
```

This will call process("arg1") while $width is 80, process("arg2") while $width is 72, and process("arg3") while $width is 60.

This feature requires configuration option **permute**, see section "Configuring Getopt::Long".

Configuring Getopt::Long

Getopt::Long can be configured by calling subroutine Getopt::Long::Configure(). This subroutine takes a list of quoted strings, each specifying a configuration option to be enabled, e.g. ignore_case. To disable, prefix with no or no_, e.g. no_ignore_case. Case does not matter. Multiple calls to Configure() are possible.

Alternatively, as of version 2.24, the configuration options may be passed together with the use statement:

```
use Getopt::Long qw(:config no_ignore_case bundling);
```

The following options are available:

- default

  This option causes all configuration options to be reset to their default values.

- posix_default

  This option causes all configuration options to be reset to their default values as if the environment variable POSIXLY_CORRECT had been set.

- auto_abbrev

  Allow option names to be abbreviated to uniqueness. Default is enabled unless environment variable POSIXLY_CORRECT has been set, in which case `auto_abbrev` is disabled.

- getopt_compat

  Allow `+` to start options. Default is enabled unless environment variable POSIXLY_CORRECT has been set, in which case `getopt_compat` is disabled.

- gnu_compat

  `gnu_compat` controls whether `--opt=` is allowed, and what it should do. Without `gnu_compat`, `--opt=` gives an error. With `gnu_compat`, `--opt=` will give option `opt` an empty value. This is the way GNU getopt_long() does it.

  Note that for options with optional arguments, `--opt value` is still accepted, even though GNU getopt_long() requires writing `--opt=value` in this case.

- gnu_getopt

  This is a short way of setting `gnu_compat` `bundling` `permute` `no_getopt_compat`. With `gnu_getopt`, command line handling should be reasonably compatible with GNU getopt_long().

- require_order

  Whether command line arguments are allowed to be mixed with options. Default is disabled unless environment variable POSIXLY_CORRECT has been set, in which case `require_order` is enabled.

  See also `permute`, which is the opposite of `require_order`.

- permute

  Whether command line arguments are allowed to be mixed with options. Default is enabled unless environment variable POSIXLY_CORRECT has been set, in which case `permute` is disabled. Note that `permute` is the opposite of `require_order`.

  If `permute` is enabled, this means that

  ```
  --foo arg1 --bar arg2 arg3
  ```

  is equivalent to

  ```
  --foo --bar arg1 arg2 arg3
  ```

If an argument callback routine is specified, @ARGV will always be empty upon successful return of GetOptions() since all options have been processed. The only exception is when `--` is used:

```
--foo arg1 --bar arg2 -- arg3
```

This will call the callback routine for arg1 and arg2, and then terminate GetOptions() leaving `"arg3"` in @ARGV.

If `require_order` is enabled, options processing terminates when the first non-option is encountered.

```
--foo arg1 --bar arg2 arg3
```

is equivalent to

```
--foo -- arg1 --bar arg2 arg3
```

If `pass_through` is also enabled, options processing will terminate at the first unrecognized option, or non-option, whichever comes first.

- bundling (default: disabled)

  Enabling this option will allow single-character options to be bundled. To distinguish bundles from long option names, long options (and any of their auto-abbreviated shortened forms) *must* be introduced with `--` and bundles with `-`.

  Note that, if you have options `a`, `l` and `all`, and auto_abbrev enabled, possible arguments and option settings are:

```
using argument               sets option(s)
------------------------------------------
-a, --a                      a
-l, --l                      l
-al, -la, -ala, -all,...     a, l
--al, --all                  all
```

  The surprising part is that `--a` sets option `a` (due to auto completion), not `all`.

  Note: disabling `bundling` also disables `bundling_override`.

- bundling_override (default: disabled)

  If `bundling_override` is enabled, bundling is enabled as with `bundling` but now long option names override option bundles.

  Note: disabling `bundling_override` also disables `bundling`.

  Note: Using option bundling can easily lead to unexpected results, especially when mixing long options and bundles. Caveat emptor.

- ignore_case (default: enabled)

  If enabled, case is ignored when matching option names. If, however, bundling is enabled as well, single character options will be treated case-sensitive.

With `ignore_case`, option specifications for options that only differ in case, e.g., `"foo"` and `"Foo"`, will be flagged as duplicates.

Note: disabling `ignore_case` also disables `ignore_case_always`.

- ignore_case_always (default: disabled)

  When bundling is in effect, case is ignored on single-character options also.

  Note: disabling `ignore_case_always` also disables `ignore_case`.

- auto_version (default:disabled)

  Automatically provide support for the –**version** option if the application did not specify a handler for this option itself.

  Getopt::Long will provide a standard version message that includes the program name, its version (if $main::VERSION is defined), and the versions of Getopt::Long and Perl. The message will be written to standard output and processing will terminate.

  `auto_version` will be enabled if the calling program explicitly specified a version number higher than 2.32 in the `use` or `require` statement.

- auto_help (default:disabled)

  Automatically provide support for the –**help** and **-?** options if the application did not specify a handler for this option itself.

  Getopt::Long will provide a help message using module *Pod::Usage*. The message, derived from the SYNOPSIS POD section, will be written to standard output and processing will terminate.

  `auto_help` will be enabled if the calling program explicitly specified a version number higher than 2.32 in the `use` or `require` statement.

- pass_through (default: disabled)

  With `pass_through` anything that is unknown, ambiguous or supplied with an invalid option will not be flagged as an error. Instead the unknown option(s) will be passed to the catchall `<>` if present, otherwise through to `@ARGV`. This makes it possible to write wrapper scripts that process only part of the user supplied command line arguments, and pass the remaining options to some other program.

  If `require_order` is enabled, options processing will terminate at the first unrecognized option, or non-option, whichever comes first and all remaining arguments are passed to `@ARGV` instead of the catchall `<>` if present. However, if `permute` is enabled instead, results can become confusing.

  Note that the options terminator (default `--`), if present, will also be passed through in `@ARGV`.

- prefix

  The string that starts options. If a constant string is not sufficient, see `prefix_pattern`.

- prefix_pattern

A Perl pattern that identifies the strings that introduce options. Default is `--|-|\+` unless environment variable POSIXLY_CORRECT has been set, in which case it is `--|-`.

- long_prefix_pattern

  A Perl pattern that allows the disambiguation of long and short prefixes. Default is `--`.

  Typically you only need to set this if you are using nonstandard prefixes and want some or all of them to have the same semantics as '–' does under normal circumstances.

  For example, setting prefix_pattern to `--|-|\+|\/` and long_prefix_pattern to `--|\/` would add Win32 style argument handling.

- debug (default: disabled)

  Enable debugging output.

Exportable Methods

- VersionMessage

  This subroutine provides a standard version message. Its argument can be:

  – A string containing the text of a message to print *before* printing the standard message.

  – A numeric value corresponding to the desired exit status.

  – A reference to a hash.

  If more than one argument is given then the entire argument list is assumed to be a hash. If a hash is supplied (either as a reference or as a list) it should contain one or more elements with the following keys:

  – `-message`

  – `-msg`

    The text of a message to print immediately prior to printing the program's usage message.

  – `-exitval`

    The desired exit status to pass to the **exit()** function. This should be an integer, or else the string "NOEXIT" to indicate that control should simply be returned without terminating the invoking process.

  – `-output`

    A reference to a filehandle, or the pathname of a file to which the usage message should be written. The default is `\*STDERR` unless the exit value is less than 2 (in which case the default is `\*STDOUT`).

  You cannot tie this routine directly to an option, e.g.:

```
GetOptions("version" => \&VersionMessage);
```

  Use this instead:

```
GetOptions("version" => sub { VersionMessage() });
```

- HelpMessage

  This subroutine produces a standard help message, derived from the program's POD section SYNOPSIS using *Pod::Usage*. It takes the same arguments as VersionMessage(). In particular, you cannot tie it directly to an option, e.g.:

```
GetOptions("help" => \&HelpMessage);
```

  Use this instead:

```
GetOptions("help" => sub { HelpMessage() });
```

Return values and Errors

Configuration errors and errors in the option definitions are signalled using die() and will terminate the calling program unless the call to Getopt::Long::GetOptions() was embedded in `eval { ... }`, or die() was trapped using `$SIG{__DIE__}`.

GetOptions returns true to indicate success. It returns false when the function detected one or more errors during option parsing. These errors are signalled using warn() and can be trapped with `$SIG{__WARN__}`.

Legacy

The earliest development of `newgetopt.pl` started in 1990, with Perl version 4. As a result, its development, and the development of Getopt::Long, has gone through several stages. Since backward compatibility has always been extremely important, the current version of Getopt::Long still supports a lot of constructs that nowadays are no longer necessary or otherwise unwanted. This section describes briefly some of these 'features'.

Default destinations

When no destination is specified for an option, GetOptions will store the resultant value in a global variable named `opt_`*XXX*, where *XXX* is the primary name of this option. When a program executes under `use strict` (recommended), these variables must be pre-declared with our().

```
our $opt_length = 0;
GetOptions ('length=i');    # will store in $opt_length
```

To yield a usable Perl variable, characters that are not part of the syntax for variables are translated to underscores. For example, `--fpp-struct-return` will set the variable `$opt_fpp_struct_return`. Note that this variable resides in the namespace of the calling program, not necessarily `main`. For example:

```
GetOptions ("size=i", "sizes=i@");
```

with command line "-size 10 -sizes 24 -sizes 48" will perform the equivalent of the assignments

```
$opt_size = 10;
@opt_sizes = (24, 48);
```

Alternative option starters

A string of alternative option starter characters may be passed as the first argument (or the first argument after a leading hash reference argument).

```
my $len = 0;
GetOptions ('/', 'length=i' => $len);
```

Now the command line may look like:

```
/length 24 -- arg
```

Note that to terminate options processing still requires a double dash `--`.

GetOptions() will not interpret a leading `"<>"` as option starters if the next argument is a reference. To force `"<"` and `">"` as option starters, use `"><"`. Confusing? Well, **using a starter argument is strongly deprecated** anyway.

Configuration variables

Previous versions of Getopt::Long used variables for the purpose of configuring. Although manipulating these variables still work, it is strongly encouraged to use the `Configure` routine that was introduced in version 2.17. Besides, it is much easier.

Tips and Techniques

Pushing multiple values in a hash option

Sometimes you want to combine the best of hashes and arrays. For example, the command line:

```
--list add=first --list add=second --list add=third
```

where each successive 'list add' option will push the value of add into array ref $list->{'add'}. The result would be like

```
$list->{add} = [qw(first second third)];
```

This can be accomplished with a destination routine:

```
GetOptions('list=s%' =>
            sub { push(@{$list{$_[1]}}, $_[2]) });
```

Troubleshooting

GetOptions does not return a false result when an option is not supplied

That's why they're called 'options'.

GetOptions does not split the command line correctly

The command line is not split by GetOptions, but by the command line interpreter (CLI). On Unix, this is the shell. On Windows, it is COMMAND.COM or CMD.EXE. Other operating systems have other CLIs.

It is important to know that these CLIs may behave different when the command line contains special characters, in particular quotes or backslashes. For example, with Unix shells you can use single quotes (') and double quotes (") to group words together. The following alternatives are equivalent on Unix:

```
"two words"
'two words'
two\ words
```

In case of doubt, insert the following statement in front of your Perl program:

```
print STDERR (join("|",@ARGV),"\n");
```

to verify how your CLI passes the arguments to the program.

Undefined subroutine &main::GetOptions called

Are you running Windows, and did you write

```
use GetOpt::Long;
```

(note the capital 'O')?

How do I put a "-?" option into a Getopt::Long?

You can only obtain this using an alias, and Getopt::Long of at least version 2.13.

```
use Getopt::Long;
GetOptions ("help|?");    # -help and -? will both set $opt_help
```

Other characters that can't appear in Perl identifiers are also supported in aliases with Getopt::Long of at version 2.39. Note that the characters `!`, `|`, `+`, `=`, and `:` can only appear as the first (or only) character of an alias.

As of version 2.32 Getopt::Long provides auto-help, a quick and easy way to add the options –help and -? to your program, and handle them.

See `auto_help` in section "Configuring Getopt::Long".

AUTHOR

Johan Vromans jvromans@squirrel.nl

COPYRIGHT AND DISCLAIMER

This program is Copyright 1990,2015,2023 by Johan Vromans. This program is free software; you can redistribute it and/or modify it under the terms of the Perl Artistic License or the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

If you do not have a copy of the GNU General Public License write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Modules

- `Getopt::Long::Parser` - Getopt::Long object-oriented interface

Getopt::Long::Parser

Getopt::Long object-oriented interface

SYNOPSIS

```
use Getopt::Long::Parser;
my $p = Getopt::Long::Parser->new;
$p->configure( %options );
if ( $p->getoptions( @options ) ) { ... }
if ( $p->getoptionsfromarray( \@array, @options ) ) { ... }
```

Configuration options can be passed to the constructor:

```
my $p = Getopt::Long::Parser->new( config => [ %options ] );
```

DESCRIPTION

`Getopt::Long::Parser` is an object-oriented interface to *Getopt::Long*. See its documentation for configuration and use.

Note that `Getopt::Long` and `Getopt::Long::Parser` are not object-oriented. `Getopt::Long::Parser` emulates an object-oriented interface, which should be okay for most purposes.

CONSTRUCTOR

```
my $p = Getopt::Long::Parser->new( %options );
```

The constructor takes an optional hash with parameters.

- config

  An array reference with configuration settings. See *"Configuring Getopt::Long" in Getopt::Long* for all possible settings.

METHODS

In the examples, $p is assumed to be the result of a call to the constructor.

configure

```
$p->configure( %settings );
```

Update the current config settings. See *"Configuring Getopt::Long" in Getopt::Long* for all possible settings.

getoptionsfromarray

```
my $res = $p->getoptionsfromarray( $aref, @opts );
```

getoptions

```
my $res = $p->getoptions( @opts );
```

The same as `getoptionsfromarray( \@ARGV, @opts )`.

SEE ALSO

Getopt::Long

AUTHOR

Johan Vromans jvromans@squirrel.nl

COPYRIGHT AND DISCLAIMER

This program is Copyright 1990,2015,2023 by Johan Vromans. This program is free software; you can redistribute it and/or modify it under the same terms as Perl.

Getopt::Std

Process single-character switches with switch clustering

SYNOPSIS

```
use Getopt::Std;

getopts('oif:');  # -o & -i are boolean flags, -f takes an argument
                  # Sets $opt_* global variables as a side effect
getopts('oif:', \my %opts);  # Options as above, values in %opts
getopt('oDI');    # -o, -D & -I take arguments
                  # Sets $opt_* global variables as a side effect
getopt('oDI', \my %opts);    # -o, -D & -I take arg, values in %opts
```

DESCRIPTION

The `getopts()` function processes single-character switches with switch clustering. Pass one argument which is a string containing all switches to be recognized. For each switch found, if an argument is expected and provided, `getopts()` sets `$opt_x` (where `x` is the switch name) to the value of the argument. If an argument is expected but none is provided, `$opt_x` is set to an undefined value. If a switch does not take an argument, `$opt_x` is set to 1.

Switches which take an argument don't care whether there is a space between the switch and the argument. If unspecified switches are found on the command-line, the user will be warned that an unknown option was given.

The `getopts()` function returns true unless an invalid option was found.

The `getopt()` function is similar, but its argument is a string containing all switches that take an argument. If no argument is provided for a switch, say, `y`, the corresponding `$opt_y` will be set to an undefined value. Unspecified switches are silently accepted. Use of `getopt()` is **not recommended**.

Note that, if your code is running under the recommended `use strict vars` pragma, you will need to declare these package variables with `our`:

```
our($opt_x, $opt_y);
```

For those of you who don't like additional global variables being created, `getopt()` and `getopts()` will also accept a hash reference as an optional second argument. Hash keys will be `x` (where `x` is the switch name) with key values the value of the argument or 1 if no argument is specified.

To allow programs to process arguments that look like switches, but aren't, both functions will stop processing switches when they see the argument `--`. The `--` will be removed from `@ARGV`.

`--help` and `--version`

If `-` is not a recognized switch letter, `getopts()` supports arguments `--help` and `--version`. If `main::HELP_MESSAGE()` and/or `main::VERSION_MESSAGE()` are defined, they are called; the arguments are the output file handle, the name of option-processing package, its version, and the switches string. If the subroutines are not defined, an attempt is made to generate intelligent messages; for best results, define `$main::VERSION`.

If embedded documentation (in pod format, see perlpod) is detected in the script, `--help` will also show how to access the documentation.

Note that due to excessive paranoia, if `$Getopt::Std::STANDARD_HELP_VERSION` isn't true (the default is false), then the messages are printed on `STDERR`, and the processing continues after the messages are printed. This being the opposite of the standard-conforming behaviour, it is strongly recommended to set `$Getopt::Std::STANDARD_HELP_VERSION` to true.

One can change the output file handle of the messages by setting `$Getopt::Std::OUTPUT_HELP_VERSION`. One can print the messages of `--help` (without the `Usage:` line) and `--version` by calling functions `help_mess()` and `version_mess()` with the switches string as an argument.

5.2.52 Guard

Guard - run a piece of code when a scope ends or an object goes away.

```
use Guard;

{
    scope_guard { print "leaving the block\n" };
    ...
}                       # prints here, however the block is left

my $g = guard { unlink $tempfile };
...
undef $g;               # runs now - or whenever the last copy goes
```

`scope_guard BLOCK` runs the block when the enclosing scope is left: at its end, through `return`, `last`, or an exception unwinding past it.

`guard BLOCK` returns an object that runs the block when it is destroyed - when the last reference to it goes away. It can be stored, passed around and kept in a data structure, and the code runs exactly once, when nothing holds it any more. `Guard::cancel($guard)` disarms one: its code will then never run.

The code runs with `$@` saved and restored, so a guard firing while an exception is on its way out does not overwrite it. If the guard's own code dies, the exception is handed to `$Guard::DIED`, which by default `warn`s it; the exception never escapes the guard.

This is a native implementation of the CPAN module Guard 1.023; its `Guard.pm` is used unchanged.

Differences from upstream

One, where perl 5.44 with Guard.xs fails: `Guard::cancel(\1)` - a reference to a plain number or undef - dies "Guard::cancel called on a non-guard object", as for every other non-guard; perl5 reads past the value and crashes some of the time (t/81-xs-native/Guard/300-guard-surface.t).

Functions

Other Functions

scope_guard

`scope_guard BLOCK`: run BLOCK when the enclosing scope is left - at its end, by `return`, `last`/`next`, or an exception passing through. Several in one scope run in reverse order of creation. Exported by default.

guard

`guard BLOCK`: an object that runs BLOCK when it is destroyed, that is when the last reference to it goes away - so the code's lifetime can be tied to a data structure rather than a scope. Exported by default.

cancel

`Guard::cancel($guard)`: disarm a guard object made by `guard` - its code will never run. Anything else dies "Guard::cancel called on a non-guard object". Not exported; it is also the method `$guard->cancel`.

scope_guard

`scope_guard BLOCK`: run BLOCK when the enclosing scope is left - at its end, by `return`, `last`/`next`, or an exception passing through. Several in one scope run in reverse order of creation. Exported by default.

```
{
    my $lock = acquire();
    scope_guard { release($lock) };
    work();                 # release runs even if this dies
}
```

5.2.53 HTML

Namespace containing the following modules:

- `HTML::Entities` - Encode or decode strings with HTML entities
- `HTML::Filter` - Filter HTML text through the parser
- `HTML::HeadParser` - Parse section of a HTML document
- `HTML::LinkExtor` - Extract links from an HTML document
- `HTML::Parser` - Parse HTML into events: start and end tags, text, comments, declarations, processing instructions.
- `HTML::PullParser` - Alternative HTML::Parser interface
- `HTML::Tagset` - data tables useful in parsing HTML
- `HTML::TokeParser` - Alternative HTML::Parser interface

HTML::Entities

Encode or decode strings with HTML entities

SYNOPSIS

```
use HTML::Entities;

$a = "V&aring;re norske tegn b&oslash;r &#230res";
decode_entities($a);
encode_entities($a, "\200-\377");
```

For example, this:

```
$input = "vis-à-vis Beyoncé's naïve\npapier-mâché résumé";
print encode_entities($input), "\n"
```

Prints this out:

```
vis-&agrave;-vis Beyonc&eacute;'s na&iuml;ve
papier-m&acirc;ch&eacute; r&eacute;sum&eacute;
```

DESCRIPTION

This module deals with encoding and decoding of strings with HTML character entities. The module provides the following functions:

- decode_entities($string, ...)

  This routine replaces HTML entities found in the $string with the corresponding Unicode character. Unrecognized entities are left alone.

  If multiple strings are provided as argument they are each decoded separately and the same number of strings are returned.

  If called in void context the arguments are decoded in-place.

  This routine is exported by default.

- _decode_entities($string, \%entity2char)

- _decode_entities($string, \%entity2char, $expand_prefix)

  This will in-place replace HTML entities in $string. The %entity2char hash must be provided. Named entities not found in the %entity2char hash are left alone. Numeric entities are expanded unless their value overflow.

  The keys in %entity2char are the entity names to be expanded and their values are what they should expand into. The values do not have to be single character strings. If a key has ";" as suffix, then occurrences in $string are only expanded if properly terminated with ";". Entities without ";" will be expanded regardless of how they are terminated for compatibility with how common browsers treat entities in the Latin-1 range.

  If $expand_prefix is TRUE then entities without trailing ";" in

  unrecognized name. The longest matching name in %entity2char will be used. This is mainly present for compatibility with an MSIE misfeature.

  ```
  $string = "foo&nbspbar";
  _decode_entities($string, { nb => "@", nbsp => "\xA0" }, 1);
  print $string;  # will print "foo bar"
  ```

  An entity whose value is the string being decoded expands to the value that string had when the call began. This routine does not scan replacement text again, so any entities inside it are left alone.

  Looking up a value can run Perl code, through a tied hash or an overloaded object. If that code modifies the string being decoded, this routine dies.

  This routine is exported by default.

- encode_entities($string)
- encode_entities($string, $unsafe_chars)

  This routine replaces unsafe characters in $string with their entity representation. A second argument can be given to specify which characters to consider unsafe. The unsafe characters is specified using the regular expression character class syntax (what you find within brackets in regular expressions).

  The default set of characters to encode are control chars, high-bit chars, and the <, &, >, ' and " characters. But this, for example, would encode *just* the <, &, >, and " characters:

```
$encoded = encode_entities($input, '<>&"');
```

  and this would only encode non-plain ASCII:

```
$encoded = encode_entities($input, '^\n\x20-\x25\x27-\x7e');
```

  This routine is exported by default.

- encode_entities_numeric($string)
- encode_entities_numeric($string, $unsafe_chars)

  This routine works just like encode_entities, except that the replacement entities are always &#x_hexnum_; and never &_entname_;. For example, `encode_entities("r\xF4le")` returns “rôle”, but `encode_entities_numeric("r\xF4le")` returns “rôle”.

  This routine is *not* exported by default. But you can always export it with `use HTML::Entities qw(encode_entities_numeric);` or even `use HTML::Entities qw(:DEFAULT encode_entities_numeric);`

All these routines modify the string passed as the first argument, if called in a void context. In scalar and array contexts, the encoded or decoded string is returned (without changing the input string).

If you prefer not to import these routines into your namespace, you can call them as:

```
use HTML::Entities ();
$decoded = HTML::Entities::decode($a);
$encoded = HTML::Entities::encode($a);
$encoded = HTML::Entities::encode_numeric($a);
```

The module can also export the %char2entity and the %entity2char hashes, which contain the mapping from all characters to the corresponding entities (and vice versa, respectively).

COPYRIGHT

Copyright 1995-2006 Gisle Aas. All rights reserved.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

Functions

Entities

decode_entities

Replace the entities in the strings: in place in void context, else returning decoded copies.

UNICODE_SUPPORT

Always 1: entities decode to Unicode characters.

HTML::Filter

Filter HTML text through the parser

NOTE

This module is deprecated. The `HTML::Parser` now provides the functionally of `HTML::Filter` much more efficiently with the `default` handler.

SYNOPSIS

```
require HTML::Filter;
$p = HTML::Filter->new->parse_file("index.html");
```

DESCRIPTION

`HTML::Filter` is an HTML parser that by default prints the original text of each HTML element (a slow version of cat(1) basically). The callback methods may be overridden to modify the filtering for some HTML elements and you can override output() method which is called to print the HTML text.

`HTML::Filter` is a subclass of `HTML::Parser`. This means that the document should be given to the parser by calling the $p->parse() or $p->parse_file() methods.

EXAMPLES

The first example is a filter that will remove all comments from an HTML file. This is achieved by simply overriding the comment method to do nothing.

```
package CommentStripper;
require HTML::Filter;
@ISA=qw(HTML::Filter);
sub comment { }  # ignore comments
```

The second example shows a filter that will remove any s found in the HTML file. We specialize the start() and end() methods to count table tags and then make output not happen when inside a table.

```
package TableStripper;
require HTML::Filter;
@ISA=qw(HTML::Filter);
sub start
{
   my $self = shift;
   $self->{table_seen}++ if $_[0] eq "table";
   $self->SUPER::start(@_);
}

sub end
{
   my $self = shift;
   $self->SUPER::end(@_);
   $self->{table_seen}-- if $_[0] eq "table";
}

sub output
{
    my $self = shift;
    unless ($self->{table_seen}) {
        $self->SUPER::output(@_);
    }
}
```

If you want to collect the parsed text internally you might want to do something like this:

```
package FilterIntoString;
require HTML::Filter;
@ISA=qw(HTML::Filter);
sub output { push(@{$_[0]->{fhtml}}, $_[1]) }
sub filtered_html { join("", @{$_[0]->{fhtml}}) }
```

SEE ALSO

HTML::Parser

COPYRIGHT

Copyright 1997-1999 Gisle Aas.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

HTML::HeadParser

Parse section of a HTML document

SYNOPSIS

```
require HTML::HeadParser;
$p = HTML::HeadParser->new;
$p->parse($text) and  print "not finished";

$p->header('Title')           # to access <title>....</title>
$p->header('Content-Base')    # to access <base href="http://...">
$p->header('Foo')             # to access <meta http-equiv="Foo"␣
↪content="...">
$p->header('X-Meta-Author')   # to access <meta name="author"␣
↪content="...">
$p->header('X-Meta-Charset') # to access <meta charset="...">
```

DESCRIPTION

The `HTML::HeadParser` is a specialized (and lightweight) `HTML::Parser` that will only parse the ...</HEAD> section of an HTML document. The parse() method will return a FALSE value as soon as some element or body text are found, and should not be called again after this.

Note that the `HTML::HeadParser` might get confused if raw undecoded UTF-8 is passed to the parse() method. Make sure the strings are properly decoded before passing them on.

The `HTML::HeadParser` keeps a reference to a header object, and the parser will update this header object as the various elements of the

- Content-Base:

  The *Content-Base* header is initialized from the <base href="..."> element.

- Title:

  The *Title* header is initialized from the <title>...</title> element.

- Isindex:

  The *Isindex* header will be added if there is a <isindex> element in the <head>. The header value is initialized from the *prompt* attribute if it is present. If no *prompt* attribute is given it will have '?' as the value.

- X-Meta-Foo:

  All <meta> elements containing a `name` attribute will result in headers using the prefix `X-Meta-` appended with the value of the `name` attribute as the name of the header, and the value of the `content` attribute as the pushed header value.

  <meta> elements containing a `http-equiv` attribute will result in headers as in above, but without the `X-Meta-` prefix in the header name.

  <meta> elements containing a `charset` attribute will result in an `X-Meta-Charset` header, using the value of the `charset` attribute as the pushed header value.

  The ':' character can't be represented in header field names, so if the meta element contains this char it's substituted with '-' before forming the field name.

METHODS

The following methods (in addition to those provided by the superclass) are available:

- $hp = HTML::HeadParser->new
- $hp = HTML::HeadParser->new($header)

  The object constructor. The optional $header argument should be a reference to an object that implement the header() and push_header() methods as defined by the `HTTP::Headers` class. Normally it will be of some class that is a or delegates to the `HTTP::Headers` class.

  If no $header is given `HTML::HeadParser` will create an `HTTP::Headers` object by itself (initially empty).

- $hp->header;

  Returns a reference to the header object.

- $hp->header($key)

  Returns a header value. It is just a shorter way to write `$hp->header->header($key)`.

EXAMPLE

```
$h = HTTP::Headers->new;
$p = HTML::HeadParser->new($h);
$p->parse(<<EOT);
<title>Stupid example</title>
<base href="http://www.linpro.no/lwp/">
Normal text starts here.
EOT
undef $p;
print $h->title;   # should print "Stupid example"
```

SEE ALSO

HTML::Parser, HTTP::Headers

The `HTTP::Headers` class is distributed as part of the *libwww-perl* package. If you don't have that distribution installed you need to provide the $header argument to the `HTML::HeadParser` constructor with your own object that implements the documented protocol.

COPYRIGHT

Copyright 1996-2001 Gisle Aas. All rights reserved.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

HTML::LinkExtor

Extract links from an HTML document

SYNOPSIS

```
require HTML::LinkExtor;
$p = HTML::LinkExtor->new(\&cb, "http://www.perl.org/");
sub cb {
    my($tag, %links) = @_;
    print "$tag @{[%links]}\n";
}
$p->parse_file("index.html");
```

DESCRIPTION

HTML::LinkExtor is an HTML parser that extracts links from an HTML document. The *HTML::LinkExtor* is a subclass of *HTML::Parser*. This means that the document should be given to the parser by calling the $p->parse() or $p->parse_file() methods.

- $p = HTML::LinkExtor->new
- $p = HTML::LinkExtor->new($callback)
- $p = HTML::LinkExtor->new($callback, $base)

  The constructor takes two optional arguments. The first is a reference to a callback routine. It will be called as links are found. If a callback is not provided, then links are just accumulated internally and can be retrieved by calling the $p->links() method.

  The $base argument is an optional base URL used to absolutize all URLs found. You need to have the *URI* module installed if you provide $base.

  The callback is called with the lowercase tag name as first argument, and then all link attributes as separate key/value pairs. All non-link attributes are removed.

- $p->links

  Returns a list of all links found in the document. The returned values will be anonymous arrays with the following elements:

  ```
  [$tag, $attr => $url1, $attr2 => $url2,...]
  ```

  The $p->links method will also truncate the internal link list. This means that if the method is called twice without any parsing between them the second call will return an empty list.

  Also note that $p->links will always be empty if a callback routine was provided when the *HTML::LinkExtor* was created.

EXAMPLE

This is an example showing how you can extract links from a document received using LWP:

```
use LWP::UserAgent;
use HTML::LinkExtor;
use URI::URL;

$url = "http://www.perl.org/";  # for instance
$ua = LWP::UserAgent->new;

# Set up a callback that collect image links
my @imgs = ();
sub callback {
   my($tag, %attr) = @_;
   return if $tag ne 'img';  # we only look closer at <img ...>
   push(@imgs, values %attr);
}

# Make the parser.  Unfortunately, we don't know the base yet
# (it might be different from $url)
$p = HTML::LinkExtor->new(\&callback);

# Request document and parse it as it arrives
$res = $ua->request(HTTP::Request->new(GET => $url),
                    sub {$p->parse($_[0])});

# Expand all image URLs to absolute ones
my $base = $res->base;
@imgs = map { $_ = url($_, $base)->abs; } @imgs;

# Print them out
print join("\n", @imgs), "\n";
```

SEE ALSO

HTML::Parser, *HTML::Tagset*, LWP, URI::URL

COPYRIGHT

Copyright 1996-2001 Gisle Aas.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

HTML::Parser

Parse HTML into events: start and end tags, text, comments, declarations, processing instructions.

`HTML::Parser` reads HTML the way browsers tolerate it - unquoted and unclosed attributes, stray <, comments that end at the first > in a damaged document - and reports what it finds to handlers you register, with exactly the pieces you ask for (`'tagname, attr'`, `'dtext'`, `'offset, length'`, ...). It parses incrementally: feed the document in chunks as it arrives, then `eof`. `HTML::Entities` decodes and encodes character entities, and `HTML::TokeParser`, `HTML::PullParser`, `HTML::LinkExtor`, `HTML::HeadParser` and `HTML::Filter` are built on the parser.

```
use HTML::Parser;
binmode STDOUT, ':utf8';

my $p = HTML::Parser->new(
    api_version => 3,
    start_h => [sub { print "start $_[0] @{[ %{$_[1]} ]}\n" },
↪'tagname, attr'],
    text_h  => [sub { print "text [$_[0]]\n" if $_[0] =~ /\S/ },
↪'dtext'],
);
$p->parse('<p class=x>caf&eacute; &amp; <b>bar</b></p>');
$p->eof;
# start p class x
# text [café & ]
# start b
# text [bar]
```

This is a native implementation of the XS half of HTML-Parser 3.86 (`hparser.c`, `util.c`, `Parser.xs`, with marked sections enabled as upstream builds them); its `HTML/Parser.pm`, `HTML/Entities.pm` and the other modules of the dist are used unchanged.

Differences from upstream

None in behaviour.

Functions

Parsing

parse

Parse a chunk of the document, or chunks from a code reference until it returns nothing. Returns the parser, or undef when a handler called `eof` meanwhile.

eof

End the document: flush what is buffered and report the end. Called from a handler, parsing stops after the current event.

Configuration

strict_comment

Get, and with an argument set, one of the boolean options: strict_comment, strict_names, xml_mode, unbroken_text, marked_sections, attr_encoded, case_sensitive, strict_end, closing_plaintext, utf8_mode, empty_element_tags, xml_pic, backquote.

Also available as HTML::Parser::strict_names, HTML::Parser::xml_mode, HTML::Parser::unbroken_text, HTML::Parser::marked_sections, HTML::Parser::attr_encoded, HTML::Parser::case_sensitive, HTML::Parser::strict_end, HTML::Parser::closing_plaintext, HTML::Parser::utf8_mode, HTML::Parser::empty_element_tags, HTML::Parser::xml_pic, HTML::Parser::backquote.

boolean_attribute_value

Get, and with an argument set, the value a boolean attribute (one written without =value) gets; by default its name.

report_tags

Set the tags whose events are reported (report_tags), those whose tags are ignored (ignore_tags), or those whose whole elements are (ignore_elements); no list clears it.

Also available as HTML::Parser::ignore_tags, HTML::Parser::ignore_elements.

Handlers

handler

Get the handler of an event; with more arguments set it (a code reference, a method name or an array to accumulate into) and the argspec naming what it is passed.

HTML::PullParser

Alternative HTML::Parser interface

SYNOPSIS

```
use HTML::PullParser;

$p = HTML::PullParser->new(file => "index.html",
                           start => 'event, tagname, @attr',
                           end   => 'event, tagname',
                           ignore_elements => [qw(script style)],
                          ) || die "Can't open: $!";
while (my $token = $p->get_token) {
    #...do something with $token
}
```

DESCRIPTION

The HTML::PullParser is an alternative interface to the HTML::Parser class. It basically turns the HTML::Parser inside out. You associate a file (or any IO::Handle object or string) with the parser at construction time and then repeatedly call $parser->get_token to obtain the tags and text found in the parsed document.

The following methods are provided:

- $p = HTML::PullParser->new(file => $file, %options)
- $p = HTML::PullParser->new(doc => \$doc, %options)

  A `HTML::PullParser` can be made to parse from either a file or a literal document based on whether the `file` or `doc` option is passed to the parser's constructor.

  The `file` passed in can either be a file name or a file handle object. If a file name is passed, and it can't be opened for reading, then the constructor will return an undefined value and `$!` will tell you why it failed. Otherwise the argument is taken to be some object that the `HTML::PullParser` can read() from when it needs more data. The stream will be read() until EOF, but not closed.

  A `doc` can be passed plain or as a reference to a scalar. If a reference is passed then the value of this scalar should not be changed before all tokens have been extracted.

  Next the information to be returned for the different token types must be set up. This is done by simply associating an argspec (as defined in *HTML::Parser*) with the events you have an interest in. For instance, if you want `start` tokens to be reported as the string `'S'` followed by the tagname and the attributes you might pass an `start`-option like this:

```
$p = HTML::PullParser->new(
       doc   => $document_to_parse,
       start => '"S", tagname, @attr',
       end   => '"E", tagname',
     );
```

  At last other `HTML::Parser` options, like `ignore_tags`, and `unbroken_text`, can be passed in. Note that you should not use the *event*_h options to set up parser handlers. That would confuse the inner logic of `HTML::PullParser`.

- $token = $p->get_token

  This method will return the next *token* found in the HTML document, or `undef` at the end of the document. The token is returned as an array reference. The content of this array match the argspec set up during `HTML::PullParser` construction.

- $p->unget_token(@tokens)

  If you find out you have read too many tokens you can push them back, so that they are returned again the next time $p->get_token is called.

EXAMPLES

The ‘eg/hform’ script shows how we might parse the form section of HTML::Documents using HTML::PullParser.

SEE ALSO

HTML::Parser, *HTML::TokeParser*

COPYRIGHT

Copyright 1998-2001 Gisle Aas.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

HTML::Tagset

data tables useful in parsing HTML

VERSION

Version 3.24

SYNOPSIS

```
use HTML::Tagset;
# Then use any of the items in the HTML::Tagset package
#  as need arises
```

DESCRIPTION

This module contains several data tables useful in various kinds of HTML parsing operations.

Note that all tag names used are lowercase.

In the following documentation, a “hashset” is a hash being used as a set – the hash conveys that its keys are there, and the actual values associated with the keys are not significant. (But what values are there, are always true.)

VARIABLES

Note that none of these variables are exported.

hashset %HTML::Tagset::emptyElement

This hashset has as values the tag-names (GIs) of elements that cannot have content. (For example, “base”, “br”, “hr”.) So `$HTML::Tagset::emptyElement{'hr'}` exists and is true. `$HTML::Tagset::emptyElement{'dl'}` does not exist, and so is not true.

hashset %HTML::Tagset::optionalEndTag

This hashset lists tag-names for elements that can have content, but whose end-tags are generally, "safely", omissible. Example: `$HTML::Tagset::emptyElement{'li'}` exists and is true.

hash %HTML::Tagset::linkElements

Values in this hash are tagnames for elements that might contain links, and the value for each is a reference to an array of the names of attributes whose values can be links.

hash %HTML::Tagset::boolean_attr

This hash (not hashset) lists what attributes of what elements can be printed without showing the value (for example, the "noshade" attribute of "hr" elements). For elements with only one such attribute, its value is simply that attribute name. For elements with many such attributes, the value is a reference to a hashset containing all such attributes.

hashset %HTML::Tagset::isPhraseMarkup

This hashset contains all phrasal-level elements.

hashset %HTML::Tagset::is_Possible_Strict_P_Content

This hashset contains all phrasal-level elements that be content of a P element, for a strict model of HTML.

hashset %HTML::Tagset::isHeadElement

This hashset contains all elements that elements that should be present only in the 'head' element of an HTML document.

hashset %HTML::Tagset::isList

This hashset contains all elements that can contain "li" elements.

hashset %HTML::Tagset::isTableElement

This hashset contains all elements that are to be found only in/under a "table" element.

hashset %HTML::Tagset::isFormElement

This hashset contains all elements that are to be found only in/under a "form" element.

hashset %HTML::Tagset::isBodyElement

This hashset contains all elements that are to be found only in/under the "body" element of an HTML document.

hashset %HTML::Tagset::isHeadOrBodyElement

This hashset includes all elements that I notice can fall either in the head or in the body.

hashset %HTML::Tagset::isKnown

This hashset lists all known HTML elements.

hashset %HTML::Tagset::canTighten

This hashset lists elements that might have ignorable whitespace as children or siblings.

array @HTML::Tagset::p_closure_barriers

This array has a meaning that I have only seen a need for in `HTML::TreeBuilder`, but I include it here on the off chance that someone might find it of use:

When we see a "<p>" token, we go lookup up the lineage for a p element we might have to minimize. At first sight, we might say that if there's a p anywhere in the lineage of this new p, it should be closed. But that's wrong. Consider this document:

```
<html>
  <head>
    <title>foo</title>
  </head>
  <body>
    <p>foo
      <table>
        <tr>
          <td>
             foo
             <p>bar
          </td>
        </tr>
      </table>
    </p>
  </body>
</html>
```

The second p is quite legally inside a much higher p.

My formalization of the reason why this is legal, but this:

```
<p>foo<p>bar</p></p>
```

isn't, is that something about the table constitutes a "barrier" to the application of the rule about what p must minimize.

So `@HTML::Tagset::p_closure_barriers` is the list of all such barrier-tags.

hashset %isCDATA_Parent

This hashset includes all elements whose content is CDATA.

CAVEATS

You may find it useful to alter the behavior of modules (like `HTML::Element` or `HTML::TreeBuilder`) that use `HTML::Tagset`'s data tables by altering the data tables themselves. You are welcome to try, but be careful; and be aware that different modules may or may react differently to the data tables being changed.

Note that it may be inappropriate to use these tables for *producing* HTML – for example, `%isHeadOrBodyElement` lists the tagnames for all elements that can appear either in the head or in the body, such as "script". That doesn't mean that I am saying your code that produces HTML should feel free to put script elements in either place! If you are producing programs that spit out HTML, you should be *intimately* familiar with the DTDs for HTML or XHTML (available at `http://www.w3.org/`), and you should slavishly obey them, not the data tables in this document.

SEE ALSO

HTML::Element, HTML::TreeBuilder, *HTML::LinkExtor*

COPYRIGHT & LICENSE

Copyright 1995-2000 Gisle Aas.

Copyright 2000-2005 Sean M. Burke.

Copyright 2005-2024 Andy Lester.

This library is free software; you can redistribute it and/or modify it under the terms of the Artistic License version 2.0.

ACKNOWLEDGEMENTS

Most of the code/data in this module was adapted from code written by Gisle Aas for `HTML::Element`, `HTML::TreeBuilder`, and `HTML::LinkExtor`. Then it was maintained by Sean M. Burke.

AUTHOR

Current maintainer: Andy Lester, `<andy at petdance.com>`

BUGS

Please report any bugs or feature requests to `bug-html-tagset at rt.cpan.org`, or through the web interface at http://rt.cpan.org/NoAuth/ReportBug.html?Queue=HTML-Tagset. I will be notified, and then you'll automatically be notified of progress on your bug as I make changes.

HTML::TokeParser

Alternative HTML::Parser interface

SYNOPSIS

```
require HTML::TokeParser;
$p = HTML::TokeParser->new("index.html") ||
     die "Can't open: $!";
$p->empty_element_tags(1);  # configure its behaviour

while (my $token = $p->get_token) {
    #...
}
```

DESCRIPTION

The `HTML::TokeParser` is an alternative interface to the `HTML::Parser` class. It is an `HTML::PullParser` subclass with a predeclared set of token types. If you wish the tokens to be reported differently you probably want to use the `HTML::PullParser` directly.

The following methods are available:

- $p = HTML::TokeParser->new($filename, %opt);
- $p = HTML::TokeParser->new($filehandle, %opt);
- $p = HTML::TokeParser->new(\$document, %opt);

  The object constructor argument is either a file name, a file handle object, or the complete document to be parsed. Extra options can be provided as key/value pairs and are processed as documented by the base classes.

  If the argument is a plain scalar, then it is taken as the name of a file to be opened and parsed. If the file can't be opened for reading, then the constructor will return `undef` and `$!` will tell you why it failed.

  If the argument is a reference to a plain scalar, then this scalar is taken to be the literal document to parse. The value of this scalar should not be changed before all tokens have been extracted.

  Otherwise the argument is taken to be some object that the `HTML::TokeParser` can read() from when it needs more data. Typically it will be a filehandle of some kind. The stream will be read() until EOF, but not closed.

  A newly constructed `HTML::TokeParser` differ from its base classes by having the `unbroken_text` attribute enabled by default. See *HTML::Parser* for a description of this and other attributes that influence how the document is parsed. It is often a good idea to enable `empty_element_tags` behaviour.

  Note that the parsing result will likely not be valid if raw undecoded UTF-8 is used as a source. When parsing UTF-8 encoded files turn on UTF-8 decoding:

```
open(my $fh, "<:utf8", "index.html") || die "Can't open 'index.
 ↪html': $!";
my $p = HTML::TokeParser->new( $fh );
# ...
```

If a $filename is passed to the constructor the file will be opened in raw mode and the parsing result will only be valid if its content is Latin-1 or pure ASCII.

If parsing from an UTF-8 encoded string buffer decode it first:

```
utf8::decode($document);
my $p = HTML::TokeParser->new( \$document );
# ...
```

- $p->get_token

  This method will return the next *token* found in the HTML document, or `undef` at the end of the document. The token is returned as an array reference. The first element of the array will be a string denoting the type of this token: "S" for start tag, "E" for end tag, "T" for text, "C" for comment, "D" for declaration, and "PI" for process instructions. The rest of the token array depend on the type like this:

```
["S",  $tag, $attr, $attrseq, $text]
["E",  $tag, $text]
["T",  $text, $is_data]
["C",  $text]
["D",  $text]
["PI", $token0, $text]
```

  where $attr is a hash reference, $attrseq is an array reference and the rest are plain scalars. The *"Argspec" in HTML::Parser* explains the details.

- $p->unget_token(@tokens)

  If you find you have read too many tokens you can push them back, so that they are returned the next time $p->get_token is called.

- $p->get_tag

- $p->get_tag(@tags)

  This method returns the next start or end tag (skipping any other tokens), or `undef` if there are no more tags in the document. If one or more arguments are given, then we skip tokens until one of the specified tag types is found. For example:

```
$p->get_tag("font", "/font");
```

  will find the next start or end tag for a font-element.

  The tag information is returned as an array reference in the same form as for $p->get_token above, but the type code (first element) is missing. A start tag will be returned like this:

```
[$tag, $attr, $attrseq, $text]
```

  The tagname of end tags are prefixed with "/", i.e. end tag is returned like this:

```
["/$tag", $text]
```

- $p->get_text
- $p->get_text(@endtags)

  This method returns all text found at the current position. It will return a zero length string if the next token is not text. Any entities will be converted to their corresponding character.

  If one or more arguments are given, then we return all text occurring before the first of the specified tags found. For example:

```
$p->get_text("p", "br");
```

  will return the text up to either a paragraph of line break element.

  The text might span tags that should be *textified*. This is controlled by the $p->{textify} attribute, which is a hash that defines how certain tags can be treated as text. If the name of a start tag matches a key in this hash then this tag is converted to text. The hash value is used to specify which tag attribute to obtain the text from. If this tag attribute is missing, then the upper case name of the tag enclosed in brackets is returned, e.g. “[IMG]”. The hash value can also be a subroutine reference. In this case the routine is called with the start tag token content as its argument and the return value is treated as the text.

  The default $p->{textify} value is:

```
{img => "alt", applet => "alt"}
```

  This means that and tags are treated as text, and that the text to substitute can be found in the ALT attribute.

- $p->get_trimmed_text
- $p->get_trimmed_text(@endtags)

  Same as $p->get_text above, but will collapse any sequences of white space to a single space character. Leading and trailing white space is removed.

- $p->get_phrase

  This will return all text found at the current position ignoring any phrasal-level tags. Text is extracted until the first non phrasal-level tag. Textification of tags is the same as for get_text(). This method will collapse white space in the same way as get_trimmed_text() does.

  The definition of phrasal-level tags is obtained from the HTML::Tagset module.

EXAMPLES

This example extracts all links from a document. It will print one line for each link, containing the URL and the textual description between the ...</A> tags:

```
use HTML::TokeParser;
$p = HTML::TokeParser->new(shift||"index.html");

while (my $token = $p->get_tag("a")) {
    my $url = $token->[1]{href} || "-";
    my $text = $p->get_trimmed_text("/a");
    print "$url\t$text\n";
}
```

This example extract the from the document:

```
use HTML::TokeParser;
$p = HTML::TokeParser->new(shift||"index.html");
if ($p->get_tag("title")) {
    my $title = $p->get_trimmed_text;
    print "Title: $title\n";
}
```

SEE ALSO

HTML::PullParser, *HTML::Parser*

COPYRIGHT

Copyright 1998-2005 Gisle Aas.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

5.2.54 HTTP

Namespace containing the following modules:

- `HTTP::Parser`
- `HTTP::Tiny` - A small, simple, correct HTTP/1.1 client

HTTP::Parser

Namespace containing the following modules:

- `HTTP::Parser::XS` - Parse an HTTP request or response head out of a byte buffer - the parser Plack, Starman and Twiggy sit on.

HTTP::Parser::XS

Parse an HTTP request or response head out of a byte buffer - the parser Plack, Starman and Twiggy sit on.

Two entry points, both non-blocking and both operating on a buffer the caller owns:

- `parse_http_request($buf, \%env)` fills `%env` with a PSGI environment and returns the number of bytes the request head occupies, `-2` if the head is not complete yet, `-1` if it is malformed. A server calls it again with more bytes on `-2`.
- `parse_http_response($buf, $format, \%special)` returns `($ret, $minor_version, $status, $message, $headers)` for a client.

```
use HTTP::Parser::XS qw(parse_http_request parse_http_response␣
↪HEADERS_AS_HASHREF);

my $req = "GET /search?q=perl%20glob HTTP/1.1\r\nHost: example.com\
↪r\n"
        . "X-Trace: 1\r\nX-Trace: 2\r\n\r\nbody";
my %env;
my $len = parse_http_request($req, \%env);
print "$len bytes of head; $env{REQUEST_METHOD} $env{PATH_INFO} ?
↪$env{QUERY_STRING}\n";
# 81 bytes of head; GET /search ? q=perl%20glob
print "host $env{HTTP_HOST}, trace '$env{HTTP_X_TRACE}'\n";   # host␣
↪example.com, trace '1, 2'
print parse_http_request("GET / HTTP/1.1\r\nHost:", {}), "\n";  # -
↪2: not complete yet

my ($ret, $minor, $status, $msg, $headers) =
    parse_http_response("HTTP/1.0 404 Not Found\r\nContent-Length:␣
↪0\r\n\r\n", HEADERS_AS_HASHREF);
print "$ret: HTTP/1.$minor $status $msg, length $headers->{'content-
↪length'}\n";
# 45: HTTP/1.0 404 Not Found, length 0
```

`$format` is `HEADERS_AS_HASHREF` (a hash of lower-cased names; a repeated header's values in an array reference), `HEADERS_AS_ARRAYREF` (name, value pairs in order) or `HEADERS_NONE`. This is a native implementation of HTTP::Parser::XS 0.17; its `XS.pm` is used unchanged, and nothing is exported by default.

The PSGI environment

`parse_http_request` writes `REQUEST_METHOD`, `REQUEST_URI`, `SCRIPT_NAME` (always `""`), `PATH_INFO`, `QUERY_STRING` and `SERVER_PROTOCOL`, then one key per header. Header names are upper-cased, `-` becomes `_`, and everything except `Content-Type` and `Content-Length` is prefixed `HTTP_`; those two land as `CONTENT_TYPE` and `CONTENT_LENGTH` with no prefix, as CGI and PSGI require. A header sent twice is joined with `", "`.

`REQUEST_URI` keeps the request target verbatim. `PATH_INFO` is the part before `?` with `%XX` escapes decoded, `QUERY_STRING` the part after; a `#` fragment and everything after it is dropped from both but not from `REQUEST_URI`.

Limits

At most 128 headers, and a header name of at most 1024 bytes. The two parsers differ on what happens past the name limit: the request parser fails the whole request (`-1`, and %env is cleared), the response parser silently skips the offending header and carries on.

$BACKEND

`'xs'`, set by XS.pm once XSLoader::load has loaded this native code. `PERL_ONLY=1` or `PERL_HTTP_PARSER_XS=pp` make XS.pm load the distribution's pure-Perl `HTTP::Parser::XS::PP` instead, and then it is `'pp'`.

Vectorised tokenizer

The byte scanners run AVX2 (32B/iteration) or SSE2 (16B) blocks before falling back to picohttpparser's own eight-way unrolled loop and then a byte tail. Upstream drives PCMPESTRI instead, but only under `-msse4.2`, which its Makefile.PL never passes - the shipped `.so` has no vector instruction in it at all - so every CPAN install runs the scalar loop.

Speed relative to the reference XS on a Skylake Xeon E3-1505M (`--release`, processes interleaved, min of 5 - sequential runs drift enough between processes to invent a 10% result):

workload	speedup vs perl5 XS (>1 = pperl faster)
64 KB header value, `HEADERS_NONE`	6.01x
64 KB header value	4.23x
4 KB header value	2.15x
1 KB header name	1.95x
4 KB request target	1.25x
realistic 10-header GET	**1.00x**
response, `HEADERS_AS_HASHREF`	0.88x

The long-value rows are where the vector scan pays. A realistic request is not bound by scanning: most of its cost is creating the hash entry and the string for each PSGI key, so there the two run at the same speed.

`PPERL_HTTP_PARSER_SIMD=avx2|sse2|scalar` forces a kernel, for example to compare them on one machine; all three give identical results.

Differences from upstream

None in what is parsed; only the speed differs.

Functions

Request parsing

parse_http_request

`parse_http_request($buf, \%env)` - XS.xs `parse_http_request`.

Response parsing

parse_http_response

`parse_http_response($buf, $header_format, \%special_headers)` - XS.xs parse_http_response.

HTTP::Tiny

A small, simple, correct HTTP/1.1 client

VERSION

version 0.096

SYNOPSIS

```
use HTTP::Tiny;

my $response = HTTP::Tiny->new->get('http://example.com/');

die "Failed!\n" unless $response->{success};

print "$response->{status} $response->{reason}\n";

while (my ($k, $v) = each %{$response->{headers}}) {
    for (ref $v eq 'ARRAY' ? @$v : $v) {
        print "$k: $_\n";
    }
}

print $response->{content} if length $response->{content};
```

DESCRIPTION

This is a very simple HTTP/1.1 client, designed for doing simple requests without the overhead of a large framework like LWP::UserAgent.

It is more correct and more complete than HTTP::Lite. It supports proxies and redirection. It also correctly resumes after EINTR.

If *IO::Socket::IP* 0.25 or later is installed, HTTP::Tiny will use it instead of *IO::Socket::INET* for transparent support for both IPv4 and IPv6.

Cookie support requires HTTP::CookieJar or an equivalent class.

METHODS

new

```
$http = HTTP::Tiny->new( %attributes );
```

This constructor returns a new HTTP::Tiny object. Valid attributes include:

- `agent` — A user-agent string (defaults to 'HTTP-Tiny/$VERSION'). If `agent` — ends in a space character, the default user-agent string is appended.
- `allow_credentialed_redirects` - If a `3XX` redirects to a different scheme, host or port, by default HTTP::Tiny will strip away caller-supplied `Authorization`, `Cookie` and `Proxy-Authorization` headers from the redirected request and from all subsequent requests in the chain. Set this to a true value to revert to the legacy behavior of forwarding those headers. Default is `false`.
- `allow_downgrade` — If a `3XX` redirect changes the scheme from `https` to plain `http`, HTTP::Tiny will by default refuse to follow it, returning the `3XX` response. Set this to a true value to revert to the legacy behavior of redirecting `https` to `http`. Default is `false`.
- `cookie_jar` — An instance of HTTP::CookieJar — or equivalent class that supports the `add` and `cookie_header` methods
- `default_headers` — A hashref of default headers to apply to requests
- `local_address` — The local IP address to bind to
- `keep_alive` — Whether to reuse the last connection (if for the same scheme, host and port) (defaults to 1)
- `keep_alive_timeout` — How many seconds to keep a connection available for after a request (defaults to 0, unlimited)
- `max_redirect` — Maximum number of redirects allowed (defaults to 5)
- `max_size` — Maximum response size in bytes (only when not using a data callback). If defined, requests with responses larger than this will return a 599 status code.
- `http_proxy` — URL of a proxy server to use for HTTP connections (default is `$ENV{http_proxy}` — if set)
- `https_proxy` — URL of a proxy server to use for HTTPS connections (default is `$ENV{https_proxy}` — if set)
- `proxy` — URL of a generic proxy server for both HTTP and HTTPS connections (default is `$ENV{all_proxy}` — if set)
- `no_proxy` — List of domain suffixes that should not be proxied. Must be a comma-separated string or an array reference. (default is `$ENV{no_proxy}` —)
- `timeout` — Request timeout in seconds (default is 60) If a socket open, read or write takes longer than the timeout, the request response status code will be 599.
- `verify_SSL` — A boolean that indicates whether to validate the TLS/SSL certificate of an `https` — connection (default is true). Changed from false to true in version 0.083.
- `SSL_options` — A hashref of `SSL_*` — options to pass through to *IO::Socket::SSL*

- `$ENV{PERL_HTTP_TINY_SSL_INSECURE_BY_DEFAULT}` - Changes the default certificate verification behavior to not check server identity if set to 1. Only effective if `verify_SSL` is not set. Added in version 0.083.

An accessor/mutator method exists for each attribute.

Passing an explicit `undef` for `proxy`, `http_proxy` or `https_proxy` will prevent getting the corresponding proxies from the environment.

Errors during request execution will result in a pseudo-HTTP status code of 599 and a reason of "Internal Exception". The content field in the response will contain the text of the error.

The `keep_alive` parameter enables a persistent connection, but only to a single destination scheme, host and port. If any connection-relevant attributes are modified via accessor, or if the process ID or thread ID change, the persistent connection will be dropped. If you want persistent connections across multiple destinations, use multiple HTTP::Tiny objects.

The `keep_alive_timeout` parameter allows you to control how long a keep alive connection will be considered for reuse. By setting this lower than the server keep alive time, this allows you to avoid race conditions where the server closes the connection while preparing to write the request on a reused persistent connection.

See "TLS/SSL SUPPORT" for more on the `verify_SSL` and `SSL_options` attributes.

get|head|put|post|patch|delete

```
$response = $http->get($url);
$response = $http->get($url, \%options);
$response = $http->head($url);
```

These methods are shorthand for calling `request()` for the given method. The URL must have unsafe characters escaped and international domain names encoded. See `request()` for valid options and a description of the response.

The `success` field of the response will be true if the status code is 2XX.

post_form

```
$response = $http->post_form($url, $form_data);
$response = $http->post_form($url, $form_data, \%options);
```

This method executes a `POST` request and sends the key/value pairs from a form data hash or array reference to the given URL with a `content-type` of `application/x-www-form-urlencoded`. If data is provided as an array reference, the order is preserved; if provided as a hash reference, the terms are sorted by key for consistency. See documentation for the `www_form_urlencode` method for details on the encoding.

The URL must have unsafe characters escaped and international domain names encoded. See `request()` for valid options and a description of the response. Any `content-type` header or content in the options hashref will be ignored.

The `success` field of the response will be true if the status code is 2XX.

mirror

```
$response = $http->mirror($url, $file, \%options)
if ( $response->{success} ) {
    print "$file is up to date\n";
}
```

Executes a `GET` request for the URL and saves the response body to the file name provided. The URL must have unsafe characters escaped and international domain names encoded. If the file already exists, the request will include an `If-Modified-Since` header with the modification timestamp of the file. You may specify a different `If-Modified-Since` header yourself in the `$options->{headers}` hash.

The `success` field of the response will be true if the status code is `2XX` or if the status code is `304` (unmodified).

If the file was modified and the server response includes a properly formatted `Last-Modified` header, the file modification time will be updated accordingly.

request

```
$response = $http->request($method, $url);
$response = $http->request($method, $url, \%options);
```

Executes an HTTP request of the given method type ('GET', 'HEAD', 'POST', 'PUT', etc.) on the given URL. The URL must have unsafe characters escaped and international domain names encoded.

NOTE: Method names are **case-sensitive** per the HTTP/1.1 specification. Don't use `get` when you really want `GET`. See "LIMITATIONS" for how this applies to redirection.

If the URL includes a "user:password" stanza, they will be used for Basic-style authorization headers. For example:

```
$http->request('GET', 'http://Aladdin:open sesame@example.com/');
```

If the "user:password" stanza contains reserved characters, they must be percent-escaped:

```
$http->request('GET', 'http://john%40example.com:password@example.
↪com/');
```

Caller-supplied `Authorization`, `Cookie` and `Proxy-Authorization` headers are stripped on cross-origin redirects. See "new"'s `allow_credentialed_redirects` attribute to opt out.

A hashref of options may be appended to modify the request.

Valid options are:

- `headers` — A hashref containing headers to include with the request. If the value for a header is an array reference, the header will be output multiple times with each value in the array. These headers over-write any default headers.

- `content` — A scalar to include as the body of the request OR a code reference that will be called iteratively to produce the body of the request
- `trailer_callback` — A code reference that will be called if it exists to provide a hashref of trailing headers (only used with chunked transfer-encoding)
- `data_callback` — A code reference that will be called for each chunks of the response body received.
- `peer` — Override host resolution and force all connections to go only to a specific peer address, regardless of the URL of the request. This will include any redirections! This options should be used with extreme caution (e.g. debugging or very special circumstances). It can be given as either a scalar or a code reference that will receive the hostname and whose response will be taken as the address.

The `Host` header is generated from the URL in accordance with RFC 2616. It is a fatal error to specify `Host` in the `headers` option. Other headers may be ignored or overwritten if necessary for transport compliance.

If the `content` option is a code reference, it will be called iteratively to provide the content body of the request. It should return the empty string or undef when the iterator is exhausted.

If the `content` option is the empty string, no `content-type` or `content-length` headers will be generated.

If the `data_callback` option is provided, it will be called iteratively until the entire response body is received. The first argument will be a string containing a chunk of the response body, the second argument will be the in-progress response hash reference, as described below. (This allows customizing the action of the callback based on the `status` or `headers` received prior to the content body.)

Content data in the request/response is handled as "raw bytes". Any encoding/decoding (with associated headers) are the responsibility of the caller.

The `request` method returns a hashref containing the response. The hashref will have the following keys:

- `success` — Boolean indicating whether the operation returned a 2XX status code
- `url` — URL that provided the response. This is the URL of the request unless there were redirections, in which case it is the last URL queried in a redirection chain
- `status` — The HTTP status code of the response
- `reason` — The response phrase returned by the server
- `content` — The body of the response. If the response does not have any content or if a data callback is provided to consume the response body, this will be the empty string
- `headers` — A hashref of header fields. All header field names will be normalized to be lower case. If a header is repeated, the value will be an arrayref; it will otherwise be a scalar string containing the value
- `protocol` - If this field exists, it is the protocol of the response such as HTTP/1.0 or HTTP/1.1

- `redirects` If this field exists, it is an arrayref of response hash references from redirects in the same order that redirections occurred. If it does not exist, then no redirections occurred.

On an error during the execution of the request, the `status` field will contain 599, and the `content` field will contain the text of the error.

www_form_urlencode

```
$params = $http->www_form_urlencode( $data );
$response = $http->get("http://example.com/query?$params");
```

This method converts the key/value pairs from a data hash or array reference into a `x-www-form-urlencoded` string. The keys and values from the data reference will be UTF-8 encoded and escaped per RFC 3986. If a value is an array reference, the key will be repeated with each of the values of the array reference. If data is provided as a hash reference, the key/value pairs in the resulting string will be sorted by key and value for consistent ordering.

can_ssl

```
$ok          = HTTP::Tiny->can_ssl;
($ok, $why) = HTTP::Tiny->can_ssl;
($ok, $why) = $http->can_ssl;
```

Indicates if SSL support is available. When called as a class object, it checks for the correct version of *Net::SSLeay* and *IO::Socket::SSL*. When called as an object methods, if `SSL_verify` is true or if `SSL_verify_mode` is set in `SSL_options`, it checks that a CA file is available.

In scalar context, returns a boolean indicating if SSL is available. In list context, returns the boolean and a (possibly multi-line) string of errors indicating why SSL isn't available.

connected

```
$host = $http->connected;
($host, $port) = $http->connected;
```

Indicates if a connection to a peer is being kept alive, per the `keep_alive` option.

In scalar context, returns the peer host and port, joined with a colon, or `undef` (if no peer is connected). In list context, returns the peer host and port or an empty list (if no peer is connected).

Note: This method cannot reliably be used to discover whether the remote host has closed its end of the socket.

TLS/SSL SUPPORT

Direct `https` connections are supported only if *IO::Socket::SSL* 1.56 or greater and *Net::SSLeay* 1.49 or greater are installed. An error will occur if new enough versions of these modules are not installed or if the TLS encryption fails. You can also use `HTTP::Tiny::can_ssl()` utility function that returns boolean to see if the required modules are installed.

An `https` connection may be made via an `http` proxy that supports the CONNECT command (i.e. RFC 2817). You may not proxy `https` via a proxy that itself requires `https` to communicate.

TLS/SSL provides two distinct capabilities:

- Encrypted communication channel
- Verification of server identity

By default, HTTP::Tiny verifies server identity.

This was changed in version 0.083 due to security concerns. The previous default behavior can be enabled by setting `$ENV{PERL_HTTP_TINY_SSL_INSECURE_BY_DEFAULT}` to 1.

Verification is done by checking that that the TLS/SSL connection has a valid certificate corresponding to the host name of the connection and that the certificate has been verified by a CA. Assuming you trust the CA, this will protect against machine-in-the-middle attacks.

Certificate verification requires a file or directory containing trusted CA certificates.

`IO::Socket::SSL::default_ca()` is called to detect the default location of your CA certificates. This also supports the environment variables `SSL_CERT_FILE` and `SSL_CERT_DIR`, and will fail over to Mozilla::CA if no certs are found.

If `IO::Socket::SSL::default_ca()` is not able to find usable CA certificates, HTTP::Tiny will search several well-known system-specific default locations for a CA certificate file as a last resort:

- /etc/ssl/certs/ca-certificates.crt
- /etc/pki/tls/certs/ca-bundle.crt
- /etc/ssl/ca-bundle.pem
- /etc/openssl/certs/ca-certificates.crt
- /etc/ssl/cert.pem
- /usr/local/share/certs/ca-root-nss.crt
- /etc/pki/tls/cacert.pem
- /etc/certs/ca-certificates.crt

An error will be occur if `verify_SSL` is true and no CA certificate file is available.

If you desire complete control over TLS/SSL connections, the `SSL_options` attribute lets you provide a hash reference that will be passed through to `IO::Socket::SSL::start_SSL()`, overriding any options set by HTTP::Tiny. For example, to provide your own trusted CA file:

```
SSL_options => {
    SSL_ca_file => $file_path,
}
```

The `SSL_options` attribute could also be used for such things as providing a client certificate for authentication to a server or controlling the choice of cipher used for the TLS/SSL connection. See *IO::Socket::SSL* documentation for details.

PROXY SUPPORT

HTTP::Tiny can proxy both `http` and `https` requests. Only Basic proxy authorization is supported and it must be provided as part of the proxy URL: `http://user:pass@proxy.example.com/`.

HTTP::Tiny supports the following proxy environment variables:

- http_proxy or HTTP_PROXY
- https_proxy or HTTPS_PROXY
- all_proxy or ALL_PROXY

If the `REQUEST_METHOD` environment variable is set, then this might be a CGI process and `HTTP_PROXY` would be set from the `Proxy:` header, which is a security risk. If `REQUEST_METHOD` is set, `HTTP_PROXY` (the upper case variant only) is ignored, but `CGI_HTTP_PROXY` is considered instead.

Tunnelling `https` over an `http` proxy using the CONNECT method is supported. If your proxy uses `https` itself, you can not tunnel `https` over it.

Be warned that proxying an `https` connection opens you to the risk of a man-in-the-middle attack by the proxy server.

The `no_proxy` environment variable is supported in the format of a comma-separated list of domain extensions proxy should not be used for.

Proxy arguments passed to `new` will override their corresponding environment variables.

LIMITATIONS

HTTP::Tiny is *conditionally compliant* with the HTTP/1.1 specifications:

- “Message Syntax and Routing” [RFC7230]
- “Semantics and Content” [RFC7231]
- “Conditional Requests” [RFC7232]
- “Range Requests” [RFC7233]
- “Caching” [RFC7234]
- “Authentication” [RFC7235]

It attempts to meet all “MUST” requirements of the specification, but does not implement all “SHOULD” requirements. (Note: it was developed against the earlier RFC 2616 speci-

fication and may not yet meet the revised RFC 7230-7235 spec.) Additionally, HTTP::Tiny supports the PATCH method of RFC 5789.

Some particular limitations of note include:

- HTTP::Tiny focuses on correct transport. Users are responsible for ensuring that user-defined headers and content are compliant with the HTTP/1.1 specification.
- Users must ensure that URLs are properly escaped for unsafe characters and that international domain names are properly encoded to ASCII. See URI::Escape, URI::_punycode and Net::IDN::Encode.
- Redirection is very strict against the specification. Redirection is only automatic for response codes 301, 302, 307 and 308 if the request method is 'GET' or 'HEAD'. Response code 303 is always converted into a 'GET' redirection, as mandated by the specification. There is no automatic support for status 305 ("Use proxy") redirections.
- There is no provision for delaying a request body using an Expect header. Unexpected 1XX responses are silently ignored as per the specification.
- Only 'chunked' Transfer-Encoding is supported.
- There is no support for a Request-URI of '*' for the 'OPTIONS' request.
- Headers mentioned in the RFCs and some other, well-known headers are generated with their canonical case. Other headers are sent in the case provided by the user. Except for control headers (which are sent first), headers are sent in arbitrary order.

Despite the limitations listed above, HTTP::Tiny is considered feature-complete. New feature requests should be directed to HTTP::Tiny::UA.

SEE ALSO

- HTTP::Tiny::UA - Higher level UA features for HTTP::Tiny
- HTTP::Thin - HTTP::Tiny wrapper with HTTP::Request/HTTP::Response compatibility
- HTTP::Tiny::Mech - Wrap WWW::Mechanize instance in HTTP::Tiny compatible interface
- *IO::Socket::IP* - Required for IPv6 support
- *IO::Socket::SSL* - Required for SSL support
- LWP::UserAgent - If HTTP::Tiny isn't enough for you, this is the "standard" way to do things
- *Net::SSLeay* - Required for SSL support

SUPPORT

Bugs / Feature Requests

Please report any bugs or feature requests through the issue tracker at https://github.com/Perl-Toolchain-Gang/HTTP-Tiny/issues. You will be notified automatically of any progress on your issue.

Source Code

This is open source software. The code repository is available for public review and contribution under the terms of the license.

https://github.com/Perl-Toolchain-Gang/HTTP-Tiny

```
git clone https://github.com/Perl-Toolchain-Gang/HTTP-Tiny.git
```

AUTHORS

- Christian Hansen chansen@cpan.org
- David Golden dagolden@cpan.org

CONTRIBUTORS

- Alan Gardner gardner@pythian.com
- Alessandro Ghedini al3xbio@gmail.com
- A. Sinan Unur nanis@cpan.org
- Brad Gilbert bgills@cpan.org
- brian m. carlson sandals@crustytoothpaste.net
- Chris Nehren apeiron@cpan.org
- Chris Weyl cweyl@alumni.drew.edu
- Claes Jakobsson claes@surfar.nu
- Clinton Gormley clint@traveljury.com
- Craig A. Berry craigberry@mac.com
- Dan grinnz@gmail.com
- David Golden xdg@xdg.me
- David Mitchell davem@iabyn.com
- Dean Pearce pearce@pythian.com
- Edward Zborowski ed@rubensteintech.com
- Felipe Gasper felipe@felipegasper.com
- Graham Knop haarg@haarg.org
- Greg Kennedy kennedy.greg@gmail.com
- James E Keenan jkeenan@cpan.org
- James Raspass jraspass@gmail.com
- Jeremy Mates jmates@cpan.org
- Jess Robinson castaway@desert-island.me.uk
- Karen Etheridge ether@cpan.org

- Lukas Eklund leklund@gmail.com
- Martin J. Evans mjegh@ntlworld.com
- Martin-Louis Bright mlbright@gmail.com
- Matthew Horsfall wolfsage@gmail.com
- Michael R. Davis mrdvt92@users.noreply.github.com
- Michael Stevens michael.stevens@dianomi.com
- Mike Doherty doherty@cpan.org
- Nicolas Rochelemagne rochelemagne@cpanel.net
- Olaf Alders olaf@wundersolutions.com
- Olivier Mengué dolmen@cpan.org
- Petr Písař ppisar@redhat.com
- Philippe Bruhat (BooK) book@cpan.org
- Rob Mueller robm@fastmailteam.com
- sanjay-cpu snjkmr32@gmail.com
- Serguei Trouchelle stro@cpan.org
- Shoichi Kaji skaji@cpan.org
- SkyMarshal skymarshal1729@gmail.com
- Sören Kornetzki soeren.kornetzki@delti.com
- Steve Grazzini steve.grazzini@grantstreet.com
- Stig Palmquist git@stig.io
- Syohei YOSHIDA syohex@gmail.com
- Tatsuhiko Miyagawa miyagawa@bulknews.net
- Tom Hukins tom@eborcom.com
- Tony Cook tony@develop-help.com
- Xavier Guimard yadd@debian.org

COPYRIGHT AND LICENSE

This software is copyright (c) 2026 by Christian Hansen.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

5.2.55 Hash

Namespace containing the following modules:

- `Hash::Util` - General-utility hash subroutines - lock and unlock keys or values, inspect bucket layout, and seed the hash function.

Hash::Util

General-utility hash subroutines - lock and unlock keys or values, inspect bucket layout, and seed the hash function.

A *restricted* hash is an ordinary hash that rejects new keys and, optionally, refuses writes to its values. You build one by calling `lock_keys` on a populated hash; from that point on a write to any key not already present croaks. Locking individual values with `lock_value`, or the whole hash with `lock_hash`, makes the existing values immutable as well. Every lock has a matching unlock.

Alongside the locking primitives the module exposes a handful of introspection helpers: `bucket_ratio`, `num_buckets`, `used_buckets`, `bucket_info`, `bucket_array`, and the summary reports `bucket_stats` and `bucket_stats_formatted`. They report how the hash is laid out internally and are useful when tuning the size of very large hashes.

`hash_seed` and `hash_value` expose the hash function itself - the random per-process seed, and the integer hash of a given string under that seed (or a caller-supplied seed). Both are sensitive: handing them to untrusted code enables algorithmic-complexity attacks against any hash in the program.

Every `ref`-suffixed variant (`lock_ref_keys`, `lock_hashref`, `hashref_locked`, `legal_ref_keys`, `hidden_ref_keys`, etc.) does the same work as its plain counterpart but takes an already-taken hash reference rather than relying on the `\%` prototype.

Synopsis

```
use Hash::Util qw(lock_keys unlock_keys lock_value hash_locked␣
 ↪hidden_keys);

my %h = (foo => 1, bar => 2);
lock_keys(%h);                      # fix the keyset
$h{foo} = 10;                       # ok
eval { $h{baz} = 99 };              # croaks - attempt to access␣
 ↪disallowed key

lock_value(%h, 'foo');              # make $h{foo} immutable too
print hash_locked(%h) ? "yes" : "no";   # yes

unlock_keys(%h);                    # lift the keyset restriction
```

Locking keys

`lock_keys(%hash)` restricts the hash to its current keys; `lock_keys(%hash, @keys)` restricts it to `@keys`. Afterwards, assigning to a key outside the set croaks with `Attempt to access disallowed key '...' in a restricted hash`. With an explicit list, an existing key that is not in the list makes the call croak with `Hash has key '...' which is not in the new key set`; keys in the list that are not present yet become hidden placeholders, legal to assign to later. Every key is compared as a string, so `lock_keys(%h, 1)` allows the key `"1"`.

`delete` and `exists` still work on a locked hash and do not change the legal keyset: a deleted key can be assigned again, a new key still cannot. A locked hash cannot be

`blessed`; bless it first.

`unlock_keys(%hash)` lifts the keyset restriction. Values locked with `lock_value` stay locked, and placeholders stay in place until the next `lock_keys`, which clears them before it locks.

`lock_keys_plus(%hash, @additional_keys)` locks the hash to its current keys plus `@additional_keys` - the same as `lock_keys(%hash, @additional_keys, keys %hash)`.

All of them return a reference to the hash.

Locking values

`lock_value(%hash, $key)` makes the value under `$key` read-only: a later assignment croaks with `Modification of a read-only value attempted`. Unless the keyset is locked too, the key can still be deleted, and a later assignment then creates a fresh writable value; for that reason, with warnings enabled, locking a value in an unlocked hash warns `Cannot usefully lock values in an unlocked hash`. `unlock_value(%hash, $key)` makes the value writable again.

`lock_hash(%hash)` locks the keyset and every value; `unlock_hash` undoes both. `lock_hash_recurse` and `unlock_hash_recurse` do the same and descend into every value that is a hash reference - only into hashes, so a hash of arrays of hashes has only its outermost hash locked.

All of them return a reference to the hash.

Querying the lock state

`hash_locked(%hash)` is true when the keyset is locked and `hash_unlocked(%hash)` is its negation. Both return Perl's boolean true and false values, so a false result prints as the empty string. Value locks alone do not count.

`legal_keys(%hash)` lists every key the hash may carry - the visible keys plus the hidden placeholders - and `hidden_keys(%hash)` lists the placeholders only: legal keys that fail both `exists` and `defined`. On an unrestricted hash `legal_keys` equals `keys %hash` and `hidden_keys` is empty.

Hash-reference forms

Each function above has a form that takes a hash reference instead of relying on the `\%` prototype: `lock_ref_keys`, `unlock_ref_keys`, `lock_ref_keys_plus`, `lock_ref_value`, `unlock_ref_value`, `lock_hashref`, `unlock_hashref`, `lock_hashref_recurse`, `unlock_hashref_recurse`, `hashref_locked`, `hashref_unlocked`, `legal_ref_keys` and `hidden_ref_keys`.

Bucket statistics

`bucket_stats(\%hash)` extends `bucket_info` with derived quantities:

```
my ($keys, $buckets, $used, $quality, $utilization_ratio,
    $collision_pct, $mean, $stddev, @length_counts)
    = bucket_stats(\%hash);
```

`$quality` is the Red Dragon quality score (ideal `1.0`, acceptable around `0.95` to `1.05`), `$utilization_ratio` the fraction of used buckets, `$collision_pct` the fraction of keys that share a bucket, and `$mean` and `$stddev` describe the chain lengths of the used buckets. For an empty hash only the first three values are returned.

`bucket_stats_formatted(\%hash)` renders the same data as a report: a summary with the quality score read as "Good", "Poor" or "Bad", then two bar charts - chain lengths, which is what a failed lookup costs, and key positions within their chains, which is what a successful one costs. Useful when deciding whether to pre-size a large hash with `keys(%h) = N`.

Security

`hash_seed`, `hash_value`, and `bucket_array` reveal enough about the hash function to enable collision attacks. Do not expose their output to code you do not trust.

This is a native implementation of the XS half of Hash::Util 0.32; its `Hash/Util.pm` is used unchanged. Nothing is exported by default.

Differences from upstream

None.

Modules

- `Hash::Util::FieldHash` - Support for inside-out classes: hashes keyed by the identity of a reference, whose entries go away with the object.

Functions

Hash locking

`all_keys`

Split a restricted hash's keys into visible and hidden sets, writing the results into the caller's arrays.

`hidden_ref_keys`

List the hidden placeholder keys of a restricted hash - legal keys that currently have no value.

`legal_ref_keys`

List every key a restricted hash is allowed to carry - the visible keys plus the hidden placeholders.

Synopsis

```
my @legal = legal_ref_keys(\%hash);
my @legal = legal_keys(%hash);          # the \% form
```

On an unrestricted hash this is equivalent to `keys %hash`.

hv_store

Install an *alias* from `$hash{$key}` to an existing scalar, instead of copying its value in the usual way.

lock_keys

Restrict a hash to its current keys, or to an explicit list.

unlock_keys

Lift the keyset restriction imposed by `lock_keys`.

lock_ref_keys

Hash-reference variant of `lock_keys`.

unlock_ref_keys

Hash-reference variant of `unlock_keys`.

lock_ref_keys_plus

Lock the hash to the union of its current keys and a given extra list.

lock_value

Make the value under a given key immutable.

unlock_value

Undo a previous `lock_value`, making the value at `$key` writable again.

lock_ref_value

Hash-reference variant of `lock_value`.

Synopsis

```
lock_ref_value($href, $key);
```

Same behaviour as `lock_value` but takes a hash reference.

unlock_ref_value

Hash-reference variant of unlock_value.

Synopsis

```
unlock_ref_value($href, $key);
```

Same behaviour as unlock_value but takes a hash reference.

lock_hash

Lock the whole hash: both its keyset and every value.

unlock_hash

Undo a previous lock_hash, making every key and value writable again.

Synopsis

```
unlock_hash(%hash);
```

Clears the read-only flag on each existing value and lifts the keyset restriction. Returns a reference to the hash.

lock_hashref

Hash-reference variant of lock_hash.

Synopsis

```
lock_hashref($href);
```

Same behaviour as lock_hash but takes a hash reference.

unlock_hashref

Hash-reference variant of unlock_hash.

Synopsis

```
unlock_hashref($href);
```

Same behaviour as unlock_hash but takes a hash reference.

lock_hash_recurse

Lock a hash and every nested hash it references, transitively.

unlock_hash_recurse

Undo a previous `lock_hash_recurse`, unlocking every nested hash it touched.

Synopsis

```
unlock_hash_recurse(%hash);
```

Traverses the same subset of references as `lock_hash_recurse` - hash-valued entries only. Returns a reference to the hash.

lock_hashref_recurse

Hash-reference variant of `lock_hash_recurse`.

Synopsis

```
lock_hashref_recurse($href);
```

Same behaviour as `lock_hash_recurse` but takes a hash reference.

unlock_hashref_recurse

Hash-reference variant of `unlock_hash_recurse`.

Synopsis

```
unlock_hashref_recurse($href);
```

Same behaviour as `unlock_hash_recurse` but takes a hash reference.

hash_locked

Report whether the hash's keyset is currently locked.

hash_unlocked

Report whether the hash's keyset is *not* currently locked.

hashref_locked

Hash-reference variant of `hash_locked`.

Synopsis

```
hashref_locked($href) and print "restricted\n";
```

Same behaviour as `hash_locked` but takes a hash reference.

hashref_unlocked

Hash-reference variant of hash_unlocked.

Synopsis

```
hashref_unlocked($href) and print "open\n";
```

Same behaviour as hash_unlocked but takes a hash reference.

hidden_keys

List the “hidden” placeholder keys of a restricted hash - legal keys that currently have no value assigned.

legal_keys

List every key the restricted hash *is allowed to carry* - visible keys plus hidden placeholders.

Bucket introspection

hash_traversal_mask

Read or set the per-hash bucket-traversal mask used by keys, values, and each.

bucket_info

Return a flat list summarising the hash’s bucket layout.

bucket_array

Return a packed array-of-arrays describing which key lives in which bucket.

bucket_ratio

Return a "used/total" string describing how many of the hash’s internal buckets hold at least one key.

num_buckets

Return the total number of buckets the hash holds (or would hold if its bucket array were allocated).

used_buckets

Return the count of buckets that currently hold at least one key.

bucket_stats

Return a rich statistical summary of the hash's bucket layout.

bucket_stats_formatted

Return a multi-line human-readable report of `bucket_stats`, complete with ASCII bar-charts.

Hash function

hash_seed

Return the per-process random bytes that seed the hash function.

hash_value

Return the integer hash value of a string under the current (or a supplied) seed.

Hash::Util::FieldHash

Support for inside-out classes: hashes keyed by the identity of a reference, whose entries go away with the object.

A *field hash* takes a reference as a key and stores it under the reference's id, so the key survives stringification overloads and re-blessing. When the object is destroyed, its entry disappears from every field hash it was used in. That is the bookkeeping an inside-out class needs: each attribute is one field hash, keyed by the object.

An *idhash* only replaces a reference key by its id; registering the object with `register` adds the garbage collection.

Synopsis

```
use Hash::Util::FieldHash qw(fieldhash id id_2obj register idhash);

package Person;
fieldhash my %name;
sub new  { my $self = bless \ do { my $o }, shift; $name{$self} =␣
 ↪shift; $self }
sub name { $name{$_[0]} }

package main;
my $p = Person->new('Ada');
print $p->name, "\n";                # Ada
my $id = id($p);                     # the key %name stores it under
print id_2obj($id) == $p ? "same\n" : "other\n";
undef $p;                            # %name is empty again
```

Functions

`fieldhash(%h)` and `fieldhashes(\%a, \%b, ...)` turn hashes into field hashes; `idhash` and `idhashes` turn them into idhashes. Each returns a reference to the hash, or nothing when the argument is not a hash. A hash that already is one of the two keeps its kind.

`id($ref)` is the id of a reference, the number a field hash stores it under; a non-reference comes back unchanged. `id_2obj($id)` is the registered object with that id, or `undef`. `register($obj, @hashrefs)` registers `$obj` and marks each hash in `@hashrefs` for the cleanup of its id; it returns a new reference to `$obj`.

This is a native implementation of the XS half of Hash::Util::FieldHash 1.27; its `Hash/Util/FieldHash.pm` is used unchanged. Nothing is exported by default.

Differences from upstream

None.

Functions

Field hashes

`register`

Register an object for garbage collection in the given hashes.

Object ids

`id`

The id of a reference: the number a field hash stores it under.

`id_2obj`

The registered object that has a given id.

id

The id of a reference: the number a field hash stores it under.

Synopsis

```
use Hash::Util::FieldHash qw(id);
my $id = id($obj);
```

What you get back

For a reference, the address of what it points to, as a number - the same value `Scalar::Util::refaddr` gives. The id stays the object's for its lifetime. Anything that is not a reference comes back unchanged, where `refaddr` gives `undef`.

id_2obj

The registered object that has a given id.

Synopsis

```
use Hash::Util::FieldHash qw(id id_2obj);
my $obj = id_2obj($id);
```

What you get back

A new reference to the object whose `id` is `$id`, or `undef`. Only objects that have been used as a field hash key or passed to `register` are known; for any other id, and once the object is gone, the answer is `undef`.

register

Register an object for garbage collection in the given hashes.

Synopsis

```
use Hash::Util::FieldHash qw(id register);
my %name;
register($obj, \%name);
$name{id $obj} = 'Ada';     # deleted when $obj goes away
```

What you get back

A new reference to the object. Every hash reference after the first argument is marked, so that the object's entry is deleted from it when the object is destroyed; arguments that are not hash references are ignored. Field hashes do this themselves; `register` is for idhashes and ordinary hashes keyed by `id`. A first argument that is not a reference dies with `Attempt to register a non-ref`.

all_keys

Split a restricted hash's keys into visible and hidden sets, writing the results into the caller's arrays.

Synopsis

```
all_keys(%hash, @visible, @hidden);
```

Clears both arrays first, then fills `@visible` with every key that passes `exists` and `@hidden` with every legal-but-unassigned placeholder key. Returns a reference to the hash.

On an unrestricted hash `@hidden` comes back empty and `@visible` matches `keys %hash`. Like `hidden_keys`, the behaviour is experimental and depends on the current restricted-hash implementation.

hidden_ref_keys

List the hidden placeholder keys of a restricted hash - legal keys that currently have no value.

Synopsis

```
my @hidden = hidden_ref_keys(\%hash);
my @hidden = hidden_keys(%hash);        # the \% form
```

A hidden key is allowed by the hash's keyset but fails both `exists` and `defined` - typically the ghost left behind by a `delete` on a locked hash, or a key reserved in advance by `lock_keys_plus`. Assigning to it brings it into the visible keyset.

Returns the empty list on an unrestricted hash. The feature is experimental and closely tied to the current restricted-hash implementation; do not build durable contracts on top of it.

hv_store

Install an *alias* from `$hash{$key}` to an existing scalar, instead of copying its value in the usual way.

Synopsis

```
my $sv = 0;
hv_store(%hash, $key, $sv) or die "store failed";
$hash{$key} = 1;
print $sv;          # prints 1 - $sv and $hash{$key} share one
 ↪scalar
```

After the call, `$sv` and `$hash{$key}` refer to the same scalar: a later assignment through either name is visible through the other. Returns true on success and false if the store was rejected.

This is a low-level primitive with narrow uses; ordinary code should assign through the hash normally.

hash_seed

Return the per-process random bytes that seed the hash function.

Synopsis

```
my $seed = hash_seed();
```

The result is a byte string whose length depends on the hash function the interpreter is built with: 32 bytes for pperl, as for a default build of perl 5.44. Two interpreters started from the same program see different values; a single run sees the same value throughout its life.

The seed is sensitive. Anyone who learns it can construct keys that collide in your hashes and mount an algorithmic-complexity attack. Keep it out of logs and error messages.

hash_value

Return the integer hash value of a string under the current (or a supplied) seed.

Synopsis

```
my $h = hash_value($string);
my $h = hash_value($string, $seed);
```

With one argument, hashes `$string` with the interpreter's per-process seed - the same seed reported by `hash_seed`. With a second argument, hashes it with `$seed` instead, letting you compute what the hash value *would* be under a different seed. The custom seed must be at least as long as `hash_seed` produces; a shorter `$seed` croaks with `seed len must be at least N long only got M bytes`.

`$string` is used as a string whatever it holds, so `hash_value(42)` equals `hash_value("42")`.

The one-argument value is stable for the lifetime of a single interpreter and unstable across runs, Perl versions, build options, and architectures. Do not persist it.

Like `hash_seed`, the result is sensitive information that can be used to engineer collisions. Treat it accordingly.

hash_traversal_mask

Read or set the per-hash bucket-traversal mask used by `keys`, `values`, and `each`.

Synopsis

```
my $mask = hash_traversal_mask(\%hash);
hash_traversal_mask(\%hash, 1234);
```

Every hash carries a U32 mask that is XOR'd with each bucket index during iteration, so two hashes with the same contents still visit their keys in different orders. Called with one argument, the function returns the current mask. Called with two, it sets the mask - pinning the iteration order of that particular hash for reproducible test output.

Pinning the mask does not make two *different* hashes iterate identically: keys that collide into the same bucket may still appear in different orders because chain order depends on insertion history.

Returns undef for a hash that has never been given a mask, and nothing at all for an argument that is not a reference to a plain hash.

bucket_info

Return a flat list summarising the hash's bucket layout.

Synopsis

```
my ($keys, $buckets, $used, @length_counts) = bucket_info(\%hash);
```

The first three values are the number of keys, the total bucket count, and the number of used buckets. Each `$length_counts[$n]` that follows is the number of buckets holding exactly `$n` keys, so `$length_counts[0]` is the empty-bucket count, `$length_counts[1]` the count of single-key buckets, and so on.

Given undef, it reports on the interpreter's shared string table. Returns the empty list for any other argument that is not a reference to a plain, non-magical hash. When the hash's bucket array has not been allocated yet, only the three leading values are returned.

bucket_array

Return a packed array-of-arrays describing which key lives in which bucket.

Synopsis

```
my $array = bucket_array(\%hash);
```

Each element of the returned array reference is either an integer `$n` - meaning `$n` consecutive empty buckets at that position - or an array reference listing the keys stored in a single bucket. Placeholder keys are reported as normal keys. The order within a bucket is Perl's insert-first chain order and changes across rehashes.

Intended for debugging and diagnostics only. The layout reveals enough about the hash function to support collision attacks, so do not leak the output to untrusted code.

Given undef, it reports on the interpreter's shared string table. Returns nothing when the argument is not a reference to a plain hash, or when the hash has no bucket array yet.

bucket_ratio

Return a `"used/total"` string describing how many of the hash's internal buckets hold at least one key.

Synopsis

```
my $r = bucket_ratio(%hash);     # e.g. "5/8"
```

Behaves like the pre-5.25 `scalar(%hash)`: for a tied hash it calls the SCALAR tie method; for a normal empty hash it returns `0`; otherwise it reports the count of non-empty buckets followed by a slash followed by the total bucket count.

num_buckets

Return the total number of buckets the hash holds (or would hold if its bucket array were allocated).

Synopsis

```
my $n = num_buckets(%hash);
```

A freshly created hash may report a non-zero bucket count even before its bucket array has been materialised - Perl allocates buckets lazily. The value is always a power of two.

used_buckets

Return the count of buckets that currently hold at least one key.

Synopsis

```
my $used = used_buckets(%hash);
```

The value is computed from scratch on every call by walking the bucket array - there is no cache. Avoid calling it inside a hot loop on a large hash.

bucket_stats

Return a rich statistical summary of the hash's bucket layout.

Synopsis

```
my ($keys, $buckets, $used, $quality, $utilization_ratio,
    $collision_pct, $mean, $stddev, @length_counts)
    = bucket_stats(\%hash);
```

Extends `bucket_info` with derived quantities: `$quality` is the Red-Dragon quality score (ideal `1.0`, acceptable around `0.95` to `1.05`, worse as the number grows), `$utilization_ratio` is the fraction of used buckets, `$collision_pct` is the fraction of keys that are not first in their bucket chain (`($keys - $used) / $keys`), and `$mean` and `$stddev` describe the bucket-chain-length distribution over used buckets.

For an empty hash, only `$keys`, `$buckets`, and `$used` are returned - the statistical block is skipped.

bucket_stats_formatted

Return a multi-line human-readable report of `bucket_stats`, complete with ASCII bar-charts.

Synopsis

```
print bucket_stats_formatted(\%hash);
```

The first paragraph of the report gives the summary line (key count, used buckets, total buckets, quality score translated into “Good”, “Poor”, or “Bad”), followed by utilisation and collision percentages and the chain-length mean and standard deviation. Two bar-charts follow: one showing the distribution of bucket chain lengths (how costly a missed lookup is), and one showing how many keys sit at each position in their chain (how costly a found lookup is).

Useful when deciding whether to pre-size a large hash with `keys(%h) = N` to cut collisions.

lock_keys

Restrict a hash to its current keys, or to an explicit list.

Synopsis

```
lock_keys(%hash);              # pin the current keyset
lock_keys(%hash, @keys);       # pin to the given keyset
```

With one argument, subsequent attempts to assign to a key not already in `%hash` croak with `Attempt to access disallowed key '...' in a restricted hash`. With more arguments, the extra keys specify the allowed set; existing keys that do not appear in that set make the call croak with `Hash has key '...' which is not in the new key set`. Keys in the allowed set that are not yet present are accepted later (their slots start as hidden placeholders).

`delete` and `exists` still work on a locked hash and do not change the legal keyset - a deleted key can be reassigned, but a brand-new key still cannot. Returns a reference to the hash.

A locked hash cannot be `blessed`; bless before locking if you need both.

unlock_keys

Lift the keyset restriction imposed by `lock_keys`.

Synopsis

```
unlock_keys(%hash);
```

Any keys locked with `lock_value` or `lock_hash` remain locked - this undoes only the keyset restriction. Hidden placeholder keys introduced while the hash was locked stay

in place; use `hidden_keys` to see them and ordinary assignment to bring them into the visible keyset.

Returns a reference to the hash.

lock_ref_keys

Hash-reference variant of `lock_keys`.

Synopsis

```
lock_ref_keys($href);
lock_ref_keys($href, @keys);
```

Exactly the same effect as `lock_keys`, but takes an already-taken hash reference rather than relying on the `\%` prototype. Useful when the hash only exists as a reference - for example, an anonymous hash or a field on an object.

unlock_ref_keys

Hash-reference variant of `unlock_keys`.

Synopsis

```
unlock_ref_keys($href);
```

Lifts the keyset restriction on the referenced hash. See `unlock_keys` for the full story on placeholders and values that remain locked.

lock_ref_keys_plus

Lock the hash to the union of its current keys and a given extra list.

Synopsis

```
lock_ref_keys_plus($href, @additional_keys);
lock_keys_plus(%hash, @additional_keys);      # plain-hash form
```

Equivalent to `lock_keys($hash, @additional_keys, keys %hash)` without having to list the existing keys yourself. Extra keys that are not yet present in the hash become hidden placeholders - they are legal to assign to later but do not yet appear in `keys`.

Returns a reference to the hash.

lock_value

Make the value under a given key immutable.

Synopsis

```
lock_value(%hash, $key);
```

Any subsequent attempt to overwrite `$hash{$key}` croaks with `Modification of a read-only value attempted`. The key itself remains deletable unless `lock_keys` has also been applied - once the key is deleted, a later assignment creates a fresh (writable) value. A key not already present is created with a read-only `undef` value, or croaks if a locked keyset does not allow it. On a hash whose keyset is not locked, `lock_value` warns `Cannot usefully lock values in an unlocked hash` when warnings are enabled.

Returns a reference to the hash.

unlock_value

Undo a previous `lock_value`, making the value at `$key` writable again.

Synopsis

```
unlock_value(%hash, $key);
```

Does nothing (silently) if the value was never locked; an absent key is created with an `undef` value, or croaks if a locked keyset does not allow it. Returns a reference to the hash.

lock_hash

Lock the whole hash: both its keyset and every value.

Synopsis

```
lock_hash(%hash);
```

Equivalent to `lock_keys(%hash)` followed by `lock_value` on each existing key. After the call, no key may be added and no value overwritten, and because every value is read-only, `delete` croaks with “Attempt to delete readonly key”.

Returns a reference to the hash.

lock_hash_recurse

Lock a hash and every nested hash it references, transitively.

Synopsis

```
lock_hash_recurse(%hash);
```

Recurses only through hash-valued entries - a hash-of-hashes is locked throughout, but a hash-of-arrays-of-hashes has only the outermost hash locked because the recursion

does not descend into arrays. Reach for `Data::Visitor` or write your own walker if you need deeper traversal.

Returns a reference to the hash.

hash_locked

Report whether the hash's keyset is currently locked.

Synopsis

```
hash_locked(%hash) and print "restricted\n";
```

Returns 1 when `lock_keys` (or an equivalent) has been applied to the hash, the empty string otherwise. Value-level locks alone do not count - this only reports the keyset restriction.

hash_unlocked

Report whether the hash's keyset is *not* currently locked.

Synopsis

```
hash_unlocked(%hash) and print "open\n";
```

The logical inverse of `hash_locked`: returns 1 on an unlocked hash, the empty string on a locked one. Handy in conditions where testing for the open state reads more naturally than negating the locked one.

hidden_keys

List the "hidden" placeholder keys of a restricted hash - legal keys that currently have no value assigned.

Synopsis

```
my @hidden = hidden_keys(%hash);
```

A hidden key is one that is allowed by the hash's keyset but fails both `exists` and `defined` - typically the ghost left behind after a `delete` on a locked hash, or a key reserved in advance by `lock_keys_plus`. Assigning to a hidden key brings it into the visible keyset.

Returns the empty list on a hash that was never restricted; placeholders left by an earlier lock are still listed after `unlock_keys`. The feature is experimental and closely tied to the current restricted-hash implementation; do not build durable contracts on top of it.

legal_keys

List every key the restricted hash *is allowed to carry* - visible keys plus hidden placeholders.

Synopsis

```
my @legal = legal_keys(%hash);
```

On a hash that was never restricted this is equivalent to `keys %hash`. On a restricted hash it is the visible keys (`keys %hash`) merged with whatever `hidden_keys` would report.

5.2.56 I18N

Namespace containing the following modules:

- `I18N::Collate` - compare 8-bit scalar data according to the current locale
- `I18N::LangTags` - functions for dealing with RFC3066-style language tags
- `I18N::Langinfo` - Query locale information - country, language, and cultural conventions from the current locale.

I18N::Collate

compare 8-bit scalar data according to the current locale

SYNOPSIS

```
use I18N::Collate;
setlocale(LC_COLLATE, 'locale-of-your-choice');
$s1 = I18N::Collate->new("scalar_data_1");
$s2 = I18N::Collate->new("scalar_data_2");
```

DESCRIPTION

```
***

WARNING: starting from the Perl version 5.003_06
the I18N::Collate interface for comparing 8-bit scalar data
according to the current locale

      HAS BEEN DEPRECATED

That is, please do not use it anymore for any new applications
and please migrate the old applications away from it because its
functionality was integrated into the Perl core language in the
release 5.003_06.

```

(continues on next page)

(continued from previous page)

```
See the perllocale manual page for further information.

***
```

This module provides you with objects that will collate according to your national character set, provided that the POSIX setlocale() function is supported on your system.

You can compare $s1 and $s2 above with

```
$s1 le $s2
```

to extract the data itself, you'll need a dereference: $$s1

This module uses POSIX::setlocale(). The basic collation conversion is done by strxfrm() which terminates at NUL characters being a decent C routine. collate_xfrm() handles embedded NUL characters gracefully.

The available locales depend on your operating system; try whether `locale -a` shows them or man pages for "locale" or "nlsinfo" or the direct approach `ls /usr/lib/nls/loc` or `ls /usr/lib/nls` or `ls /usr/lib/locale`. Not all the locales that your vendor supports are necessarily installed: please consult your operating system's documentation and possibly your local system administration. The locale names are probably something like `xx_XX.(ISO)?8859-N` or `xx_XX.(ISO)?8859N`, for example `fr_CH.ISO8859-1` is the Swiss (CH) variant of French (fr), ISO Latin (8859) 1 (-1) which is the Western European character set.

I18N::LangTags

functions for dealing with RFC3066-style language tags

SYNOPSIS

```
use I18N::LangTags();
```

...or specify whichever of those functions you want to import, like so:

```
use I18N::LangTags qw(implicate_supers similarity_language_tag);
```

All the exportable functions are listed below – you're free to import only some, or none at all. By default, none are imported. If you say:

```
use I18N::LangTags qw(:ALL)
```

...then all are exported. (This saves you from having to use something less obvious like `use I18N::LangTags qw(/./)`.)

If you don't import any of these functions, assume a `&I18N::LangTags::` in front of all the function names in the following examples.

DESCRIPTION

Language tags are a formalism, described in RFC 3066 (obsoleting 1766), for declaring what language form (language and possibly dialect) a given chunk of information is in.

This library provides functions for common tasks involving language tags as they are needed in a variety of protocols and applications.

Please see the "See Also" references for a thorough explanation of how to correctly use language tags.

- the function is_language_tag($lang1)

  Returns true iff $lang1 is a formally valid language tag.

```
is_language_tag("fr")           is TRUE
is_language_tag("x-jicarilla")   is FALSE
    (Subtags can be 8 chars long at most -- 'jicarilla' is 9)

is_language_tag("sgn-US")    is TRUE
    (That's American Sign Language)

is_language_tag("i-Klikitat")    is TRUE
    (True without regard to the fact noone has actually
     registered Klikitat -- it's a formally valid tag)

is_language_tag("fr-patois")     is TRUE
    (Formally valid -- altho descriptively weak!)

is_language_tag("Spanish")       is FALSE
is_language_tag("french-patois") is FALSE
    (No good -- first subtag has to match
     /^([xXiI]|[a-zA-Z]{2,3})$/ -- see RFC3066)

is_language_tag("x-borg-prot2532") is TRUE
    (Yes, subtags can contain digits, as of RFC3066)
```

- the function extract_language_tags($whatever)

  Returns a list of whatever looks like formally valid language tags in $whatever. Not very smart, so don't get too creative with what you want to feed it.

```
extract_language_tags("fr, fr-ca, i-mingo")
  returns:   ('fr', 'fr-ca', 'i-mingo')

extract_language_tags("It's like this: I'm in fr -- French!")
  returns:   ('It', 'in', 'fr')
(So don't just feed it any old thing.)
```

  The output is untainted. If you don't know what tainting is, don't worry about it.

- the function same_language_tag($lang1, $lang2)

  Returns true iff $lang1 and $lang2 are acceptable variant tags representing the same language-form.

```
same_language_tag('x-kadara', 'i-kadara')  is TRUE
   (The x/i- alternation doesn't matter)
same_language_tag('X-KADARA', 'i-kadara')  is TRUE
   (...and neither does case)
same_language_tag('en',       'en-US')     is FALSE
   (all-English is not the SAME as US English)
same_language_tag('x-kadara', 'x-kadar')   is FALSE
   (these are totally unrelated tags)
same_language_tag('no-bok',    'nb')       is TRUE
   (no-bok is a legacy tag for nb (Norwegian Bokmal))
```

same_language_tag works by just seeing whether encode_language_tag($lang1) is the same as encode_language_tag($lang2).

(Yes, I know this function is named a bit oddly. Call it historic reasons.)

- the function similarity_language_tag($lang1, $lang2)

  Returns an integer representing the degree of similarity between tags $lang1 and $lang2 (the order of which does not matter), where similarity is the number of common elements on the left, without regard to case and to x/i- alternation.

```
similarity_language_tag('fr', 'fr-ca')          is 1
   (one element in common)
similarity_language_tag('fr-ca', 'fr-FR')       is 1
   (one element in common)

similarity_language_tag('fr-CA-joual',
                        'fr-CA-PEI')            is 2
similarity_language_tag('fr-CA-joual', 'fr-CA')  is 2
   (two elements in common)

similarity_language_tag('x-kadara', 'i-kadara')  is 1
   (x/i- doesn't matter)

similarity_language_tag('en',       'x-kadar')   is 0
similarity_language_tag('x-kadara', 'x-kadar')   is 0
   (unrelated tags -- no similarity)

similarity_language_tag('i-cree-syllabic',
                        'i-cherokee-syllabic')   is 0
   (no B<leftmost> elements in common!)
```

- the function is_dialect_of($lang1, $lang2)

  Returns true iff language tag $lang1 represents a subform of language tag $lang2.

  Get the order right! It doesn't work the other way around!

```
is_dialect_of('en-US', 'en')           is TRUE
  (American English IS a dialect of all-English)

is_dialect_of('fr-CA-joual', 'fr-CA')   is TRUE
```

(continues on next page)

(continued from previous page)

```
is_dialect_of('fr-CA-joual', 'fr')      is TRUE
  (Joual is a dialect of (a dialect of) French)

is_dialect_of('en', 'en-US')            is FALSE
  (all-English is a NOT dialect of American English)

is_dialect_of('fr', 'en-CA')            is FALSE

is_dialect_of('en',    'en'   )         is TRUE
is_dialect_of('en-US', 'en-US')         is TRUE
  (B<Note:> these are degenerate cases)

is_dialect_of('i-mingo-tom', 'x-Mingo') is TRUE
  (the x/i thing doesn't matter, nor does case)

is_dialect_of('nn', 'no')               is TRUE
  (because 'nn' (New Norse) is aliased to 'no-nyn',
   as a special legacy case, and 'no-nyn' is a
   subform of 'no' (Norwegian))
```

- the function super_languages($lang1)

  Returns a list of language tags that are superordinate tags to $lang1 – it gets this by removing subtags from the end of $lang1 until nothing (or just “i” or “x”) is left.

```
super_languages("fr-CA-joual")  is  ("fr-CA", "fr")

super_languages("en-AU")  is  ("en")

super_languages("en")  is  empty-list, ()

super_languages("i-cherokee")  is  empty-list, ()
 ...not ("i"), which would be illegal as well as pointless.
```

  If $lang1 is not a valid language tag, returns empty-list in a list context, undef in a scalar context.

  A notable and rather unavoidable problem with this method: “x-mingo-tom” has an “x” because the whole tag isn’t an IANA-registered tag – but super_languages(‘x-mingo-tom’) is (‘x-mingo’) – which isn’t really right, since ‘i-mingo’ is registered. But this module has no way of knowing that. (But note that same_language_tag(‘x-mingo’, ‘i-mingo’) is TRUE.)

  More importantly, you assume *at your peril* that superordinates of $lang1 are mutually intelligible with $lang1. Consider this carefully.

- the function locale2language_tag($locale_identifier)

  This takes a locale name (like “en”, “en_US”, or “en_US.ISO8859-1”) and maps it to a language tag. If it’s not mappable (as with, notably, “C” and “POSIX”), this returns empty-list in a list context, or undef in a scalar context.

```
locale2language_tag("en") is "en"

locale2language_tag("en_US") is "en-US"

locale2language_tag("en_US.ISO8859-1") is "en-US"

locale2language_tag("C") is undef or ()

locale2language_tag("POSIX") is undef or ()

locale2language_tag("POSIX") is undef or ()
```

I'm not totally sure that locale names map satisfactorily to language tags. Think REAL hard about how you use this. YOU HAVE BEEN WARNED.

The output is untainted. If you don't know what tainting is, don't worry about it.

- the function encode_language_tag($lang1)

  This function, if given a language tag, returns an encoding of it such that:

  * tags representing different languages never get the same encoding.

  * tags representing the same language always get the same encoding.

  * an encoding of a formally valid language tag always is a string value that is defined, has length, and is true if considered as a boolean.

  Note that the encoding itself is **not** a formally valid language tag. Note also that you cannot, currently, go from an encoding back to a language tag that it's an encoding of.

  Note also that you **must** consider the encoded value as atomic; i.e., you should not consider it as anything but an opaque, unanalysable string value. (The internals of the encoding method may change in future versions, as the language tagging standard changes over time.)

  `encode_language_tag` returns undef if given anything other than a formally valid language tag.

  The reason `encode_language_tag` exists is because different language tags may represent the same language; this is normally treatable with `same_language_tag`, but consider this situation:

  You have a data file that expresses greetings in different languages. Its format is "[language tag]=[how to say 'Hello']", like:

```
en-US=Hiho
fr=Bonjour
i-mingo=Hau'
```

  And suppose you write a program that reads that file and then runs as a daemon, answering client requests that specify a language tag and then expect the string that says how to greet in that language. So an interaction looks like:

```
greeting-client asks:    fr
greeting-server answers: Bonjour
```

So far so good. But suppose the way you're implementing this is:

```
my %greetings;
die unless open(IN, "<", "in.dat");
while(<IN>) {
  chomp;
  next unless /^([^=]+)=(.+)/s;
  my($lang, $expr) = ($1, $2);
  $greetings{$lang} = $expr;
}
close(IN);
```

at which point %greetings has the contents:

```
"en-US"   => "Hiho"
"fr"      => "Bonjour"
"i-mingo" => "Hau'"
```

And suppose then that you answer client requests for language $wanted by just looking up $greetings{$wanted}.

If the client asks for "fr", that will look up successfully in

"i-mingo", that will look up successfully in %greetings, to the value "Hau'".

But if the client asks for "i-Mingo" or "x-mingo", or "Fr", then the lookup in %greetings fails. That's the Wrong Thing.

You could instead do lookups on $wanted with:

```
use I18N::LangTags qw(same_language_tag);
my $response = '';
foreach my $l2 (keys %greetings) {
  if(same_language_tag($wanted, $l2)) {
    $response = $greetings{$l2};
    last;
  }
}
```

But that's rather inefficient. A better way to do it is to start your program with:

```
use I18N::LangTags qw(encode_language_tag);
my %greetings;
die unless open(IN, "<", "in.dat");
while(<IN>) {
  chomp;
  next unless /^([^=]+)=(.+)/s;
  my($lang, $expr) = ($1, $2);
  $greetings{
             encode_language_tag($lang)
```

(continues on next page)

(continued from previous page)

```
            } = $expr;
}
close(IN);
```

and then just answer client requests for language $wanted by just looking up

```
$greetings{encode_language_tag($wanted)}
```

And that does the Right Thing.

- the function alternate_language_tags($lang1)

  This function, if given a language tag, returns all language tags that are alternate forms of this language tag. (I.e., tags which refer to the same language.) This is meant to handle legacy tags caused by the minor changes in language tag standards over the years; and the x-/i- alternation is also dealt with.

  Note that this function does *not* try to equate new (and never-used, and unusable) ISO639-2 three-letter tags to old (and still in use) ISO639-1 two-letter equivalents – like “ara” -> “ar” – because “ara” has *never* been in use as an Internet language tag, and RFC 3066 stipulates that it never should be, since a shorter tag (“ar”) exists.

  Examples:

```
alternate_language_tags('no-bok')       is ('nb')
alternate_language_tags('nb')           is ('no-bok')
alternate_language_tags('he')           is ('iw')
alternate_language_tags('iw')           is ('he')
alternate_language_tags('i-hakka')      is ('zh-hakka', 'x-hakka
↪')
alternate_language_tags('zh-hakka')     is ('i-hakka', 'x-hakka
↪')
alternate_language_tags('en')           is ()
alternate_language_tags('x-mingo-tom')  is ('i-mingo-tom')
alternate_language_tags('x-klikitat')   is ('i-klikitat')
alternate_language_tags('i-klikitat')   is ('x-klikitat')
```

  This function returns empty-list if given anything other than a formally valid language tag.

- the function @langs = panic_languages(@accept_languages)

  This function takes a list of 0 or more language tags that constitute a given user’s Accept-Language list, and returns a list of tags for *other* (non-super) languages that are probably acceptable to the user, to be used *if all else fails*.

  For example, if a user accepts only ‘ca’ (Catalan) and ‘es’ (Spanish), and the documents/interfaces you have available are just in German, Italian, and Chinese, then the user will most likely want the Italian one (and not the Chinese or German one!), instead of getting nothing. So `panic_languages('ca', 'es')` returns a list containing ‘it’ (Italian).

  English (‘en’) is *always* in the return list, but whether it’s at the very end or not depends on the input languages. This function works by consulting an internal table that stipulates what common languages are “close” to each other.

A useful construct you might consider using is:

```
@fallbacks = super_languages(@accept_languages);
push @fallbacks, panic_languages(
  @accept_languages, @fallbacks,
);
```

- the function implicate_supers(...languages...)

  This takes a list of strings (which are presumed to be language-tags; strings that aren't, are ignored); and after each one, this function inserts super-ordinate forms that don't already appear in the list. The original list, plus these insertions, is returned.

  In other words, it takes this:

```
pt-br de-DE en-US fr pt-br-janeiro
```

  and returns this:

```
pt-br pt de-DE de en-US en fr pt-br-janeiro
```

  This function is most useful in the idiom

```
implicate_supers( I18N::LangTags::Detect::detect() );
```

  (See *I18N::LangTags::Detect*.)

- the function implicate_supers_strictly(...languages...)

  This works like `implicate_supers` except that the implicated forms are added to the end of the return list.

  In other words, implicate_supers_strictly takes a list of strings (which are presumed to be language-tags; strings that aren't, are ignored) and after the whole given list, it inserts the super-ordinate forms of all given tags, minus any tags that already appear in the input list.

  In other words, it takes this:

```
pt-br de-DE en-US fr pt-br-janeiro
```

  and returns this:

```
pt-br de-DE en-US fr pt-br-janeiro pt de en
```

  The reason this function has "_strictly" in its name is that when you're processing an Accept-Language list according to the RFCs, if you interpret the RFCs quite strictly, then you would use implicate_supers_strictly, but for normal use (i.e., common-sense use, as far as I'm concerned) you'd use implicate_supers.

ABOUT LOWERCASING

I've considered making all the above functions that output language tags return all those tags strictly in lowercase. Having all your language tags in lowercase does make some things easier. But you might as well just lowercase as you like, or call `encode_language_tag($lang1)` where appropriate.

ABOUT UNICODE PLAINTEXT LANGUAGE TAGS

In some future version of I18N::LangTags, I plan to include support for RFC2482-style language tags – which are basically just normal language tags with their ASCII characters shifted into Plane 14.

SEE ALSO

* *I18N::LangTags::List*

* RFC 3066, http://www.ietf.org/rfc/rfc3066.txt, "Tags for the Identification of Languages". (Obsoletes RFC 1766)

* RFC 2277, http://www.ietf.org/rfc/rfc2277.txt, "IETF Policy on Character Sets and Languages".

* RFC 2231, http://www.ietf.org/rfc/rfc2231.txt, "MIME Parameter Value and Encoded Word Extensions: Character Sets, Languages, and Continuations".

* RFC 2482, http://www.ietf.org/rfc/rfc2482.txt, "Language Tagging in Unicode Plain Text".

* Locale::Codes, in http://www.perl.com/CPAN/modules/by-module/Locale/

* ISO 639-2, "Codes for the representation of names of languages", including two-letter and three-letter codes, http://www.loc.gov/standards/iso639-2/php/code_list.php

* The IANA list of registered languages (hopefully up-to-date), http://www.iana.org/assignments/language-tags

COPYRIGHT

Copyright (c) 1998+ Sean M. Burke. All rights reserved.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

The programs and documentation in this dist are distributed in the hope that they will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Sean M. Burke `sburke@cpan.org`

Modules

- `I18N::LangTags::Detect` - detect the user's language preferences
- `I18N::LangTags::List` - tags and names for human languages

I18N::LangTags::Detect

detect the user's language preferences

SYNOPSIS

```
use I18N::LangTags::Detect;
my @user_wants = I18N::LangTags::Detect::detect();
```

DESCRIPTION

It is a common problem to want to detect what language(s) the user would prefer output in.

FUNCTIONS

This module defines one public function, `I18N::LangTags::Detect::detect()`. This function is not exported (nor is even exportable), and it takes no parameters.

In scalar context, the function returns the most preferred language tag (or undef if no preference was seen).

In list context (which is usually what you want), the function returns a (possibly empty) list of language tags representing (best first) what languages the user apparently would accept output in. You will probably want to pass the output of this through `I18N::LangTags::implicate_supers_tightly(...)` or `I18N::LangTags::implicate_supers(...)`, like so:

```
my @languages =
  I18N::LangTags::implicate_supers_tightly(
    I18N::LangTags::Detect::detect()
  );
```

ENVIRONMENT

This module looks at several environment variables: REQUEST_METHOD, HTTP_ACCEPT_LANGUAGE, LANGUAGE, LC_ALL, LC_MESSAGES, and LANG.

It will also use the Win32::Locale module, if it's installed and IGNORE_WIN32_LOCALE is not set to a true value in the environment.

SEE ALSO

I18N::LangTags, Win32::Locale, *Locale::Maketext*.

(This module's core code started out as a routine in Locale::Maketext; but I moved it here once I realized it was more generally useful.)

COPYRIGHT

Copyright (c) 1998-2004 Sean M. Burke. All rights reserved.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

The programs and documentation in this dist are distributed in the hope that they will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Sean M. Burke `sburke@cpan.org`

I18N::LangTags::List

tags and names for human languages

SYNOPSIS

```
use I18N::LangTags::List;
print "Parlez-vous... ", join(', ',
    I18N::LangTags::List::name('elx') || 'unknown_language',
    I18N::LangTags::List::name('ar-Kw') || 'unknown_language',
    I18N::LangTags::List::name('en') || 'unknown_language',
    I18N::LangTags::List::name('en-CA') || 'unknown_language',
  ), "?\n";
```

prints:

```
Parlez-vous... Elamite, Kuwait Arabic, English, Canadian English?
```

DESCRIPTION

This module provides a function `I18N::LangTags::List::name( _langtag_ )` that takes a language tag (see *I18N::LangTags*) and returns the best attempt at an English name for it, or undef if it can't make sense of the tag.

The function I18N::LangTags::List::name(...) is not exported.

This module also provides a function `I18N::LangTags::List::is_decent( _langtag_ )` that returns true iff the language tag is syntactically valid and is for general use (like "fr" or "fr-ca", below). That is, it returns false for tags that are syntactically invalid and for tags, like "aus", that are listed in brackets below. This function is not exported.

The map of tags-to-names that it uses is accessible as

that follows in this documentation, which should be useful to you even if you don't use this module.

ABOUT LANGUAGE TAGS

Internet language tags, as defined in RFC 3066, are a formalism for denoting human languages. The two-letter ISO 639-1 language codes are well known (as "en" for English), as are their forms when qualified by a country code ("en-US"). Less well-known are the arbitrary-length non-ISO codes (like "i-mingo"), and the recently (in 2001) introduced three-letter ISO-639-2 codes.

Remember these important facts:

- Language tags are not locale IDs. A locale ID is written with a "_" instead of a "-", (almost?) always matches `m/^\w\w_\w\w\b/`, and *means* something different than a language tag. A language tag denotes a language. A locale ID denotes a language *as used in* a particular place, in combination with non-linguistic location-specific information such as what currency is used there. Locales *also* often denote character set information, as in "en_US.ISO8859-1".
- Language tags are not for computer languages.
- "Dialect" is not a useful term, since there is no objective criterion for establishing when two language-forms are dialects of eachother, or are separate languages.
- Language tags are not case-sensitive. en-US, en-us, En-Us, etc., are all the same tag, and denote the same language.
- Not every language tag really refers to a single language. Some language tags refer to conditions: i-default (system-message text in English plus maybe other languages), und (undetermined language). Others (notably lots of the three-letter codes) are bibliographic tags that classify whole groups of languages, as with cus "Cushitic (Other)" (i.e., a language that has been classed as Cushtic, but which has no more specific code) or the even less linguistically coherent sai for "South American Indian (Other)". Though useful in bibliography, **SUCH TAGS ARE NOT FOR GENERAL USE**. For further guidance, email me.
- Language tags are not country codes. In fact, they are often distinct codes, as with language tag ja for Japanese, and ISO 3166 country code `.jp` for Japan.

LIST OF LANGUAGES

The first part of each item is the language tag, between {...}. It is followed by an English name for the language or language-group. Language tags that I judge to be not for general use, are bracketed.

This list is in alphabetical order by English name of the language.

- {ab} : Abkhazian

  eq Abkhaz
- {ace} : Achinese
- {ach} : Acoli

- {ada} : Adangme
- {ady} : Adyghe

  eq Adygei
- {aa} : Afar
- {afh} : Afrihili

  (Artificial)
- {af} : Afrikaans
- [{afa} : Afro-Asiatic (Other)]
- {ak} : Akan

  (Formerly “aka”.)
- {akk} : Akkadian

  (Historical)
- {sq} : Albanian
- {ale} : Aleut
- [{alg} : Algonquian languages]

  NOT Algonquin!
- [{tut} : Altaic (Other)]
- {am} : Amharic

  NOT Aramaic!
- {i-ami} : Ami

  eq Amis. eq ‘Amis. eq Pangca.
- [{apa} : Apache languages]
- {ar} : Arabic

  Many forms are mutually un-intelligible in spoken media. Notable forms: {ar-ae} UAE Arabic; {ar-bh} Bahrain Arabic; {ar-dz} Algerian Arabic; {ar-eg} Egyptian Arabic; {ar-iq} Iraqi Arabic; {ar-jo} Jordanian Arabic; {ar-kw} Kuwait Arabic; {ar-lb} Lebanese Arabic; {ar-ly} Libyan Arabic; {ar-ma} Moroccan Arabic; {ar-om} Omani Arabic; {ar-qa} Qatari Arabic; {ar-sa} Sauda Arabic; {ar-sy} Syrian Arabic; {ar-tn} Tunisian Arabic; {ar-ye} Yemen Arabic.
- {arc} : Aramaic

  NOT Amharic! NOT Samaritan Aramaic!
- {arp} : Arapaho
- {arn} : Araucanian
- {arw} : Arawak
- {hy} : Armenian
- {an} : Aragonese

- [{art} : Artificial (Other)]
- {ast} : Asturian

  eq Bable.
- {as} : Assamese
- [{ath} : Athapascan languages]

  eq Athabaskan. eq Athapaskan. eq Athabascan.
- [{aus} : Australian languages]
- [{map} : Austronesian (Other)]
- {av} : Avaric

  (Formerly “ava”.)
- {ae} : Avestan

  eq Zend
- {awa} : Awadhi
- {ay} : Aymara
- {az} : Azerbaijani

  eq Azeri

  Notable forms: {az-arab} Azerbaijani in Arabic script; {az-cyrl} Azerbaijani in Cyrillic script; {az-latn} Azerbaijani in Latin script.
- {ban} : Balinese
- [{bat} : Baltic (Other)]
- {bal} : Baluchi
- {bm} : Bambara

  (Formerly “bam”.)
- [{bai} : Bamileke languages]
- {bad} : Banda
- [{bnt} : Bantu (Other)]
- {bas} : Basa
- {ba} : Bashkir
- {eu} : Basque
- {btk} : Batak (Indonesia)
- {bej} : Beja
- {be} : Belarusian

  eq Belarussian. eq Byelarussian. eq Belorussian. eq Byelorussian. eq White Russian. eq White Ruthenian. NOT Ruthenian!
- {bem} : Bemba

- {bn} : Bengali

  eq Bangla.
- [{ber} : Berber (Other)]
- {bho} : Bhojpuri
- {bh} : Bihari
- {bik} : Bikol
- {bin} : Bini
- {bi} : Bislama

  eq Bichelamar.
- {bs} : Bosnian
- {bra} : Braj
- {br} : Breton
- {bug} : Buginese
- {bg} : Bulgarian
- {i-bnn} : Bunun
- {bua} : Buriat
- {my} : Burmese
- {cad} : Caddo
- {car} : Carib
- {ca} : Catalan

  eq Catalán. eq Catalonian.
- [{cau} : Caucasian (Other)]
- {ceb} : Cebuano
- [{cel} : Celtic (Other)]

  Notable forms: {cel-gaulish} Gaulish (Historical)
- [{cai} : Central American Indian (Other)]
- {chg} : Chagatai

  (Historical?)
- [{cmc} : Chamic languages]
- {ch} : Chamorro
- {ce} : Chechen
- {chr} : Cherokee

  eq Tsalagi
- {chy} : Cheyenne

- {chb} : Chibcha

  (Historical) NOT Chibchan (which is a language family).

- {ny} : Chichewa

  eq Nyanja. eq Chinyanja.

- {zh} : Chinese

  Many forms are mutually un-intelligible in spoken media. Notable forms: {zh-hans} Chinese, in simplified script; {zh-hant} Chinese, in traditional script; {zh-tw} Taiwan Chinese; {zh-cn} PRC Chinese; {zh-sg} Singapore Chinese; {zh-mo} Macau Chinese; {zh-hk} Hong Kong Chinese; {zh-guoyu} Mandarin [Putonghua/Guoyu]; {zh-hakka} Hakka [formerly "i-hakka"]; {zh-min} Hokkien; {zh-min-nan} Southern Hokkien; {zh-wuu} Shanghaiese; {zh-xiang} Hunanese; {zh-gan} Gan; {zh-yue} Cantonese.

- {chn} : Chinook Jargon

  eq Chinook Wawa.

- {chp} : Chipewyan
- {cho} : Choctaw
- {cu} : Church Slavic

  eq Old Church Slavonic.

- {chk} : Chuukese

  eq Trukese. eq Chuuk. eq Truk. eq Ruk.

- {cv} : Chuvash
- {cop} : Coptic
- {kw} : Cornish
- {co} : Corsican

  eq Corse.

- {cr} : Cree

  NOT Creek! (Formerly "cre".)

- {mus} : Creek

  NOT Cree!

- [{cpe} : English-based Creoles and pidgins (Other)]
- [{cpf} : French-based Creoles and pidgins (Other)]
- [{cpp} : Portuguese-based Creoles and pidgins (Other)]
- [{crp} : Creoles and pidgins (Other)]
- {hr} : Croatian

  eq Croat.

- [{cus} : Cushitic (Other)]

- {cs} : Czech
- {dak} : Dakota

  eq Nakota. eq Latoka.
- {da} : Danish
- {dar} : Dargwa
- {day} : Dayak
- {i-default} : Default (Fallthru) Language

  Defined in RFC 2277, this is for tagging text (which must include English text, and might/should include text in other appropriate languages) that is emitted in a context where language-negotiation wasn't possible – in SMTP mail failure messages, for example.
- {del} : Delaware
- {din} : Dinka
- {dv} : Divehi

  eq Maldivian. (Formerly "div".)
- {doi} : Dogri

  NOT Dogrib!
- {dgr} : Dogrib

  NOT Dogri!
- [{dra} : Dravidian (Other)]
- {dua} : Duala
- {nl} : Dutch

  eq Netherlander. Notable forms: {nl-nl} Netherlands Dutch; {nl-be} Belgian Dutch.
- {dum} : Middle Dutch (ca.1050-1350)

  (Historical)
- {dyu} : Dyula
- {dz} : Dzongkha
- {efi} : Efik
- {egy} : Ancient Egyptian

  (Historical)
- {eka} : Ekajuk
- {elx} : Elamite

  (Historical)
- {en} : English

  Notable forms: {en-au} Australian English; {en-bz} Belize English; {en-ca} Canadian English; {en-gb} UK English; {en-ie} Irish English; {en-jm} Jamaican English;

{en-nz} New Zealand English; {en-ph} Philippine English; {en-tt} Trinidad English; {en-us} US English; {en-za} South African English; {en-zw} Zimbabwe English.

- {enm} : Old English (1100-1500)

  (Historical)

- {ang} : Old English (ca.450-1100)

  eq Anglo-Saxon. (Historical)

- {i-enochian} : Enochian (Artificial)
- {myv} : Erzya
- {eo} : Esperanto

  (Artificial)

- {et} : Estonian
- {ee} : Ewe

  (Formerly “ewe”.)

- {ewo} : Ewondo
- {fan} : Fang
- {fat} : Fanti
- {fo} : Faroese
- {fj} : Fijian
- {fi} : Finnish
- [{fiu} : Finno-Ugrian (Other)]

  eq Finno-Ugric. NOT Ugaritic!

- {fon} : Fon
- {fr} : French

  Notable forms: {fr-fr} France French; {fr-be} Belgian French; {fr-ca} Canadian French; {fr-ch} Swiss French; {fr-lu} Luxembourg French; {fr-mc} Monaco French.

- {frm} : Middle French (ca.1400-1600)

  (Historical)

- {fro} : Old French (842-ca.1400)

  (Historical)

- {fy} : Frisian
- {fur} : Friulian
- {ff} : Fulah

  (Formerly “ful”.)

- {gaa} : Ga

- {gd} : Scots Gaelic

  NOT Scots!

- {gl} : Gallegan

  eq Galician

- {lg} : Ganda

  (Formerly “lug”.)

- {gay} : Gayo
- {gba} : Gbaya
- {gez} : Geez

  eq Ge’ez

- {ka} : Georgian
- {de} : German

  Notable forms: {de-at} Austrian German; {de-be} Belgian German; {de-ch} Swiss German; {de-de} Germany German; {de-li} Liechtenstein German; {de-lu} Luxembourg German.

- {gmh} : Middle High German (ca.1050-1500)

  (Historical)

- {goh} : Old High German (ca.750-1050)

  (Historical)

- [{gem} : Germanic (Other)]
- {gil} : Gilbertese
- {gon} : Gondi
- {gor} : Gorontalo
- {got} : Gothic

  (Historical)

- {grb} : Grebo
- {grc} : Ancient Greek

  (Historical) (Until 15th century or so.)

- {el} : Modern Greek

  (Since 15th century or so.)

- {gn} : Guarani

  Guaraní

- {gu} : Gujarati
- {gwi} : Gwich’in

  eq Gwichin

- {hai} : Haida
- {ht} : Haitian

  eq Haitian Creole
- {ha} : Hausa
- {haw} : Hawaiian

  Hawai'ian
- {he} : Hebrew

  (Formerly "iw".)
- {hz} : Herero
- {hil} : Hiligaynon
- {him} : Himachali
- {hi} : Hindi
- {ho} : Hiri Motu
- {hit} : Hittite

  (Historical)
- {hmn} : Hmong
- {hu} : Hungarian
- {hup} : Hupa
- {iba} : Iban
- {is} : Icelandic
- {io} : Ido

  (Artificial)
- {ig} : Igbo

  (Formerly "ibo".)
- {ijo} : Ijo
- {ilo} : Iloko
- [{inc} : Indic (Other)]
- [{ine} : Indo-European (Other)]
- {id} : Indonesian

  (Formerly "in".)
- {inh} : Ingush
- {ia} : Interlingua (International Auxiliary Language Association)

  (Artificial) NOT Interlingue!

- {ie} : Interlingue

  (Artificial) NOT Interlingua!
- {iu} : Inuktitut

  A subform of “Eskimo”.
- {ik} : Inupiaq

  A subform of “Eskimo”.
- [{ira} : Iranian (Other)]
- {ga} : Irish
- {mga} : Middle Irish (900-1200)

  (Historical)
- {sga} : Old Irish (to 900)

  (Historical)
- [{iro} : Iroquoian languages]
- {it} : Italian

  Notable forms: {it-it} Italy Italian; {it-ch} Swiss Italian.
- {ja} : Japanese

  (NOT “jp”!)
- {jv} : Javanese

  (Formerly “jw” because of a typo.)
- {jrb} : Judeo-Arabic
- {jpr} : Judeo-Persian
- {kbd} : Kabardian
- {kab} : Kabyle
- {kac} : Kachin
- {kl} : Kalaallisut

  eq Greenlandic “Eskimo”
- {xal} : Kalmyk
- {kam} : Kamba
- {kn} : Kannada

  eq Kanarese. NOT Canadian!
- {kr} : Kanuri

  (Formerly “kau”.)
- {krc} : Karachay-Balkar
- {kaa} : Kara-Kalpak

- {kar} : Karen
- {ks} : Kashmiri
- {csb} : Kashubian

  eq Kashub
- {kaw} : Kawi
- {kk} : Kazakh
- {kha} : Khasi
- {km} : Khmer

  eq Cambodian. eq Kampuchean.
- [{khi} : Khoisan (Other)]
- {kho} : Khotanese
- {ki} : Kikuyu

  eq Gikuyu.
- {kmb} : Kimbundu
- {rw} : Kinyarwanda
- {ky} : Kirghiz
- {i-klingon} : Klingon
- {kv} : Komi
- {kg} : Kongo

  (Formerly “kon”.)
- {kok} : Konkani
- {ko} : Korean
- {kos} : Kosraean
- {kpe} : Kpelle
- {kro} : Kru
- {kj} : Kuanyama
- {kum} : Kumyk
- {ku} : Kurdish
- {kru} : Kurukh
- {kut} : Kutenai
- {lad} : Ladino

  eq Judeo-Spanish. NOT Ladin (a minority language in Italy).
- {lah} : Lahnda

  NOT Lamba!

- {lam} : Lamba

  NOT Lahnda!
- {lo} : Lao

  eq Laotian.
- {la} : Latin

  (Historical) NOT Ladin! NOT Ladino!
- {lv} : Latvian

  eq Lettish.
- {lb} : Letzeburgesch

  eq Luxemburgian, eq Luxemburger. (Formerly “i-lux”.)
- {lez} : Lezghian
- {li} : Limburgish

  eq Limburger, eq Limburgan. NOT Letzeburgesch!
- {ln} : Lingala
- {lt} : Lithuanian
- {nds} : Low German

  eq Low Saxon. eq Low German. eq Low Saxon.
- {art-lojban} : Lojban (Artificial)
- {loz} : Lozi
- {lu} : Luba-Katanga

  (Formerly “lub”.)
- {lua} : Luba-Lulua
- {lui} : Luiseno

  eq Luiseño.
- {lun} : Lunda
- {luo} : Luo (Kenya and Tanzania)
- {lus} : Lushai
- {mk} : Macedonian

  eq the modern Slavic language spoken in what was Yugoslavia. NOT the form of Greek spoken in Greek Macedonia!
- {mad} : Madurese
- {mag} : Magahi
- {mai} : Maithili
- {mak} : Makasar
- {mg} : Malagasy

- {ms} : Malay

  NOT Malayalam!
- {ml} : Malayalam

  NOT Malay!
- {mt} : Maltese
- {mnc} : Manchu
- {mdr} : Mandar

  NOT Mandarin!
- {man} : Mandingo
- {mni} : Manipuri

  eq Meithei.
- [{mno} : Manobo languages]
- {gv} : Manx
- {mi} : Maori

  NOT Mari!
- {mr} : Marathi
- {chm} : Mari

  NOT Maori!
- {mh} : Marshall

  eq Marshallese.
- {mwr} : Marwari
- {mas} : Masai
- [{myn} : Mayan languages]
- {men} : Mende
- {mic} : Micmac
- {min} : Minangkabau
- {i-mingo} : Mingo

  eq the Irquoian language West Virginia Seneca. NOT New York Seneca!
- [{mis} : Miscellaneous languages]

  Don't use this.
- {moh} : Mohawk
- {mdf} : Moksha
- {mo} : Moldavian

  eq Moldovan.

- [{mkh} : Mon-Khmer (Other)]
- {lol} : Mongo
- {mn} : Mongolian

  eq Mongol.
- {mos} : Mossi
- [{mul} : Multiple languages]

  Not for normal use.
- [{mun} : Munda languages]
- {nah} : Nahuatl
- {nap} : Neapolitan
- {na} : Nauru
- {nv} : Navajo

  eq Navaho. (Formerly “i-navajo”.)
- {nd} : North Ndebele
- {nr} : South Ndebele
- {ng} : Ndonga
- {ne} : Nepali

  eq Nepalese. Notable forms: {ne-np} Nepal Nepali; {ne-in} India Nepali.
- {new} : Newari
- {nia} : Nias
- [{nic} : Niger-Kordofanian (Other)]
- [{ssa} : Nilo-Saharan (Other)]
- {niu} : Niuean
- {nog} : Nogai
- {non} : Old Norse

  (Historical)
- [{nai} : North American Indian]

  Do not use this.
- {no} : Norwegian

  Note the two following forms:
- {nb} : Norwegian Bokmal

  eq Bokmål, (A form of Norwegian.) (Formerly “no-bok”.)
- {nn} : Norwegian Nynorsk

  (A form of Norwegian.) (Formerly “no-nyn”.)

- [{nub} : Nubian languages]
- {nym} : Nyamwezi
- {nyn} : Nyankole
- {nyo} : Nyoro
- {nzi} : Nzima
- {oc} : Occitan (post 1500)

  eq Provençal, eq Provencal
- {oj} : Ojibwa

  eq Ojibwe. (Formerly “oji”.)
- {or} : Oriya
- {om} : Oromo
- {osa} : Osage
- {os} : Ossetian; Ossetic
- [{oto} : Otomian languages]

  Group of languages collectively called “Otomí”.
- {pal} : Pahlavi

  eq Pahlevi
- {i-pwn} : Paiwan

  eq Pariwan
- {pau} : Palauan
- {pi} : Pali

  (Historical?)
- {pam} : Pampanga
- {pag} : Pangasinan
- {pa} : Panjabi

  eq Punjabi
- {pap} : Papiamento

  eq Papiamentu.
- [{paa} : Papuan (Other)]
- {fa} : Persian

  eq Farsi. eq Iranian.
- {peo} : Old Persian (ca.600-400 B.C.)
- [{phi} : Philippine (Other)]

- {phn} : Phoenician

  (Historical)

- {pon} : Pohnpeian

  NOT Pompeiian!

- {pl} : Polish

- {pt} : Portuguese

  eq Portugese. Notable forms: {pt-pt} Portugal Portuguese; {pt-br} Brazilian Portuguese.

- [{pra} : Prakrit languages]

- {pro} : Old Provencal (to 1500)

  eq Old Provençal. (Historical.)

- {ps} : Pushto

  eq Pashto. eq Pushtu.

- {qu} : Quechua

  eq Quecha.

- {rm} : Raeto-Romance

  eq Romansh.

- {raj} : Rajasthani

- {rap} : Rapanui

- {rar} : Rarotongan

- [{qaa - qtz} : Reserved for local use.]

- [{roa} : Romance (Other)]

  NOT Romanian! NOT Romany! NOT Romansh!

- {ro} : Romanian

  eq Rumanian. NOT Romany!

- {rom} : Romany

  eq Rom. NOT Romanian!

- {rn} : Rundi

- {ru} : Russian

  NOT White Russian! NOT Rusyn!

- [{sal} : Salishan languages]

  Large language group.

- {sam} : Samaritan Aramaic

  NOT Aramaic!

- {se} : Northern Sami

  eq Lappish. eq Lapp. eq (Northern) Saami.
- {sma} : Southern Sami
- {smn} : Inari Sami
- {smj} : Lule Sami
- {sms} : Skolt Sami
- [{smi} : Sami languages (Other)]
- {sm} : Samoan
- {sad} : Sandawe
- {sg} : Sango
- {sa} : Sanskrit

  (Historical)
- {sat} : Santali
- {sc} : Sardinian

  eq Sard.
- {sas} : Sasak
- {sco} : Scots

  NOT Scots Gaelic!
- {sel} : Selkup
- [{sem} : Semitic (Other)]
- {sr} : Serbian

  eq Serb. NOT Sorbian.

  Notable forms: {sr-cyrl} : Serbian in Cyrillic script; {sr-latn} : Serbian in Latin script.
- {srr} : Serer
- {shn} : Shan
- {sn} : Shona
- {sid} : Sidamo
- {sgn-...} : Sign Languages

  Always use with a subtag. Notable forms: {sgn-gb} British Sign Language (BSL); {sgn-ie} Irish Sign Language (ESL); {sgn-ni} Nicaraguan Sign Language (ISN); {sgn-us} American Sign Language (ASL).

  (And so on with other country codes as the subtag.)
- {bla} : Siksika

  eq Blackfoot. eq Pikanii.

- {sd} : Sindhi
- {si} : Sinhalese

  eq Sinhala.
- [{sit} : Sino-Tibetan (Other)]
- [{sio} : Siouan languages]
- {den} : Slave (Athapascan)

  ("Slavey" is a subform.)
- [{sla} : Slavic (Other)]
- {sk} : Slovak

  eq Slovakian.
- {sl} : Slovenian

  eq Slovene.
- {sog} : Sogdian
- {so} : Somali
- {son} : Songhai
- {snk} : Soninke
- {wen} : Sorbian languages

  eq Wendish. eq Sorb. eq Lusatian. eq Wend. NOT Venda! NOT Serbian!
- {nso} : Northern Sotho
- {st} : Southern Sotho

  eq Sutu. eq Sesotho.
- [{sai} : South American Indian (Other)]
- {es} : Spanish

  Notable forms: {es-ar} Argentine Spanish; {es-bo} Bolivian Spanish; {es-cl} Chilean Spanish; {es-co} Colombian Spanish; {es-do} Dominican Spanish; {es-ec} Ecuadorian Spanish; {es-es} Spain Spanish; {es-gt} Guatemalan Spanish; {es-hn} Honduran Spanish; {es-mx} Mexican Spanish; {es-pa} Panamanian Spanish; {es-pe} Peruvian Spanish; {es-pr} Puerto Rican Spanish; {es-py} Paraguay Spanish; {es-sv} Salvadoran Spanish; {es-us} US Spanish; {es-uy} Uruguayan Spanish; {es-ve} Venezuelan Spanish.
- {suk} : Sukuma
- {sux} : Sumerian

  (Historical)
- {su} : Sundanese
- {sus} : Susu

- {sw} : Swahili

  eq Kiswahili
- {ss} : Swati
- {sv} : Swedish

  Notable forms: {sv-se} Sweden Swedish; {sv-fi} Finland Swedish.
- {syr} : Syriac
- {tl} : Tagalog
- {ty} : Tahitian
- [{tai} : Tai (Other)]

  NOT Thai!
- {tg} : Tajik
- {tmh} : Tamashek
- {ta} : Tamil
- {i-tao} : Tao

  eq Yami.
- {tt} : Tatar
- {i-tay} : Tayal

  eq Atayal. eq Atayan.
- {te} : Telugu
- {ter} : Tereno
- {tet} : Tetum
- {th} : Thai

  NOT Tai!
- {bo} : Tibetan
- {tig} : Tigre
- {ti} : Tigrinya
- {tem} : Timne

  eq Themne. eq Timene.
- {tiv} : Tiv
- {tli} : Tlingit
- {tpi} : Tok Pisin
- {tkl} : Tokelau
- {tog} : Tonga (Nyasa)

  NOT Tsonga!

- {to} : Tonga (Tonga Islands)

  (Pronounced “Tong-a”, not “Tong-ga”)

  NOT Tsonga!
- {tsi} : Tsimshian

  eq Sm’algyax
- {ts} : Tsonga

  NOT Tonga!
- {i-tsu} : Tsou
- {tn} : Tswana

  Same as Setswana.
- {tum} : Tumbuka
- [{tup} : Tupi languages]
- {tr} : Turkish

  (Typically in Roman script)
- {ota} : Ottoman Turkish (1500-1928)

  (Typically in Arabic script) (Historical)
- {crh} : Crimean Turkish

  eq Crimean Tatar
- {tk} : Turkmen

  eq Turkmeni.
- {tvl} : Tuvalu
- {tyv} : Tuvinian

  eq Tuvan. eq Tuvin.
- {tw} : Twi
- {udm} : Udmurt
- {uga} : Ugaritic

  NOT Ugric!
- {ug} : Uighur
- {uk} : Ukrainian
- {umb} : Umbundu
- {und} : Undetermined

  Not a tag for normal use.
- {ur} : Urdu

- {uz} : Uzbek

  eq Özbek

  Notable forms: {uz-cyrl} Uzbek in Cyrillic script; {uz-latn} Uzbek in Latin script.
- {vai} : Vai
- {ve} : Venda

  NOT Wendish! NOT Wend! NOT Avestan! (Formerly “ven”.)
- {vi} : Vietnamese

  eq Viet.
- {vo} : Volapuk

  eq Volapük. (Artificial)
- {vot} : Votic

  eq Votian. eq Vod.
- [{wak} : Wakashan languages]
- {wa} : Walloon
- {wal} : Walamo

  eq Wolaytta.
- {war} : Waray

  Presumably the Philippine language Waray-Waray (Samareño), not the smaller Philippine language Waray Sorsogon, nor the extinct Australian language Waray.
- {was} : Washo

  eq Washoe
- {cy} : Welsh
- {wo} : Wolof
- {x-...} : Unregistered (Semi-Private Use)

  “x-” is a prefix for language tags that are not registered with ISO or IANA. Example, x-double-dutch
- {xh} : Xhosa
- {sah} : Yakut
- {yao} : Yao

  (The Yao in Malawi?)
- {yap} : Yapese

  eq Yap
- {ii} : Sichuan Yi
- {yi} : Yiddish

  Formerly “ji”. Usually in Hebrew script.

Notable forms: {yi-latn} Yiddish in Latin script

- {yo} : Yoruba
- [{ypk} : Yupik languages]

  Several “Eskimo” languages.
- {znd} : Zande
- [{zap} : Zapotec]

  (A group of languages.)
- {zen} : Zenaga

  NOT Zend.
- {za} : Zhuang
- {zu} : Zulu
- {zun} : Zuni

  eq Zuñi

SEE ALSO

I18N::LangTags and its “See Also” section.

COPYRIGHT AND DISCLAIMER

Copyright (c) 2001+ Sean M. Burke. All rights reserved.

You can redistribute and/or modify this document under the same terms as Perl itself.

This document is provided in the hope that it will be useful, but without any warranty; without even the implied warranty of accuracy, authoritativeness, completeness, merchantability, or fitness for a particular purpose.

Email any corrections or questions to me.

AUTHOR

Sean M. Burke, sburke@cpan.org

I18N::Langinfo

Query locale information - country, language, and cultural conventions from the current locale.

Exports `langinfo()` together with the item constants it takes: the character set name, the date and time formats, the day and month names, the numeric and currency separators, the era formats, the locale’s yes/no expressions, and glibc’s `_NL_*` extension items (postal address, identification, measurement, name, paper and telephone formats).

```
use I18N::Langinfo qw(langinfo CODESET DAY_1 MON_1 RADIXCHAR␣
 ↪YESEXPR);
use POSIX qw(setlocale LC_ALL);

setlocale(LC_ALL, "C");
printf "%s | %s %s | radix '%s' | yes %s\n",
    langinfo(CODESET), langinfo(DAY_1), langinfo(MON_1),␣
 ↪langinfo(RADIXCHAR), langinfo(YESEXPR);
# ANSI_X3.4-1968 | Sunday January | radix '.' | yes ^[yY]
```

The answers are the C library's for the locale in effect, so they change with `setlocale`. This is a native implementation of the XS half of I18N::Langinfo 0.24; its `.pm` is used unchanged, and `use I18N::Langinfo` exports `langinfo` alone - the item constants are imported by name.

Differences from upstream

None.

Functions

Other Functions

`langinfo`

`langinfo($item)` - the string the current locale gives for one `nl_langinfo` item code.

langinfo

`langinfo($item)` - the string the current locale gives for one `nl_langinfo` item code.

`$item` is one of the constants this module exports - `CODESET`, `D_FMT`, `T_FMT`, `DAY_1` .. `DAY_7`, `MON_1` .. `MON_12`, `RADIXCHAR`, `THOUSEP`, `CRNCYSTR`, `YESEXPR` and the rest. The answer comes from the locale in effect, so it follows `setlocale`; `RADIXCHAR` and `THOUSEP` come from the underlying `LC_NUMERIC` locale even though Perl keeps the radix a dot. A string that is UTF-8 in a UTF-8 locale comes back with the UTF-8 flag on. Called with no argument it uses `$_`. An item the C library has nothing for yields the empty string.

5.2.57 IO

Load the core IO modules - `IO::Handle`, `IO::File`, `IO::Seekable`, `IO::Pipe`, `IO::Socket`, `IO::Dir` - in one `use`.

This is the module every `IO::` class rests on: `IO::Handle`, `IO::File`, `IO::Seekable`, `IO::Dir`, `IO::Socket` and `IO::Pipe` all require it, so `use IO;` is enough to have the whole family available.

`IO` itself defines nothing you call as `IO::something`. Every method lives in the class it belongs to - `IO::Seekable`, `IO::File`, `IO::Handle`, `IO::Socket`, `IO::Poll` - and `use IO qw(File Handle)` simply loads the named subset.

It also defines the numeric constants those classes need: `POLLIN`, `POLLPRI`, `POLLOUT`, `POLLRDNORM`, `POLLWRNORM`, `POLLRDBAND`, `POLLWRBAND`, `POLLNORM`, `POLLERR`, `POLLHUP`

and POLLNVAL in IO::Poll, and _IOFBF, _IOLBF, _IONBF, SEEK_SET, SEEK_CUR and SEEK_END in IO::Handle.

```
use strict;
use IO qw(Handle File Seekable);
use IO::Seekable;                  # exports SEEK_SET, SEEK_CUR,␣
 ↪SEEK_END

my $fh = IO::File->new($path, "w+") or die $!;
$fh->autoflush(1);
$fh->print("hello\n");
$fh->seek(0, SEEK_SET);
print "read back: ", $fh->getline;                # read back: hello
printf "SEEK_END %d, _IONBF %d\n", SEEK_END, IO::Handle::_IONBF;   #␣
 ↪2, 2
```

This is a native implementation of the XS half of IO 1.56 - the handle methods that need C (blocking, ungetc, error, clearerr, untaint, flush, sync, setbuf, setvbuf, getline/getlines/gets, IO::Seekable's getpos/setpos, IO::File->new_tmpfile, IO::Poll's _poll and IO::Socket's sockatmark). Every IO::* module's Perl is upstream's, unchanged.

Differences from upstream

None.

Modules

- IO::Compress - read/write compressed data in multiple formats
- IO::Dir - supply object methods for directory handles
- IO::File - supply object methods for filehandles
- IO::Handle - supply object methods for I/O handles
- IO::Pipe - supply object methods for pipes
- IO::Poll - Object interface to system poll call
- IO::Pty - Pseudo TTY object class
- IO::Seekable - supply seek based methods for I/O objects
- IO::Select - OO interface to the select system call
- IO::Socket - Object interface to socket communications
- IO::Tty - Pseudo-terminals and the terminal constants: the compiled part under IO::Pty (a pty master and its slave as handles) and IO::Tty.
- IO::Uncompress
- IO::Zlib - IO:: style interface to *Compress::Zlib*

IO::Compress

read/write compressed data in multiple formats

DESCRIPTION

This is a stub module. It contains no code.

AUTHOR

Paul Marquess `pmqs@cpan.org`.

COPYRIGHT

Copyright (c) 2011-2026 Paul Marquess. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

Modules

- `IO::Compress::Base` - Base Class for IO::Compress modules
- `IO::Compress::Bzip2` - Write bzip2 files/buffers
- `IO::Compress::Deflate` - Write RFC 1950 files/buffers
- `IO::Compress::Gzip` - Write RFC 1952 files/buffers
- `IO::Compress::RawDeflate` - Write RFC 1951 files/buffers
- `IO::Compress::Zip` - Write zip files/buffers

IO::Compress::Base

Base Class for IO::Compress modules

SYNOPSIS

```
use IO::Compress::Base ;
```

DESCRIPTION

This module is not intended for direct use in application code. Its sole purpose is to be sub-classed by IO::Compress modules.

SUPPORT

General feedback/questions/bug reports should be sent to https://github.com/pmqs/IO-Compress/issues (preferred) or https://rt.cpan.org/Public/Dist/Display.html?Name=IO-Compress.

SEE ALSO

Compress::Zlib, *IO::Compress::Gzip*, *IO::Uncompress::Gunzip*, *IO::Compress::Deflate*, *IO::Uncompress::Inflate*, *IO::Compress::RawDeflate*, *IO::Uncompress::RawInflate*, *IO::Compress::Bzip2*, *IO::Uncompress::Bunzip2*, IO::Compress::Lzma, IO::Uncompress::UnLzma, IO::Compress::Xz, IO::Uncompress::UnXz, IO::Compress::Lzip, IO::Uncompress::UnLzip, IO::Compress::Lzop, IO::Uncompress::UnLzop, IO::Compress::Lzf, IO::Uncompress::UnLzf, IO::Compress::Zstd, IO::Uncompress::UnZstd, *IO::Uncompress::AnyInflate*, *IO::Uncompress::AnyUncompress*

IO::Compress::FAQ

File::GlobMapper, Archive::Zip, *Archive::Tar*, *IO::Zlib*

AUTHOR

This module was written by Paul Marquess, `pmqs@cpan.org`.

MODIFICATION HISTORY

See the Changes file.

COPYRIGHT AND LICENSE

Copyright (c) 2005-2026 Paul Marquess. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

IO::Compress::Bzip2

Write bzip2 files/buffers

SYNOPSIS

```
use IO::Compress::Bzip2 qw(bzip2 $Bzip2Error) ;

my $status = bzip2 $input => $output [,OPTS]
    or die "bzip2 failed: $Bzip2Error\n";

my $z = IO::Compress::Bzip2->new( $output [,OPTS] )
    or die "bzip2 failed: $Bzip2Error\n";

$z->print($string);
$z->printf($format, $string);
$z->write($string);
$z->syswrite($string [, $length, $offset]);
$z->flush();
$z->tell();
$z->eof();
$z->seek($position, $whence);
```

(continues on next page)

(continued from previous page)

```
$z->binmode();
$z->fileno();
$z->opened();
$z->autoflush();
$z->input_line_number();
$z->newStream( [OPTS] );

$z->close() ;

$Bzip2Error ;

# IO::File mode

print $z $string;
printf $z $format, $string;
tell $z
eof $z
seek $z, $position, $whence
binmode $z
fileno $z
close $z ;
```

DESCRIPTION

This module provides a Perl interface that allows writing bzip2 compressed data to files or buffer.

For reading bzip2 files/buffers, see the companion module *IO::Uncompress::Bunzip2*.

Functional Interface

A top-level function, `bzip2`, is provided to carry out "one-shot" compression between buffers and/or files. For finer control over the compression process, see the "OO Interface" section.

```
use IO::Compress::Bzip2 qw(bzip2 $Bzip2Error) ;

bzip2 $input_filename_or_reference => $output_filename_or_reference␣
 ↪[,OPTS]
    or die "bzip2 failed: $Bzip2Error\n";
```

The functional interface needs Perl5.005 or better.

bzip2 $input_filename_or_reference => $output_filename_or_reference [, OPTS]

`bzip2` expects at least two parameters, `$input_filename_or_reference` and `$output_filename_or_reference` and zero or more optional parameters (see "Optional Parameters")

The `$input_filename_or_reference` parameter

The parameter, `$input_filename_or_reference`, is used to define the source of the uncompressed data.

It can take one of the following forms:

- A filename

  If the `$input_filename_or_reference` parameter is a simple scalar, it is assumed to be a filename. This file will be opened for reading and the input data will be read from it.

- A filehandle

  If the `$input_filename_or_reference` parameter is a filehandle, the input data will be read from it. The string '-' can be used as an alias for standard input.

- A scalar reference

  If `$input_filename_or_reference` is a scalar reference, the input data will be read from `$$input_filename_or_reference`.

- An array reference

  If `$input_filename_or_reference` is an array reference, each element in the array must be a filename.

  The input data will be read from each file in turn.

  The complete array will be walked to ensure that it only contains valid filenames before any data is compressed.

- An Input FileGlob string

  If `$input_filename_or_reference` is a string that is delimited by the characters "<" and ">" `bzip2` will assume that it is an *input fileglob string*. The input is the list of files that match the fileglob.

  See *File::GlobMapper* for more details.

If the `$input_filename_or_reference` parameter is any other type, `undef` will be returned.

The `$output_filename_or_reference` parameter

The parameter `$output_filename_or_reference` is used to control the destination of the compressed data. This parameter can take one of these forms.

- A filename

  If the `$output_filename_or_reference` parameter is a simple scalar, it is assumed to be a filename. This file will be opened for writing and the compressed data will be written to it.

- A filehandle

  If the `$output_filename_or_reference` parameter is a filehandle, the compressed data will be written to it. The string '-' can be used as an alias for standard output.

- A scalar reference

  If `$output_filename_or_reference` is a scalar reference, the compressed data will be stored in `$$output_filename_or_reference`.

- An Array Reference

  If `$output_filename_or_reference` is an array reference, the compressed data will be pushed onto the array.

- An Output FileGlob

  If `$output_filename_or_reference` is a string that is delimited by the characters "<" and ">" `bzip2` will assume that it is an *output fileglob string*. The output is the list of files that match the fileglob.

  When `$output_filename_or_reference` is an fileglob string, `$input_filename_or_reference` must also be a fileglob string. Anything else is an error.

  See *File::GlobMapper* for more details.

If the `$output_filename_or_reference` parameter is any other type, `undef` will be returned.

Notes

When `$input_filename_or_reference` maps to multiple files/buffers and `$output_filename_or_reference` is a single file/buffer the input files/buffers will be stored in `$output_filename_or_reference` as a concatenated series of compressed data streams.

Optional Parameters

The optional parameters for the one-shot function `bzip2` are (for the most part) identical to those used with the OO interface defined in the "Constructor Options" section. The exceptions are listed below

- `AutoClose => 0|1`

  This option applies to any input or output data streams to `bzip2` that are filehandles.

  If `AutoClose` is specified, and the value is true, it will result in all input and/or output filehandles being closed once `bzip2` has completed.

  This parameter defaults to 0.

- `BinModeIn => 0|1`

  This option is now a no-op. All files will be read in binmode.

- `Append => 0|1`

  The behaviour of this option is dependent on the type of output data stream.

  - A Buffer

    If Append is enabled, all compressed data will be append to the end of the output buffer. Otherwise the output buffer will be cleared before any compressed data is written to it.

- A Filename

  If Append is enabled, the file will be opened in append mode. Otherwise the contents of the file, if any, will be truncated before any compressed data is written to it.

- A Filehandle

  If Append is enabled, the filehandle will be positioned to the end of the file via a call to `seek` before any compressed data is written to it. Otherwise the file pointer will not be moved.

When Append is specified, and set to true, it will *append* all compressed data to the output data stream.

So when the output is a filehandle it will carry out a seek to the eof before writing any compressed data. If the output is a filename, it will be opened for appending. If the output is a buffer, all compressed data will be appended to the existing buffer.

Conversely when Append is not specified, or it is present and is set to false, it will operate as follows.

When the output is a filename, it will truncate the contents of the file before writing any compressed data. If the output is a filehandle its position will not be changed. If the output is a buffer, it will be wiped before any compressed data is output.

Defaults to 0.

Oneshot Examples

Here are a few example that show the capabilities of the module.

Streaming

This very simple command line example demonstrates the streaming capabilities of the module. The code reads data from STDIN, compresses it, and writes the compressed data to STDOUT.

```
$ echo hello world | perl -MIO::Compress::Bzip2=bzip2 -e 'bzip2 \
↪*STDIN => \*STDOUT' >output.bz2
```

The special filename "-" can be used as a standin for both `\*STDIN` and `\*STDOUT`, so the above can be rewritten as

```
$ echo hello world | perl -MIO::Compress::Bzip2=bzip2 -e 'bzip2 "-"␣
↪=> "-"' >output.bz2
```

Compressing a file from the filesystem

To read the contents of the file `file1.txt` and write the compressed data to the file `file1.txt.bz2`.

```
use strict ;
use warnings ;
```

(continues on next page)

(continued from previous page)

```
use IO::Compress::Bzip2 qw(bzip2 $Bzip2Error) ;

my $input = "file1.txt";
bzip2 $input => "$input.bz2"
    or die "bzip2 failed: $Bzip2Error\n";
```

Reading from a Filehandle and writing to an in-memory buffer

To read from an existing Perl filehandle, `$input`, and write the compressed data to a buffer, `$buffer`.

```
use strict ;
use warnings ;
use IO::Compress::Bzip2 qw(bzip2 $Bzip2Error) ;
use IO::File ;

my $input = IO::File->new( "<file1.txt" )
    or die "Cannot open 'file1.txt': $!\n" ;
my $buffer ;
bzip2 $input => \$buffer
    or die "bzip2 failed: $Bzip2Error\n";
```

Compressing multiple files

To compress all files in the directory “/my/home” that match “*.txt” and store the compressed data in the same directory

```
use strict ;
use warnings ;
use IO::Compress::Bzip2 qw(bzip2 $Bzip2Error) ;

bzip2 '</my/home/*.txt>' => '<*.bz2>'
    or die "bzip2 failed: $Bzip2Error\n";
```

and if you want to compress each file one at a time, this will do the trick

```
use strict ;
use warnings ;
use IO::Compress::Bzip2 qw(bzip2 $Bzip2Error) ;

for my $input ( glob "/my/home/*.txt" )
{
    my $output = "$input.bz2" ;
    bzip2 $input => $output
        or die "Error compressing '$input': $Bzip2Error\n";
}
```

OO Interface

Constructor

The format of the constructor for `IO::Compress::Bzip2` is shown below

```
my $z = IO::Compress::Bzip2->new( $output [,OPTS] )
    or die "IO::Compress::Bzip2 failed: $Bzip2Error\n";
```

The constructor takes one mandatory parameter, `$output`, defined below and zero or more `OPTS`, defined in "Constructor Options".

It returns an `IO::Compress::Bzip2` object on success and `undef` on failure. The variable `$Bzip2Error` will contain an error message on failure.

If you are running Perl 5.005 or better the object, `$z`, returned from IO::Compress::Bzip2 can be used exactly like an *IO::File* filehandle. This means that all normal output file operations can be carried out with `$z`. For example, to write to a compressed file/buffer you can use either of these forms

```
$z->print("hello world\n");
print $z "hello world\n";
```

Below is a simple exaple of using the OO interface to create an output file `myfile.bz2` and write some data to it.

```
my $filename = "myfile.bz2";
my $z = IO::Compress::Bzip2->new($filename)
    or die "IO::Compress::Bzip2 failed: $Bzip2Error\n";

$z->print("abcde");
$z->close();
```

See the "Examples" for more.

The mandatory parameter `$output` is used to control the destination of the compressed data. This parameter can take one of these forms.

- A filename

  If the `$output` parameter is a simple scalar, it is assumed to be a filename. This file will be opened for writing and the compressed data will be written to it.

- A filehandle

  If the `$output` parameter is a filehandle, the compressed data will be written to it. The string '-' can be used as an alias for standard output.

- A scalar reference

  If `$output` is a scalar reference, the compressed data will be stored in `$$output`.

If the `$output` parameter is any other type, `IO::Compress::Bzip2`::new will return undef.

Constructor Options

`OPTS` is any combination of zero or more the following options:

- `AutoClose => 0|1`

  This option is only valid when the `$output` parameter is a filehandle. If specified, and the value is true, it will result in the `$output` being closed once either the `close` method is called or the `IO::Compress::Bzip2` object is destroyed.

  This parameter defaults to 0.

- `Append => 0|1`

  Opens `$output` in append mode.

  The behaviour of this option is dependent on the type of `$output`.

  - A Buffer

    If `$output` is a buffer and Append is enabled, all compressed data will be append to the end of `$output`. Otherwise `$output` will be cleared before any data is written to it.

  - A Filename

    If `$output` is a filename and Append is enabled, the file will be opened in append mode. Otherwise the contents of the file, if any, will be truncated before any compressed data is written to it.

  - A Filehandle

    If `$output` is a filehandle, the file pointer will be positioned to the end of the file via a call to `seek` before any compressed data is written to it. Otherwise the file pointer will not be moved.

  This parameter defaults to 0.

- `BlockSize100K => number`

  Specify the number of 100K blocks bzip2 uses during compression.

  Valid values are from 1 to 9, where 9 is best compression.

  The default is 1.

- `WorkFactor => number`

  Specifies how much effort bzip2 should take before resorting to a slower fallback compression algorithm.

  Valid values range from 0 to 250, where 0 means use the default value 30.

  The default is 0.

- `Strict => 0|1`

  This is a placeholder option.

Examples

Streaming

This very simple command line example demonstrates the streaming capabilities of the module. The code reads data from STDIN or all the files given on the commandline, compresses it, and writes the compressed data to STDOUT.

```
use strict ;
use warnings ;
use IO::Compress::Bzip2 qw(bzip2 $Bzip2Error) ;

my $z = IO::Compress::Bzip2->new("-", Stream => 1)
    or die "IO::Compress::Bzip2 failed: $Bzip2Error\n";

while (<>) {
    $z->print("abcde");
}
$z->close();
```

Note the use of `"-"` to means `STDOUT`. Alternatively you can use `\*STDOUT`.

Compressing a file from the filesystem

To read the contents of the file `file1.txt` and write the compressed data to the file `file1.txt.bz2` there are a few options

Start by creating the compression object and opening the input file

```
use strict ;
use warnings ;
use IO::Compress::Bzip2 qw(bzip2 $Bzip2Error) ;

my $input = "file1.txt";
my $z = IO::Compress::Bzip2->new("file1.txt.bz2")
    or die "IO::Compress::Bzip2 failed: $Bzip2Error\n";

# open the input file
open my $fh, "<", "file1.txt"
    or die "Cannot open file1.txt: $!\n";

# loop through the input file & write to the compressed file
while (<$fh>) {
    $z->print($_);
}

# not forgetting to close the compressed file
$z->close();
```

Methods

print

Usage is

```
$z->print($data)
print $z $data
```

Compresses and outputs the contents of the `$data` parameter. This has the same behaviour as the `print` built-in.

Returns true if successful.

printf

Usage is

```
$z->printf($format, $data)
printf $z $format, $data
```

Compresses and outputs the contents of the `$data` parameter.

Returns true if successful.

syswrite

Usage is

```
$z->syswrite $data
$z->syswrite $data, $length
$z->syswrite $data, $length, $offset
```

Compresses and outputs the contents of the `$data` parameter.

Returns the number of uncompressed bytes written, or `undef` if unsuccessful.

write

Usage is

```
$z->write $data
$z->write $data, $length
$z->write $data, $length, $offset
```

Compresses and outputs the contents of the `$data` parameter.

Returns the number of uncompressed bytes written, or `undef` if unsuccessful.

flush

Usage is

```
$z->flush;
```

Flushes any pending compressed data to the output file/buffer.

TODO

Returns true on success.

tell

Usage is

```
$z->tell()
tell $z
```

Returns the uncompressed file offset.

eof

Usage is

```
$z->eof();
eof($z);
```

Returns true if the `close` method has been called.

seek

```
$z->seek($position, $whence);
seek($z, $position, $whence);
```

Provides a sub-set of the `seek` functionality, with the restriction that it is only legal to seek forward in the output file/buffer. It is a fatal error to attempt to seek backward.

Empty parts of the file/buffer will have NULL (0x00) bytes written to them.

The `$whence` parameter takes one the usual values, namely SEEK_SET, SEEK_CUR or SEEK_END.

Returns 1 on success, 0 on failure.

binmode

Usage is

```
$z->binmode
binmode $z ;
```

This is a noop provided for completeness.

opened

```
$z->opened()
```

Returns true if the object currently refers to a opened file/buffer.

autoflush

```
my $prev = $z->autoflush()
my $prev = $z->autoflush(EXPR)
```

If the `$z` object is associated with a file or a filehandle, this method returns the current autoflush setting for the underlying filehandle. If `EXPR` is present, and is non-zero, it will enable flushing after every write/print operation.

If `$z` is associated with a buffer, this method has no effect and always returns `undef`.

Note that the special variable `$|` **cannot** be used to set or retrieve the autoflush setting.

input_line_number

```
$z->input_line_number()
$z->input_line_number(EXPR)
```

This method always returns `undef` when compressing.

fileno

```
$z->fileno()
fileno($z)
```

If the `$z` object is associated with a file or a filehandle, `fileno` will return the underlying file descriptor. Once the `close` method is called `fileno` will return `undef`.

If the `$z` object is associated with a buffer, this method will return `undef`.

close

```
$z->close() ;
close $z ;
```

Flushes any pending compressed data and then closes the output file/buffer.

For most versions of Perl this method will be automatically invoked if the IO::Compress::Bzip2 object is destroyed (either explicitly or by the variable with the reference to the object going out of scope). The exceptions are Perl versions 5.005 through 5.00504 and 5.8.0. In these cases, the `close` method will be called automatically, but not until global destruction of all live objects when the program is terminating.

Therefore, if you want your scripts to be able to run on all versions of Perl, you should call `close` explicitly and not rely on automatic closing.

Returns true on success, otherwise 0.

If the `AutoClose` option has been enabled when the IO::Compress::Bzip2 object was created, and the object is associated with a file, the underlying file will also be closed.

newStream([OPTS])

Usage is

```
$z->newStream( [OPTS] )
```

Closes the current compressed data stream and starts a new one.

OPTS consists of any of the options that are available when creating the `$z` object.

See the "Constructor Options" section for more details.

Importing

No symbolic constants are required by IO::Compress::Bzip2 at present.

- :all

  Imports `bzip2` and `$Bzip2Error`. Same as doing this

  ```
  use IO::Compress::Bzip2 qw(bzip2 $Bzip2Error) ;
  ```

EXAMPLES

Apache::GZip Revisited

See IO::Compress::FAQ

Working with Net::FTP

See IO::Compress::FAQ

SUPPORT

General feedback/questions/bug reports should be sent to https://github.com/pmqs/IO-Compress/issues (preferred) or https://rt.cpan.org/Public/Dist/Display.html?Name=IO-Compress.

SEE ALSO

Compress::Zlib, *IO::Compress::Gzip*, *IO::Uncompress::Gunzip*, *IO::Compress::Deflate*, *IO::Uncompress::Inflate*, *IO::Compress::RawDeflate*, *IO::Uncompress::RawInflate*, *IO::Uncompress::Bunzip2*, IO::Compress::Lzma, IO::Uncompress::UnLzma, IO::Compress::Xz, IO::Uncompress::UnXz, IO::Compress::Lzip, IO::Uncompress::UnLzip, IO::Compress::Lzop, IO::Uncompress::UnLzop, IO::Compress::Lzf, IO::Uncompress::UnLzf, IO::Compress::Zstd, IO::Uncompress::UnZstd, *IO::Uncompress::AnyInflate*, *IO::Uncompress::AnyUncompress*

IO::Compress::FAQ

File::GlobMapper, Archive::Zip, *Archive::Tar*, *IO::Zlib*

The primary site for the bzip2 program is https://sourceware.org/bzip2/.

See the module Compress::Bzip2

AUTHOR

This module was written by Paul Marquess, `pmqs@cpan.org`.

MODIFICATION HISTORY

See the Changes file.

COPYRIGHT AND LICENSE

Copyright (c) 2005-2026 Paul Marquess. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

IO::Compress::Deflate

Write RFC 1950 files/buffers

SYNOPSIS

```
use IO::Compress::Deflate qw(deflate $DeflateError) ;

my $status = deflate $input => $output [,OPTS]
    or die "deflate failed: $DeflateError\n";

my $z = IO::Compress::Deflate->new( $output [,OPTS] )
    or die "deflate failed: $DeflateError\n";

$z->print($string);
$z->printf($format, $string);
$z->write($string);
$z->syswrite($string [, $length, $offset]);
$z->flush();
$z->tell();
$z->eof();
$z->seek($position, $whence);
$z->binmode();
$z->fileno();
$z->opened();
$z->autoflush();
$z->input_line_number();
$z->newStream( [OPTS] );
```

(continues on next page)

(continued from previous page)

```
$z->deflateParams();

$z->close() ;

$DeflateError ;

# IO::File mode

print $z $string;
printf $z $format, $string;
tell $z
eof $z
seek $z, $position, $whence
binmode $z
fileno $z
close $z ;
```

DESCRIPTION

This module provides a Perl interface that allows writing compressed data to files or buffer as defined in RFC 1950.

For reading RFC 1950 files/buffers, see the companion module *IO::Uncompress::Inflate*.

Functional Interface

A top-level function, `deflate`, is provided to carry out "one-shot" compression between buffers and/or files. For finer control over the compression process, see the "OO Interface" section.

```
use IO::Compress::Deflate qw(deflate $DeflateError) ;

deflate $input_filename_or_reference => $output_filename_or_
 ↪reference [,OPTS]
    or die "deflate failed: $DeflateError\n";
```

The functional interface needs Perl5.005 or better.

deflate $input_filename_or_reference => $output_filename_or_reference [, OPTS]

`deflate` expects at least two parameters, `$input_filename_or_reference` and `$output_filename_or_reference` and zero or more optional parameters (see "Optional Parameters")

The $input_filename_or_reference parameter

The parameter, $input_filename_or_reference, is used to define the source of the uncompressed data.

It can take one of the following forms:

- A filename

  If the $input_filename_or_reference parameter is a simple scalar, it is assumed to be a filename. This file will be opened for reading and the input data will be read from it.

- A filehandle

  If the $input_filename_or_reference parameter is a filehandle, the input data will be read from it. The string '-' can be used as an alias for standard input.

- A scalar reference

  If $input_filename_or_reference is a scalar reference, the input data will be read from $$input_filename_or_reference.

- An array reference

  If $input_filename_or_reference is an array reference, each element in the array must be a filename.

  The input data will be read from each file in turn.

  The complete array will be walked to ensure that it only contains valid filenames before any data is compressed.

- An Input FileGlob string

  If $input_filename_or_reference is a string that is delimited by the characters "<" and ">" deflate will assume that it is an *input fileglob string*. The input is the list of files that match the fileglob.

  See *File::GlobMapper* for more details.

If the $input_filename_or_reference parameter is any other type, undef will be returned.

The $output_filename_or_reference parameter

The parameter $output_filename_or_reference is used to control the destination of the compressed data. This parameter can take one of these forms.

- A filename

  If the $output_filename_or_reference parameter is a simple scalar, it is assumed to be a filename. This file will be opened for writing and the compressed data will be written to it.

- A filehandle

  If the $output_filename_or_reference parameter is a filehandle, the compressed data will be written to it. The string '-' can be used as an alias for standard output.

- A scalar reference

  If `$output_filename_or_reference` is a scalar reference, the compressed data will be stored in `$$output_filename_or_reference`.

- An Array Reference

  If `$output_filename_or_reference` is an array reference, the compressed data will be pushed onto the array.

- An Output FileGlob

  If `$output_filename_or_reference` is a string that is delimited by the characters "<" and ">" `deflate` will assume that it is an *output fileglob string*. The output is the list of files that match the fileglob.

  When `$output_filename_or_reference` is an fileglob string, `$input_filename_or_reference` must also be a fileglob string. Anything else is an error.

  See *File::GlobMapper* for more details.

If the `$output_filename_or_reference` parameter is any other type, `undef` will be returned.

Notes

When `$input_filename_or_reference` maps to multiple files/buffers and `$output_filename_or_reference` is a single file/buffer the input files/buffers will be stored in `$output_filename_or_reference` as a concatenated series of compressed data streams.

Optional Parameters

The optional parameters for the one-shot function `deflate` are (for the most part) identical to those used with the OO interface defined in the "Constructor Options" section. The exceptions are listed below

- `AutoClose => 0|1`

  This option applies to any input or output data streams to `deflate` that are filehandles.

  If `AutoClose` is specified, and the value is true, it will result in all input and/or output filehandles being closed once `deflate` has completed.

  This parameter defaults to 0.

- `BinModeIn => 0|1`

  This option is now a no-op. All files will be read in binmode.

- `Append => 0|1`

  The behaviour of this option is dependent on the type of output data stream.

  - A Buffer

    If Append is enabled, all compressed data will be append to the end of the output buffer. Otherwise the output buffer will be cleared before any compressed data is written to it.

- A Filename

  If Append is enabled, the file will be opened in append mode. Otherwise the contents of the file, if any, will be truncated before any compressed data is written to it.

- A Filehandle

  If Append is enabled, the filehandle will be positioned to the end of the file via a call to `seek` before any compressed data is written to it. Otherwise the file pointer will not be moved.

When Append is specified, and set to true, it will *append* all compressed data to the output data stream.

So when the output is a filehandle it will carry out a seek to the eof before writing any compressed data. If the output is a filename, it will be opened for appending. If the output is a buffer, all compressed data will be appended to the existing buffer.

Conversely when Append is not specified, or it is present and is set to false, it will operate as follows.

When the output is a filename, it will truncate the contents of the file before writing any compressed data. If the output is a filehandle its position will not be changed. If the output is a buffer, it will be wiped before any compressed data is output.

Defaults to 0.

Oneshot Examples

Here are a few example that show the capabilities of the module.

Streaming

This very simple command line example demonstrates the streaming capabilities of the module. The code reads data from STDIN, compresses it, and writes the compressed data to STDOUT.

```
$ echo hello world | perl -MIO::Compress::Deflate=deflate -e
↪'deflate \*STDIN => \*STDOUT' >output.1950
```

The special filename "-" can be used as a standin for both `\*STDIN` and `\*STDOUT`, so the above can be rewritten as

```
$ echo hello world | perl -MIO::Compress::Deflate=deflate -e
↪'deflate "-" => "-"' >output.1950
```

Compressing a file from the filesystem

To read the contents of the file `file1.txt` and write the compressed data to the file `file1.txt.1950`.

```
use strict ;
use warnings ;
```

(continues on next page)

(continued from previous page)

```
use IO::Compress::Deflate qw(deflate $DeflateError) ;

my $input = "file1.txt";
deflate $input => "$input.1950"
    or die "deflate failed: $DeflateError\n";
```

Reading from a Filehandle and writing to an in-memory buffer

To read from an existing Perl filehandle, `$input`, and write the compressed data to a buffer, `$buffer`.

```
use strict ;
use warnings ;
use IO::Compress::Deflate qw(deflate $DeflateError) ;
use IO::File ;

my $input = IO::File->new( "<file1.txt" )
    or die "Cannot open 'file1.txt': $!\n" ;
my $buffer ;
deflate $input => \$buffer
    or die "deflate failed: $DeflateError\n";
```

Compressing multiple files

To compress all files in the directory “/my/home” that match “*.txt” and store the compressed data in the same directory

```
use strict ;
use warnings ;
use IO::Compress::Deflate qw(deflate $DeflateError) ;

deflate '</my/home/*.txt>' => '<*.1950>'
    or die "deflate failed: $DeflateError\n";
```

and if you want to compress each file one at a time, this will do the trick

```
use strict ;
use warnings ;
use IO::Compress::Deflate qw(deflate $DeflateError) ;

for my $input ( glob "/my/home/*.txt" )
{
    my $output = "$input.1950" ;
    deflate $input => $output
        or die "Error compressing '$input': $DeflateError\n";
}
```

OO Interface

Constructor

The format of the constructor for `IO::Compress::Deflate` is shown below

```
my $z = IO::Compress::Deflate->new( $output [,OPTS] )
    or die "IO::Compress::Deflate failed: $DeflateError\n";
```

The constructor takes one mandatory parameter, `$output`, defined below and zero or more `OPTS`, defined in "Constructor Options".

It returns an `IO::Compress::Deflate` object on success and `undef` on failure. The variable `$DeflateError` will contain an error message on failure.

If you are running Perl 5.005 or better the object, `$z`, returned from IO::Compress::Deflate can be used exactly like an *IO::File* filehandle. This means that all normal output file operations can be carried out with `$z`. For example, to write to a compressed file/buffer you can use either of these forms

```
$z->print("hello world\n");
print $z "hello world\n";
```

Below is a simple exaple of using the OO interface to create an output file `myfile.1950` and write some data to it.

```
my $filename = "myfile.1950";
my $z = IO::Compress::Deflate->new($filename)
    or die "IO::Compress::Deflate failed: $DeflateError\n";

$z->print("abcde");
$z->close();
```

See the "Examples" for more.

The mandatory parameter `$output` is used to control the destination of the compressed data. This parameter can take one of these forms.

- A filename

  If the `$output` parameter is a simple scalar, it is assumed to be a filename. This file will be opened for writing and the compressed data will be written to it.

- A filehandle

  If the `$output` parameter is a filehandle, the compressed data will be written to it. The string '-' can be used as an alias for standard output.

- A scalar reference

  If `$output` is a scalar reference, the compressed data will be stored in `$$output`.

If the `$output` parameter is any other type, `IO::Compress::Deflate`::new will return undef.

Constructor Options

`OPTS` is any combination of zero or more the following options:

- `AutoClose => 0|1`

  This option is only valid when the `$output` parameter is a filehandle. If specified, and the value is true, it will result in the `$output` being closed once either the `close` method is called or the `IO::Compress::Deflate` object is destroyed.

  This parameter defaults to 0.

- `Append => 0|1`

  Opens `$output` in append mode.

  The behaviour of this option is dependent on the type of `$output`.

  - A Buffer

    If `$output` is a buffer and Append is enabled, all compressed data will be append to the end of `$output`. Otherwise `$output` will be cleared before any data is written to it.

  - A Filename

    If `$output` is a filename and Append is enabled, the file will be opened in append mode. Otherwise the contents of the file, if any, will be truncated before any compressed data is written to it.

  - A Filehandle

    If `$output` is a filehandle, the file pointer will be positioned to the end of the file via a call to `seek` before any compressed data is written to it. Otherwise the file pointer will not be moved.

  This parameter defaults to 0.

- `Merge => 0|1`

  This option is used to compress input data and append it to an existing compressed data stream in `$output`. The end result is a single compressed data stream stored in `$output`.

  It is a fatal error to attempt to use this option when `$output` is not an RFC 1950 data stream.

  There are a number of other limitations with the `Merge` option:

  1. This module needs to have been built with zlib 1.2.1 or better to work. A fatal error will be thrown if `Merge` is used with an older version of zlib.

  2. If `$output` is a file or a filehandle, it must be seekable.

  This parameter defaults to 0.

- -Level

  Defines the compression level used by zlib. The value should either be a number between 0 and 9 (0 means no compression and 9 is maximum compression), or one of the symbolic constants defined below.

```
Z_NO_COMPRESSION
Z_BEST_SPEED
Z_BEST_COMPRESSION
Z_DEFAULT_COMPRESSION
```

The default is Z_DEFAULT_COMPRESSION.

Note, these constants are not imported by `IO::Compress::Deflate` by default.

```
use IO::Compress::Deflate qw(:strategy);
use IO::Compress::Deflate qw(:constants);
use IO::Compress::Deflate qw(:all);
```

- -Strategy

  Defines the strategy used to tune the compression. Use one of the symbolic constants defined below.

  ```
  Z_FILTERED
  Z_HUFFMAN_ONLY
  Z_RLE
  Z_FIXED
  Z_DEFAULT_STRATEGY
  ```

  The default is Z_DEFAULT_STRATEGY.

- `Strict => 0|1`

  This is a placeholder option.

Examples

Streaming

This very simple command line example demonstrates the streaming capabilities of the module. The code reads data from STDIN or all the files given on the commandline, compresses it, and writes the compressed data to STDOUT.

```
use strict ;
use warnings ;
use IO::Compress::Deflate qw(deflate $DeflateError) ;

my $z = IO::Compress::Deflate->new("-", Stream => 1)
    or die "IO::Compress::Deflate failed: $DeflateError\n";

while (<>) {
    $z->print("abcde");
}
$z->close();
```

Note the use of `"-"` to means `STDOUT`. Alternatively you can use `\*STDOUT`.

Compressing a file from the filesystem

To read the contents of the file `file1.txt` and write the compressed data to the file `file1.txt.1950` there are a few options

Start by creating the compression object and opening the input file

```
use strict ;
use warnings ;
use IO::Compress::Deflate qw(deflate $DeflateError) ;

my $input = "file1.txt";
my $z = IO::Compress::Deflate->new("file1.txt.1950")
    or die "IO::Compress::Deflate failed: $DeflateError\n";

# open the input file
open my $fh, "<", "file1.txt"
    or die "Cannot open file1.txt: $!\n";

# loop through the input file & write to the compressed file
while (<$fh>) {
    $z->print($_);
}

# not forgetting to close the compressed file
$z->close();
```

Methods

print

Usage is

```
$z->print($data)
print $z $data
```

Compresses and outputs the contents of the `$data` parameter. This has the same behaviour as the `print` built-in.

Returns true if successful.

printf

Usage is

```
$z->printf($format, $data)
printf $z $format, $data
```

Compresses and outputs the contents of the `$data` parameter.

Returns true if successful.

syswrite

Usage is

```
$z->syswrite $data
$z->syswrite $data, $length
$z->syswrite $data, $length, $offset
```

Compresses and outputs the contents of the `$data` parameter.

Returns the number of uncompressed bytes written, or `undef` if unsuccessful.

write

Usage is

```
$z->write $data
$z->write $data, $length
$z->write $data, $length, $offset
```

Compresses and outputs the contents of the `$data` parameter.

Returns the number of uncompressed bytes written, or `undef` if unsuccessful.

flush

Usage is

```
$z->flush;
$z->flush($flush_type);
```

Flushes any pending compressed data to the output file/buffer.

This method takes an optional parameter, `$flush_type`, that controls how the flushing will be carried out. By default the `$flush_type` used is `Z_FINISH`. Other valid values for `$flush_type` are `Z_NO_FLUSH`, `Z_SYNC_FLUSH`, `Z_FULL_FLUSH` and `Z_BLOCK`. It is strongly recommended that you only set the `flush_type` parameter if you fully understand the implications of what it does - overuse of `flush` can seriously degrade the level of compression achieved. See the `zlib` documentation for details.

Returns true on success.

tell

Usage is

```
$z->tell()
tell $z
```

Returns the uncompressed file offset.

eof

Usage is

```
$z->eof();
eof($z);
```

Returns true if the `close` method has been called.

seek

```
$z->seek($position, $whence);
seek($z, $position, $whence);
```

Provides a sub-set of the `seek` functionality, with the restriction that it is only legal to seek forward in the output file/buffer. It is a fatal error to attempt to seek backward.

Empty parts of the file/buffer will have NULL (0x00) bytes written to them.

The `$whence` parameter takes one the usual values, namely SEEK_SET, SEEK_CUR or SEEK_END.

Returns 1 on success, 0 on failure.

binmode

Usage is

```
$z->binmode
binmode $z ;
```

This is a noop provided for completeness.

opened

```
$z->opened()
```

Returns true if the object currently refers to a opened file/buffer.

autoflush

```
my $prev = $z->autoflush()
my $prev = $z->autoflush(EXPR)
```

If the `$z` object is associated with a file or a filehandle, this method returns the current autoflush setting for the underlying filehandle. If `EXPR` is present, and is non-zero, it will enable flushing after every write/print operation.

If `$z` is associated with a buffer, this method has no effect and always returns `undef`.

Note that the special variable `$|` **cannot** be used to set or retrieve the autoflush setting.

input_line_number

```
$z->input_line_number()
$z->input_line_number(EXPR)
```

This method always returns undef when compressing.

fileno

```
$z->fileno()
fileno($z)
```

If the $z object is associated with a file or a filehandle, fileno will return the underlying file descriptor. Once the close method is called fileno will return undef.

If the $z object is associated with a buffer, this method will return undef.

close

```
$z->close() ;
close $z ;
```

Flushes any pending compressed data and then closes the output file/buffer.

For most versions of Perl this method will be automatically invoked if the IO::Compress::Deflate object is destroyed (either explicitly or by the variable with the reference to the object going out of scope). The exceptions are Perl versions 5.005 through 5.00504 and 5.8.0. In these cases, the close method will be called automatically, but not until global destruction of all live objects when the program is terminating.

Therefore, if you want your scripts to be able to run on all versions of Perl, you should call close explicitly and not rely on automatic closing.

Returns true on success, otherwise 0.

If the AutoClose option has been enabled when the IO::Compress::Deflate object was created, and the object is associated with a file, the underlying file will also be closed.

newStream([OPTS])

Usage is

```
$z->newStream( [OPTS] )
```

Closes the current compressed data stream and starts a new one.

OPTS consists of any of the options that are available when creating the $z object.

See the “Constructor Options” section for more details.

deflateParams

Usage is

```
$z->deflateParams
```

TODO

Importing

A number of symbolic constants are required by some methods in `IO::Compress::Deflate`. None are imported by default.

- :all

  Imports `deflate`, `$DeflateError` and all symbolic constants that can be used by `IO::Compress::Deflate`. Same as doing this

  ```
  use IO::Compress::Deflate qw(deflate $DeflateError :constants) ;
  ```

- :constants

  Import all symbolic constants. Same as doing this

  ```
  use IO::Compress::Deflate qw(:flush :level :strategy) ;
  ```

- :flush

  These symbolic constants are used by the `flush` method.

  ```
  Z_NO_FLUSH
  Z_PARTIAL_FLUSH
  Z_SYNC_FLUSH
  Z_FULL_FLUSH
  Z_FINISH
  Z_BLOCK
  ```

- :level

  These symbolic constants are used by the `Level` option in the constructor.

  ```
  Z_NO_COMPRESSION
  Z_BEST_SPEED
  Z_BEST_COMPRESSION
  Z_DEFAULT_COMPRESSION
  ```

- :strategy

  These symbolic constants are used by the `Strategy` option in the constructor.

  ```
  Z_FILTERED
  Z_HUFFMAN_ONLY
  Z_RLE
  Z_FIXED
  Z_DEFAULT_STRATEGY
  ```

EXAMPLES

Apache::GZip Revisited

See IO::Compress::FAQ

Working with Net::FTP

See IO::Compress::FAQ

SUPPORT

General feedback/questions/bug reports should be sent to https://github.com/pmqs/IO-Compress/issues (preferred) or https://rt.cpan.org/Public/Dist/Display.html?Name=IO-Compress.

SEE ALSO

Compress::Zlib, *IO::Compress::Gzip*, *IO::Uncompress::Gunzip*, *IO::Uncompress::Inflate*, *IO::Compress::RawDeflate*, *IO::Uncompress::RawInflate*, *IO::Compress::Bzip2*, *IO::Uncompress::Bunzip2*, IO::Compress::Lzma, IO::Uncompress::UnLzma, IO::Compress::Xz, IO::Uncompress::UnXz, IO::Compress::Lzip, IO::Uncompress::UnLzip, IO::Compress::Lzop, IO::Uncompress::UnLzop, IO::Compress::Lzf, IO::Uncompress::UnLzf, IO::Compress::Zstd, IO::Uncompress::UnZstd, *IO::Uncompress::AnyInflate*, *IO::Uncompress::AnyUncompress*

IO::Compress::FAQ

File::GlobMapper, Archive::Zip, *Archive::Tar*, *IO::Zlib*

For RFC 1950, 1951 and 1952 see https://datatracker.ietf.org/doc/html/rfc1950, https://datatracker.ietf.org/doc/html/rfc1951 and https://datatracker.ietf.org/doc/html/rfc1952

The *zlib* compression library was written by Jean-loup Gailly `gzip@prep.ai.mit.edu` and Mark Adler `madler@alumni.caltech.edu`.

The primary site for the *zlib* compression library is http://www.zlib.org.

The primary site for the *zlib-ng* compression library is https://github.com/zlib-ng/zlib-ng.

The primary site for gzip is http://www.gzip.org.

AUTHOR

This module was written by Paul Marquess, `pmqs@cpan.org`.

MODIFICATION HISTORY

See the Changes file.

COPYRIGHT AND LICENSE

Copyright (c) 2005-2026 Paul Marquess. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

IO::Compress::Gzip

Write RFC 1952 files/buffers

SYNOPSIS

```
use IO::Compress::Gzip qw(gzip $GzipError) ;

my $status = gzip $input => $output [,OPTS]
    or die "gzip failed: $GzipError\n";

my $z = IO::Compress::Gzip->new( $output [,OPTS] )
    or die "gzip failed: $GzipError\n";

$z->print($string);
$z->printf($format, $string);
$z->write($string);
$z->syswrite($string [, $length, $offset]);
$z->flush();
$z->tell();
$z->eof();
$z->seek($position, $whence);
$z->binmode();
$z->fileno();
$z->opened();
$z->autoflush();
$z->input_line_number();
$z->newStream( [OPTS] );

$z->deflateParams();

$z->close() ;

$GzipError ;

# IO::File mode

print $z $string;
printf $z $format, $string;
tell $z
eof $z
seek $z, $position, $whence
binmode $z
```

(continues on next page)

(continued from previous page)

```
fileno $z
close $z ;
```

DESCRIPTION

This module provides a Perl interface that allows writing compressed data to files or buffer as defined in RFC 1952.

All the gzip headers defined in RFC 1952 can be created using this module.

For reading RFC 1952 files/buffers, see the companion module *IO::Uncompress::Gunzip*.

Functional Interface

A top-level function, `gzip`, is provided to carry out "one-shot" compression between buffers and/or files. For finer control over the compression process, see the "OO Interface" section.

```
use IO::Compress::Gzip qw(gzip $GzipError) ;

gzip $input_filename_or_reference => $output_filename_or_reference␣
↪[,OPTS]
    or die "gzip failed: $GzipError\n";
```

The functional interface needs Perl5.005 or better.

gzip $input_filename_or_reference => $output_filename_or_reference [, OPTS]

`gzip` expects at least two parameters, `$input_filename_or_reference` and `$output_filename_or_reference` and zero or more optional parameters (see "Optional Parameters")

The `$input_filename_or_reference` parameter

The parameter, `$input_filename_or_reference`, is used to define the source of the uncompressed data.

It can take one of the following forms:

- A filename

  If the `$input_filename_or_reference` parameter is a simple scalar, it is assumed to be a filename. This file will be opened for reading and the input data will be read from it.

- A filehandle

  If the `$input_filename_or_reference` parameter is a filehandle, the input data will be read from it. The string '-' can be used as an alias for standard input.

- A scalar reference

  If `$input_filename_or_reference` is a scalar reference, the input data will be read from `$$input_filename_or_reference`.

- An array reference

  If `$input_filename_or_reference` is an array reference, each element in the array must be a filename.

  The input data will be read from each file in turn.

  The complete array will be walked to ensure that it only contains valid filenames before any data is compressed.

- An Input FileGlob string

  If `$input_filename_or_reference` is a string that is delimited by the characters "<" and ">" `gzip` will assume that it is an *input fileglob string*. The input is the list of files that match the fileglob.

  See *File::GlobMapper* for more details.

If the `$input_filename_or_reference` parameter is any other type, `undef` will be returned.

In addition, if `$input_filename_or_reference` is a simple filename, the default values for the `Name` and `Time` options will be sourced from that file.

If you do not want to use these defaults they can be overridden by explicitly setting the `Name` and `Time` options or by setting the `Minimal` parameter.

The `$output_filename_or_reference` parameter

The parameter `$output_filename_or_reference` is used to control the destination of the compressed data. This parameter can take one of these forms.

- A filename

  If the `$output_filename_or_reference` parameter is a simple scalar, it is assumed to be a filename. This file will be opened for writing and the compressed data will be written to it.

- A filehandle

  If the `$output_filename_or_reference` parameter is a filehandle, the compressed data will be written to it. The string '-' can be used as an alias for standard output.

- A scalar reference

  If `$output_filename_or_reference` is a scalar reference, the compressed data will be stored in `$$output_filename_or_reference`.

- An Array Reference

  If `$output_filename_or_reference` is an array reference, the compressed data will be pushed onto the array.

- An Output FileGlob

  If `$output_filename_or_reference` is a string that is delimited by the characters "<" and ">" `gzip` will assume that it is an *output fileglob string*. The output is the list of files that match the fileglob.

When `$output_filename_or_reference` is an fileglob string, `$input_filename_or_reference` must also be a fileglob string. Anything else is an error.

See *File::GlobMapper* for more details.

If the `$output_filename_or_reference` parameter is any other type, `undef` will be returned.

Notes

When `$input_filename_or_reference` maps to multiple files/buffers and `$output_filename_or_reference` is a single file/buffer the input files/buffers will be stored in `$output_filename_or_reference` as a concatenated series of compressed data streams.

Optional Parameters

The optional parameters for the one-shot function `gzip` are (for the most part) identical to those used with the OO interface defined in the "Constructor Options" section. The exceptions are listed below

- `AutoClose => 0|1`

  This option applies to any input or output data streams to `gzip` that are filehandles.

  If `AutoClose` is specified, and the value is true, it will result in all input and/or output filehandles being closed once `gzip` has completed.

  This parameter defaults to 0.

- `BinModeIn => 0|1`

  This option is now a no-op. All files will be read in binmode.

- `Append => 0|1`

  The behaviour of this option is dependent on the type of output data stream.

  - A Buffer

    If Append is enabled, all compressed data will be append to the end of the output buffer. Otherwise the output buffer will be cleared before any compressed data is written to it.

  - A Filename

    If Append is enabled, the file will be opened in append mode. Otherwise the contents of the file, if any, will be truncated before any compressed data is written to it.

  - A Filehandle

    If Append is enabled, the filehandle will be positioned to the end of the file via a call to `seek` before any compressed data is written to it. Otherwise the file pointer will not be moved.

  When Append is specified, and set to true, it will *append* all compressed data to the output data stream.

So when the output is a filehandle it will carry out a seek to the eof before writing any compressed data. If the output is a filename, it will be opened for appending. If the output is a buffer, all compressed data will be appended to the existing buffer.

Conversely when Append is not specified, or it is present and is set to false, it will operate as follows.

When the output is a filename, it will truncate the contents of the file before writing any compressed data. If the output is a filehandle its position will not be changed. If the output is a buffer, it will be wiped before any compressed data is output.

Defaults to 0.

Oneshot Examples

Here are a few example that show the capabilities of the module.

Streaming

This very simple command line example demonstrates the streaming capabilities of the module. The code reads data from STDIN, compresses it, and writes the compressed data to STDOUT.

```
$ echo hello world | perl -MIO::Compress::Gzip=gzip -e 'gzip \*STDIN
 ↪=> \*STDOUT' >output.gz
```

The special filename "-" can be used as a standin for both `\*STDIN` and `\*STDOUT`, so the above can be rewritten as

```
$ echo hello world | perl -MIO::Compress::Gzip=gzip -e 'gzip "-" =>
 ↪"-"' >output.gz
```

Compressing a file from the filesystem

To read the contents of the file `file1.txt` and write the compressed data to the file `file1.txt.gz`.

```
use strict ;
use warnings ;
use IO::Compress::Gzip qw(gzip $GzipError) ;

my $input = "file1.txt";
gzip $input => "$input.gz"
    or die "gzip failed: $GzipError\n";
```

Reading from a Filehandle and writing to an in-memory buffer

To read from an existing Perl filehandle, `$input`, and write the compressed data to a buffer, `$buffer`.

```
use strict ;
use warnings ;
use IO::Compress::Gzip qw(gzip $GzipError) ;
use IO::File ;

my $input = IO::File->new( "<file1.txt" )
    or die "Cannot open 'file1.txt': $!\n" ;
my $buffer ;
gzip $input => \$buffer
    or die "gzip failed: $GzipError\n";
```

Compressing multiple files

To compress all files in the directory "/my/home" that match "*.txt" and store the compressed data in the same directory

```
use strict ;
use warnings ;
use IO::Compress::Gzip qw(gzip $GzipError) ;

gzip '</my/home/*.txt>' => '<*.gz>'
    or die "gzip failed: $GzipError\n";
```

and if you want to compress each file one at a time, this will do the trick

```
use strict ;
use warnings ;
use IO::Compress::Gzip qw(gzip $GzipError) ;

for my $input ( glob "/my/home/*.txt" )
{
    my $output = "$input.gz" ;
    gzip $input => $output
        or die "Error compressing '$input': $GzipError\n";
}
```

OO Interface

Constructor

The format of the constructor for `IO::Compress::Gzip` is shown below

```
my $z = IO::Compress::Gzip->new( $output [,OPTS] )
    or die "IO::Compress::Gzip failed: $GzipError\n";
```

The constructor takes one mandatory parameter, `$output`, defined below and zero or more `OPTS`, defined in "Constructor Options".

It returns an `IO::Compress::Gzip` object on success and `undef` on failure. The variable `$GzipError` will contain an error message on failure.

If you are running Perl 5.005 or better the object, `$z`, returned from IO::Compress::Gzip can be used exactly like an *IO::File* filehandle. This means that all normal output file operations can be carried out with `$z`. For example, to write to a compressed file/buffer you can use either of these forms

```
$z->print("hello world\n");
print $z "hello world\n";
```

Below is a simple exaple of using the OO interface to create an output file `myfile.gz` and write some data to it.

```
my $filename = "myfile.gz";
my $z = IO::Compress::Gzip->new($filename)
    or die "IO::Compress::Gzip failed: $GzipError\n";

$z->print("abcde");
$z->close();
```

See the "Examples" for more.

The mandatory parameter `$output` is used to control the destination of the compressed data. This parameter can take one of these forms.

- A filename

  If the `$output` parameter is a simple scalar, it is assumed to be a filename. This file will be opened for writing and the compressed data will be written to it.

- A filehandle

  If the `$output` parameter is a filehandle, the compressed data will be written to it. The string '-' can be used as an alias for standard output.

- A scalar reference

  If `$output` is a scalar reference, the compressed data will be stored in `$$output`.

If the `$output` parameter is any other type, `IO::Compress::Gzip`::new will return undef.

Constructor Options

`OPTS` is any combination of zero or more the following options:

- `AutoClose => 0|1`

  This option is only valid when the `$output` parameter is a filehandle. If specified, and the value is true, it will result in the `$output` being closed once either the `close` method is called or the `IO::Compress::Gzip` object is destroyed.

  This parameter defaults to 0.

- `Append => 0|1`

  Opens `$output` in append mode.

  The behaviour of this option is dependent on the type of `$output`.

- A Buffer

  If `$output` is a buffer and Append is enabled, all compressed data will be append to the end of `$output`. Otherwise `$output` will be cleared before any data is written to it.

- A Filename

  If `$output` is a filename and Append is enabled, the file will be opened in append mode. Otherwise the contents of the file, if any, will be truncated before any compressed data is written to it.

- A Filehandle

  If `$output` is a filehandle, the file pointer will be positioned to the end of the file via a call to `seek` before any compressed data is written to it. Otherwise the file pointer will not be moved.

This parameter defaults to 0.

- `Merge => 0|1`

  This option is used to compress input data and append it to an existing compressed data stream in `$output`. The end result is a single compressed data stream stored in `$output`.

  It is a fatal error to attempt to use this option when `$output` is not an RFC 1952 data stream.

  There are a number of other limitations with the `Merge` option:

  1. This module needs to have been built with zlib 1.2.1 or better to work. A fatal error will be thrown if `Merge` is used with an older version of zlib.
  2. If `$output` is a file or a filehandle, it must be seekable.

  This parameter defaults to 0.

- -Level

  Defines the compression level used by zlib. The value should either be a number between 0 and 9 (0 means no compression and 9 is maximum compression), or one of the symbolic constants defined below.

```
Z_NO_COMPRESSION
Z_BEST_SPEED
Z_BEST_COMPRESSION
Z_DEFAULT_COMPRESSION
```

  The default is Z_DEFAULT_COMPRESSION.

  Note, these constants are not imported by `IO::Compress::Gzip` by default.

```
use IO::Compress::Gzip qw(:strategy);
use IO::Compress::Gzip qw(:constants);
use IO::Compress::Gzip qw(:all);
```

- -Strategy

Defines the strategy used to tune the compression. Use one of the symbolic constants defined below.

```
Z_FILTERED
Z_HUFFMAN_ONLY
Z_RLE
Z_FIXED
Z_DEFAULT_STRATEGY
```

The default is Z_DEFAULT_STRATEGY.

- `Minimal => 0|1`

  If specified, this option will force the creation of the smallest possible compliant gzip header (which is exactly 10 bytes long) as defined in RFC 1952.

  See the section titled “Compliance” in RFC 1952 for a definition of the values used for the fields in the gzip header.

  All other parameters that control the content of the gzip header will be ignored if this parameter is set to 1.

  This parameter defaults to 0.

- `Comment => $comment`

  Stores the contents of `$comment` in the COMMENT field in the gzip header. By default, no comment field is written to the gzip file.

  If the `-Strict` option is enabled, the comment can only consist of ISO 8859-1 characters plus line feed.

  If the `-Strict` option is disabled, the comment field can contain any character except NULL. If any null characters are present, the field will be truncated at the first NULL.

- `Name => $string`

  Stores the contents of `$string` in the gzip NAME header field. If `Name` is not specified, no gzip NAME field will be created.

  If the `-Strict` option is enabled, `$string` can only consist of ISO 8859-1 characters.

  If `-Strict` is disabled, then `$string` can contain any character except NULL. If any null characters are present, the field will be truncated at the first NULL.

- `Time => $number`

  Sets the MTIME field in the gzip header to $number.

  This field defaults to the time the `IO::Compress::Gzip` object was created if this option is not specified.

- `TextFlag => 0|1`

  This parameter controls the setting of the FLG.FTEXT bit in the gzip header. It is used to signal that the data stored in the gzip file/buffer is probably text.

  The default is 0.

- `HeaderCRC => 0|1`

  When true this parameter will set the FLG.FHCRC bit to 1 in the gzip header and set the CRC16 header field to the CRC of the complete gzip header except the CRC16 field itself.

  Note that gzip files created with the `HeaderCRC` flag set to 1 cannot be read by most, if not all, of the standard gunzip utilities, most notably gzip version 1.2.4. You should therefore avoid using this option if you want to maximize the portability of your gzip files.

  This parameter defaults to 0.

- `OS_Code => $value`

  Stores `$value` in the gzip OS header field. A number between 0 and 255 is valid.

  If not specified, this parameter defaults to the OS code of the Operating System this module was built on. The value 3 is used as a catch-all for all Unix variants and unknown Operating Systems.

- `ExtraField => $data`

  This parameter allows additional metadata to be stored in the ExtraField in the gzip header. An RFC 1952 compliant ExtraField consists of zero or more subfields. Each subfield consists of a two byte header followed by the subfield data.

  The list of subfields can be supplied in any of the following formats

```
-ExtraField => [$id1, $data1,
                $id2, $data2,
                ...
               ]
-ExtraField => [ [$id1 => $data1],
                 [$id2 => $data2],
                 ...
               ]
-ExtraField => { $id1 => $data1,
                 $id2 => $data2,
                 ...
               }
```

  Where `$id1`, `$id2` are two byte subfield ID's. The second byte of the ID cannot be 0, unless the `Strict` option has been disabled.

  If you use the hash syntax, you have no control over the order in which the Extra-SubFields are stored, plus you cannot have SubFields with duplicate ID.

  Alternatively the list of subfields can by supplied as a scalar, thus

```
-ExtraField => $rawdata
```

  If you use the raw format, and the `Strict` option is enabled, `IO::Compress::Gzip` will check that `$rawdata` consists of zero or more conformant sub-fields. When `Strict` is disabled, `$rawdata` can consist of any arbitrary byte stream.

  The maximum size of the Extra Field 65535 bytes.

- `ExtraFlags => $value`

  Sets the XFL byte in the gzip header to `$value`.

  If this option is not present, the value stored in XFL field will be determined by the setting of the `Level` option.

  If `Level => Z_BEST_SPEED` has been specified then XFL is set to 2. If `Level => Z_BEST_COMPRESSION` has been specified then XFL is set to 4. Otherwise XFL is set to 0.

- `Strict => 0|1`

  `Strict` will optionally police the values supplied with other options to ensure they are compliant with RFC1952.

  This option is enabled by default.

  If `Strict` is enabled the following behaviour will be policed:

  – The value supplied with the `Name` option can only contain ISO 8859-1 characters.

  – The value supplied with the `Comment` option can only contain ISO 8859-1 characters plus line-feed.

  – The values supplied with the `-Name` and `-Comment` options cannot contain multiple embedded nulls.

  – If an `ExtraField` option is specified and it is a simple scalar, it must conform to the sub-field structure as defined in RFC 1952.

  – If an `ExtraField` option is specified the second byte of the ID will be checked in each subfield to ensure that it does not contain the reserved value 0x00.

  When `Strict` is disabled the following behaviour will be policed:

  – The value supplied with `-Name` option can contain any character except NULL.

  – The value supplied with `-Comment` option can contain any character except NULL.

  – The values supplied with the `-Name` and `-Comment` options can contain multiple embedded nulls. The string written to the gzip header will consist of the characters up to, but not including, the first embedded NULL.

  – If an `ExtraField` option is specified and it is a simple scalar, the structure will not be checked. The only error is if the length is too big.

  – The ID header in an `ExtraField` sub-field can consist of any two bytes.

Examples

Streaming

This very simple command line example demonstrates the streaming capabilities of the module. The code reads data from STDIN or all the files given on the commandline, compresses it, and writes the compressed data to STDOUT.

```
use strict ;
use warnings ;
use IO::Compress::Gzip qw(gzip $GzipError) ;

my $z = IO::Compress::Gzip->new("-", Stream => 1)
    or die "IO::Compress::Gzip failed: $GzipError\n";

while (<>) {
    $z->print("abcde");
}
$z->close();
```

Note the use of `"-"` to means STDOUT. Alternatively you can use `\*STDOUT`.

Compressing a file from the filesystem

To read the contents of the file `file1.txt` and write the compressed data to the file `file1.txt.gz` there are a few options

Start by creating the compression object and opening the input file

```
use strict ;
use warnings ;
use IO::Compress::Gzip qw(gzip $GzipError) ;

my $input = "file1.txt";
my $z = IO::Compress::Gzip->new("file1.txt.gz")
    or die "IO::Compress::Gzip failed: $GzipError\n";

# open the input file
open my $fh, "<", "file1.txt"
    or die "Cannot open file1.txt: $!\n";

# loop through the input file & write to the compressed file
while (<$fh>) {
    $z->print($_);
}

# not forgetting to close the compressed file
$z->close();
```

Methods

print

Usage is

```
$z->print($data)
print $z $data
```

Compresses and outputs the contents of the `$data` parameter. This has the same behaviour as the `print` built-in.

Returns true if successful.

printf

Usage is

```
$z->printf($format, $data)
printf $z $format, $data
```

Compresses and outputs the contents of the `$data` parameter.

Returns true if successful.

syswrite

Usage is

```
$z->syswrite $data
$z->syswrite $data, $length
$z->syswrite $data, $length, $offset
```

Compresses and outputs the contents of the `$data` parameter.

Returns the number of uncompressed bytes written, or `undef` if unsuccessful.

write

Usage is

```
$z->write $data
$z->write $data, $length
$z->write $data, $length, $offset
```

Compresses and outputs the contents of the `$data` parameter.

Returns the number of uncompressed bytes written, or `undef` if unsuccessful.

flush

Usage is

```
$z->flush;
$z->flush($flush_type);
```

Flushes any pending compressed data to the output file/buffer.

This method takes an optional parameter, `$flush_type`, that controls how the flushing will be carried out. By default the `$flush_type` used is `Z_FINISH`. Other valid values for `$flush_type` are `Z_NO_FLUSH`, `Z_SYNC_FLUSH`, `Z_FULL_FLUSH` and `Z_BLOCK`. It is

strongly recommended that you only set the `flush_type` parameter if you fully understand the implications of what it does - overuse of `flush` can seriously degrade the level of compression achieved. See the `zlib` documentation for details.

Returns true on success.

tell

Usage is

```
$z->tell()
tell $z
```

Returns the uncompressed file offset.

eof

Usage is

```
$z->eof();
eof($z);
```

Returns true if the `close` method has been called.

seek

```
$z->seek($position, $whence);
seek($z, $position, $whence);
```

Provides a sub-set of the `seek` functionality, with the restriction that it is only legal to seek forward in the output file/buffer. It is a fatal error to attempt to seek backward.

Empty parts of the file/buffer will have NULL (0x00) bytes written to them.

The `$whence` parameter takes one the usual values, namely SEEK_SET, SEEK_CUR or SEEK_END.

Returns 1 on success, 0 on failure.

binmode

Usage is

```
$z->binmode
binmode $z ;
```

This is a noop provided for completeness.

opened

```
$z->opened()
```

Returns true if the object currently refers to a opened file/buffer.

autoflush

```
my $prev = $z->autoflush()
my $prev = $z->autoflush(EXPR)
```

If the `$z` object is associated with a file or a filehandle, this method returns the current autoflush setting for the underlying filehandle. If `EXPR` is present, and is non-zero, it will enable flushing after every write/print operation.

If `$z` is associated with a buffer, this method has no effect and always returns `undef`.

Note that the special variable `$|` **cannot** be used to set or retrieve the autoflush setting.

input_line_number

```
$z->input_line_number()
$z->input_line_number(EXPR)
```

This method always returns `undef` when compressing.

fileno

```
$z->fileno()
fileno($z)
```

If the `$z` object is associated with a file or a filehandle, `fileno` will return the underlying file descriptor. Once the `close` method is called `fileno` will return `undef`.

If the `$z` object is associated with a buffer, this method will return `undef`.

close

```
$z->close() ;
close $z ;
```

Flushes any pending compressed data and then closes the output file/buffer.

For most versions of Perl this method will be automatically invoked if the IO::Compress::Gzip object is destroyed (either explicitly or by the variable with the reference to the object going out of scope). The exceptions are Perl versions 5.005 through 5.00504 and 5.8.0. In these cases, the `close` method will be called automatically, but not until global destruction of all live objects when the program is terminating.

Therefore, if you want your scripts to be able to run on all versions of Perl, you should call `close` explicitly and not rely on automatic closing.

Returns true on success, otherwise 0.

If the `AutoClose` option has been enabled when the IO::Compress::Gzip object was created, and the object is associated with a file, the underlying file will also be closed.

newStream([OPTS])

Usage is

```
$z->newStream( [OPTS] )
```

Closes the current compressed data stream and starts a new one.

OPTS consists of any of the options that are available when creating the `$z` object.

See the "Constructor Options" section for more details.

deflateParams

Usage is

```
$z->deflateParams
```

TODO

Importing

A number of symbolic constants are required by some methods in `IO::Compress::Gzip`. None are imported by default.

- :all

  Imports `gzip`, `$GzipError` and all symbolic constants that can be used by `IO::Compress::Gzip`. Same as doing this

  ```
  use IO::Compress::Gzip qw(gzip $GzipError :constants) ;
  ```

- :constants

  Import all symbolic constants. Same as doing this

  ```
  use IO::Compress::Gzip qw(:flush :level :strategy) ;
  ```

- :flush

  These symbolic constants are used by the `flush` method.

  ```
  Z_NO_FLUSH
  Z_PARTIAL_FLUSH
  Z_SYNC_FLUSH
  Z_FULL_FLUSH
  Z_FINISH
  Z_BLOCK
  ```

- :level

  These symbolic constants are used by the `Level` option in the constructor.

```
Z_NO_COMPRESSION
Z_BEST_SPEED
Z_BEST_COMPRESSION
Z_DEFAULT_COMPRESSION
```

- :strategy

  These symbolic constants are used by the `Strategy` option in the constructor.

```
Z_FILTERED
Z_HUFFMAN_ONLY
Z_RLE
Z_FIXED
Z_DEFAULT_STRATEGY
```

EXAMPLES

Apache::GZip Revisited

See IO::Compress::FAQ

Working with Net::FTP

See IO::Compress::FAQ

SUPPORT

General feedback/questions/bug reports should be sent to https://github.com/pmqs/IO-Copress/issues (preferred) or https://rt.cpan.org/Public/Dist/Display.html?Name=IO-Copress.

SEE ALSO

Compress::Zlib, *IO::Uncompress::Gunzip*, *IO::Compress::Deflate*, *IO::Uncompress::Inflate*, *IO::Compress::RawDeflate*, *IO::Uncompress::RawInflate*, *IO::Compress::Bzip2*, *IO::Uncompress::Bunzip2*, IO::Compress::Lzma, IO::Uncompress::UnLzma, IO::Compress::Xz, IO::Uncompress::UnXz, IO::Compress::Lzip, IO::Uncompress::UnLzip, IO::Compress::Lzop, IO::Uncompress::UnLzop, IO::Compress::Lzf, IO::Uncompress::UnLzf, IO::Compress::Zstd, IO::Uncompress::UnZstd, *IO::Uncompress::AnyInflate*, *IO::Uncompress::AnyUncompress*

IO::Compress::FAQ

File::GlobMapper, Archive::Zip, *Archive::Tar*, *IO::Zlib*

For RFC 1950, 1951 and 1952 see https://datatracker.ietf.org/doc/html/rfc1950, https://datatracker.ietf.org/doc/html/rfc1951 and https://datatracker.ietf.org/doc/html/rfc1952

The *zlib* compression library was written by Jean-loup Gailly `gzip@prep.ai.mit.edu` and Mark Adler `madler@alumni.caltech.edu`.

The primary site for the *zlib* compression library is http://www.zlib.org.

The primary site for the *zlib-ng* compression library is https://github.com/zlib-ng/zlib-ng.

The primary site for gzip is http://www.gzip.org.

AUTHOR

This module was written by Paul Marquess, `pmqs@cpan.org`.

MODIFICATION HISTORY

See the Changes file.

COPYRIGHT AND LICENSE

Copyright (c) 2005-2026 Paul Marquess. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

IO::Compress::RawDeflate

Write RFC 1951 files/buffers

SYNOPSIS

```
use IO::Compress::RawDeflate qw(rawdeflate $RawDeflateError) ;

my $status = rawdeflate $input => $output [,OPTS]
    or die "rawdeflate failed: $RawDeflateError\n";

my $z = IO::Compress::RawDeflate->new( $output [,OPTS] )
    or die "rawdeflate failed: $RawDeflateError\n";

$z->print($string);
$z->printf($format, $string);
$z->write($string);
$z->syswrite($string [, $length, $offset]);
$z->flush();
$z->tell();
$z->eof();
$z->seek($position, $whence);
$z->binmode();
$z->fileno();
$z->opened();
$z->autoflush();
$z->input_line_number();
$z->newStream( [OPTS] );

$z->deflateParams();
```

(continues on next page)

(continued from previous page)

```
$z->close() ;

$RawDeflateError ;

# IO::File mode

print $z $string;
printf $z $format, $string;
tell $z
eof $z
seek $z, $position, $whence
binmode $z
fileno $z
close $z ;
```

DESCRIPTION

This module provides a Perl interface that allows writing compressed data to files or buffer as defined in RFC 1951.

Note that RFC 1951 data is not a good choice of compression format to use in isolation, especially if you want to auto-detect it.

For reading RFC 1951 files/buffers, see the companion module *IO::Uncompress::RawInflate*.

Functional Interface

A top-level function, `rawdeflate`, is provided to carry out "one-shot" compression between buffers and/or files. For finer control over the compression process, see the "OO Interface" section.

```
use IO::Compress::RawDeflate qw(rawdeflate $RawDeflateError) ;

rawdeflate $input_filename_or_reference => $output_filename_or_
 ↪reference [,OPTS]
    or die "rawdeflate failed: $RawDeflateError\n";
```

The functional interface needs Perl5.005 or better.

rawdeflate $input_filename_or_reference => $output_filename_or_reference [, OPTS]

`rawdeflate` expects at least two parameters, `$input_filename_or_reference` and `$output_filename_or_reference` and zero or more optional parameters (see "Optional Parameters")

The `$input_filename_or_reference` parameter

The parameter, `$input_filename_or_reference`, is used to define the source of the uncompressed data.

It can take one of the following forms:

- A filename

  If the `$input_filename_or_reference` parameter is a simple scalar, it is assumed to be a filename. This file will be opened for reading and the input data will be read from it.

- A filehandle

  If the `$input_filename_or_reference` parameter is a filehandle, the input data will be read from it. The string '-' can be used as an alias for standard input.

- A scalar reference

  If `$input_filename_or_reference` is a scalar reference, the input data will be read from `$$input_filename_or_reference`.

- An array reference

  If `$input_filename_or_reference` is an array reference, each element in the array must be a filename.

  The input data will be read from each file in turn.

  The complete array will be walked to ensure that it only contains valid filenames before any data is compressed.

- An Input FileGlob string

  If `$input_filename_or_reference` is a string that is delimited by the characters "<" and ">" `rawdeflate` will assume that it is an *input fileglob string*. The input is the list of files that match the fileglob.

  See *File::GlobMapper* for more details.

If the `$input_filename_or_reference` parameter is any other type, `undef` will be returned.

The `$output_filename_or_reference` parameter

The parameter `$output_filename_or_reference` is used to control the destination of the compressed data. This parameter can take one of these forms.

- A filename

  If the `$output_filename_or_reference` parameter is a simple scalar, it is assumed to be a filename. This file will be opened for writing and the compressed data will be written to it.

- A filehandle

  If the `$output_filename_or_reference` parameter is a filehandle, the compressed data will be written to it. The string '-' can be used as an alias for standard output.

- A scalar reference

  If `$output_filename_or_reference` is a scalar reference, the compressed data will be stored in `$$output_filename_or_reference`.

- An Array Reference

  If `$output_filename_or_reference` is an array reference, the compressed data will be pushed onto the array.

- An Output FileGlob

  If `$output_filename_or_reference` is a string that is delimited by the characters "<" and ">" `rawdeflate` will assume that it is an *output fileglob string*. The output is the list of files that match the fileglob.

  When `$output_filename_or_reference` is an fileglob string, `$input_filename_or_reference` must also be a fileglob string. Anything else is an error.

  See *File::GlobMapper* for more details.

If the `$output_filename_or_reference` parameter is any other type, `undef` will be returned.

Notes

When `$input_filename_or_reference` maps to multiple files/buffers and `$output_filename_or_reference` is a single file/buffer the input files/buffers will be stored in `$output_filename_or_reference` as a concatenated series of compressed data streams.

Optional Parameters

The optional parameters for the one-shot function `rawdeflate` are (for the most part) identical to those used with the OO interface defined in the "Constructor Options" section. The exceptions are listed below

- `AutoClose => 0|1`

  This option applies to any input or output data streams to `rawdeflate` that are filehandles.

  If `AutoClose` is specified, and the value is true, it will result in all input and/or output filehandles being closed once `rawdeflate` has completed.

  This parameter defaults to 0.

- `BinModeIn => 0|1`

  This option is now a no-op. All files will be read in binmode.

- `Append => 0|1`

  The behaviour of this option is dependent on the type of output data stream.

  - A Buffer

    If Append is enabled, all compressed data will be append to the end of the output buffer. Otherwise the output buffer will be cleared before any compressed data is written to it.

- A Filename

  If Append is enabled, the file will be opened in append mode. Otherwise the contents of the file, if any, will be truncated before any compressed data is written to it.

- A Filehandle

  If Append is enabled, the filehandle will be positioned to the end of the file via a call to `seek` before any compressed data is written to it. Otherwise the file pointer will not be moved.

When Append is specified, and set to true, it will *append* all compressed data to the output data stream.

So when the output is a filehandle it will carry out a seek to the eof before writing any compressed data. If the output is a filename, it will be opened for appending. If the output is a buffer, all compressed data will be appended to the existing buffer.

Conversely when Append is not specified, or it is present and is set to false, it will operate as follows.

When the output is a filename, it will truncate the contents of the file before writing any compressed data. If the output is a filehandle its position will not be changed. If the output is a buffer, it will be wiped before any compressed data is output.

Defaults to 0.

Oneshot Examples

Here are a few example that show the capabilities of the module.

Streaming

This very simple command line example demonstrates the streaming capabilities of the module. The code reads data from STDIN, compresses it, and writes the compressed data to STDOUT.

```
$ echo hello world | perl -MIO::Compress::RawDeflate=rawdeflate -e
↪'rawdeflate \*STDIN => \*STDOUT' >output.1951
```

The special filename "-" can be used as a standin for both `\*STDIN` and `\*STDOUT`, so the above can be rewritten as

```
$ echo hello world | perl -MIO::Compress::RawDeflate=rawdeflate -e
↪'rawdeflate "-" => "-"' >output.1951
```

Compressing a file from the filesystem

To read the contents of the file `file1.txt` and write the compressed data to the file `file1.txt.1951`.

```
use strict ;
use warnings ;
```

(continues on next page)

(continued from previous page)

```
use IO::Compress::RawDeflate qw(rawdeflate $RawDeflateError) ;

my $input = "file1.txt";
rawdeflate $input => "$input.1951"
    or die "rawdeflate failed: $RawDeflateError\n";
```

Reading from a Filehandle and writing to an in-memory buffer

To read from an existing Perl filehandle, `$input`, and write the compressed data to a buffer, `$buffer`.

```
use strict ;
use warnings ;
use IO::Compress::RawDeflate qw(rawdeflate $RawDeflateError) ;
use IO::File ;

my $input = IO::File->new( "<file1.txt" )
    or die "Cannot open 'file1.txt': $!\n" ;
my $buffer ;
rawdeflate $input => \$buffer
    or die "rawdeflate failed: $RawDeflateError\n";
```

Compressing multiple files

To compress all files in the directory “/my/home” that match “*.txt” and store the compressed data in the same directory

```
use strict ;
use warnings ;
use IO::Compress::RawDeflate qw(rawdeflate $RawDeflateError) ;

rawdeflate '</my/home/*.txt>' => '<*.1951>'
    or die "rawdeflate failed: $RawDeflateError\n";
```

and if you want to compress each file one at a time, this will do the trick

```
use strict ;
use warnings ;
use IO::Compress::RawDeflate qw(rawdeflate $RawDeflateError) ;

for my $input ( glob "/my/home/*.txt" )
{
    my $output = "$input.1951" ;
    rawdeflate $input => $output
        or die "Error compressing '$input': $RawDeflateError\n";
}
```

OO Interface

Constructor

The format of the constructor for `IO::Compress::RawDeflate` is shown below

```
my $z = IO::Compress::RawDeflate->new( $output [,OPTS] )
    or die "IO::Compress::RawDeflate failed: $RawDeflateError\n";
```

The constructor takes one mandatory parameter, `$output`, defined below and zero or more `OPTS`, defined in "Constructor Options".

It returns an `IO::Compress::RawDeflate` object on success and `undef` on failure. The variable `$RawDeflateError` will contain an error message on failure.

If you are running Perl 5.005 or better the object, `$z`, returned from IO::Compress::RawDeflate can be used exactly like an *IO::File* filehandle. This means that all normal output file operations can be carried out with `$z`. For example, to write to a compressed file/buffer you can use either of these forms

```
$z->print("hello world\n");
print $z "hello world\n";
```

Below is a simple exaple of using the OO interface to create an output file `myfile.1951` and write some data to it.

```
my $filename = "myfile.1951";
my $z = IO::Compress::RawDeflate->new($filename)
    or die "IO::Compress::RawDeflate failed: $RawDeflateError\n";

$z->print("abcde");
$z->close();
```

See the "Examples" for more.

The mandatory parameter `$output` is used to control the destination of the compressed data. This parameter can take one of these forms.

- A filename

  If the `$output` parameter is a simple scalar, it is assumed to be a filename. This file will be opened for writing and the compressed data will be written to it.

- A filehandle

  If the `$output` parameter is a filehandle, the compressed data will be written to it. The string '-' can be used as an alias for standard output.

- A scalar reference

  If `$output` is a scalar reference, the compressed data will be stored in `$$output`.

If the `$output` parameter is any other type, `IO::Compress::RawDeflate`::new will return undef.

Constructor Options

`OPTS` is any combination of zero or more the following options:

- `AutoClose => 0|1`

  This option is only valid when the `$output` parameter is a filehandle. If specified, and the value is true, it will result in the `$output` being closed once either the `close` method is called or the `IO::Compress::RawDeflate` object is destroyed.

  This parameter defaults to 0.

- `Append => 0|1`

  Opens `$output` in append mode.

  The behaviour of this option is dependent on the type of `$output`.

  - A Buffer

    If `$output` is a buffer and Append is enabled, all compressed data will be append to the end of `$output`. Otherwise `$output` will be cleared before any data is written to it.

  - A Filename

    If `$output` is a filename and Append is enabled, the file will be opened in append mode. Otherwise the contents of the file, if any, will be truncated before any compressed data is written to it.

  - A Filehandle

    If `$output` is a filehandle, the file pointer will be positioned to the end of the file via a call to `seek` before any compressed data is written to it. Otherwise the file pointer will not be moved.

  This parameter defaults to 0.

- `Merge => 0|1`

  This option is used to compress input data and append it to an existing compressed data stream in `$output`. The end result is a single compressed data stream stored in `$output`.

  It is a fatal error to attempt to use this option when `$output` is not an RFC 1951 data stream.

  There are a number of other limitations with the `Merge` option:

  1. This module needs to have been built with zlib 1.2.1 or better to work. A fatal error will be thrown if `Merge` is used with an older version of zlib.

  2. If `$output` is a file or a filehandle, it must be seekable.

  This parameter defaults to 0.

- -Level

  Defines the compression level used by zlib. The value should either be a number between 0 and 9 (0 means no compression and 9 is maximum compression), or one of the symbolic constants defined below.

```
Z_NO_COMPRESSION
Z_BEST_SPEED
Z_BEST_COMPRESSION
Z_DEFAULT_COMPRESSION
```

The default is Z_DEFAULT_COMPRESSION.

Note, these constants are not imported by `IO::Compress::RawDeflate` by default.

```
use IO::Compress::RawDeflate qw(:strategy);
use IO::Compress::RawDeflate qw(:constants);
use IO::Compress::RawDeflate qw(:all);
```

- -Strategy

  Defines the strategy used to tune the compression. Use one of the symbolic constants defined below.

  ```
  Z_FILTERED
  Z_HUFFMAN_ONLY
  Z_RLE
  Z_FIXED
  Z_DEFAULT_STRATEGY
  ```

  The default is Z_DEFAULT_STRATEGY.

- `Strict => 0|1`

  This is a placeholder option.

Examples

Streaming

This very simple command line example demonstrates the streaming capabilities of the module. The code reads data from STDIN or all the files given on the commandline, compresses it, and writes the compressed data to STDOUT.

```
use strict ;
use warnings ;
use IO::Compress::RawDeflate qw(rawdeflate $RawDeflateError) ;

my $z = IO::Compress::RawDeflate->new("-", Stream => 1)
    or die "IO::Compress::RawDeflate failed: $RawDeflateError\n";

while (<>) {
    $z->print("abcde");
}
$z->close();
```

Note the use of `"-"` to means STDOUT. Alternatively you can use `\*STDOUT`.

Compressing a file from the filesystem

To read the contents of the file `file1.txt` and write the compressed data to the file `file1.txt.1951` there are a few options

Start by creating the compression object and opening the input file

```
use strict ;
use warnings ;
use IO::Compress::RawDeflate qw(rawdeflate $RawDeflateError) ;

my $input = "file1.txt";
my $z = IO::Compress::RawDeflate->new("file1.txt.1951")
    or die "IO::Compress::RawDeflate failed: $RawDeflateError\n";

# open the input file
open my $fh, "<", "file1.txt"
    or die "Cannot open file1.txt: $!\n";

# loop through the input file & write to the compressed file
while (<$fh>) {
    $z->print($_);
}

# not forgetting to close the compressed file
$z->close();
```

Methods

print

Usage is

```
$z->print($data)
print $z $data
```

Compresses and outputs the contents of the `$data` parameter. This has the same behaviour as the `print` built-in.

Returns true if successful.

printf

Usage is

```
$z->printf($format, $data)
printf $z $format, $data
```

Compresses and outputs the contents of the `$data` parameter.

Returns true if successful.

syswrite

Usage is

```
$z->syswrite $data
$z->syswrite $data, $length
$z->syswrite $data, $length, $offset
```

Compresses and outputs the contents of the `$data` parameter.

Returns the number of uncompressed bytes written, or `undef` if unsuccessful.

write

Usage is

```
$z->write $data
$z->write $data, $length
$z->write $data, $length, $offset
```

Compresses and outputs the contents of the `$data` parameter.

Returns the number of uncompressed bytes written, or `undef` if unsuccessful.

flush

Usage is

```
$z->flush;
$z->flush($flush_type);
```

Flushes any pending compressed data to the output file/buffer.

This method takes an optional parameter, `$flush_type`, that controls how the flushing will be carried out. By default the `$flush_type` used is `Z_FINISH`. Other valid values for `$flush_type` are `Z_NO_FLUSH`, `Z_SYNC_FLUSH`, `Z_FULL_FLUSH` and `Z_BLOCK`. It is strongly recommended that you only set the `flush_type` parameter if you fully understand the implications of what it does - overuse of `flush` can seriously degrade the level of compression achieved. See the `zlib` documentation for details.

Returns true on success.

tell

Usage is

```
$z->tell()
tell $z
```

Returns the uncompressed file offset.

eof

Usage is

```
$z->eof();
eof($z);
```

Returns true if the `close` method has been called.

seek

```
$z->seek($position, $whence);
seek($z, $position, $whence);
```

Provides a sub-set of the `seek` functionality, with the restriction that it is only legal to seek forward in the output file/buffer. It is a fatal error to attempt to seek backward.

Empty parts of the file/buffer will have NULL (0x00) bytes written to them.

The `$whence` parameter takes one the usual values, namely SEEK_SET, SEEK_CUR or SEEK_END.

Returns 1 on success, 0 on failure.

binmode

Usage is

```
$z->binmode
binmode $z ;
```

This is a noop provided for completeness.

opened

```
$z->opened()
```

Returns true if the object currently refers to a opened file/buffer.

autoflush

```
my $prev = $z->autoflush()
my $prev = $z->autoflush(EXPR)
```

If the `$z` object is associated with a file or a filehandle, this method returns the current autoflush setting for the underlying filehandle. If `EXPR` is present, and is non-zero, it will enable flushing after every write/print operation.

If `$z` is associated with a buffer, this method has no effect and always returns `undef`.

Note that the special variable `$|` **cannot** be used to set or retrieve the autoflush setting.

input_line_number

```
$z->input_line_number()
$z->input_line_number(EXPR)
```

This method always returns undef when compressing.

fileno

```
$z->fileno()
fileno($z)
```

If the $z object is associated with a file or a filehandle, fileno will return the underlying file descriptor. Once the close method is called fileno will return undef.

If the $z object is associated with a buffer, this method will return undef.

close

```
$z->close() ;
close $z ;
```

Flushes any pending compressed data and then closes the output file/buffer.

For most versions of Perl this method will be automatically invoked if the IO::Compress::RawDeflate object is destroyed (either explicitly or by the variable with the reference to the object going out of scope). The exceptions are Perl versions 5.005 through 5.00504 and 5.8.0. In these cases, the close method will be called automatically, but not until global destruction of all live objects when the program is terminating.

Therefore, if you want your scripts to be able to run on all versions of Perl, you should call close explicitly and not rely on automatic closing.

Returns true on success, otherwise 0.

If the AutoClose option has been enabled when the IO::Compress::RawDeflate object was created, and the object is associated with a file, the underlying file will also be closed.

newStream([OPTS])

Usage is

```
$z->newStream( [OPTS] )
```

Closes the current compressed data stream and starts a new one.

OPTS consists of any of the options that are available when creating the $z object.

See the “Constructor Options” section for more details.

deflateParams

Usage is

```
$z->deflateParams
```

TODO

Importing

A number of symbolic constants are required by some methods in `IO::Compress::RawDeflate`. None are imported by default.

- :all

  Imports `rawdeflate`, `$RawDeflateError` and all symbolic constants that can be used by `IO::Compress::RawDeflate`. Same as doing this

  ```
  use IO::Compress::RawDeflate qw(rawdeflate $RawDeflateError␣
  ↪:constants) ;
  ```

- :constants

  Import all symbolic constants. Same as doing this

  ```
  use IO::Compress::RawDeflate qw(:flush :level :strategy) ;
  ```

- :flush

  These symbolic constants are used by the `flush` method.

  ```
  Z_NO_FLUSH
  Z_PARTIAL_FLUSH
  Z_SYNC_FLUSH
  Z_FULL_FLUSH
  Z_FINISH
  Z_BLOCK
  ```

- :level

  These symbolic constants are used by the `Level` option in the constructor.

  ```
  Z_NO_COMPRESSION
  Z_BEST_SPEED
  Z_BEST_COMPRESSION
  Z_DEFAULT_COMPRESSION
  ```

- :strategy

  These symbolic constants are used by the `Strategy` option in the constructor.

  ```
  Z_FILTERED
  Z_HUFFMAN_ONLY
  Z_RLE
  ```

(continues on next page)

(continued from previous page)

```
Z_FIXED
Z_DEFAULT_STRATEGY
```

EXAMPLES

Apache::GZip Revisited

See IO::Compress::FAQ

Working with Net::FTP

See IO::Compress::FAQ

SUPPORT

General feedback/questions/bug reports should be sent to https://github.com/pmqs/IO-Compress/issues (preferred) or https://rt.cpan.org/Public/Dist/Display.html?Name=IO-Compress.

SEE ALSO

Compress::Zlib, *IO::Compress::Gzip*, *IO::Uncompress::Gunzip*, *IO::Compress::Deflate*, *IO::Uncompress::Inflate*, *IO::Uncompress::RawInflate*, *IO::Compress::Bzip2*, *IO::Uncompress::Bunzip2*, IO::Compress::Lzma, IO::Uncompress::UnLzma, IO::Compress::Xz, IO::Uncompress::UnXz, IO::Compress::Lzip, IO::Uncompress::UnLzip, IO::Compress::Lzop, IO::Uncompress::UnLzop, IO::Compress::Lzf, IO::Uncompress::UnLzf, IO::Compress::Zstd, IO::Uncompress::UnZstd, *IO::Uncompress::AnyInflate*, *IO::Uncompress::AnyUncompress*

IO::Compress::FAQ

File::GlobMapper, Archive::Zip, *Archive::Tar*, *IO::Zlib*

For RFC 1950, 1951 and 1952 see https://datatracker.ietf.org/doc/html/rfc1950, https://datatracker.ietf.org/doc/html/rfc1951 and https://datatracker.ietf.org/doc/html/rfc1952

The *zlib* compression library was written by Jean-loup Gailly `gzip@prep.ai.mit.edu` and Mark Adler `madler@alumni.caltech.edu`.

The primary site for the *zlib* compression library is http://www.zlib.org.

The primary site for the *zlib-ng* compression library is https://github.com/zlib-ng/zlib-ng.

The primary site for gzip is http://www.gzip.org.

AUTHOR

This module was written by Paul Marquess, `pmqs@cpan.org`.

MODIFICATION HISTORY

See the Changes file.

COPYRIGHT AND LICENSE

Copyright (c) 2005-2026 Paul Marquess. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

IO::Compress::Zip

Write zip files/buffers

SYNOPSIS

```
use IO::Compress::Zip qw(zip $ZipError) ;

my $status = zip $input => $output [,OPTS]
    or die "zip failed: $ZipError\n";

my $z = IO::Compress::Zip->new( $output [,OPTS] )
    or die "zip failed: $ZipError\n";

$z->print($string);
$z->printf($format, $string);
$z->write($string);
$z->syswrite($string [, $length, $offset]);
$z->flush();
$z->tell();
$z->eof();
$z->seek($position, $whence);
$z->binmode();
$z->fileno();
$z->opened();
$z->autoflush();
$z->input_line_number();
$z->newStream( [OPTS] );

$z->deflateParams();

$z->close() ;

$ZipError ;

# IO::File mode

print $z $string;
printf $z $format, $string;
```

(continues on next page)

(continued from previous page)

```
tell $z
eof $z
seek $z, $position, $whence
binmode $z
fileno $z
close $z ;
```

DESCRIPTION

This module provides a Perl interface that allows writing zip compressed data to files or buffer.

The primary purpose of this module is to provide streaming write access to zip files and buffers.

At present the following compression methods are supported by IO::Compress::Zip

- Store (0)
- Deflate (8)
- Bzip2 (12)

  To write Bzip2 content, the module `IO::Uncompress::Bunzip2` must be installed.
- Lzma (14)

  To write LZMA content, the module `IO::Uncompress::UnLzma` must be installed.
- Zstandard (93)

  To write Zstandard content, the module `IO::Compress::Zstd` must be installed.
- Xz (95)

  To write Xz content, the module `IO::Uncompress::UnXz` must be installed.

For reading zip files/buffers, see the companion module *IO::Uncompress::Unzip*.

Functional Interface

A top-level function, `zip`, is provided to carry out "one-shot" compression between buffers and/or files. For finer control over the compression process, see the "OO Interface" section.

```
use IO::Compress::Zip qw(zip $ZipError) ;

zip $input_filename_or_reference => $output_filename_or_reference [,
↪OPTS]
    or die "zip failed: $ZipError\n";
```

The functional interface needs Perl5.005 or better.

zip $input_filename_or_reference => $output_filename_or_reference [, OPTS]

`zip` expects at least two parameters, `$input_filename_or_reference` and `$output_filename_or_reference` and zero or more optional parameters (see "Optional Parameters")

The `$input_filename_or_reference` parameter

The parameter, `$input_filename_or_reference`, is used to define the source of the uncompressed data.

It can take one of the following forms:

- A filename

  If the `$input_filename_or_reference` parameter is a simple scalar, it is assumed to be a filename. This file will be opened for reading and the input data will be read from it.

- A filehandle

  If the `$input_filename_or_reference` parameter is a filehandle, the input data will be read from it. The string '-' can be used as an alias for standard input.

- A scalar reference

  If `$input_filename_or_reference` is a scalar reference, the input data will be read from `$$input_filename_or_reference`.

- An array reference

  If `$input_filename_or_reference` is an array reference, each element in the array must be a filename.

  The input data will be read from each file in turn.

  The complete array will be walked to ensure that it only contains valid filenames before any data is compressed.

- An Input FileGlob string

  If `$input_filename_or_reference` is a string that is delimited by the characters "<" and ">" `zip` will assume that it is an *input fileglob string*. The input is the list of files that match the fileglob.

  See *File::GlobMapper* for more details.

If the `$input_filename_or_reference` parameter is any other type, `undef` will be returned.

In addition, if `$input_filename_or_reference` corresponds to a filename from the filesystem, a number of zip file header fields will be populated by default using the following attributes from the input file

- the full filename contained in `$input_filename_or_reference`
- the file protection attributes
- the UID/GID for the file
- the file timestamps

If you do not want to use these defaults they can be overridden by explicitly setting one, or more, of the `Name`, `Time`, `TextFlag`, `ExtAttr`, `exUnixN` and `exTime` options or by setting the `Minimal` parameter.

The `$output_filename_or_reference` parameter

The parameter `$output_filename_or_reference` is used to control the destination of the compressed data. This parameter can take one of these forms.

- A filename

  If the `$output_filename_or_reference` parameter is a simple scalar, it is assumed to be a filename. This file will be opened for writing and the compressed data will be written to it.

- A filehandle

  If the `$output_filename_or_reference` parameter is a filehandle, the compressed data will be written to it. The string '-' can be used as an alias for standard output.

- A scalar reference

  If `$output_filename_or_reference` is a scalar reference, the compressed data will be stored in `$$output_filename_or_reference`.

- An Array Reference

  If `$output_filename_or_reference` is an array reference, the compressed data will be pushed onto the array.

- An Output FileGlob

  If `$output_filename_or_reference` is a string that is delimited by the characters "<" and ">" `zip` will assume that it is an *output fileglob string*. The output is the list of files that match the fileglob.

  When `$output_filename_or_reference` is an fileglob string, `$input_filename_or_reference` must also be a fileglob string. Anything else is an error.

  See *File::GlobMapper* for more details.

If the `$output_filename_or_reference` parameter is any other type, `undef` will be returned.

Notes

When `$input_filename_or_reference` maps to multiple files/buffers and `$output_filename_or_reference` is a single file/buffer the input files/buffers will each be stored in `$output_filename_or_reference` as a distinct entry.

Optional Parameters

The optional parameters for the one-shot function `zip` are (for the most part) identical to those used with the OO interface defined in the "Constructor Options" section. The exceptions are listed below

- `AutoClose => 0|1`

  This option applies to any input or output data streams to `zip` that are filehandles.

  If `AutoClose` is specified, and the value is true, it will result in all input and/or output filehandles being closed once `zip` has completed.

  This parameter defaults to 0.

- `BinModeIn => 0|1`

  This option is now a no-op. All files will be read in binmode.

- `Append => 0|1`

  The behaviour of this option is dependent on the type of output data stream.

  - A Buffer

    If Append is enabled, all compressed data will be append to the end of the output buffer. Otherwise the output buffer will be cleared before any compressed data is written to it.

  - A Filename

    If Append is enabled, the file will be opened in append mode. Otherwise the contents of the file, if any, will be truncated before any compressed data is written to it.

  - A Filehandle

    If Append is enabled, the filehandle will be positioned to the end of the file via a call to `seek` before any compressed data is written to it. Otherwise the file pointer will not be moved.

  When Append is specified, and set to true, it will *append* all compressed data to the output data stream.

  So when the output is a filehandle it will carry out a seek to the eof before writing any compressed data. If the output is a filename, it will be opened for appending. If the output is a buffer, all compressed data will be appended to the existing buffer.

  Conversely when Append is not specified, or it is present and is set to false, it will operate as follows.

  When the output is a filename, it will truncate the contents of the file before writing any compressed data. If the output is a filehandle its position will not be changed. If the output is a buffer, it will be wiped before any compressed data is output.

  Defaults to 0.

Oneshot Examples

Here are a few example that show the capabilities of the module.

Streaming

This very simple command line example demonstrates the streaming capabilities of the module. The code reads data from STDIN, compresses it, and writes the compressed data to STDOUT.

```
$ echo hello world | perl -MIO::Compress::Zip=zip -e 'zip \*STDIN =>␣
↪\*STDOUT' >output.zip
```

The special filename "-" can be used as a standin for both `\*STDIN` and `\*STDOUT`, so the above can be rewritten as

```
$ echo hello world | perl -MIO::Compress::Zip=zip -e 'zip "-" => "-"
↪' >output.zip
```

One problem with creating a zip archive directly from STDIN can be demonstrated by looking at the contents of the zip file, output.zip, that we have just created.

```
$ unzip -l output.zip
Archive:  output.zip
Length      Date    Time    Name
---------  ---------- -----   ----
    12  2019-08-16 22:21
---------                     -------
    12                     1 file
```

The archive member (filename) used is the empty string.

If that doesn't suit your needs, you can explicitly set the filename used in the zip archive by specifying the Name option, like so

```
echo hello world | perl -MIO::Compress::Zip=zip -e 'zip "-" => "-",␣
↪Name => "hello.txt"' >output.zip
```

Now the contents of the zip file looks like this

```
$ unzip -l output.zip
Archive:  output.zip
Length      Date    Time    Name
---------  ---------- -----   ----
    12  2019-08-16 22:22   hello.txt
---------                     -------
    12                     1 file
```

Compressing a file from the filesystem

To read the contents of the file `file1.txt` and write the compressed data to the file `file1.txt.zip`.

```
use strict ;
use warnings ;
use IO::Compress::Zip qw(zip $ZipError) ;
```

(continues on next page)

(continued from previous page)

```
my $input = "file1.txt";
zip $input => "$input.zip"
    or die "zip failed: $ZipError\n";
```

Reading from a Filehandle and writing to an in-memory buffer

To read from an existing Perl filehandle, `$input`, and write the compressed data to a buffer, `$buffer`.

```
use strict ;
use warnings ;
use IO::Compress::Zip qw(zip $ZipError) ;
use IO::File ;

my $input = IO::File->new( "<file1.txt" )
    or die "Cannot open 'file1.txt': $!\n" ;
my $buffer ;
zip $input => \$buffer
    or die "zip failed: $ZipError\n";
```

Compressing multiple files

To create a zip file, `output.zip`, that contains the compressed contents of the files `alpha.txt` and `beta.txt`

```
use strict ;
use warnings ;
use IO::Compress::Zip qw(zip $ZipError) ;

zip [ 'alpha.txt', 'beta.txt' ] => 'output.zip'
    or die "zip failed: $ZipError\n";
```

Alternatively, rather than having to explicitly name each of the files that you want to compress, you could use a fileglob to select all the `txt` files in the current directory, as follows

```
use strict ;
use warnings ;
use IO::Compress::Zip qw(zip $ZipError) ;

my @files = <*.txt>;
zip \@files => 'output.zip'
    or die "zip failed: $ZipError\n";
```

or more succinctly

```
zip [ <*.txt> ] => 'output.zip'
    or die "zip failed: $ZipError\n";
```

OO Interface

Constructor

The format of the constructor for `IO::Compress::Zip` is shown below

```
my $z = IO::Compress::Zip->new( $output [,OPTS] )
    or die "IO::Compress::Zip failed: $ZipError\n";
```

The constructor takes one mandatory parameter, `$output`, defined below and zero or more `OPTS`, defined in "Constructor Options".

It returns an `IO::Compress::Zip` object on success and `undef` on failure. The variable `$ZipError` will contain an error message on failure.

If you are running Perl 5.005 or better the object, `$z`, returned from IO::Compress::Zip can be used exactly like an *IO::File* filehandle. This means that all normal output file operations can be carried out with `$z`. For example, to write to a compressed file/buffer you can use either of these forms

```
$z->print("hello world\n");
print $z "hello world\n";
```

Below is a simple exaple of using the OO interface to create an output file `myfile.zip` and write some data to it.

```
my $filename = "myfile.zip";
my $z = IO::Compress::Zip->new($filename)
    or die "IO::Compress::Zip failed: $ZipError\n";

$z->print("abcde");
$z->close();
```

See the "Examples" for more.

The mandatory parameter `$output` is used to control the destination of the compressed data. This parameter can take one of these forms.

- A filename

  If the `$output` parameter is a simple scalar, it is assumed to be a filename. This file will be opened for writing and the compressed data will be written to it.

- A filehandle

  If the `$output` parameter is a filehandle, the compressed data will be written to it. The string '-' can be used as an alias for standard output.

- A scalar reference

  If `$output` is a scalar reference, the compressed data will be stored in `$$output`.

If the `$output` parameter is any other type, `IO::Compress::Zip`::new will return undef.

Constructor Options

`OPTS` is any combination of zero or more the following options:

- `AutoClose => 0|1`

  This option is only valid when the `$output` parameter is a filehandle. If specified, and the value is true, it will result in the `$output` being closed once either the `close` method is called or the `IO::Compress::Zip` object is destroyed.

  This parameter defaults to 0.

- `Append => 0|1`

  Opens `$output` in append mode.

  The behaviour of this option is dependent on the type of `$output`.

  - A Buffer

    If `$output` is a buffer and Append is enabled, all compressed data will be append to the end of `$output`. Otherwise `$output` will be cleared before any data is written to it.

  - A Filename

    If `$output` is a filename and Append is enabled, the file will be opened in append mode. Otherwise the contents of the file, if any, will be truncated before any compressed data is written to it.

  - A Filehandle

    If `$output` is a filehandle, the file pointer will be positioned to the end of the file via a call to `seek` before any compressed data is written to it. Otherwise the file pointer will not be moved.

  This parameter defaults to 0.

File Naming Options

A quick bit of zip file terminology – A zip archive consists of one or more *archive members*, where each member has an associated filename, known as the *archive member name*.

The options listed in this section control how the *archive member name* (or filename) is stored the zip archive.

- `Name => $string`

  This option is used to explicitly set the *archive member name* in the zip archive to `$string`. Most of the time you don't need to make use of this option. By default when adding a filename to the zip archive, the *archive member name* will match the filename.

  You should only need to use this option if you want the *archive member name* to be different from the uncompressed filename or when the input is a filehandle or a buffer.

  The default behaviour for what *archive member name* is used when the `Name` option is *not* specified depends on the form of the `$input` parameter:

- If the `$input` parameter is a filename, the value of `$input` will be used for the *archive member name* .
- If the `$input` parameter is not a filename, the *archive member name* will be an empty string.

Note that both the `CanonicalName` and `FilterName` options can modify the value used for the *archive member name*.

Also note that you should set the `Efs` option to true if you are working with UTF8 filenames.

- `CanonicalName => 0|1`

  This option controls whether the *archive member name* is *normalized* into Unix format before being written to the zip file.

  It is recommended that you enable this option unless you really need to create a non-standard Zip file.

  This is what APPNOTE.TXT has to say on what should be stored in the zip filename header field.

```
The name of the file, with optional relative path.
The path stored should not contain a drive or
device letter, or a leading slash.  All slashes
should be forward slashes '/' as opposed to
backwards slashes '\' for compatibility with Amiga
and UNIX file systems etc.
```

  This option defaults to **false**.

- `FilterName => sub { ... }`

  This option allow the *archive member* name to be modified before it is written to the zip file.

  This option takes a parameter that must be a reference to a sub. On entry to the sub the `$_` variable will contain the name to be filtered. If no filename is available `$_` will contain an empty string.

  The value of `$_` when the sub returns will be used as the *archive member name*.

  Note that if `CanonicalName` is enabled, a normalized filename will be passed to the sub.

  If you use `FilterName` to modify the filename, it is your responsibility to keep the filename in Unix format.

  Although this option can be used with the OO interface, it is of most use with the one-shot interface. For example, the code below shows how `FilterName` can be used to remove the path component from a series of filenames before they are stored in `$zipfile`.

```
sub compressTxtFiles
{
    my $zipfile = shift ;
    my $dir     = shift ;
```

(continues on next page)

(continued from previous page)

```
    zip [ <$dir/*.txt> ] => $zipfile,
        FilterName => sub { s[^$dir/][] } ;
}
```

- `Efs => 0|1`

  This option controls setting of the "Language Encoding Flag" (EFS) in the zip archive. When set, the filename and comment fields for the zip archive MUST be valid UTF-8.

  If the string used for the filename and/or comment is not valid UTF-8 when this option is true, the script will die with a "wide character" error.

  Note that this option only works with Perl 5.8.4 or better.

  This option defaults to **false**.

Overall Zip Archive Structure

- `Minimal => 1|0`

  If specified, this option will disable the creation of all extra fields in the zip local and central headers. So the `exTime`, `exUnix2`, `exUnixN`, `ExtraFieldLocal` and `ExtraFieldCentral` options will be ignored.

  This parameter defaults to 0.

- `Stream => 0|1`

  This option controls whether the zip file/buffer output is created in streaming mode.

  Note that when outputting to a file with streaming mode disabled (`Stream` is 0), the output file must be seekable.

  The default is 1.

- `Zip64 => 0|1`

  Create a Zip64 zip file/buffer. This option is used if you want to store files larger than 4 Gig or store more than 64K files in a single zip archive.

  `Zip64` will be automatically set, as needed, if working with the one-shot interface when the input is either a filename or a scalar reference.

  If you intend to manipulate the Zip64 zip files created with this module using an external zip/unzip, make sure that it supports Zip64.

  In particular, if you are using Info-Zip you need to have zip version 3.x or better to update a Zip64 archive and unzip version 6.x to read a zip64 archive.

  The default is 0.

Deflate Compression Options

- -Level

  Defines the compression level used by zlib. The value should either be a number between 0 and 9 (0 means no compression and 9 is maximum compression), or one of the symbolic constants defined below.

```
Z_NO_COMPRESSION
Z_BEST_SPEED
Z_BEST_COMPRESSION
Z_DEFAULT_COMPRESSION
```

  The default is Z_DEFAULT_COMPRESSION.

  Note, these constants are not imported by `IO::Compress::Zip` by default.

```
use IO::Compress::Zip qw(:strategy);
use IO::Compress::Zip qw(:constants);
use IO::Compress::Zip qw(:all);
```

- -Strategy

  Defines the strategy used to tune the compression. Use one of the symbolic constants defined below.

```
Z_FILTERED
Z_HUFFMAN_ONLY
Z_RLE
Z_FIXED
Z_DEFAULT_STRATEGY
```

  The default is Z_DEFAULT_STRATEGY.

Bzip2 Compression Options

- `BlockSize100K => number`

  Specify the number of 100K blocks bzip2 uses during compression.

  Valid values are from 1 to 9, where 9 is best compression.

  This option is only valid if the `Method` is ZIP_CM_BZIP2. It is ignored otherwise.

  The default is 1.

- `WorkFactor => number`

  Specifies how much effort bzip2 should take before resorting to a slower fallback compression algorithm.

  Valid values range from 0 to 250, where 0 means use the default value 30.

  This option is only valid if the `Method` is ZIP_CM_BZIP2. It is ignored otherwise.

  The default is 0.

Lzma and Xz Compression Options

- `Preset => number`

  Used to choose the LZMA compression preset.

  Valid values are 0-9 and `LZMA_PRESET_DEFAULT`.

  0 is the fastest compression with the lowest memory usage and the lowest compression.

  9 is the slowest compression with the highest memory usage but with the best compression.

  This option is only valid if the `Method` is ZIP_CM_LZMA. It is ignored otherwise.

  Defaults to `LZMA_PRESET_DEFAULT` (6).

- `Extreme => 0|1`

  Makes LZMA compression a lot slower, but a small compression gain.

  This option is only valid if the `Method` is ZIP_CM_LZMA. It is ignored otherwise.

  Defaults to 0.

Other Options

- `Time => $number`

  Sets the last modified time field in the zip header to $number.

  This field defaults to the time the `IO::Compress::Zip` object was created if this option is not specified and the `$input` parameter is not a filename.

- `ExtAttr => $attr`

  This option controls the "external file attributes" field in the central header of the zip file. This is a 4 byte field.

  If you are running a Unix derivative this value defaults to

  ```
  0100644 << 16
  ```

  This should allow read/write access to any files that are extracted from the zip file/buffer`.

  For all other systems it defaults to 0.

- `exTime => [$atime, $mtime, $ctime]`

  This option expects an array reference with exactly three elements: `$atime`, `mtime` and `$ctime`. These correspond to the last access time, last modification time and creation time respectively.

  It uses these values to set the extended timestamp field (ID is "UT") in the local zip header using the three values, $atime, $mtime, $ctime. In addition it sets the extended timestamp field in the central zip header using `$mtime`.

  If any of the three values is `undef` that time value will not be used. So, for example, to set only the `$mtime` you would use this

```
exTime => [undef, $mtime, undef]
```

If the `Minimal` option is set to true, this option will be ignored.

By default no extended time field is created.

- `exUnix2 => [$uid, $gid]`

  This option expects an array reference with exactly two elements: `$uid` and `$gid`. These values correspond to the numeric User ID (UID) and Group ID (GID) of the owner of the files respectively.

  When the `exUnix2` option is present it will trigger the creation of a Unix2 extra field (ID is "Ux") in the local zip header. This will be populated with `$uid` and `$gid`. An empty Unix2 extra field will also be created in the central zip header.

  Note - The UID & GID are stored as 16-bit integers in the "Ux" field. Use `exUnixN` if your UID or GID are 32-bit.

  If the `Minimal` option is set to true, this option will be ignored.

  By default no Unix2 extra field is created.

- `exUnixN => [$uid, $gid]`

  This option expects an array reference with exactly two elements: `$uid` and `$gid`. These values correspond to the numeric User ID (UID) and Group ID (GID) of the owner of the files respectively.

  When the `exUnixN` option is present it will trigger the creation of a UnixN extra field (ID is "ux") in both the local and central zip headers. This will be populated with `$uid` and `$gid`. The UID & GID are stored as 32-bit integers.

  If the `Minimal` option is set to true, this option will be ignored.

  By default no UnixN extra field is created.

- `Comment => $comment`

  Stores the contents of `$comment` in the Central File Header of the zip file.

  Set the `Efs` option to true if you want to store a UTF8 comment.

  By default, no comment field is written to the zip file.

- `ZipComment => $comment`

  Stores the contents of `$comment` in the End of Central Directory record of the zip file.

  By default, no comment field is written to the zip file.

- `Method => $method`

  Controls which compression method is used. At present the compression methods supported are: Store (no compression at all), Deflate, Bzip2, Zstd, Xz and Lzma.

  The symbols ZIP_CM_STORE, ZIP_CM_DEFLATE, ZIP_CM_BZIP2, ZIP_CM_ZSTD, ZIP_CM_XZ and ZIP_CM_LZMA are used to select the compression method.

  These constants are not imported by `IO::Compress::Zip` by default.

```
use IO::Compress::Zip qw(:zip_method);
use IO::Compress::Zip qw(:constants);
use IO::Compress::Zip qw(:all);
```

Note that to create Bzip2 content, the module `IO::Compress::Bzip2` must be installed. A fatal error will be thrown if you attempt to create Bzip2 content when `IO::Compress::Bzip2` is not available.

Note that to create Lzma content, the module `IO::Compress::Lzma` must be installed. A fatal error will be thrown if you attempt to create Lzma content when `IO::Compress::Lzma` is not available.

Note that to create Xz content, the module `IO::Compress::Xz` must be installed. A fatal error will be thrown if you attempt to create Xz content when `IO::Compress::Xz` is not available.

Note that to create Zstd content, the module `IO::Compress::Zstd` must be installed. A fatal error will be thrown if you attempt to create Zstd content when `IO::Compress::Zstd` is not available.

The default method is ZIP_CM_DEFLATE.

- `TextFlag => 0|1`

  This parameter controls the setting of a bit in the zip central header. It is used to signal that the data stored in the zip file/buffer is probably text.

  In one-shot mode this flag will be set to true if the Perl `-T` operator thinks the file contains text.

  The default is 0.

- `ExtraFieldLocal => $data`
- `ExtraFieldCentral => $data`

  The `ExtraFieldLocal` option is used to store additional metadata in the local header for the zip file/buffer. The `ExtraFieldCentral` does the same for the matching central header.

  An extra field consists of zero or more subfields. Each subfield consists of a two byte header followed by the subfield data.

  The list of subfields can be supplied in any of the following formats

```
ExtraFieldLocal => [$id1, $data1,
                    $id2, $data2,
                    ...
                   ]

ExtraFieldLocal => [ [$id1 => $data1],
                     [$id2 => $data2],
                     ...
                   ]

ExtraFieldLocal => { $id1 => $data1,
                     $id2 => $data2,
```

(continues on next page)

(continued from previous page)

```
                    ...
                }
```

Where `$id1`, `$id2` are two byte subfield ID's.

If you use the hash syntax, you have no control over the order in which the Extra-SubFields are stored, plus you cannot have SubFields with duplicate ID.

Alternatively the list of subfields can by supplied as a scalar, thus

```
ExtraField => $rawdata
```

In this case `IO::Compress::Zip` will check that `$rawdata` consists of zero or more conformant sub-fields.

The Extended Time field (ID "UT"), set using the `exTime` option, and the Unix2 extra field (ID "Ux), set using the `exUnix2` option, are examples of extra fields.

If the `Minimal` option is set to true, this option will be ignored.

The maximum size of an extra field 65535 bytes.

- `Strict => 0|1`

  This is a placeholder option.

Examples

Streaming

This very simple command line example demonstrates the streaming capabilities of the module. The code reads data from STDIN or all the files given on the commandline, compresses it, and writes the compressed data to STDOUT.

```
use strict ;
use warnings ;
use IO::Compress::Zip qw(zip $ZipError) ;

my $z = IO::Compress::Zip->new("-", Stream => 1)
    or die "IO::Compress::Zip failed: $ZipError\n";

while (<>) {
    $z->print("abcde");
}
$z->close();
```

Note the use of `"-"` to means `STDOUT`. Alternatively you can use `\*STDOUT`.

One problem with creating a zip archive directly from STDIN can be demonstrated by looking at the contents of the zip file, output.zip, that we have just created (assumg you have redirected it to a file called `output.zip`).

```
$ unzip -l output.zip
Archive:  output.zip
```

(continues on next page)

(continued from previous page)

```
Length      Date    Time    Name
---------  ---------- -----   ----
    12  2019-08-16 22:21
---------                     -------
    12                    1 file
```

The archive member (filename) used is the empty string.

If that doesn't suit your needs, you can explicitly set the filename used in the zip archive by specifying the Name option, like so

```
my $z = IO::Compress::Zip->new("-", Name => "hello.txt", Stream =>␣
↪1)
```

Now the contents of the zip file looks like this

```
$ unzip -l output.zip
Archive:  output.zip
Length      Date    Time    Name
---------  ---------- -----   ----
    12  2019-08-16 22:22   hello.txt
---------                     -------
    12                    1 file
```

Compressing a file from the filesystem

To read the contents of the file `file1.txt` and write the compressed data to the file `file1.txt.zip` there are a few options

Start by creating the compression object and opening the input file

```
use strict ;
use warnings ;
use IO::Compress::Zip qw(zip $ZipError) ;

my $input = "file1.txt";
my $z = IO::Compress::Zip->new("file1.txt.zip")
    or die "IO::Compress::Zip failed: $ZipError\n";

# open the input file
open my $fh, "<", "file1.txt"
    or die "Cannot open file1.txt: $!\n";

# loop through the input file & write to the compressed file
while (<$fh>) {
    $z->print($_);
}

# not forgetting to close the compressed file
$z->close();
```

Compressing multiple files

To create a zip file, `output.zip`, that contains the compressed contents of the files `alpha.txt` and `beta.txt`

```
use strict ;
use warnings ;
use IO::Compress::Zip qw(zip $ZipError) ;

my $z = IO::Compress::Zip->new("output.zip", Name => "alpha.txt")
    or die "IO::Compress::Zip failed: $ZipError\n";

# open the input file
open my $fh, "<", "file1.txt"
    or die "Cannot open file1.txt: $!\n";

# loop through the input file & write to the compressed file
while (<$fh>) {
    $z->print($_);
}

# move to next file
$z->newStream(Name => "beta.txt")

while (<$fh>) {
    $z->print($_);
}

$z->close();
```

Methods

print

Usage is

```
$z->print($data)
print $z $data
```

Compresses and outputs the contents of the `$data` parameter. This has the same behaviour as the `print` built-in.

Returns true if successful.

printf

Usage is

```
$z->printf($format, $data)
printf $z $format, $data
```

Compresses and outputs the contents of the `$data` parameter.

Returns true if successful.

syswrite

Usage is

```
$z->syswrite $data
$z->syswrite $data, $length
$z->syswrite $data, $length, $offset
```

Compresses and outputs the contents of the `$data` parameter.

Returns the number of uncompressed bytes written, or `undef` if unsuccessful.

write

Usage is

```
$z->write $data
$z->write $data, $length
$z->write $data, $length, $offset
```

Compresses and outputs the contents of the `$data` parameter.

Returns the number of uncompressed bytes written, or `undef` if unsuccessful.

flush

Usage is

```
$z->flush;
$z->flush($flush_type);
```

Flushes any pending compressed data to the output file/buffer.

This method takes an optional parameter, `$flush_type`, that controls how the flushing will be carried out. By default the `$flush_type` used is `Z_FINISH`. Other valid values for `$flush_type` are `Z_NO_FLUSH`, `Z_SYNC_FLUSH`, `Z_FULL_FLUSH` and `Z_BLOCK`. It is strongly recommended that you only set the `flush_type` parameter if you fully understand the implications of what it does - overuse of `flush` can seriously degrade the level of compression achieved. See the `zlib` documentation for details.

Returns true on success.

tell

Usage is

```
$z->tell()
tell $z
```

Returns the uncompressed file offset.

eof

Usage is

```
$z->eof();
eof($z);
```

Returns true if the `close` method has been called.

seek

```
$z->seek($position, $whence);
seek($z, $position, $whence);
```

Provides a sub-set of the `seek` functionality, with the restriction that it is only legal to seek forward in the output file/buffer. It is a fatal error to attempt to seek backward.

Empty parts of the file/buffer will have NULL (0x00) bytes written to them.

The `$whence` parameter takes one the usual values, namely SEEK_SET, SEEK_CUR or SEEK_END.

Returns 1 on success, 0 on failure.

binmode

Usage is

```
$z->binmode
binmode $z ;
```

This is a noop provided for completeness.

opened

```
$z->opened()
```

Returns true if the object currently refers to a opened file/buffer.

autoflush

```
my $prev = $z->autoflush()
my $prev = $z->autoflush(EXPR)
```

If the `$z` object is associated with a file or a filehandle, this method returns the current autoflush setting for the underlying filehandle. If `EXPR` is present, and is non-zero, it will enable flushing after every write/print operation.

If `$z` is associated with a buffer, this method has no effect and always returns `undef`.

Note that the special variable `$|` **cannot** be used to set or retrieve the autoflush setting.

input_line_number

```
$z->input_line_number()
$z->input_line_number(EXPR)
```

This method always returns undef when compressing.

fileno

```
$z->fileno()
fileno($z)
```

If the $z object is associated with a file or a filehandle, fileno will return the underlying file descriptor. Once the close method is called fileno will return undef.

If the $z object is associated with a buffer, this method will return undef.

close

```
$z->close() ;
close $z ;
```

Flushes any pending compressed data and then closes the output file/buffer.

For most versions of Perl this method will be automatically invoked if the IO::Compress::Zip object is destroyed (either explicitly or by the variable with the reference to the object going out of scope). The exceptions are Perl versions 5.005 through 5.00504 and 5.8.0. In these cases, the close method will be called automatically, but not until global destruction of all live objects when the program is terminating.

Therefore, if you want your scripts to be able to run on all versions of Perl, you should call close explicitly and not rely on automatic closing.

Returns true on success, otherwise 0.

If the AutoClose option has been enabled when the IO::Compress::Zip object was created, and the object is associated with a file, the underlying file will also be closed.

newStream([OPTS])

Usage is

```
$z->newStream( [OPTS] )
```

Closes the current compressed data stream and starts a new one.

OPTS consists of any of the options that are available when creating the $z object.

See the “Constructor Options” section for more details.

deflateParams

Usage is

```
$z->deflateParams
```

TODO

Importing

A number of symbolic constants are required by some methods in `IO::Compress::Zip`. None are imported by default.

- :all

  Imports `zip`, `$ZipError` and all symbolic constants that can be used by `IO::Compress::Zip`. Same as doing this

  ```
  use IO::Compress::Zip qw(zip $ZipError :constants) ;
  ```

- :constants

  Import all symbolic constants. Same as doing this

  ```
  use IO::Compress::Zip qw(:flush :level :strategy :zip_method) ;
  ```

- :flush

  These symbolic constants are used by the `flush` method.

  ```
  Z_NO_FLUSH
  Z_PARTIAL_FLUSH
  Z_SYNC_FLUSH
  Z_FULL_FLUSH
  Z_FINISH
  Z_BLOCK
  ```

- :level

  These symbolic constants are used by the `Level` option in the constructor.

  ```
  Z_NO_COMPRESSION
  Z_BEST_SPEED
  Z_BEST_COMPRESSION
  Z_DEFAULT_COMPRESSION
  ```

- :strategy

  These symbolic constants are used by the `Strategy` option in the constructor.

  ```
  Z_FILTERED
  Z_HUFFMAN_ONLY
  Z_RLE
  Z_FIXED
  Z_DEFAULT_STRATEGY
  ```

- :zip_method

  These symbolic constants are used by the `Method` option in the constructor.

```
ZIP_CM_STORE
ZIP_CM_DEFLATE
ZIP_CM_BZIP2
```

EXAMPLES

Apache::GZip Revisited

See IO::Compress::FAQ

Working with Net::FTP

See IO::Compress::FAQ

SUPPORT

General feedback/questions/bug reports should be sent to https://github.com/pmqs/IO-Compress/issues (preferred) or https://rt.cpan.org/Public/Dist/Display.html?Name=IO-Compress.

SEE ALSO

Compress::Zlib, *IO::Compress::Gzip*, *IO::Uncompress::Gunzip*, *IO::Compress::Deflate*, *IO::Uncompress::Inflate*, *IO::Compress::RawDeflate*, *IO::Uncompress::RawInflate*, *IO::Compress::Bzip2*, *IO::Uncompress::Bunzip2*, IO::Compress::Lzma, IO::Uncompress::UnLzma, IO::Compress::Xz, IO::Uncompress::UnXz, IO::Compress::Lzip, IO::Uncompress::UnLzip, IO::Compress::Lzop, IO::Uncompress::UnLzop, IO::Compress::Lzf, IO::Uncompress::UnLzf, IO::Compress::Zstd, IO::Uncompress::UnZstd, *IO::Uncompress::AnyInflate*, *IO::Uncompress::AnyUncompress*

IO::Compress::FAQ

File::GlobMapper, Archive::Zip, *Archive::Tar*, *IO::Zlib*

For RFC 1950, 1951 and 1952 see https://datatracker.ietf.org/doc/html/rfc1950, https://datatracker.ietf.org/doc/html/rfc1951 and https://datatracker.ietf.org/doc/html/rfc1952

The *zlib* compression library was written by Jean-loup Gailly `gzip@prep.ai.mit.edu` and Mark Adler `madler@alumni.caltech.edu`.

The primary site for the *zlib* compression library is http://www.zlib.org.

The primary site for the *zlib-ng* compression library is https://github.com/zlib-ng/zlib-ng.

The primary site for gzip is http://www.gzip.org.

AUTHOR

This module was written by Paul Marquess, pmqs@cpan.org.

MODIFICATION HISTORY

See the Changes file.

COPYRIGHT AND LICENSE

Copyright (c) 2005-2026 Paul Marquess. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

IO::Dir

supply object methods for directory handles

SYNOPSIS

```
use IO::Dir;
my $d = IO::Dir->new(".");
if (defined $d) {
    while (defined($_ = $d->read)) { something($_); }
    $d->rewind;
    while (defined($_ = $d->read)) { something_else($_); }
    undef $d;
}

tie my %dir, 'IO::Dir', ".";
foreach (keys %dir) {
    print $_, " " , $dir{$_}->size,"\n";
}
```

DESCRIPTION

The IO::Dir package provides two interfaces to perl's directory reading routines.

The first interface is an object approach. IO::Dir provides an object constructor and methods, which are just wrappers around perl's built in directory reading routines.

- new ([DIRNAME])

  new is the constructor for IO::Dir objects. It accepts one optional argument which, if given, new will pass to open

The following methods are wrappers for the directory related functions built into perl (the trailing 'dir' has been removed from the names). See *perlfunc* for details of these functions.

- open (DIRNAME)

- read ()
- seek (POS)
- tell ()
- rewind ()
- close ()

`IO::Dir` also provides an interface to reading directories via a tied hash. The tied hash extends the interface beyond just the directory reading routines by the use of `lstat`, from the `File::stat` package, `unlink`, `rmdir` and `utime`.

- tie %hash, 'IO::Dir', DIRNAME [, OPTIONS]

The keys of the hash will be the names of the entries in the directory. Reading a value from the hash will be the result of calling `File::stat::lstat`. Deleting an element from the hash will delete the corresponding file or subdirectory, provided that `DIR_UNLINK` is included in the `OPTIONS`.

Assigning to an entry in the hash will cause the time stamps of the file to be modified. If the file does not exist then it will be created. Assigning a single integer to a hash element will cause both the access and modification times to be changed to that value. Alternatively a reference to an array of two values can be passed. The first array element will be used to set the access time and the second element will be used to set the modification time.

SEE ALSO

File::stat

AUTHOR

Graham Barr. Currently maintained by the Perl Porters. Please report all bugs at https://github.com/Perl/perl5/issues.

COPYRIGHT

Copyright (c) 1997-2003 Graham Barr gbarr@pobox.com. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

IO::File

supply object methods for filehandles

SYNOPSIS

```
use IO::File;

my $fh = IO::File->new();
if ($fh->open("< file")) {
```

(continues on next page)

(continued from previous page)

```
    print <$fh>;
    $fh->close;
}

my $fh = IO::File->new("> file");
if (defined $fh) {
    print $fh "bar\n";
    $fh->close;
}

my $fh = IO::File->new("file", "r");
if (defined $fh) {
    print <$fh>;
    undef $fh;       # automatically closes the file
}

my $fh = IO::File->new("file", O_WRONLY|O_APPEND);
if (defined $fh) {
    print $fh "corge\n";

    my $pos = $fh->getpos;
    $fh->setpos($pos);

    undef $fh;       # automatically closes the file
}

autoflush STDOUT 1;
```

DESCRIPTION

`IO::File` inherits from `IO::Handle` and `IO::Seekable`. It extends these classes with methods that are specific to file handles.

CONSTRUCTOR

- new (FILENAME [,MODE [,PERMS]])

  Creates an `IO::File`. If it receives any parameters, they are passed to the method open; if the open fails, the object is destroyed. Otherwise, it is returned to the caller.

- new_tmpfile

  Creates an `IO::File` opened for read/write on a newly created temporary file. On systems where this is possible, the temporary file is anonymous (i.e. it is unlinked after creation, but held open). If the temporary file cannot be created or opened, the `IO::File` object is destroyed. Otherwise, it is returned to the caller.

METHODS

- open(FILENAME [,MODE [,PERMS]])
- open(FILENAME, IOLAYERS)

  open accepts one, two or three parameters. With one parameter, it is just a front end for the built-in open function. With two or three parameters, the first parameter is a filename that may include whitespace or other special characters, and the second parameter is the open mode, optionally followed by a file permission value.

  If `IO::File::open` receives a Perl mode string (">", "+<", etc.) or an ANSI C fopen() mode string ("w", "r+", etc.), it uses the basic Perl open operator (but protects any special characters).

  If `IO::File::open` is given a numeric mode, it passes that mode and the optional permissions value to the Perl `sysopen` operator. The permissions default to 0666.

  If `IO::File::open` is given a mode that includes the `:` character, it passes all the three arguments to the three-argument open operator.

  For convenience, `IO::File` exports the O_XXX constants from the Fcntl module, if this module is available.

NOTE

Some operating systems may perform `IO::File::new()` or `IO::File::open()` on a directory without errors. This behavior is not portable and not suggested for use. Using `opendir()` and `readdir()` or `IO::Dir` are suggested instead.

SEE ALSO

perlfunc, *"I/O Operators" in perlop*, *IO::Handle*, *IO::Seekable*, *IO::Dir*

HISTORY

Derived from FileHandle.pm by Graham Barr <gbarr@pobox.com>.

Functions

Other Functions

new_tmpfile

Open an anonymous temporary file and return it as an `IO::File`.

IO::Handle

supply object methods for I/O handles

SYNOPSIS

```
use IO::Handle;

my $io = IO::Handle->new();
if ($io->fdopen(fileno(STDIN),"r")) {
    print $io->getline;
    $io->close;
}

my $io = IO::Handle->new();
if ($io->fdopen(fileno(STDOUT),"w")) {
    $io->print("Some text\n");
}

# setvbuf is not available by default on Perls 5.8.0 and later.
use IO::Handle '_IOLBF';
$io->setvbuf(my $buffer_var, _IOLBF, 1024);

undef $io;       # automatically closes the file if it's open

autoflush STDOUT 1;
```

DESCRIPTION

`IO::Handle` is the base class for all other IO handle classes. It is not intended that objects of `IO::Handle` would be created directly, but instead `IO::Handle` is inherited from by several other classes in the IO hierarchy.

If you are reading this documentation, looking for a replacement for the `FileHandle` package, then I suggest you read the documentation for `IO::File` too.

CONSTRUCTOR

- new ()

  Creates a new `IO::Handle` object.

- new_from_fd (FD, MODE)

  Creates an `IO::Handle` like `new` does. It requires two parameters, which are passed to the method `fdopen`; if the fdopen fails, the object is destroyed. Otherwise, it is returned to the caller.

METHODS

See *perlfunc* for complete descriptions of each of the following supported `IO::Handle` methods, which are just front ends for the corresponding built-in functions:

```
$io->close
$io->eof
$io->fcntl( FUNCTION, SCALAR )
```

(continues on next page)

(continued from previous page)

```
$io->fileno
$io->format_write( [FORMAT_NAME] )
$io->getc
$io->ioctl( FUNCTION, SCALAR )
$io->read ( BUF, LEN, [OFFSET] )
$io->print ( ARGS )
$io->printf ( FMT, [ARGS] )
$io->say ( ARGS )
$io->stat
$io->sysread ( BUF, LEN, [OFFSET] )
$io->syswrite ( BUF, [LEN, [OFFSET]] )
$io->truncate ( LEN )
```

See *perlvar* for complete descriptions of each of the following supported `IO::Handle` methods. All of them return the previous value of the attribute and takes an optional single argument that when given will set the value. If no argument is given the previous value is unchanged (except for $io->autoflush will actually turn ON autoflush by default).

```
$io->autoflush ( [BOOL] )                    $|
$io->format_page_number( [NUM] )             $%
$io->format_lines_per_page( [NUM] )          $=
$io->format_lines_left( [NUM] )              $-
$io->format_name( [STR] )                    $~
$io->format_top_name( [STR] )                $^
$io->input_line_number( [NUM])               $.
```

The following methods are not supported on a per-filehandle basis.

```
IO::Handle->format_line_break_characters( [STR] ) $:
IO::Handle->format_formfeed( [STR])               $^L
IO::Handle->output_field_separator( [STR] )       $,
IO::Handle->output_record_separator( [STR] )      $\

IO::Handle->input_record_separator( [STR] )       $/
```

Furthermore, for doing normal I/O you might need these:

- $io->fdopen (FD, MODE)

  `fdopen` is like an ordinary open except that its first parameter is not a filename but rather a file handle name, an IO::Handle object, or a file descriptor number. (For the documentation of the open method, see *IO::File*.)

- $io->opened

  Returns true if the object is currently a valid file descriptor, false otherwise.

- $io->getline

  This works like <$io> described in *"I/O Operators" in perlop* except that it's more readable and can be safely called in a list context but still returns just one line. If used as the conditional within a `while` or C-style `for` loop, however, you will need to emulate the functionality of <$io> with `defined($_ = $io->getline)`.

- $io->getlines

  This works like <$io> when called in a list context to read all the remaining lines in a file, except that it's more readable. It will also croak() if accidentally called in a scalar context.

- $io->ungetc (ORD)

  Pushes a character with the given ordinal value back onto the given handle's input stream. Only one character of pushback per handle is guaranteed.

- $io->write (BUF, LEN [, OFFSET])

  This `write` is somewhat like `write` found in C, in that it is the opposite of read. The wrapper for the perl `write` function is called `format_write`. However, whilst the C `write` function returns the number of bytes written, this `write` function simply returns true if successful (like `print`). A more C-like `write` is `syswrite` (see above).

- $io->error

  Returns a true value if the given handle has experienced any errors since it was opened or since the last call to `clearerr`, or if the handle is invalid. It only returns false for a valid handle with no outstanding errors.

- $io->clearerr

  Clear the given handle's error indicator. Returns -1 if the handle is invalid, 0 otherwise.

- $io->sync

  `sync` synchronizes a file's in-memory state with that on the physical medium. `sync` does not operate at the perlio api level, but operates on the file descriptor (similar to sysread, sysseek and systell). This means that any data held at the perlio api level will not be synchronized. To synchronize data that is buffered at the perlio api level you must use the flush method. `sync` is not implemented on all platforms. Returns "0 but true" on success, `undef` on error, `undef` for an invalid handle. See fsync(3c).

- $io->flush

  `flush` causes perl to flush any buffered data at the perlio api level. Any unread data in the buffer will be discarded, and any unwritten data will be written to the underlying file descriptor. Returns "0 but true" on success, `undef` on error.

- $io->printflush (ARGS)

  Turns on autoflush, print ARGS and then restores the autoflush status of the `IO::Handle` object. Returns the return value from print.

- $io->blocking ([BOOL])

  If called with an argument `blocking` will turn on non-blocking IO if `BOOL` is false, and turn it off if `BOOL` is true.

  `blocking` will return the value of the previous setting, or the current setting if `BOOL` is not given.

  If an error occurs `blocking` will return undef and `$!` will be set.

- binmode([LAYER])

  `binmode` sets `binmode` on the underlying `IO` object, as documented in `perldoc -f binmode`.

  `binmode` accepts one optional parameter, which is the layer to be passed on to the `binmode` call.

If the C functions setbuf() and/or setvbuf() are available, then `IO::Handle::setbuf` and `IO::Handle::setvbuf` set the buffering policy for an IO::Handle. The calling sequences for the Perl functions are the same as their C counterparts–including the constants `_IOFBF`, `_IOLBF`, and `_IONBF` for setvbuf()–except that the buffer parameter specifies a scalar variable to use as a buffer. You should only change the buffer before any I/O, or immediately after calling flush.

WARNING: The IO::Handle::setvbuf() is not available by default on Perls 5.8.0 and later because setvbuf() is rather specific to using the stdio library, while Perl prefers the new perlio subsystem instead.

WARNING: A variable used as a buffer by `setbuf` or `setvbuf` **must not be modified** in any way until the IO::Handle is closed or `setbuf` or `setvbuf` is called again, or memory corruption may result! Remember that the order of global destruction is undefined, so even if your buffer variable remains in scope until program termination, it may be undefined before the file IO::Handle is closed. Note that you need to import the constants `_IOFBF`, `_IOLBF`, and `_IONBF` explicitly. Like C, setbuf returns nothing. setvbuf returns "0 but true", on success, `undef` on failure.

Lastly, there is a special method for working under **-T** and setuid/gid scripts:

- $io->untaint

  Marks the object as taint-clean, and as such data read from it will also be considered taint-clean. Note that this is a very trusting action to take, and appropriate consideration for the data source and potential vulnerability should be kept in mind. Returns 0 on success, -1 if setting the taint-clean flag failed. (eg invalid handle)

NOTE

An `IO::Handle` object is a reference to a symbol/GLOB reference (see the *Symbol* package). Some modules that inherit from `IO::Handle` may want to keep object related variables in the hash table part of the GLOB. In an attempt to prevent modules trampling on each other I propose the that any such module should prefix its variables with its own name separated by _'s. For example the IO::Socket module keeps a `timeout` variable in 'io_socket_timeout'.

SEE ALSO

perlfunc, *"I/O Operators" in perlop*, *IO::File*

BUGS

Due to backwards compatibility, all filehandles resemble objects of class `IO::Handle`, or actually classes derived from that class. They actually aren't. Which means you can't derive your own class from `IO::Handle` and inherit those methods.

HISTORY

Derived from FileHandle.pm by Graham Barr <gbarr@pobox.com>

Functions

Other Functions

blocking

Query or set blocking mode on the handle.

ungetc

Push one character back onto the handle, to be read again next.

error

Report whether the handle's error indicator is set.

clearerr

Clear the handle's error and end-of-file indicators.

untaint

Mark the handle as a trusted source, so data read from it is not tainted.

flush

Write out any buffered data on the handle.

setbuf

Not available on this build: croaks when called on a handle open for writing.

setvbuf

Not available on this build: croaks when called.

sync

Flush the handle's data through to the storage device.

getlines

Read every remaining line from the handle and return them as a list.

getline

Read one line from the handle and return it, or undef at end of file.

gets

An old name for `getline`.

IO::Pipe

supply object methods for pipes

SYNOPSIS

```
use IO::Pipe;

$pipe = IO::Pipe->new();

if($pid = fork()) { # Parent
    $pipe->reader();

    while(<$pipe>) {
        ...
    }

}
elsif(defined $pid) { # Child
    $pipe->writer();

    print $pipe ...
}

or

$pipe = IO::Pipe->new();

$pipe->reader(qw(ls -l));

while(<$pipe>) {
    ...
}
```

DESCRIPTION

`IO::Pipe` provides an interface to creating pipes between processes.

CONSTRUCTOR

- new ([READER, WRITER])

  Creates an `IO::Pipe`, which is a reference to a newly created symbol (see the *Symbol* package). `IO::Pipe::new` optionally takes two arguments, which should be objects blessed into `IO::Handle`, or a subclass thereof. These two objects will be used for the system call to `pipe`. If no arguments are given then method `handles` is called on the new `IO::Pipe` object.

  These two handles are held in the array part of the GLOB until either `reader` or `writer` is called.

METHODS

- reader ([ARGS])

  The object is re-blessed into a sub-class of `IO::Handle`, and becomes a handle at the reading end of the pipe. If `ARGS` are given then `fork` is called and `ARGS` are passed to exec.

- writer ([ARGS])

  The object is re-blessed into a sub-class of `IO::Handle`, and becomes a handle at the writing end of the pipe. If `ARGS` are given then `fork` is called and `ARGS` are passed to exec.

- handles ()

  This method is called during construction by `IO::Pipe::new` on the newly created `IO::Pipe` object. It returns an array of two objects blessed into `IO::Pipe::End`, or a subclass thereof.

SEE ALSO

IO::Handle

AUTHOR

Graham Barr. Currently maintained by the Perl Porters. Please report all bugs at https://github.com/Perl/perl5/issues.

COPYRIGHT

Copyright (c) 1996-8 Graham Barr gbarr@pobox.com. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

IO::Poll

Object interface to system poll call

SYNOPSIS

```
use IO::Poll qw(POLLRDNORM POLLWRNORM POLLIN POLLHUP);

$poll = IO::Poll->new();

$poll->mask($input_handle => POLLIN);
$poll->mask($output_handle => POLLOUT);

$poll->poll($timeout);

$ev = $poll->events($input);
```

DESCRIPTION

`IO::Poll` is a simple interface to the system level poll routine.

METHODS

- mask (IO [, EVENT_MASK])

  If EVENT_MASK is given, then, if EVENT_MASK is non-zero, IO is added to the list of file descriptors and the next call to poll will check for any event specified in EVENT_MASK. If EVENT_MASK is zero then IO will be removed from the list of file descriptors.

  If EVENT_MASK is not given then the return value will be the current event mask value for IO.

- poll ([TIMEOUT])

  Call the system level poll routine. If TIMEOUT is not specified then the call will block. Returns the number of handles which had events happen, or -1 on error.

- events (IO)

  Returns the event mask which represents the events that happened on IO during the last call to `poll`.

- remove (IO)

  Remove IO from the list of file descriptors for the next poll.

- handles([EVENT_MASK])

  Returns a list of handles. If EVENT_MASK is not given then a list of all handles known will be returned. If EVENT_MASK is given then a list of handles will be returned which had one of the events specified by EVENT_MASK happen during the last call to `poll`.

SEE ALSO

poll(2), *IO::Handle*, *IO::Select*

AUTHOR

Graham Barr. Currently maintained by the Perl Porters. Please report all bugs at https://github.com/Perl/perl5/issues.

COPYRIGHT

Copyright (c) 1997-8 Graham Barr gbarr@pobox.com. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

IO::Pty

Pseudo TTY object class

VERSION

1.31

SYNOPSIS

```
use IO::Pty;

$pty = IO::Pty->new;

$slave  = $pty->slave;

foreach $val (1..10) {
    print $pty "$val\n";
    $_ = <$slave>;
    print "$_";
}

close($slave);
```

DESCRIPTION

`IO::Pty` provides an interface to allow the creation of a pseudo tty.

`IO::Pty` inherits from *IO::Handle* and so provide all the methods defined by the *IO::Handle* package.

Please note that pty creation is very system-dependent. If you have problems, see *IO::Tty* for help.

CONSTRUCTOR

- new

  The new constructor takes no arguments and returns a new file object which is the master side of the pseudo tty.

METHODS

- ttyname()

  Returns the name of the slave pseudo tty. On UNIX machines this will be the pathname of the device. Use this name for informational purpose only, to get a slave filehandle, use slave().

- slave()

  The `slave` method will return the slave filehandle of the given master pty, opening it anew if necessary. If IO::Stty is installed, you can then call `$slave->stty()` to modify the terminal settings.

- close_slave()

  The slave filehandle will be closed and destroyed. This is necessary in the parent after forking to get rid of the open filehandle, otherwise the parent will not notice if the child exits. Subsequent calls of `slave()` will return a newly opened slave filehandle.

- make_slave_controlling_terminal()

  This will set the slave filehandle as the controlling terminal of the current process, which will become a session leader, so this should only be called by a child process after a fork(), e.g. in the callback to `sync_exec()` (see Proc::SyncExec). See the `try` script (also `test.pl`) for an example how to correctly spawn a subprocess.

- set_raw()

  Will set the pty to raw. Note that this is a one-way operation, you need IO::Stty to set the terminal settings to anything else.

  On some systems, the master pty is not a tty. This method checks for that and returns success anyway on such systems. Note that this method must be called on the slave, and probably should be called on the master, just to be sure, i.e.

```
$pty->slave->set_raw();
$pty->set_raw();
```

- clone_winsize_from(*FH)

  Gets the terminal size from filehandle FH (which must be a terminal) and transfers it to the pty. Returns true on success and undef on failure. Note that this must be called upon the *slave*, i.e.

```
$pty->slave->clone_winsize_from(\*STDIN);
```

  On some systems, the master pty also isatty. I actually have no idea if setting terminal sizes there is passed through to the slave, so if this method is called for a master that is not a tty, it silently returns OK.

See the `try` script for example code how to propagate SIGWINCH.

- get_winsize()

  Returns the terminal size, in a 4-element list.

```
($row, $col, $xpixel, $ypixel) = $tty->get_winsize()
```

- set_winsize($row, $col, $xpixel, $ypixel)

  Sets the terminal size. If not specified, `$xpixel` and `$ypixel` are set to 0. As with `clone_winsize_from`, this must be called upon the *slave*.

SEE ALSO

IO::Tty, *IO::Tty::Constant*, *IO::Handle*, Expect, Proc::SyncExec

Source code and issue tracker at https://github.com/cpan-authors/IO-Tty.

AUTHORS

Originally by Graham Barr <gbarr@pobox.com>, based on the Ptty module by Nick Ing-Simmons <nik@tiuk.ti.com>.

Heavily rewritten by Roland Giersig <RGiersig@cpan.org>.

Currently maintained by Todd Rinaldo.

Contains copyrighted stuff from openssh v3.0p1, authored by Tatu Ylonen ylo@cs.hut.fi, Markus Friedl and Todd C. Miller Todd.Miller@courtesan.com.

COPYRIGHT

Now all code is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

Nevertheless the above AUTHORS retain their copyrights to the various parts and want to receive credit if their source code is used. See the source for details.

DISCLAIMER

THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

In other words: Use at your own risk. Provided as is. Your mileage may vary. Read the source, Luke!

And finally, just to be sure:

Any Use of This Product, in Any Manner Whatsoever, Will Increase the Amount of Disorder in the Universe. Although No Liability Is Implied Herein, the Consumer Is Warned That This Process Will Ultimately Lead to the Heat Death of the Universe.

Functions

Pseudo-terminals

pty_allocate

A new pseudo-terminal: its master and slave descriptors and the slave's name.

pty_allocate

A new pseudo-terminal: its master and slave descriptors and the slave's name.

Synopsis

```
my ($master_fd, $slave_fd, $name) = IO::Pty::pty_allocate();
```

What you get back

Three values, both descriptors above 2 - or an empty list when no way of getting a pty worked. `IO::Pty->new` calls it; with `use warnings` each way that failed warns, and with `$IO::Tty::DEBUG` true each one tried is named on stderr.

IO::Seekable

supply seek based methods for I/O objects

SYNOPSIS

```
use IO::Seekable;
package IO::Something;
@ISA = qw(IO::Seekable);
```

DESCRIPTION

`IO::Seekable` does not have a constructor of its own as it is intended to be inherited by other `IO::Handle` based objects. It provides methods which allow seeking of the file descriptors.

- $io->getpos

  Returns an opaque value that represents the current position of the IO::File, or `undef` if this is not possible (eg an unseekable stream such as a terminal, pipe or socket). If the fgetpos() function is available in your C library it is used to implements getpos, else perl emulates getpos using C's ftell() function.

- $io->setpos

  Uses the value of a previous getpos call to return to a previously visited position. Returns “0 but true” on success, `undef` on failure.

See *perlfunc* for complete descriptions of each of the following supported `IO::Seekable` methods, which are just front ends for the corresponding built-in functions:

- $io->seek (POS, WHENCE)

  Seek the IO::File to position POS, relative to WHENCE:

  – WHENCE=0 (SEEK_SET)

    POS is absolute position. (Seek relative to the start of the file)

  – WHENCE=1 (SEEK_CUR)

    POS is an offset from the current position. (Seek relative to current)

  – WHENCE=2 (SEEK_END)

    POS is an offset from the end of the file. (Seek relative to end)

  The SEEK_* constants can be imported from the `Fcntl` module if you don’t wish to use the numbers `0` 1 or 2 in your code.

  Returns 1 upon success, `0` otherwise.

- $io->sysseek(POS, WHENCE)

  Similar to $io->seek, but sets the IO::File’s position using the system call lseek(2) directly, so will confuse most perl IO operators except sysread and syswrite (see *perlfunc* for full details)

  Returns the new position, or `undef` on failure. A position of zero is returned as the string `"0 but true"`

- $io->tell

  Returns the IO::File’s current position, or -1 on error.

SEE ALSO

perlfunc, *“I/O Operators” in perlop*, *IO::Handle IO::File*

HISTORY

Derived from FileHandle.pm by Graham Barr gbarr@pobox.com

Functions

Other Functions

getpos

Return an opaque marker for the handle’s current position.

setpos

Return the handle to a position previously recorded by `getpos`.

IO::Select

OO interface to the `select` system call

SYNOPSIS

```
use IO::Select;

my $s = IO::Select->new();

$s->add(\*STDIN);
$s->add($some_handle);

my @ready1 = $s->can_read($timeout);

my @ready2 = IO::Select->new(@handles)->can_read(0);
```

DESCRIPTION

The `IO::Select` package implements an object approach to the system `select` function call. It allows the user to see what IO handles, see *IO::Handle*, are ready for reading, writing or have an exception pending.

CONSTRUCTOR

- new ([HANDLES])

  The constructor creates a new object and optionally initialises it with a set of handles.

METHODS

- add (HANDLES)

  Add the list of handles to the `IO::Select` object. It is these values that will be returned when an event occurs. `IO::Select` keeps these values in a cache which is indexed by the `fileno` of the handle, so if more than one handle with the same `fileno` is specified then only the last one is cached.

  Each handle can be an `IO::Handle` object, an integer or an array reference where the first element is an `IO::Handle` or an integer.

- remove (HANDLES)

  Remove all the given handles from the object. This method also works by the `fileno` of the handles. So the exact handles that were added need not be passed, just handles that have an equivalent `fileno`

- exists (HANDLE)

  Returns a true value (actually the handle itself) if it is present. Returns undef otherwise.

- handles

  Return an array of all registered handles.

- can_read ([TIMEOUT])

  Return an array of handles that are ready for reading. `TIMEOUT` is the maximum amount of time to wait before returning an empty list (with `$!` unchanged), in seconds, possibly fractional. If `TIMEOUT` is not given and any handles are registered then the call will block indefinitely. Upon error, an empty list is returned, with `$!` set to indicate the error. To distinguish between timeout and error, set `$!` to zero before calling this method, and check it after an empty list is returned.

- can_write ([TIMEOUT])

  Same as `can_read` except check for handles that can be written to.

- has_exception ([TIMEOUT])

  Same as `can_read` except check for handles that have an exception condition, for example pending out-of-band data.

- count ()

  Returns the number of handles that the object will check for when one of the `can_` methods is called or the object is passed to the `select` static method.

- bits()

  Return the bit string suitable as argument to the core select() call.

- select (READ, WRITE, EXCEPTION [, TIMEOUT])

  `select` is a static method, that is you call it with the package name like `new`. `READ`, `WRITE` and `EXCEPTION` are either `undef` or `IO::Select` objects. `TIMEOUT` is optional and has the same effect as for the core select call.

  If at least one handle is ready for the specified kind of operation, the result will be an array of 3 elements, each a reference to an array which will hold the handles that are ready for reading, writing and have exceptions respectively. Upon timeout, an empty list is returned, with `$!` unchanged. Upon error, an empty list is returned, with `$!` set to indicate the error. To distinguish between timeout and error, set `$!` to zero before calling this method, and check it after an empty list is returned.

EXAMPLE

Here is a short example which shows how `IO::Select` could be used to write a server which communicates with several sockets while also listening for more connections on a listen socket

```
use IO::Select;
use IO::Socket;
```

(continues on next page)

(continued from previous page)

```
my $lsn = IO::Socket::INET->new(Listen => 1, LocalPort => 8080);
my $sel = IO::Select->new( $lsn );

while(my @ready = $sel->can_read) {
    foreach my $fh (@ready) {
        if($fh == $lsn) {
            # Create a new socket
            my $new = $lsn->accept;
            $sel->add($new);
        }
        else {
            # Process socket

            # Maybe we have finished with the socket
            $sel->remove($fh);
            $fh->close;
        }
    }
}
```

AUTHOR

Graham Barr. Currently maintained by the Perl Porters. Please report all bugs at https://github.com/Perl/perl5/issues.

COPYRIGHT

Copyright (c) 1997-8 Graham Barr gbarr@pobox.com. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

IO::Socket

Object interface to socket communications

SYNOPSIS

```
use strict;
use warnings;

use IO::Socket qw(AF_INET AF_UNIX);

# create a new AF_INET socket
my $sock = IO::Socket->new(Domain => AF_INET);
# which is the same as
$sock = IO::Socket::INET->new();

# create a new AF_UNIX socket
```

(continues on next page)

(continued from previous page)

```
$sock = IO::Socket->new(Domain => AF_UNIX);
# which is the same as
$sock = IO::Socket::UNIX->new();
```

DESCRIPTION

`IO::Socket` provides an object-oriented, *IO::Handle*-based interface to creating and using sockets via *Socket*, which provides a near one-to-one interface to the C socket library.

`IO::Socket` is a base class that really only defines methods for those operations which are common to all types of sockets. Operations which are specific to a particular socket domain have methods defined in subclasses of `IO::Socket`. See *IO::Socket::INET*, *IO::Socket::UNIX*, and *IO::Socket::IP* for examples of such a subclass.

`IO::Socket` will export all functions (and constants) defined by *Socket*.

CONSTRUCTOR ARGUMENTS

Given that `IO::Socket` doesn't have attributes in the traditional sense, the following arguments, rather than attributes, can be passed into the constructor.

Constructor arguments should be passed in `Key => 'Value'` pairs.

The only required argument is *"Domain" in IO::Socket*.

Blocking

```
my $sock = IO::Socket->new(..., Blocking => 1);
$sock = IO::Socket->new(..., Blocking => 0);
```

If defined but false, the socket will be set to non-blocking mode. If not specified it defaults to 1 (blocking mode).

Domain

```
my $sock = IO::Socket->new(Domain => IO::Socket::AF_INET);
$sock = IO::Socket->new(Domain => IO::Socket::AF_UNIX);
```

The socket domain will define which subclass of `IO::Socket` to use. The two options available along with this distribution are `AF_INET` and `AF_UNIX`.

`AF_INET` is for the internet address family of sockets and is handled via *IO::Socket::INET*. `AF_INET` sockets are bound to an internet address and port.

`AF_UNIX` is for the unix domain socket and is handled via *IO::Socket::UNIX*. `AF_UNIX` sockets are bound to the file system as their address name space.

This argument is **required**. All other arguments are optional.

Listen

```
my $sock = IO::Socket->new(..., Listen => 5);
```

Listen should be an integer value or left unset.

If provided, this argument will place the socket into listening mode. New connections can then be accepted using the *"accept" in IO::Socket* method. The value given is used as the `listen(2)` queue size.

If the `Listen` argument is given, but false, the queue size will be set to 5.

Timeout

```
my $sock = IO::Socket->new(..., Timeout => 5);
```

The timeout value, in seconds, for this socket connection. How exactly this value is utilized is defined in the socket domain subclasses that make use of the value.

Type

```
my $sock = IO::Socket->new(..., Type => IO::Socket::SOCK_STREAM);
```

The socket type that will be used. These are usually `SOCK_STREAM`, `SOCK_DGRAM`, or `SOCK_RAW`. If this argument is left undefined an attempt will be made to infer the type from the service name.

For example, you'll usually use `SOCK_STREAM` with a `tcp` connection and `SOCK_DGRAM` with a udp connection.

CONSTRUCTORS

`IO::Socket` extends the *IO::Handle* constructor.

new

```
my $sock = IO::Socket->new();

# get a new IO::Socket::INET instance
$sock = IO::Socket->new(Domain => IO::Socket::AF_INET);
# get a new IO::Socket::UNIX instance
$sock = IO::Socket->new(Domain => IO::Socket::AF_UNIX);

# Domain is the only required argument
$sock = IO::Socket->new(
    Domain => IO::Socket::AF_INET, # AF_INET, AF_UNIX
    Type => IO::Socket::SOCK_STREAM, # SOCK_STREAM, SOCK_DGRAM, ...
    Proto => 'tcp', # 'tcp', 'udp', IPPROTO_TCP, IPPROTO_UDP
    # and so on...
);
```

Creates an `IO::Socket`, which is a reference to a newly created symbol (see the *Symbol* package). `new` optionally takes arguments, these arguments are defined in *"CONSTRUCTOR ARGUMENTS" in IO::Socket*.

Any of the *"CONSTRUCTOR ARGUMENTS" in IO::Socket* may be passed to the constructor, but if any arguments are provided, then one of them must be the *"Domain" in IO::Socket* argument. The *"Domain" in IO::Socket* argument can, by default, be either `AF_INET` or `AF_UNIX`. Other domains can be used if a proper subclass for the domain family is registered. All other arguments will be passed to the `configuration` method of the package for that domain.

If the constructor fails it will return `undef` and set the `$errstr` package variable to contain an error message.

```
$sock = IO::Socket->new(...)
    or die "Cannot create socket - $IO::Socket::errstr\n";
```

For legacy reasons the error message is also set into the global `$@` variable, and you may still find older code which looks here instead.

```
$sock = IO::Socket->new(...)
    or die "Cannot create socket - $@\n";
```

METHODS

`IO::Socket` inherits all methods from *IO::Handle* and implements the following new ones.

accept

```
my $client_sock = $sock->accept();
my $inet_sock = $sock->accept('IO::Socket::INET');
```

The accept method will perform the system call `accept` on the socket and return a new object. The new object will be created in the same class as the listen socket, unless a specific package name is specified. This object can be used to communicate with the client that was trying to connect.

This differs slightly from the `accept` function in *perlfunc*.

In a scalar context the new socket is returned, or `undef` upon failure. In a list context a two-element array is returned containing the new socket and the peer address; the list will be empty upon failure.

atmark

```
my $integer = $sock->atmark();
# read in some data on a given socket
my $data;
$sock->read($data, 1024) until $sock->atmark;
```

(continues on next page)

(continued from previous page)

```
# or, export the function to use:
use IO::Socket 'sockatmark';
$sock->read($data, 1024) until sockatmark($sock);
```

True if the socket is currently positioned at the urgent data mark, false otherwise. If your system doesn't yet implement `sockatmark` this will throw an exception.

If your system does not support `sockatmark`, the `use` declaration will fail at compile time.

autoflush

```
# by default, autoflush will be turned on when referenced
$sock->autoflush(); # turns on autoflush
# turn off autoflush
$sock->autoflush(0);
# turn on autoflush
$sock->autoflush(1);
```

This attribute isn't overridden from *IO::Handle*'s implementation. However, since we turn it on by default, it's worth mentioning here.

bind

```
use Socket qw(pack_sockaddr_in);
my $port = 3000;
my $ip_address = '0.0.0.0';
my $packed_addr = pack_sockaddr_in($port, $ip_address);
$sock->bind($packed_addr);
```

Binds a network address to a socket, just as `bind(2)` does. Returns true if it succeeded, false otherwise. You should provide a packed address of the appropriate type for the socket.

connected

```
my $peer_addr = $sock->connected();
if ($peer_addr) {
    say "We're connected to $peer_addr";
}
```

If the socket is in a connected state, the peer address is returned. If the socket is not in a connected state, `undef` is returned.

Note that this method considers a half-open TCP socket to be "in a connected state". Specifically, it does not distinguish between the **ESTABLISHED** and **CLOSE-WAIT** TCP states; it returns the peer address, rather than `undef`, in either case. Thus, in general, it cannot be used to reliably learn whether the peer has initiated a graceful shutdown because in most cases (see below) the local TCP state machine remains in **CLOSE-WAIT**

until the local application calls *"shutdown" in IO::Socket* or `close`. Only at that point does this function return `undef`.

The "in most cases" hedge is because local TCP state machine behavior may depend on the peer's socket options. In particular, if the peer socket has `SO_LINGER` enabled with a zero timeout, then the peer's `close` will generate a `RST` segment. Upon receipt of that segment, the local TCP transitions immediately to **CLOSED**, and in that state, this method *will* return `undef`.

getsockopt

```
my $value = $sock->getsockopt(SOL_SOCKET, SO_REUSEADDR);
my $buf = $socket->getsockopt(SOL_SOCKET, SO_RCVBUF);
say "Receive buffer is $buf bytes";
```

Get an option associated with the socket. Levels other than `SOL_SOCKET` may be specified here. As a convenience, this method will unpack a byte buffer of the correct size back into a number.

listen

```
$sock->listen(5);
```

Does the same thing that the `listen(2)` system call does. Returns true if it succeeded, false otherwise. Listens to a socket with a given queue size.

peername

```
my $sockaddr_in = $sock->peername();
```

Returns the packed `sockaddr` address of the other end of the socket connection. It calls `getpeername`.

protocol

```
my $proto = $sock->protocol();
```

Returns the number for the protocol being used on the socket, if known. If the protocol is unknown, as with an `AF_UNIX` socket, zero is returned.

recv

```
my $buffer = "";
my $length = 1024;
my $flags = 0; # default. optional
$sock->recv($buffer, $length);
$sock->recv($buffer, $length, $flags);
```

Similar in functionality to *"recv" in perlfunc*.

Receives a message on a socket. Attempts to receive `$length` characters of data into `$buffer` from the specified socket. `$buffer` will be grown or shrunk to the length actually read. Takes the same flags as the system call of the same name. Returns the address of the sender if socket's protocol supports this; returns an empty string otherwise. If there's an error, returns `undef`. This call is actually implemented in terms of the `recvfrom(2)` system call.

Flags are ORed together values, such as `MSG_BCAST`, `MSG_OOB`, `MSG_TRUNC`. The default value for the flags is `0`.

The cached value of *"peername" in IO::Socket* is updated with the result of `recv`.

Note: In Perl v5.30 and newer, if the socket has been marked as `:utf8`, `recv` will throw an exception. The `:encoding(...)` layer implicitly introduces the `:utf8` layer. See *"binmode" in perlfunc*.

Note: In Perl versions older than v5.30, depending on the status of the socket, either (8-bit) bytes or characters are received. By default all sockets operate on bytes, but for example if the socket has been changed using *"binmode" in perlfunc* to operate with the `:encoding(UTF-8)` I/O layer (see the *"open" in perlfunc* pragma), the I/O will operate on UTF8-encoded Unicode characters, not bytes. Similarly for the `:encoding` layer: in that case pretty much any characters can be read.

send

```
my $message = "Hello, world!";
my $flags = 0; # defaults to zero
my $to = '0.0.0.0'; # optional destination
my $sent = $sock->send($message);
$sent = $sock->send($message, $flags);
$sent = $sock->send($message, $flags, $to);
```

Similar in functionality to *"send" in perlfunc*.

Sends a message on a socket. Attempts to send the scalar message to the socket. Takes the same flags as the system call of the same name. On unconnected sockets, you must specify a destination to send to, in which case it does a `sendto(2)` syscall. Returns the number of characters sent, or `undef` on error. The `sendmsg(2)` syscall is currently unimplemented.

The `flags` option is optional and defaults to `0`.

After a successful send with `$to`, further calls to `send` on an unconnected socket without `$to` will send to the same address, and `$to` will be used as the result of *"peername" in IO::Socket*.

Note: In Perl v5.30 and newer, if the socket has been marked as `:utf8`, `send` will throw an exception. The `:encoding(...)` layer implicitly introduces the `:utf8` layer. See *"binmode" in perlfunc*.

Note: In Perl versions older than v5.30, depending on the status of the socket, either (8-bit) bytes or characters are sent. By default all sockets operate on bytes, but for example if the socket has been changed using *"binmode" in perlfunc* to operate with the `:encoding(UTF-8)` I/O layer (see the *"open" in perlfunc* pragma), the I/O will operate

on UTF8-encoded Unicode characters, not bytes. Similarly for the `:encoding` layer: in that case pretty much any characters can be sent.

setsockopt

```
$sock->setsockopt(SOL_SOCKET, SO_REUSEADDR, 1);
$sock->setsockopt(SOL_SOCKET, SO_RCVBUF, 64*1024);
```

Set option associated with the socket. Levels other than `SOL_SOCKET` may be specified here. As a convenience, this method will convert a number into a packed byte buffer.

shutdown

```
$sock->shutdown(SHUT_RD); # we stopped reading data
$sock->shutdown(SHUT_WR); # we stopped writing data
$sock->shutdown(SHUT_RDWR); # we stopped using this socket
```

Shuts down a socket connection in the manner indicated by the value passed in, which has the same interpretation as in the syscall of the same name.

This is useful with sockets when you want to tell the other side you're done writing but not done reading, or vice versa. It's also a more insistent form of `close` because it also disables the file descriptor in any forked copies in other processes.

Returns `1` for success; on error, returns `undef` if the socket is not a valid filehandle, or returns `0` and sets `$!` for any other failure.

sockdomain

```
my $domain = $sock->sockdomain();
```

Returns the number for the socket domain type. For example, for an `AF_INET` socket the value of `&AF_INET` will be returned.

socket

```
my $sock = IO::Socket->new(); # no values given
# now let's actually get a socket with the socket method
# domain, type, and protocol are required
$sock = $sock->socket(AF_INET, SOCK_STREAM, 'tcp');
```

Opens a socket of the specified kind and returns it. Domain, type, and protocol are specified the same as for the syscall of the same name.

socketpair

```
my ($r, $w) = $sock->socketpair(AF_UNIX, SOCK_STREAM, PF_UNSPEC);
($r, $w) = IO::Socket::UNIX
    ->socketpair(AF_UNIX, SOCK_STREAM, PF_UNSPEC);
```

Will return a list of two sockets created (read and write), or an empty list on failure.

Differs slightly from `socketpair` in *perlfunc* in that the argument list is a bit simpler.

sockname

```
my $packed_addr = $sock->sockname();
```

Returns the packed `sockaddr` address of this end of the connection. It's the same as `getsockname(2)`.

sockopt

```
my $value = $sock->sockopt(SO_REUSEADDR);
$sock->sockopt(SO_REUSEADDR, 1);
```

Unified method to both set and get options in the `SOL_SOCKET` level. If called with one argument then *"getsockopt" in IO::Socket* is called, otherwise *"setsockopt" in IO::Socket* is called.

socktype

```
my $type = $sock->socktype();
```

Returns the number for the socket type. For example, for a `SOCK_STREAM` socket the value of `&SOCK_STREAM` will be returned.

timeout

```
my $seconds = $sock->timeout();
my $old_val = $sock->timeout(5); # set new and return old value
```

Set or get the timeout value (in seconds) associated with this socket. If called without any arguments then the current setting is returned. If called with an argument the current setting is changed and the previous value returned.

This method is available to all `IO::Socket` implementations but may or may not be used by the individual domain subclasses.

EXAMPLES

Let's create a TCP server on `localhost:3333`.

```
use strict;
use warnings;
use feature 'say';

use IO::Socket qw(AF_INET AF_UNIX SOCK_STREAM SHUT_WR);

my $server = IO::Socket->new(
```

(continues on next page)

(continued from previous page)

```
    Domain => AF_INET,
    Type => SOCK_STREAM,
    Proto => 'tcp',
    LocalHost => '0.0.0.0',
    LocalPort => 3333,
    ReusePort => 1,
    Listen => 5,
) || die "Can't open socket: $IO::Socket::errstr";
say "Waiting on 3333";

while (1) {
    # waiting for a new client connection
    my $client = $server->accept();

    # get information about a newly connected client
    my $client_address = $client->peerhost();
    my $client_port = $client->peerport();
    say "Connection from $client_address:$client_port";

    # read up to 1024 characters from the connected client
    my $data = "";
    $client->recv($data, 1024);
    say "received data: $data";

    # write response data to the connected client
    $data = "ok";
    $client->send($data);

    # notify client that response has been sent
    $client->shutdown(SHUT_WR);
}

$server->close();
```

A client for such a server could be

```
use strict;
use warnings;
use feature 'say';

use IO::Socket qw(AF_INET AF_UNIX SOCK_STREAM SHUT_WR);

my $client = IO::Socket->new(
    Domain => AF_INET,
    Type => SOCK_STREAM,
    proto => 'tcp',
    PeerPort => 3333,
    PeerHost => '0.0.0.0',
) || die "Can't open socket: $IO::Socket::errstr";
```

(continues on next page)

(continued from previous page)

```
say "Sending Hello World!";
my $size = $client->send("Hello World!");
say "Sent data of length: $size";

$client->shutdown(SHUT_WR);

my $buffer;
$client->recv($buffer, 1024);
say "Got back $buffer";

$client->close();
```

LIMITATIONS

On some systems, for an IO::Socket object created with `new_from_fd`, or created with *"accept" in IO::Socket* from such an object, the *"protocol" in IO::Socket*, *"sockdomain" in IO::Socket* and *"socktype" in IO::Socket* methods may return `undef`.

SEE ALSO

Socket, *IO::Handle*, *IO::Socket::INET*, *IO::Socket::UNIX*, *IO::Socket::IP*

AUTHOR

Graham Barr. atmark() by Lincoln Stein. Currently maintained by the Perl 5 Porters. Please report all bugs at https://github.com/Perl/perl5/issues.

COPYRIGHT

Copyright (c) 1997-8 Graham Barr gbarr@pobox.com. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

The atmark() implementation: Copyright 2001, Lincoln Stein lstein@cshl.org. This module is distributed under the same terms as Perl itself. Feel free to use, modify and redistribute it as long as you retain the correct attribution.

Modules

- `IO::Socket::INET` - Object interface for AF_INET domain sockets
- `IO::Socket::IP` - Family-neutral IP socket supporting both IPv4 and IPv6
- `IO::Socket::SSL` - SSL sockets with IO::Socket interface
- `IO::Socket::UNIX` - Object interface for AF_UNIX domain sockets

Functions

Other Functions

sockatmark

Report whether the socket is positioned at an out-of-band mark.

IO::Socket::INET

Object interface for AF_INET domain sockets

SYNOPSIS

```
use IO::Socket::INET;
```

DESCRIPTION

`IO::Socket::INET` provides an object interface to creating and using sockets in the AF_INET domain. It is built upon the *IO::Socket* interface and inherits all the methods defined by *IO::Socket*.

CONSTRUCTOR

- new ([ARGS])

  Creates an `IO::Socket::INET` object, which is a reference to a newly created symbol (see the *Symbol* package). new optionally takes arguments, these arguments are in key-value pairs.

  In addition to the key-value pairs accepted by *IO::Socket*, `IO::Socket::INET` provides.

```
PeerAddr    Remote host address          <hostname>[:<port>]
PeerHost    Synonym for PeerAddr
PeerPort    Remote port or service       <service>[(<no>)] |
 ↪<no>
LocalAddr   Local host bind address      hostname[:port]
LocalHost   Synonym for LocalAddr
LocalPort   Local host bind port         <service>[(<no>)] |
 ↪<no>
Proto       Protocol name (or number)    "tcp" | "udp" | ...
Type        Socket type              SOCK_STREAM | SOCK_DGRAM |␣
 ↪...
Listen      Queue size for listen
ReuseAddr   Set SO_REUSEADDR before binding
Reuse       Set SO_REUSEADDR before binding (deprecated,
                                          prefer ReuseAddr)
ReusePort   Set SO_REUSEPORT before binding
Broadcast   Set SO_BROADCAST before binding
Timeout     Timeout value for various operations
```

(continues on next page)

(continued from previous page)

```
MultiHomed  Try all addresses for multi-homed hosts
Blocking    Determine if connection will be blocking mode
```

If `Listen` is defined then a listen socket is created, else if the socket type, which is derived from the protocol, is SOCK_STREAM then connect() is called. If the `Listen` argument is given, but false, the queue size will be set to 5.

Although it is not illegal, the use of `MultiHomed` on a socket which is in non-blocking mode is of little use. This is because the first connect will never fail with a timeout as the connect call will not block.

The `PeerAddr` can be a hostname or the IP-address on the "xx.xx.xx.xx" form. The `PeerPort` can be a number or a symbolic service name. The service name might be followed by a number in parenthesis which is used if the service is not known by the system. The `PeerPort` specification can also be embedded in the `PeerAddr` by preceding it with a ":".

If `Proto` is not given and you specify a symbolic `PeerPort` port, then the constructor will try to derive `Proto` from the service name. As a last resort `Proto` "tcp" is assumed. The `Type` parameter will be deduced from `Proto` if not specified.

If the constructor is only passed a single argument, it is assumed to be a `PeerAddr` specification.

If `Blocking` is set to 0, the connection will be in nonblocking mode. If not specified it defaults to 1 (blocking mode).

Examples:

```
$sock = IO::Socket::INET->new(PeerAddr => 'www.perl.org',
                              PeerPort => 'http(80)',
                              Proto    => 'tcp');

$sock = IO::Socket::INET->new(PeerAddr => 'localhost:smtp(25)');

$sock = IO::Socket::INET->new(Listen    => 5,
                              LocalAddr => 'localhost',
                              LocalPort => 9000,
                              Proto     => 'tcp');

$sock = IO::Socket::INET->new('127.0.0.1:25');

$sock = IO::Socket::INET->new(
                        PeerPort  => 9999,
                        PeerAddr  => inet_ntoa(INADDR_
↪BROADCAST),
                        Proto     => 'udp',
                        LocalAddr => 'localhost',
                        Broadcast => 1 )
                      or die "Can't bind : $IO::Socket::errstr\n";
```

If the constructor fails it will return `undef` and set the `$IO::Socket::errstr` package variable to contain an error message.

```
$sock = IO::Socket::INET->new(...)
    or die "Cannot create socket - $IO::Socket::errstr\n";
```

For legacy reasons the error message is also set into the global $@ variable, and you may still find older code which looks here instead.

```
$sock = IO::Socket::INET->new(...)
    or die "Cannot create socket - $@\n";
```

METHODS

- sockaddr ()

  Return the address part of the sockaddr structure for the socket

- sockport ()

  Return the port number that the socket is using on the local host

- sockhost ()

  Return the address part of the sockaddr structure for the socket in a text form xx.xx.xx.xx

- peeraddr ()

  Return the address part of the sockaddr structure for the socket on the peer host

- peerport ()

  Return the port number for the socket on the peer host.

- peerhost ()

  Return the address part of the sockaddr structure for the socket on the peer host in a text form xx.xx.xx.xx

SEE ALSO

Socket, *IO::Socket*

AUTHOR

Graham Barr. Currently maintained by the Perl Porters. Please report all bugs at https://github.com/Perl/perl5/issues.

COPYRIGHT

Copyright (c) 1996-8 Graham Barr gbarr@pobox.com. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

IO::Socket::IP

Family-neutral IP socket supporting both IPv4 and IPv6

SYNOPSIS

```
use IO::Socket::IP;

my $sock = IO::Socket::IP->new(
   PeerHost => "www.google.com",
   PeerPort => "http",
   Type     => SOCK_STREAM,
) or die "Cannot construct socket - $IO::Socket::errstr";

my $familyname = ( $sock->sockdomain == PF_INET6 ) ? "IPv6" :
                 ( $sock->sockdomain == PF_INET  ) ? "IPv4" :
                                                     "unknown";

printf "Connected to google via %s\n", $familyname;
```

DESCRIPTION

This module provides a protocol-independent way to use IPv4 and IPv6 sockets, intended as a replacement for *IO::Socket::INET*. Most constructor arguments and methods are provided in a backward-compatible way. For a list of known differences, see the `IO::Socket::INET` INCOMPATIBILITIES section below.

It uses the `getaddrinfo(3)` function to convert hostnames and service names or port numbers into sets of possible addresses to connect to or listen on. This allows it to work for IPv6 where the system supports it, while still falling back to IPv4-only on systems which don't.

REPLACING `IO::Socket` DEFAULT BEHAVIOUR

By placing `-register` in the import list to `IO::Socket::IP`, it will register itself with *IO::Socket* as the class that handles `PF_INET`. It will also ask to handle `PF_INET6` as well, provided that constant is available.

Changing `IO::Socket`'s default behaviour means that calling the `IO::Socket` constructor with either `PF_INET` or `PF_INET6` as the `Domain` parameter will yield an `IO::Socket::IP` object.

```
use IO::Socket::IP -register;

my $sock = IO::Socket->new(
   Domain    => PF_INET6,
   LocalHost => "::1",
   Listen    => 1,
) or die "Cannot create socket - $IO::Socket::errstr\n";

print "Created a socket of type " . ref($sock) . "\n";
```

Note that `-register` is a global setting that applies to the entire program; it cannot be applied only for certain callers, removed, or limited by lexical scope.

CONSTRUCTORS

new

```
$sock = IO::Socket::IP->new( %args )
```

Creates a new `IO::Socket::IP` object, containing a newly created socket handle according to the named arguments passed. The recognised arguments are:

- PeerHost => STRING
- PeerService => STRING

  Hostname and service name for the peer to `connect()` to. The service name may be given as a port number, as a decimal string.

- PeerAddr => STRING
- PeerPort => STRING

  For symmetry with the accessor methods and compatibility with `IO::Socket::INET`, these are accepted as synonyms for `PeerHost` and `PeerService` respectively.

- PeerAddrInfo => ARRAY

  Alternate form of specifying the peer to `connect()` to. This should be an array of the form returned by `Socket::getaddrinfo`.

  This parameter takes precedence over the `Peer*`, `Family`, `Type` and `Proto` arguments.

- LocalHost => STRING
- LocalService => STRING

  Hostname and service name for the local address to `bind()` to.

- LocalAddr => STRING
- LocalPort => STRING

  For symmetry with the accessor methods and compatibility with `IO::Socket::INET`, these are accepted as synonyms for `LocalHost` and `LocalService` respectively.

- LocalAddrInfo => ARRAY

  Alternate form of specifying the local address to `bind()` to. This should be an array of the form returned by `Socket::getaddrinfo`.

  This parameter takes precedence over the `Local*`, `Family`, `Type` and `Proto` arguments.

- Family => INT

The address family to pass to `getaddrinfo` (e.g. `AF_INET`, `AF_INET6`). Normally this will be left undefined, and `getaddrinfo` will search using any address family supported by the system.

- Type => INT

  The socket type to pass to `getaddrinfo` (e.g. `SOCK_STREAM`, `SOCK_DGRAM`). Normally defined by the caller; if left undefined `getaddrinfo` may attempt to infer the type from the service name.

- Proto => STRING or INT

  The IP protocol to use for the socket (e.g. `'tcp'`, `IPPROTO_TCP`, `'udp'`,`IPPROTO_UDP`). Normally this will be left undefined, and either `getaddrinfo` or the kernel will choose an appropriate value. May be given either in string name or numeric form.

- GetAddrInfoFlags => INT

  More flags to pass to the `getaddrinfo()` function. If not supplied, a default of `AI_ADDRCONFIG` will be used.

  These flags will be combined with `AI_PASSIVE` if the `Listen` argument is given. For more information see the documentation about `getaddrinfo()` in the *Socket* module.

- Listen => INT

  If defined, puts the socket into listening mode where new connections can be accepted using the `accept` method. The value given is used as the `listen(2)` queue size.

- ReuseAddr => BOOL

  If true, set the `SO_REUSEADDR` sockopt

- ReusePort => BOOL

  If true, set the `SO_REUSEPORT` sockopt (not all OSes implement this sockopt)

- Broadcast => BOOL

  If true, set the `SO_BROADCAST` sockopt

- Sockopts => ARRAY

  An optional array of other socket options to apply after the three listed above. The value is an ARRAY containing 2- or 3-element ARRAYrefs. Each inner array relates to a single option, giving the level and option name, and an optional value. If the value element is missing, it will be given the value of a platform-sized integer 1 constant (i.e. suitable to enable most of the common boolean options).

  For example, both options given below are equivalent to setting `ReuseAddr`.

```
Sockopts => [
   [ SOL_SOCKET, SO_REUSEADDR ],
   [ SOL_SOCKET, SO_REUSEADDR, pack( "i", 1 ) ],
]
```

- V6Only => BOOL

If defined, set the IPV6_V6ONLY sockopt when creating PF_INET6 sockets to the given value. If true, a listening-mode socket will only listen on the AF_INET6 addresses; if false it will also accept connections from AF_INET addresses.

If not defined, the socket option will not be changed, and default value set by the operating system will apply. For repeatable behaviour across platforms it is recommended this value always be defined for listening-mode sockets.

Note that not all platforms support disabling this option. Some, at least OpenBSD and MirBSD, will fail with EINVAL if you attempt to disable it. To determine whether it is possible to disable, you may use the class method

```
if( IO::Socket::IP->CAN_DISABLE_V6ONLY ) {
   ...
}
else {
   ...
}
```

If your platform does not support disabling this option but you still want to listen for both AF_INET and AF_INET6 connections you will have to create two listening sockets, one bound to each protocol.

- MultiHomed

  This IO::Socket::INET-style argument is ignored, except if it is defined but false. See the IO::Socket::INET INCOMPATIBILITIES section below.

  However, the behaviour it enables is always performed by IO::Socket::IP.

- Blocking => BOOL

  If defined but false, the socket will be set to non-blocking mode. Otherwise it will default to blocking mode. See the NON-BLOCKING section below for more detail.

- Timeout => NUM

  If defined, gives a maximum time in seconds to block per connect() call when in blocking mode. If missing, no timeout is applied other than that provided by the underlying operating system. When in non-blocking mode this parameter is ignored.

  Note that if the hostname resolves to multiple address candidates, the same timeout will apply to each connection attempt individually, rather than to the operation as a whole. Further note that the timeout does not apply to the initial hostname resolve operation, if connecting by hostname.

  This behaviour is copied inspired by IO::Socket::INET; for more fine grained control over connection timeouts, consider performing a nonblocking connect directly.

If neither Type nor Proto hints are provided, a default of SOCK_STREAM and IPPROTO_TCP respectively will be set, to maintain compatibility with IO::Socket::INET. Other named arguments that are not recognised are ignored.

If neither Family nor any hosts or addresses are passed, nor any *AddrInfo, then the constructor has no information on which to decide a socket family to create. In this

case, it performs a `getaddrinfo` call with the `AI_ADDRCONFIG` flag, no host name, and a service name of `"0"`, and uses the family of the first returned result.

If the constructor fails, it will set `$IO::Socket::errstr` and `$@` to an appropriate error message; this may be from `$!` or it may be some other string; not every failure necessarily has an associated `errno` value.

new (one arg)

```
$sock = IO::Socket::IP->new( $peeraddr )
```

As a special case, if the constructor is passed a single argument (as opposed to an even-sized list of key/value pairs), it is taken to be the value of the `PeerAddr` parameter. This is parsed in the same way, according to the behaviour given in the `PeerHost` AND `LocalHost` PARSING section below.

METHODS

As well as the following methods, this class inherits all the methods in *IO::Socket* and *IO::Handle*.

sockhost_service

```
( $host, $service ) = $sock->sockhost_service( $numeric );
```

Returns the hostname and service name of the local address (that is, the socket address given by the `sockname` method).

If `$numeric` is true, these will be given in numeric form rather than being resolved into names.

The following four convenience wrappers may be used to obtain one of the two values returned here. If both host and service names are required, this method is preferable to the following wrappers, because it will call `getnameinfo(3)` only once.

sockhost

```
$addr = $sock->sockhost;
```

Return the numeric form of the local address as a textual representation

sockport

```
$port = $sock->sockport;
```

Return the numeric form of the local port number

sockhostname

```
$host = $sock->sockhostname;
```

Return the resolved name of the local address

sockservice

```
$service = $sock->sockservice;
```

Return the resolved name of the local port number

sockaddr

```
$addr = $sock->sockaddr;
```

Return the local address as a binary octet string

peerhost_service

```
( $host, $service ) = $sock->peerhost_service( $numeric );
```

Returns the hostname and service name of the peer address (that is, the socket address given by the `peername` method), similar to the `sockhost_service` method.

The following four convenience wrappers may be used to obtain one of the two values returned here. If both host and service names are required, this method is preferable to the following wrappers, because it will call `getnameinfo(3)` only once.

peerhost

```
$addr = $sock->peerhost;
```

Return the numeric form of the peer address as a textual representation

peerport

```
$port = $sock->peerport;
```

Return the numeric form of the peer port number

peerhostname

```
$host = $sock->peerhostname;
```

Return the resolved name of the peer address

peerservice

```
$service = $sock->peerservice;
```

Return the resolved name of the peer port number

peeraddr

```
$addr = $peer->peeraddr;
```

Return the peer address as a binary octet string

as_inet

```
$inet = $sock->as_inet;
```

Returns a new *IO::Socket::INET* instance wrapping the same filehandle. This may be useful in cases where it is required, for backward-compatibility, to have a real object of `IO::Socket::INET` type instead of `IO::Socket::IP`. The new object will wrap the same underlying socket filehandle as the original, so care should be taken not to continue to use both objects concurrently. Ideally the original `$sock` should be discarded after this method is called.

This method checks that the socket domain is `PF_INET` and will throw an exception if it isn't.

NON-BLOCKING

If the constructor is passed a defined but false value for the `Blocking` argument then the socket is put into non-blocking mode. When in non-blocking mode, the socket will not be set up by the time the constructor returns, because the underlying `connect(2)` syscall would otherwise have to block.

The non-blocking behaviour is an extension of the `IO::Socket::INET` API, unique to `IO::Socket::IP`, because the former does not support multi-homed non-blocking connect.

When using non-blocking mode, the caller must repeatedly check for writeability on the filehandle (for instance using `select` or `IO::Poll`). Each time the filehandle is ready to write, the `connect` method must be called, with no arguments. Note that some operating systems, most notably `MSWin32` do not report a `connect()` failure using write-ready; so you must also `select()` for exceptional status.

While `connect` returns false, the value of `$!` indicates whether it should be tried again (by being set to the value `EINPROGRESS`, or `EWOULDBLOCK` on MSWin32), or whether a permanent error has occurred (e.g. `ECONNREFUSED`).

Once the socket has been connected to the peer, `connect` will return true and the socket will now be ready to use.

Note that calls to the platform's underlying `getaddrinfo(3)` function may block. If `IO::Socket::IP` has to perform this lookup, the constructor will block even when in non-blocking mode.

To avoid this blocking behaviour, the caller should pass in the result of such a lookup using the `PeerAddrInfo` or `LocalAddrInfo` arguments. This can be achieved by using Net::LibAsyncNS, or the `getaddrinfo(3)` function can be called in a child process.

```
use IO::Socket::IP;
use Errno qw( EINPROGRESS EWOULDBLOCK );

my @peeraddrinfo = ... # Caller must obtain the getaddrinfo result
 ↪here

my $socket = IO::Socket::IP->new(
   PeerAddrInfo => \@peeraddrinfo,
   Blocking     => 0,
) or die "Cannot construct socket - $@";

while( !$socket->connect and ( $! == EINPROGRESS || $! ==
 ↪EWOULDBLOCK ) ) {
   my $wvec = '';
   vec( $wvec, fileno $socket, 1 ) = 1;
   my $evec = '';
   vec( $evec, fileno $socket, 1 ) = 1;

   select( undef, $wvec, $evec, undef ) or die "Cannot select - $!";
}

die "Cannot connect - $!" if $!;

...
```

The example above uses `select()`, but any similar mechanism should work analogously. `IO::Socket::IP` takes care when creating new socket filehandles to preserve the actual file descriptor number, so such techniques as `poll` or `epoll` should be transparent to its reallocation of a different socket underneath, perhaps in order to switch protocol family between `PF_INET` and `PF_INET6`.

For another example using `IO::Poll` and `Net::LibAsyncNS`, see the `examples/nonblocking_libasyncns.pl` file in the module distribution.

PeerHost AND LocalHost PARSING

To support the `IO::Socket::INET` API, the host and port information may be passed in a single string rather than as two separate arguments.

If either `LocalHost` or `PeerHost` (or their `...Addr` synonyms) have any of the following special forms then special parsing is applied.

The value of the `...Host` argument will be split to give both the hostname and port (or service name):

```
hostname.example.org:http    # Host name
192.0.2.1:80                 # IPv4 address
[2001:db8::1]:80             # IPv6 address
```

In each case, the port or service name (e.g. `80`) is passed as the `LocalService` or `PeerService` argument.

Either of `LocalService` or `PeerService` (or their `...Port` synonyms) can be either a service name, a decimal number, or a string containing both a service name and number, in a form such as

```
http(80)
```

In this case, the name (`http`) will be tried first, but if the resolver does not understand it then the port number (`80`) will be used instead.

If the `...Host` argument is in this special form and the corresponding `...Service` or `...Port` argument is also defined, the one parsed from the `...Host` argument will take precedence and the other will be ignored.

split_addr

```
( $host, $port ) = IO::Socket::IP->split_addr( $addr )
```

Utility method that provides the parsing functionality described above. Returns a 2-element list, containing either the split hostname and port description if it could be parsed, or the given address and `undef` if it was not recognised.

```
IO::Socket::IP->split_addr( "hostname:http" );
                             # ( "hostname",  "http" )

IO::Socket::IP->split_addr( "192.0.2.1:80" );
                             # ( "192.0.2.1", "80"   )

IO::Socket::IP->split_addr( "[2001:db8::1]:80" );
                             # ( "2001:db8::1", "80" )

IO::Socket::IP->split_addr( "something.else" );
                             # ( "something.else", undef )
```

join_addr

```
$addr = IO::Socket::IP->join_addr( $host, $port )
```

Utility method that performs the reverse of `split_addr`, returning a string formed by joining the specified host address and port number. The host address will be wrapped in `[]` brackets if required (because it is a raw IPv6 numeric address).

This can be especially useful when combined with the `sockhost_service` or `peerhost_service` methods.

```
say "Connected to ", IO::Socket::IP->join_addr( $sock->peerhost_
↪service );
```

IO::Socket::INET INCOMPATIBILITIES

- The behaviour enabled by `MultiHomed` is in fact implemented by `IO::Socket::IP` as it is required to correctly support searching for a useable address from the results of the `getaddrinfo(3)` call. The constructor will ignore the value of this argument, except if it is defined but false. An exception is thrown in this case, because that would request it disable the `getaddrinfo(3)` search behaviour in the first place.
- `IO::Socket::IP` implements both the `Blocking` and `Timeout` parameters, but it implements the interaction of both in a different way.

  In `::INET`, supplying a timeout overrides the non-blocking behaviour, meaning that the `connect()` operation will still block despite that the caller asked for a non-blocking socket. This is not explicitly specified in its documentation, nor does this author believe that is a useful behaviour - it appears to come from a quirk of implementation.

  In `::IP` therefore, the `Blocking` parameter takes precedence - if a non-blocking socket is requested, no operation will block. The `Timeout` parameter here simply defines the maximum time that a blocking `connect()` call will wait, if it blocks at all.

  In order to specifically obtain the "blocking connect then non-blocking send and receive" behaviour of specifying this combination of options to `::INET` when using `::IP`, perform first a blocking connect, then afterwards turn the socket into nonblocking mode.

```
my $sock = IO::Socket::IP->new(
   PeerHost => $peer,
   Timeout => 20,
) or die "Cannot connect - $@";

$sock->blocking( 0 );
```

  This code will behave identically under both `IO::Socket::INET` and `IO::Socket::IP`.

TODO

- Investigate whether `POSIX::dup2` upsets BSD's kqueue watchers, and if so, consider what possible workarounds might be applied.

AUTHOR

Paul Evans leonerd@leonerd.org.uk

IO::Socket::SSL

SSL sockets with IO::Socket interface

SYNOPSIS

```
use strict;
use IO::Socket::SSL;

# simple client
my $cl = IO::Socket::SSL->new('www.google.com:443');
print $cl "GET / HTTP/1.0\r\n\r\n";
print <$cl>;

# simple server
my $srv = IO::Socket::SSL->new(
    LocalAddr => '0.0.0.0:1234',
    Listen => 10,
    SSL_cert_file => 'server-cert.pem',
    SSL_key_file => 'server-key.pem',
);
$srv->accept;
```

DESCRIPTION

IO::Socket::SSL makes using SSL/TLS much easier by wrapping the necessary functionality into the familiar *IO::Socket* interface and providing secure defaults whenever possible. This way, existing applications can be made SSL-aware without much effort, at least if you do blocking I/O and don't use select or poll.

But, under the hood, SSL is a complex beast. So there are lots of methods to make it do what you need if the default behavior is not adequate. Because it is easy to inadvertently introduce critical security bugs or just hard to debug problems, I would recommend studying the following documentation carefully.

The documentation consists of the following parts:

- "Essential Information About SSL/TLS"
- "Basic SSL Client"
- "Basic SSL Server"
- "Common Usage Errors"
- "Common Problems with SSL"
- "Using Non-Blocking Sockets"
- "Advanced Usage"
- "Integration Into Own Modules"
- "Description Of Methods"

Additional documentation can be found in

- *IO::Socket::SSL::Intercept* - Doing Man-In-The-Middle with SSL
- *IO::Socket::SSL::Utils* - Useful functions for certificates etc

Essential Information About SSL/TLS

SSL (Secure Socket Layer) or its successor TLS (Transport Layer Security) are protocols to facilitate end-to-end security. These protocols are used when accessing web sites (https), delivering or retrieving email, and in lots of other use cases. In the following documentation we will refer to both SSL and TLS as simply 'SSL'.

SSL enables end-to-end security by providing two essential functions:

- Encryption

  This part encrypts the data for transit between the communicating parties, so that nobody in between can read them. It also provides tamper resistance so that nobody in between can manipulate the data.

- Identification

  This part makes sure that you talk to the right peer. If the identification is done incorrectly it is easy to mount man-in-the-middle attacks, e.g. if Alice wants to talk to Bob it would be possible for Mallory to put itself in the middle, so that Alice talks to Mallory and Mallory to Bob. All the data would still be encrypted, but not end-to-end between Alice and Bob, but only between Alice and Mallory and then between Mallory and Bob. Thus Mallory would be able to read and modify all traffic between Alice and Bob.

Identification is the part which is the hardest to understand and the easiest to get wrong.

With SSL, the Identification is usually done with **certificates** inside a **PKI** (Public Key Infrastructure). These Certificates are comparable to an identity card, which contains information about the owner of the card. The card then is somehow **signed** by the **issuer** of the card, the **CA** (Certificate Agency).

To verify the identity of the peer the following must be done inside SSL:

- Get the certificate from the peer. If the peer does not present a certificate we cannot verify it.
- Check if we trust the certificate, e.g. make sure it's not a forgery.

  We believe that a certificate is not a fake if we either know the certificate already or if we **trust** the issuer (the CA) and can verify the issuers signature on the certificate. In reality there is often a hierarchy of certificate agencies and we only directly trust the root of this hierarchy. In this case the peer not only sends his own certificate, but also all **intermediate certificates**. Verification will be done by building a **trust path** from the trusted root up to the peers certificate and checking in each step if the we can verify the issuer's signature.

  This step often causes problems because the client does not know the necessary trusted root certificates. These are usually stored in a system dependent CA store, but often the browsers have their own CA store.

- Check if the certificate is still valid. Each certificate has a lifetime and should not be used after that time because it might be compromised or the underlying cryptography got broken in the mean time.
- Check if the subject of the certificate matches the peer. This is like comparing the picture on the identity card against the person representing the identity card.

When connecting to a server this is usually done by comparing the hostname used for connecting against the names represented in the certificate. A certificate might contain multiple names or wildcards, so that it can be used for multiple hosts (e.g. *.example.com and *.example.org).

Although nobody sane would accept an identity card where the picture does not match the person we see, it is a common implementation error with SSL to omit this check or get it wrong.

- Check if the certificate was revoked by the issuer. This might be the case if the certificate was compromised somehow and now somebody else might use it to claim the wrong identity. Such revocations happened a lot after the heartbleed attack.

  For SSL there are two ways to verify a revocation, CRL and OCSP. With CRLs (Certificate Revocation List) the CA provides a list of serial numbers for revoked certificates. The client somehow has to download the list (which can be huge) and keep it up to date. With OCSP (Online Certificate Status Protocol) the client can check a single certificate directly by asking the issuer.

  Revocation is the hardest part of the verification and none of today's browsers get it fully correct. But, they are still better than most other implementations which don't implement revocation checks or leave the hard parts to the developer.

When accessing a web site with SSL or delivering mail in a secure way the identity is usually only checked one way, e.g. the client wants to make sure it talks to the right server, but the server usually does not care which client it talks to. But, sometimes the server wants to identify the client too and will request a certificate from the client which the server must verify in a similar way.

Basic SSL Client

A basic SSL client is simple:

```
my $client = IO::Socket::SSL->new('www.example.com:443')
    or die "error=$!, ssl_error=$SSL_ERROR";
```

This will take the OpenSSL default CA store as the store for the trusted CA. This usually works on UNIX systems. If there are no certificates in the store it will try use Mozilla::CA which provides the default CAs of Firefox.

In the default settings, *IO::Socket::SSL* will use a safer cipher set and SSL version, do a proper hostname check against the certificate, and use SNI (server name indication) to send the hostname inside the SSL handshake. This is necessary to work with servers which have different certificates behind the same IP address. It will also check the revocation of the certificate with OCSP, but currently only if the server provides OCSP stapling (for deeper checks see `ocsp_resolver` method).

Lots of options can be used to change ciphers, SSL version, location of CA and much more. See documentation of methods for details.

With protocols like SMTP it is necessary to upgrade an existing socket to SSL. This can be done like this:

```
my $client = IO::Socket::INET->new('mx.example.com:25') or die $!;
# .. read greeting from server
```

(continues on next page)

(continued from previous page)

```
# .. send EHLO and read response
# .. send STARTTLS command and read response
# .. if response was successful we can upgrade the socket to SSL␣
 ↪now:
IO::Socket::SSL->start_SSL($client,
    # explicitly set hostname we should use for SNI
    SSL_hostname => 'mx.example.com'
) or die $SSL_ERROR;
```

A more complete example for a simple HTTP client:

```
my $client = IO::Socket::SSL->new(
    # where to connect
    PeerHost => "www.example.com",
    PeerPort => "https",

    # certificate verification - VERIFY_PEER is default
    SSL_verify_mode => SSL_VERIFY_PEER,

    # location of CA store
    # need only be given if default store should not be used
    SSL_ca_path => '/etc/ssl/certs', # typical CA path on Linux
    SSL_ca_file => '/etc/ssl/cert.pem', # typical CA file on BSD

    # or just use default path on system:
    IO::Socket::SSL::default_ca(), # either explicitly
    # or implicitly by not giving SSL_ca_*

    # easy hostname verification
    # It will use PeerHost as default name a verification
    # scheme as default, which is safe enough for most purposes.
    SSL_verifycn_name => 'foo.bar',
    SSL_verifycn_scheme => 'http',

    # SNI support - defaults to PeerHost
    SSL_hostname => 'foo.bar',

) or die "failed connect or ssl handshake: $!,$SSL_ERROR";

# send and receive over SSL connection
print $client "GET / HTTP/1.0\r\n\r\n";
print <$client>;
```

And to do revocation checks with OCSP (only available with OpenSSL 1.0.0 or higher and *Net::SSLeay* 1.59 or higher):

```
# default will try OCSP stapling and check only leaf certificate
my $client = IO::Socket::SSL->new($dst);

# better yet: require checking of full chain
```

(continues on next page)

(continued from previous page)

```
my $client = IO::Socket::SSL->new(
    PeerAddr => $dst,
    SSL_ocsp_mode => SSL_OCSP_FULL_CHAIN,
);

# even better: make OCSP errors fatal
# (this will probably fail with lots of sites because of bad OCSP␣
↪setups)
# also use common OCSP response cache
my $ocsp_cache = IO::Socket::SSL::OCSP_Cache->new;
my $client = IO::Socket::SSL->new(
    PeerAddr => $dst,
    SSL_ocsp_mode => SSL_OCSP_FULL_CHAIN|SSL_OCSP_FAIL_HARD,
    SSL_ocsp_cache => $ocsp_cache,
);

# disable OCSP stapling in case server has problems with it
my $client = IO::Socket::SSL->new(
    PeerAddr => $dst,
    SSL_ocsp_mode => SSL_OCSP_NO_STAPLE,
);

# check any certificates which are not yet checked by OCSP stapling␣
↪or
# where we have already cached results. For your own resolving␣
↪combine
# $ocsp->requests with $ocsp->add_response(uri,response).
my $ocsp = $client->ocsp_resolver();
my $errors = $ocsp->resolve_blocking();
if ($errors) {
    warn "OCSP verification failed: $errors";
    close($client);
}
```

Basic SSL Server

A basic SSL server looks similar to other *IO::Socket* servers, only that it also contains settings for certificate and key:

```
# simple server
my $server = IO::Socket::SSL->new(
    # where to listen
    LocalAddr => '127.0.0.1',
    LocalPort => 8080,
    Listen => 10,

    # which certificate to offer
    # with SNI support there can be different certificates per␣
↪hostname
```

(continues on next page)

(continued from previous page)

```
    SSL_cert_file => 'cert.pem',
    SSL_key_file => 'key.pem',
) or die "failed to listen: $!";

# accept client
my $client = $server->accept or die
    "failed to accept or ssl handshake: $!,$SSL_ERROR";
```

This will automatically use a secure set of ciphers and SSL version and also supports Forward Secrecy with (Elliptic-Curve) Diffie-Hellmann Key Exchange.

If you are doing a forking or threading server, we recommend that you do the SSL handshake inside the new process/thread so that the master is free for new connections. We recommend this because a client with improper or slow SSL handshake could make the server block in the handshake which would be bad to do on the listening socket:

```
# inet server
my $server = IO::Socket::INET->new(
    # where to listen
    LocalAddr => '127.0.0.1',
    LocalPort => 8080,
    Listen => 10,
);

# accept client
my $client = $server->accept or die;

# SSL upgrade client (in new process/thread)
IO::Socket::SSL->start_SSL($client,
    SSL_server => 1,
    SSL_cert_file => 'cert.pem',
    SSL_key_file => 'key.pem',
) or die "failed to ssl handshake: $SSL_ERROR";
```

Like with normal sockets, neither forking nor threading servers scale well. It is recommended to use non-blocking sockets instead, see “Using Non-Blocking Sockets”

Common Usage Errors

This is a list of typical errors seen with the use of *IO::Socket::SSL*:

- Disabling verification with `SSL_verify_mode`.

  As described in “Essential Information About SSL/TLS”, a proper identification of the peer is essential and failing to verify makes Man-In-The-Middle attacks possible.

  Nevertheless, lots of scripts and even public modules or applications disable verification, because it is probably the easiest way to make the thing work and usually nobody notices any security problems anyway.

  If the verification does not succeed with the default settings, one can do the following:

  - Make sure the needed CAs are in the store, maybe use `SSL_ca_file` or `SSL_ca_path` to specify a different CA store.
  - If the validation fails because the certificate is self-signed and that's what you expect, you can use the `SSL_fingerprint` option to accept specific leaf certificates by their certificate or pubkey fingerprint.
  - If the validation failed because the hostname does not match and you cannot access the host with the name given in the certificate, you can use `SSL_verifycn_name` to specify the hostname you expect in the certificate.

  A common error pattern is also to disable verification if they found no CA store (different modules look at different "default" places). Because *IO::Socket::SSL* is now able to provide a usable CA store on most platforms (UNIX, Mac OSX and Windows) it is better to use the defaults provided by *IO::Socket::SSL*. If necessary these can be checked with the `default_ca` method.

- Polling of SSL sockets (e.g. select, poll and other event loops).

  If you sysread one byte on a normal socket it will result in a syscall to read one byte. Thus, if more than one byte is available on the socket it will be kept in the network stack of your OS and the next select or poll call will return the socket as readable. But, with SSL you don't deliver single bytes. Multiple data bytes are packaged and encrypted together in an SSL frame. Decryption can only be done on the whole frame, so a sysread for one byte actually reads the complete SSL frame from the socket, decrypts it and returns the first decrypted byte. Further sysreads will return more bytes from the same frame until all bytes are returned and the next SSL frame will be read from the socket.

  Thus, in order to decide if you can read more data (e.g. if sysread will block) you must check if there are still data in the current SSL frame by calling `pending` and if there are no data pending you might check the underlying socket with select or poll. Another way might be if you try to sysread at least 16kByte all the time. 16kByte is the maximum size of an SSL frame and because sysread returns data from only a single SSL frame you can guarantee that there are no pending data.

  Additionally, contrary to plain sockets the data delivered on the socket are not necessarily application payload. It might be a TLS handshake, it might just be the beginning of a TLS record or it might be TLS session tickets which are send after the TLS handshake in TLS 1.3. In such situations select will return that data are available for read since it only looks at the plain socket. A sysread on the IO::Socket::SSL socket will not return any data though since it is an abstraction which only returns application data. This causes the sysread to hang in case the socket was blocking or to return an error with EAGAIN on non-blocking sockets. Applications using select or similar should therefore set the socket to non-blocking and also expect that the sysread might temporarily fail with EAGAIN.

  See also "Using Non-Blocking Sockets".

- Expecting exactly the same behavior as plain sockets.

  IO::Socket::SSL tries to emulate the usual socket behavior as good as possible, but full emulation can not be done. Specifically a read on the SSL socket might also result in a write on the TCP socket or a write on the SSL socket might result in a read on the TCP socket. Also `accept` and **close** on the SSL socket will result in writing and reading data to the TCP socket too.

Especially the hidden writes might result in a connection reset if the underlying TCP socket is already closed by the peer. Unless signal PIPE is explicitly handled by the application this will usually result in the application crashing. It is thus recommended to explicitly IGNORE signal PIPE so that the errors get propagated as EPIPE instead of causing a crash of the application.

- Set 'SSL_version' or 'SSL_cipher_list' to a "better" value.

  IO::Socket::SSL tries to set these values to reasonable, secure values which are compatible with the rest of the world. But, there are some scripts or modules out there which tried to be smart and get more secure or compatible settings. Unfortunately, they did this years ago and never updated these values, so they are still forced to do only 'TLSv1' (instead of also using TLSv12 or TLSv11). Or they set 'HIGH' as the cipher list and thought they were secure, but did not notice that 'HIGH' includes anonymous ciphers, e.g. without identification of the peer.

  So it is recommended to leave the settings at the secure defaults which *IO::Socket::SSL* sets and which get updated from time to time to better fit the real world.

- Make SSL settings inaccessible by the user, together with bad builtin settings.

  Some modules use *IO::Socket::SSL*, but don't make the SSL settings available to the user. This is often combined with bad builtin settings or defaults (like switching verification off).

  Thus the user needs to hack around these restrictions by using `set_args_filter_hack` or similar.

- Use of constants as strings.

  Constants like `SSL_VERIFY_PEER` or `SSL_WANT_READ` should be used as constants and not be put inside quotes, because they represent numerical values.

- Forking and handling the socket in parent and child.

  A **fork** of the process will duplicate the internal user space SSL state of the socket. If both master and child interact with the socket by using their own SSL state strange error messages will happen. Such interaction includes explicit or implicit **close** of the SSL socket. To avoid this the socket should be explicitly closed with **SSL_no_shutdown**.

- Forking and executing a new process.

  Since the SSL state is stored in user space it will be duplicated by a **fork** but it will be lost when doing **exec**. This means it is not possible to simply redirect stdin and stdout for the new process to the SSL socket by duplicating the relevant file handles. Instead explicitly exchanging plain data between child-process and SSL socket are needed.

Common Problems with SSL

SSL is a complex protocol with multiple implementations and each of these has their own quirks. While most of these implementations work together, it often gets problematic with older versions, minimal versions in load balancers, or plain wrong setups.

Unfortunately these problems are hard to debug. Helpful for debugging are a knowledge of SSL internals, wireshark and the use of the debug settings of *IO::Socket::SSL* and

Net::SSLeay, which can both be set with `$IO::Socket::SSL::DEBUG`. The following debugs levels are defined, but used not in any consistent way:

- 0 - No debugging (default).
- 1 - Print out errors from *IO::Socket::SSL* and ciphers from *Net::SSLeay*.
- 2 - Print also information about call flow from *IO::Socket::SSL*, progress information from *Net::SSLeay* and state information from OpenSSL.
- 3 - Print also some data dumps from *IO::Socket::SSL* and from *Net::SSLeay*.

Also, `analyze-ssl.pl` from the ssl-tools repository at https://github.com/noxxi/p5-ssl-tools might be a helpful tool when debugging SSL problems, as do the `openssl` command line tool and a check with a different SSL implementation (e.g. a web browser).

The following problems are not uncommon:

- Bad server setup: missing intermediate certificates.

  It is a regular problem that administrators fail to include all necessary certificates into their server setup, e.g. everything needed to build the trust chain from the trusted root. If they check the setup with the browser everything looks ok, because browsers work around these problems by caching any intermediate certificates and apply them to new connections if certificates are missing.

  But, fresh browser profiles which have never seen these intermediates cannot fill in the missing certificates and fail to verify; the same is true with *IO::Socket::SSL*.

- Old versions of servers or load balancers which do not understand specific TLS versions or croak on specific data.

  From time to time one encounters an SSL peer, which just closes the connection inside the SSL handshake. This can usually be worked around by downgrading the SSL version, e.g. by setting `SSL_version`. Modern Browsers usually deal with such servers by automatically downgrading the SSL version and repeat the connection attempt until they succeed.

  Worse servers do not close the underlying TCP connection but instead just drop the relevant packet. This is harder to detect because it looks like a stalled connection. But downgrading the SSL version often works here too.

  A cause of such problems are often load balancers or security devices, which have hardware acceleration and only a minimal (and less robust) SSL stack. They can often be detected because they support much fewer ciphers than other implementations.

- Bad or old OpenSSL versions.

  IO::Socket::SSL uses OpenSSL with the help of the *Net::SSLeay* library. It is recommend to have a recent version of this library, because it has more features and usually fewer known bugs.

- Validation of client certificates fail.

  Make sure that the purpose of the certificate allows use as ssl client (check with `openssl x509 -purpose`, that the necessary root certificate is in the path specified by `SSL_ca*` (or the default path) and that any intermediate certificates needed to build the trust chain are sent by the client.

- Validation of self-signed certificate fails even if it is given with `SSL_ca*` argument.

  The `SSL_ca*` arguments do not give a general trust store for arbitrary certificates but only specify a store for CA certificates which then can be used to verify other certificates. This especially means that certificates which are not a CA get simply ignored, notably self-signed certificates which do not also have the CA-flag set.

  This behavior of OpenSSL differs from the more general trust-store concept which can be found in browsers and where it is possible to simply added arbitrary certificates (CA or not) as trusted.

Using Non-Blocking Sockets

If you have a non-blocking socket, the expected behavior on read, write, accept or connect is to set `$!` to EWOULDBLOCK if the operation cannot be completed immediately. Note that EWOULDBLOCK is the same as EAGAIN on UNIX systems, but is different on Windows.

With SSL, handshakes might occur at any time, even within an established connection. In these cases it is necessary to finish the handshake before you can read or write data. This might result in situations where you want to read but must first finish the write of a handshake or where you want to write but must first finish a read. In these cases `$!` is set to EAGAIN like expected, and additionally `$SSL_ERROR` is set to either SSL_WANT_READ or SSL_WANT_WRITE. Thus if you get EWOULDBLOCK on a SSL socket you must check `$SSL_ERROR` for SSL_WANT_* and adapt your event mask accordingly.

Using readline on non-blocking sockets does not make much sense and I would advise against using it. And, while the behavior is not documented for other *IO::Socket* classes, it will try to emulate the behavior seen there, e.g. to return the received data instead of blocking, even if the line is not complete. If an unrecoverable error occurs it will return nothing, even if it already received some data.

Also, I would advise against using `accept` with a non-blocking SSL object because it might block and this is not what most would expect. The reason for this is that `accept` on a non-blocking TCP socket (e.g. *IO::Socket::IP*, *IO::Socket::INET*..) results in a new TCP socket which does not inherit the non-blocking behavior of the master socket. And thus, the initial SSL handshake on the new socket inside `IO::Socket::SSL::accept` will be done in a blocking way. To work around this you are safer by doing a TCP accept and later upgrade the TCP socket in a non-blocking way with `start_SSL` and `accept_SSL`.

```
my $cl = IO::Socket::SSL->new($dst);
$cl->blocking(0);
my $sel = IO::Select->new($cl);
while (1) {
    # with SSL a call for reading n bytes does not result in reading␣
↪of n
    # bytes from the socket, but instead it must read at least one␣
↪full SSL
    # frame. If the socket has no new bytes, but there are␣
↪unprocessed data
    # from the SSL frame can_read will block!

    # wait for data on socket
```

(continues on next page)

(continued from previous page)

```
    $sel->can_read();

    # new data on socket or eof
    READ:
    # this does not read only 1 byte from socket, but reads the␣
↪complete SSL
    # frame and then just returns one byte. On subsequent calls it␣
↪than
    # returns more byte of the same SSL frame until it needs to read␣
↪the
    # next frame.
    my $n = sysread( $cl,my $buf,1);
    if ( ! defined $n ) {
        die $! if not $!{EWOULDBLOCK};
        next if $SSL_ERROR == SSL_WANT_READ;
        if ( $SSL_ERROR == SSL_WANT_WRITE ) {
            # need to write data on renegotiation
            $sel->can_write;
            next;
        }
        die "something went wrong: $SSL_ERROR";
    } elsif ( ! $n ) {
        last; # eof
    } else {
        # read next bytes
        # we might have still data within the current SSL frame
        # thus first process these data instead of waiting on the␣
↪underlying
        # socket object
        goto READ if $cl->pending;    # goto sysread
        next;                         # goto $sel->can_read
    }
}
```

Additionally there are differences to plain sockets when using select, poll, kqueue or similar technologies to get notified if data are available. Relying only on these calls is not sufficient in all cases since unread data might be internally buffered in the SSL stack. To detect such buffering **pending()** need to be used. Alternatively the buffering can be avoided by using **sysread** with the maximum size of an SSL frame. See “Common Usage Errors” for details.

Advanced Usage

SNI Support

Newer extensions to SSL can distinguish between multiple hostnames on the same IP address using Server Name Indication (SNI).

Support for SNI on the client side was added somewhere in the OpenSSL 0.9.8 series, but with 1.0 a bug was fixed when the server could not decide about its hostname. Therefore client side SNI is only supported with OpenSSL 1.0 or higher in

IO::Socket::SSL. With a supported version, SNI is used automatically on the client side, if it can determine the hostname from `PeerAddr` or `PeerHost` (which are synonyms in the underlying IO::Socket:: classes and thus should never be set both or at least not to different values). On unsupported OpenSSL versions it will silently not use SNI. The hostname can also be given explicitly given with `SSL_hostname`, but in this case it will throw in error, if SNI is not supported. To check for support you might call `IO::Socket::SSL->can_client_sni()`.

On the server side, earlier versions of OpenSSL are supported, but only together with *Net::SSLeay* version >= 1.50. To check for support you might call `IO::Socket::SSL->can_server_sni()`. If server side SNI is supported, you might specify different certificates per host with `SSL_cert*` and `SSL_key*`, and check the requested name using `get_servername`.

Talk Plain and SSL With The Same Socket

It is often required to first exchange some plain data and then upgrade the socket to SSL after some kind of STARTTLS command. Protocols like FTPS even need a way to downgrade the socket again back to plain.

The common way to do this would be to create a normal socket and use `start_SSL` to upgrade and stop_SSL to downgrade:

```
my $sock = IO::Socket::INET->new(...) or die $!;
... exchange plain data on $sock until starttls command ...
IO::Socket::SSL->start_SSL($sock,%sslargs) or die $SSL_ERROR;
... now $sock is an IO::Socket::SSL object ...
... exchange data with SSL on $sock until stoptls command ...
$sock->stop_SSL or die $SSL_ERROR;
... now $sock is again an IO::Socket::INET object ...
```

But, lots of modules just derive directly from *IO::Socket::INET*. While this base class can be replaced with *IO::Socket::SSL*, these modules cannot easily support different base classes for SSL and plain data and switch between these classes on a starttls command.

To help in this case, *IO::Socket::SSL* can be reduced to a plain socket on startup, and connect_SSL/accept_SSL/start_SSL can be used to enable SSL and `stop_SSL` to talk plain again:

```
my $sock = IO::Socket::SSL->new(
    PeerAddr => ...
    SSL_startHandshake => 0,
    %sslargs
) or die $!;
... exchange plain data on $sock until starttls command ...
$sock->connect_SSL or die $SSL_ERROR;
... now $sock is an IO::Socket::SSL object ...
... exchange data with SSL on $sock until stoptls command ...
$sock->stop_SSL or die $SSL_ERROR;
... $sock is still an IO::Socket::SSL object ...
... but data exchanged again in plain ...
```

Integration Into Own Modules

IO::Socket::SSL behaves similarly to other *IO::Socket* modules and thus could be integrated in the same way, but you have to take special care when using non-blocking I/O (like for handling timeouts) or using select or poll. Please study the documentation on how to deal with these differences.

Also, it is recommended to not set or touch most of the `SSL_*` options, so that they keep their secure defaults. It is also recommended to let the user override these SSL specific settings without the need of global settings or hacks like `set_args_filter_hack`.

The notable exception is `SSL_verifycn_scheme`. This should be set to the hostname verification scheme required by the module or protocol.

Description Of Methods

IO::Socket::SSL inherits from another *IO::Socket* module. The choice of the super class depends on the installed modules:

- If *IO::Socket::IP* with at least version 0.20 is installed it will use this module as super class, transparently providing IPv6 and IPv4 support.
- If IO::Socket::INET6 is installed it will use this module as super class, transparently providing IPv6 and IPv4 support.
- Otherwise it will fall back to *IO::Socket::INET*, which is a perl core module. With *IO::Socket::INET* you only get IPv4 support.

Please be aware that with the IPv6 capable super classes, it will look first for the IPv6 address of a given hostname. If the resolver provides an IPv6 address, but the host cannot be reached by IPv6, there will be no automatic fallback to IPv4. To avoid these problems you can enforce IPv4 for a specific socket by using the `Domain` or `Family` option with the value AF_INET as described in *IO::Socket::IP*. Alternatively you can enforce IPv4 globally by loading *IO::Socket::SSL* with the option 'inet4', in which case it will use the IPv4 only class *IO::Socket::INET* as the super class.

IO::Socket::SSL will provide all of the methods of its super class, but sometimes it will override them to match the behavior expected from SSL or to provide additional arguments.

The new or changed methods are described below, but please also read the section about SSL specific error handling.

- Error Handling

  If an SSL specific error occurs, the global variable `$SSL_ERROR` will be set. If the error occurred on an existing SSL socket, the method `errstr` will give access to the latest socket specific error. Both `$SSL_ERROR` and the `errstr` method give a dualvar similar to `$!`, e.g. providing an error number in numeric context or an error description in string context.

- **new(...)**

  Creates a new *IO::Socket::SSL* object. You may use all the friendly options that came bundled with the super class (e.g. *IO::Socket::IP*, *IO::Socket::INET*, ...) plus (optionally) the ones described below. If you don't specify any SSL related options it will

do its best in using secure defaults, e.g. choosing good ciphers, enabling proper verification, etc.

- SSL_server

  Set this option to a true value if the socket should be used as a server. If this is not explicitly set it is assumed if the `Listen` parameter is given when creating the socket.

- SSL_hostname

  This can be given to specify the hostname used for SNI, which is needed if you have multiple SSL hostnames on the same IP address. If not given it will try to determine the hostname from `PeerAddr`, which will fail if only an IP was given or if this argument is used within `start_SSL`.

  If you want to disable SNI, set this argument to ‘’.

  Currently only supported for the client side and will be ignored for the server side.

  See section “SNI Support” for details of SNI the support.

- SSL_startHandshake

  If this option is set to false (defaults to true) it will not start the SSL handshake yet. This has to be done later with `accept_SSL` or `connect_SSL`. Before the handshake is started read/write/etc. can be used to exchange plain data.

- SSL_keepSocketOnError

  If this option is set to true (defaults to false) it will not close the underlying TCP socket on errors. In most cases there is no real use for this behavior since both sides of the TCP connection will probably have a different idea of the current state of the connection.

- SSL_ca | SSL_ca_file | SSL_ca_path

  Usually you want to verify that the peer certificate has been signed by a trusted certificate authority. In this case you should use this option to specify the file (`SSL_ca_file`) or directory (`SSL_ca_path`) containing the certificate(s) of the trusted certificate authorities.

  `SSL_ca_path` can also be an array or a string containing multiple path, where the path are separated by the platform specific separator. This separator is `;` on DOS, Windows, Netware, `,` on VMS and `:` for all the other systems. If multiple path are given at least one of these must be accessible.

  You can also give a list of X509* certificate handles (like you get from *Net::SSLeay* or IO::Socket::SSL::Utils::PEM_xxx2cert) with `SSL_ca`. These will be added to the CA store before path and file and thus take precedence. If neither SSL_ca, nor SSL_ca_file or SSL_ca_path are set it will use `default_ca()` to determine the user-set or system defaults. If you really don’t want to set a CA set SSL_ca_file or SSL_ca_path to `\undef` or SSL_ca to an empty list. (unfortunately `''` is used by some modules using *IO::Socket::SSL* when CA is not explicitly given).

- SSL_client_ca | SSL_client_ca_file

If verify_mode is VERIFY_PEER on the server side these options can be used to set the list of acceptable CAs for the client. This way the client can select they required certificate from a list of certificates. The value for these options is similar to `SSL_ca` and `SSL_ca_file`.

– SSL_fingerprint

Sometimes you have a self-signed certificate or a certificate issued by an unknown CA and you really want to accept it, but don't want to disable verification at all. In this case you can specify the fingerprint of the certificate as `'algo$hex_fingerprint'`. `algo` is a fingerprint algorithm supported by OpenSSL, e.g. 'sha1','sha256'... and `hex_fingerprint` is the hexadecimal representation of the binary fingerprint. Any colons inside the hex string will be ignored.

If you want to use the fingerprint of the pubkey inside the certificate instead of the certificate use the syntax `'algo$pub$hex_fingerprint'` instead. To get the fingerprint of an established connection you can use `get_fingerprint`.

It is also possible to skip `algo$`, i.e. only specify the fingerprint. In this case the likely algorithms will be automatically detected based on the length of the digest string.

Examples:

```
sha1$pub$f0f0c49b8a04a2dd2110e10f7806c97d87d0b26f  # pubkey␣
↪SHA-1
sha1$f0f0c49b8a04a2dd2110e10f7806c97d87d0b26f      #␣
↪certificate SHA-1
f0f0c49b8a04a2dd2110e10f7806c97d87d0b26f           #␣
↪certificate SHA-1
```

You can specify a list of fingerprints in case you have several acceptable certificates. If a fingerprint matches the topmost (i.e. leaf) certificate no additional validations can make the verification fail.

Note that the default behavior of SSL_fingerprint is to be an alternative to the normal validation. A mismatching fingerprint does not override a successful validation of the certificate, but a matching fingerprint can override an otherwise failed validation. To enforce a fingerprint see `SSL_force_fingerprint`.

– SSL_force_fingerprint

Set this to true to enforce strict certificate pinning: the connection will be rejected unless the peer certificate matches one of the fingerprints given in `SSL_fingerprint`, even if the certificate chain would otherwise be trusted.

Without this option, `SSL_fingerprint` acts as an alternative trust anchor — a matching fingerprint can unlock an otherwise-rejected certificate (e.g. self-signed or from an unknown CA), but a non-matching fingerprint does not block a certificate that passes normal chain validation. With this option, the fingerprint check is the sole trust decision: any certificate not in the fingerprint list is rejected regardless of CA trust.

– SSL_cert_file | SSL_cert | SSL_key_file | SSL_key

If you create a server you usually need to specify a server certificate which should be verified by the client. Same is true for client certificates, which should be verified by the server. The certificate can be given as a file with SSL_cert_file or as an internal representation of an X509* object (like you get from *Net::SSLeay* or IO::Socket::SSL::Utils::PEM_xxx2cert) with SSL_cert. If given as a file it will automatically detect the format. Supported file formats are PEM, DER and PKCS#12, where PEM and PKCS#12 can contain the certificate and the chain to use, while DER can only contain a single certificate.

If given as a list of X509* please note, that the all the chain certificates (e.g. all except the first) will be "consumed" by openssl and will be freed if the SSL context gets destroyed - so you should never free them yourself. But the servers certificate (e.g. the first) will not be consumed by openssl and thus must be freed by the application.

For each certificate a key is need, which can either be given as a file with SSL_key_file or as an internal representation of an EVP_PKEY* object with SSL_key (like you get from *Net::SSLeay* or IO::Socket::SSL::Utils::PEM_xxx2key). If a key was already given within the PKCS#12 file specified by SSL_cert_file it will ignore any SSL_key or SSL_key_file. If no SSL_key or SSL_key_file was given it will try to use the PEM file given with SSL_cert_file again, maybe it contains the key too.

If your SSL server should be able to use different certificates on the same IP address, depending on the name given by SNI, you can use a hash reference instead of a file with <hostname = cert_file>>.

If your SSL server should be able to use both RSA and ECDSA certificates for the same domain/IP a similar hash reference like with SNI is given. The domain names used to specify the additional certificates should be `hostname%whatever`, i.e. `hostname%ecc` or similar. This needs at least OpenSSL 1.0.2. To let the server pick the certificate based on the clients cipher preference `SSL_honor_cipher_order` should be set to false.

In case certs and keys are needed but not given it might fall back to builtin defaults, see "Defaults for Cert, Key and CA".

Examples:

```
SSL_cert_file => 'mycert.pem',
SSL_key_file => 'mykey.pem',

SSL_cert_file => {
   "foo.example.org" => 'foo-cert.pem',
   "foo.example.org%ecc" => 'foo-ecc-cert.pem',
   "bar.example.org" => 'bar-cert.pem',
   # used when nothing matches or client does not support SNI
   '' => 'default-cert.pem',
   '%ecc' => 'default-ecc-cert.pem',
},
SSL_key_file => {
   "foo.example.org" => 'foo-key.pem',
   "foo.example.org%ecc" => 'foo-ecc-key.pem',
   "bar.example.org" => 'bar-key.pem',
```

(continues on next page)

(continued from previous page)

```
    # used when nothing matches or client does not support SNI
    '' => 'default-key.pem',
    '%ecc' => 'default-ecc-key.pem',
}
```

– SSL_passwd_cb

  If your private key is encrypted, you might not want the default password prompt from Net::SSLeay. This option takes a reference to a subroutine that should return the password required to decrypt your private key.

– SSL_use_cert

  If this is true, it forces IO::Socket::SSL to use a certificate and key, even if you are setting up an SSL client. If this is set to 0 (the default), then you will only need a certificate and key if you are setting up a server.

  SSL_use_cert will implicitly be set if SSL_server is set. For convenience it is also set if it was not given but a cert was given for use (SSL_cert_file or similar).

– SSL_version

  Sets the version of the SSL protocol used to transmit data. 'SSLv23' uses a handshake compatible with SSL2.0, SSL3.0 and TLS1.x, while 'SSLv2', 'SSLv3', 'TLSv1', 'TLSv1_1', 'TLSv1_2', or 'TLSv1_3' restrict handshake and protocol to the specified version. All values are case-insensitive. Instead of 'TLSv1_1', 'TLSv1_2', and 'TLSv1_3' one can also use 'TLSv11', 'TLSv12', and 'TLSv13'. Which protocol versions are actually supported depend on the versions of OpenSSL and Net::SSLeay installed, but modern protocols like TLS 1.3 are supported by these for many years now.

  Independent from the handshake format you can limit to set of accepted SSL versions by adding !version separated by ':'.

  The default SSL_version is 'SSLv23:!TLSv1:!TLSv1_1:!SSLv3:!SSLv2'. This means, that the handshake format is compatible to SSL2.0 and higher, but that the successful handshake is limited to TLS1.2 and higher, that is no SSL2.0, SSL3.0, TLS 1.0 or TLS 1.1 because these versions have serious security issues and should not be used anymore.

  You can also use !TLSv1_1 and !TLSv1_2 to disable TLS versions 1.1 and 1.2 while still allowing TLS version 1.0. Setting the version instead to 'TLSv1' might break interaction with very old or broken clients, which expect a SSL2.0 compatible handshake. On the other side some broken clients just close the connection when they receive a TLS version 1.1 request. In this case setting the version to 'SSLv23:!SSLv2:!SSLv3:!TLSv1_1:!TLSv1_2' might help.

– SSL_cipher_list

  If this option is set the cipher list for the connection will be set to the given value, e.g. something like 'ALL:!LOW:!EXP:!aNULL'. This will only affect ciphers for TLS 1.2 and lower. See the OpenSSL documentation (https://www.openssl.org/docs/manmaster/man1/openssl-ciphers.html#CIPHER-STRINGS) for more details.

Unless you fail to contact your peer because of no shared ciphers it is recommended to leave this option at the default setting, which uses the system default but disables some insecure ciphers which might still be enabled on older systems.

In case different cipher lists are needed for different SNI hosts a hash can be given with the host as key and the cipher suite as value, similar to **SSL_cert***.

– SSL_ciphersuites

If this option is set the TLS 1.3 ciphersuites for the connection will be set to the given value. This is similar to SSL_cipher_list, but only for TLS 1.3 ciphers. See argument `-ciphersuites` in the OpenSSL documentation (https://www.openssl.org/docs/manmaster/man1/openssl-ciphers.html) for details.

Unless you fail to contact your peer because of no shared ciphers it is recommended to leave this option at the default setting, which uses the system default.

In case different cipher lists are needed for different SNI hosts a hash can be given with the host as key and the cipher suite as value, similar to **SSL_cert***.

– SSL_honor_cipher_order

If this option is true the cipher order the server specified is used instead of the order proposed by the client. This option defaults to true to make use of our secure cipher list setting.

– SSL_dh_file

To create a server which provides forward secrecy you need to either give the DH parameters or (better, because faster) the ECDH curve. This setting cares about DH parameters.

To support non-elliptic Diffie-Hellman key exchange a suitable file needs to be given here or the SSL_dh should be used with an appropriate value. See dhparam command in openssl for more information.

If neither `SSL_dh_file` nor `SSL_dh` are set a builtin DH parameter with a length of 2048 bit is used to offer DH key exchange by default. If you don't want this (e.g. disable DH key exchange) explicitly set this or the `SSL_dh` parameter to undef.

– SSL_dh

Like SSL_dh_file, but instead of giving a file you use a preloaded or generated DH*.

– SSL_ecdh_curve

To create a server which provides forward secrecy you need to either give the DH parameters or (better, because faster) the ECDH curve. This setting cares about the ECDH curve(s).

To support Elliptic Curve Diffie-Hellmann key exchange the OID or NID of at least one suitable curve needs to be provided here.

With OpenSSL 1.1.0+ this parameter defaults to `auto`, which means that it lets OpenSSL pick the best settings. If support for CTX_set_ecdh_auto is implemented in Net::SSLeay (needs at least version 1.86) it will use this to implement

the same default. Otherwise it will default to `prime256v1` (builtin of OpenSSL) in order to offer ECDH key exchange by default.

If setting groups or curves is supported by Net::SSLeay (needs at least version 1.86) then multiple curves can be given here in the order of the preference, i.e. `P-521:P-384:P-256`. When used at the client side this will include the supported curves as extension in the TLS handshake.

If you don't want to have ECDH key exchange this could be set to undef or set `SSL_ciphers` to exclude all of these ciphers.

You can check if ECDH support is available by calling `IO::Socket::SSL->can_ecdh`.

– SSL_verify_mode

This option sets the verification mode for the peer certificate. You may combine SSL_VERIFY_PEER (verify_peer), SSL_VERIFY_FAIL_IF_NO_PEER_CERT (fail verification if no peer certificate exists; ignored for clients), SSL_VERIFY_CLIENT_ONCE (verify client once; ignored for clients). Note that SSL_VERIFY_FAIL_IF_NO_PEER_CERT and SSL_VERIFY_CLIENT_ONCE must be used together with SSL_VERIFY_PEER. See OpenSSL man page for SSL_CTX_set_verify for more information.

The default is SSL_VERIFY_NONE for server (e.g. no check for client certificate) and SSL_VERIFY_PEER for client (check server certificate).

– SSL_verify_callback

If you want to verify certificates yourself, you can pass a sub reference along with this parameter to do so. When the callback is called, it will be passed:

1. a true/false value that indicates what OpenSSL thinks of the certificate,
2. a C-style memory address of the certificate store,
3. a string containing the certificate's issuer attributes and owner attributes, and
4. a string containing any errors encountered (0 if no errors).
5. a C-style memory address of the peer's own certificate (convertible to PEM form with Net::SSLeay::PEM_get_string_X509()).
6. The depth of the certificate in the chain. Depth 0 is the leaf certificate.

The function should return 1 or 0, depending on whether it thinks the certificate is valid or invalid. The default is to let OpenSSL do all of the busy work.

The callback will be called for each element in the certificate chain.

See the OpenSSL documentation for SSL_CTX_set_verify for more information.

– SSL_verifycn_scheme

The scheme is used to correctly verify the identity inside the certificate by using the hostname of the peer. See the information about the verification schemes in **verify_hostname**.

If you don't specify a scheme it will use 'default', but only complain loudly if the name verification fails instead of letting the whole certificate verification

fail. THIS WILL CHANGE, e.g. it will let the certificate verification fail in the future if the hostname does not match the certificate !!!! To override the name used in verification use **SSL_verifycn_name**.

The scheme 'default' is a superset of the usual schemes, which will accept the hostname in common name and subjectAltName and allow wildcards everywhere. While using this scheme is way more secure than no name verification at all you better should use the scheme specific to your application protocol, e.g. 'http', 'ftp'...

If you are really sure, that you don't want to verify the identity using the hostname you can use 'none' as a scheme. In this case you'd better have alternative forms of verification, like a certificate fingerprint or do a manual verification later by calling **verify_hostname** yourself.

– SSL_verifycn_publicsuffix

This option is used to specify the behavior when checking wildcards certificates for public suffixes, e.g. no wildcard certificates for *.com or *.co.uk should be accepted, while *.example.com or *.example.co.uk is ok.

If not specified it will simply use the builtin default of *IO::Socket::SSL::PublicSuffix*, you can create another object with from_string or from_file of this module.

To disable verification of public suffix set this option to `''`.

– SSL_verifycn_name

Set the name which is used in verification of hostname. If SSL_verifycn_scheme is set and no SSL_verifycn_name is given it will try to use SSL_hostname or PeerHost and PeerAddr settings and fail if no name can be determined. If SSL_verifycn_scheme is not set it will use a default scheme and warn if it cannot determine a hostname, but it will not fail.

Using PeerHost or PeerAddr works only if you create the connection directly with `IO::Socket::SSL->new`, if an IO::Socket::INET object is upgraded with **start_SSL** the name has to be given in **SSL_verifycn_name** or **SSL_hostname**.

– SSL_check_crl

If you want to verify that the peer certificate has not been revoked by the signing authority, set this value to true. OpenSSL will search for the CRL in your SSL_ca_path, or use the file specified by SSL_crl_file. See the Net::SSLeay documentation for more details.

– SSL_crl_file

If you want to specify the CRL file to be used, set this value to the pathname to be used. This must be used in addition to setting SSL_check_crl.

– SSL_ocsp_mode

Defines how certificate revocation is done using OCSP (Online Status Revocation Protocol). The default is to send a request for OCSP stapling to the server and if the server sends an OCSP response back the result will be used.

Any other OCSP checking needs to be done manually with `ocsp_resolver`.

The following flags can be combined with `|`:

* SSL_OCSP_NO_STAPLE

  Don't ask for OCSP stapling. This is the default if SSL_verify_mode is VERIFY_NONE.

* SSL_OCSP_TRY_STAPLE

  Try OCSP stapling, but don't fail if no usable stapled response is received. Both a missing staple and a broken or unverifiable staple are treated the same: the connection continues without OCSP revocation information from the staple. This is the default if SSL_verify_mode is VERIFY_PEER (the default).

* SSL_OCSP_MUST_STAPLE

  Require a valid stapled OCSP response; fail the connection if none is received. A broken or unverifiable staple is treated the same as a missing one — both cause the connection to fail with "did not receive the required stapled OCSP response". Most servers currently send no stapled OCSP response back.

* SSL_OCSP_FAIL_HARD

  Fail hard on response errors, default is to fail soft like the browsers do. Soft errors mean, that the OCSP response is not usable, e.g. no response, error response, no valid signature etc. Certificate revocations inside a verified response are considered hard errors in any case.

  Soft errors inside a stapled response are never considered hard, e.g. it is expected that in this case an OCSP request will be send to the responsible OCSP responder.

* SSL_OCSP_FULL_CHAIN

  This will set up the `ocsp_resolver` so that all certificates from the peer chain will be checked, otherwise only the leaf certificate will be checked against revocation.

– SSL_ocsp_staple_callback

  If this callback is defined, it will be called with the SSL object and the OCSP response handle obtained from the peer, e.g. `<$cb-($ssl,$resp)>>`. If the peer did not provide a stapled OCSP response the function will be called with `$resp=undef`. Because the OCSP response handle is no longer valid after leaving this function it should not by copied or freed. If access to the response is necessary after leaving this function it can be serialized with `Net::SSLeay::i2d_OCSP_RESPONSE`.

  If no such callback is provided, it will use the default one, which verifies the response and uses it to check if the certificate(s) of the connection got revoked.

– SSL_ocsp_cache

  With this option a cache can be given for caching OCSP responses, which could be shared between different SSL contexts. If not given a cache specific to the SSL context only will be used.

  You can either create a new cache with `IO::Socket::SSL::OCSP_Cache->new([size])` or implement your own cache, which needs to have methods `put($key,`

\%entry) and get($key) (returning \%entry) where entry is the hash representation of the OCSP response with fields like nextUpdate. The default implementation of the cache will consider responses valid as long as nextUpdate is less then the current time.

– SSL_psk

PSK to use instead of certificate. Requires the peer to use the same PSK.

On the client side the identity and psk need to be provided as array-ref with <[identity,psk]>. Alternatively only the psk might be provided, in which case io_socket_ssl is used as identity. On the server side a single psk might be given which will then be use no matter which identity hint was given by the client. Alternatively a hash-ref can be provided with identity as key and psk as value to serve clients with different psk based on the identity hint. The special identity '' is used to set a default psk.

To use PSK make sure to have the relevant ciphers enabled. With TLS 1.3 this is implicit, but with TLS 1.2 and lower explicitly add PSK to SSL_cipher_list since it is not enabled by default.

Note that the given psk is expected to be binary and not hexadecimal as some other tools do. To convert hex to binary use pack("H*",hexstring). Also note that the error messages one gets if either identity or psk don't match the configuration in the peer can be useless ("handshake failure") or misleading ("bad mac").

PSK support needs respective API functionality in Net::SSLeay, which was only fully included in 1.94. IO::Socket::SSL->can_psk() returns undef if functionality is not available and a hash-ref with details otherwise: key client in the hash signals support for using PSK in a SSL client (available since Net::SSLeay 1.82) while key server for support of PSK in SSL server (since Net::SSLeay 1.94).

– SSL_reuse_ctx

If you have already set the above options for a previous instance of IO::Socket::SSL, then you can reuse the SSL context of that instance by passing it as the value for the SSL_reuse_ctx parameter. You may also create a new instance of the IO::Socket::SSL::SSL_Context class, using any context options that you desire without specifying connection options, and pass that here instead.

If you use this option, all other context-related options that you pass in the same call to new() will be ignored unless the context supplied was invalid. Note that, contrary to versions of IO::Socket::SSL below v0.90, a global SSL context will not be implicitly used unless you use the set_default_context() function.

– SSL_create_ctx_callback

With this callback you can make individual settings to the context after it got created and the default setup was done. The callback will be called with the CTX object from Net::SSLeay as the single argument.

Example for limiting the server session cache size:

```
SSL_create_ctx_callback => sub {
    my $ctx = shift;
    Net::SSLeay::CTX_sess_set_cache_size($ctx,128);
}
```

– SSL_session_cache_size

  If you make repeated connections to the same host/port and the SSL renegotiation time is an issue, you can turn on client-side session caching with this option by specifying a positive cache size. For successive connections, pass the SSL_reuse_ctx option to the new() calls (or use set_default_context()) to make use of the cached sessions. The session cache size refers to the number of unique host/port pairs that can be stored at one time; the oldest sessions in the cache will be removed if new ones are added.

  This option does not effect the session cache a server has for it's clients, e.g. it does not affect SSL objects with SSL_server set.

  Note that session caching with TLS 1.3 needs at least Net::SSLeay 1.86.

– SSL_session_cache

  Specifies session cache object which should be used instead of creating a new. Overrules SSL_session_cache_size. This option is useful if you want to reuse the cache, but not the rest of the context.

  A session cache object can be created using `IO::Socket::SSL::Session_Cache->new( cachesize )`.

  Use set_default_session_cache() to set a global cache object.

– SSL_session_key

  Specifies a key to use for lookups and inserts into client-side session cache. Per default ip:port of destination will be used, but sometimes you want to share the same session over multiple ports on the same server (like with FTPS).

– SSL_session_id_context

  This gives an id for the servers session cache. It's necessary if you want clients to connect with a client certificate. If not given but SSL_verify_mode specifies the need for client certificate a context unique id will be picked.

– SSL_error_trap

  When using the accept() or connect() methods, it may be the case that the actual socket connection works but the SSL negotiation fails, as in the case of an HTTP client connecting to an HTTPS server. Passing a subroutine ref attached to this parameter allows you to gain control of the orphaned socket instead of having it be closed forcibly. The subroutine, if called, will be passed two parameters: a reference to the socket on which the SSL negotiation failed and the full text of the error message.

– SSL_npn_protocols

  If used on the server side it specifies list of protocols advertised by SSL server as an array ref, e.g. ['spdy/2','http1.1']. On the client side it specifies the

protocols offered by the client for NPN as an array ref. See also method `next_proto_negotiated`.

Next Protocol Negotiation (NPN) is available with Net::SSLeay 1.46+ and openssl-1.0.1+. NPN is unavailable in TLSv1.3 protocol. To check support you might call `IO::Socket::SSL->can_npn()`. If you use this option with an unsupported Net::SSLeay/OpenSSL it will throw an error.

– SSL_alpn_protocols

If used on the server side it specifies list of protocols supported by the SSL server as an array ref, e.g. ['http/2.0', 'spdy/3.1','http/1.1']. On the client side it specifies the protocols advertised by the client for ALPN as an array ref. See also method `alpn_selected`.

Application-Layer Protocol Negotiation (ALPN) is available with Net::SSLeay 1.56+ and openssl-1.0.2+. More details about the extension are in RFC7301. To check support you might call `IO::Socket::SSL->can_alpn()`. If you use this option with an unsupported Net::SSLeay/OpenSSL it will throw an error.

Note that some client implementations may encounter problems if both NPN and ALPN are specified. Since ALPN is intended as a replacement for NPN, try providing ALPN protocols then fall back to NPN if that fails.

– SSL_ticket_keycb => [$sub,$data] | $sub

This is a callback used for stateless session reuse (Session Tickets, RFC 5077).

This callback will be called as `$sub->($data,[$key_name])` where `$data` is the argument given to SSL_ticket_keycb (or undef) and `$key_name` depends on the mode:

* encrypt ticket

  If a ticket needs to be encrypted the callback will be called without `$key_name`. In this case it should return (`$current_key`, `$current_key_name`) where `$current_key` is the current key (32 byte random data) and `$current_key_name` the name associated with this key (exactly 16 byte). This `$current_key_name` will be incorporated into the ticket.

* decrypt ticket

  If a ticket needs to be decrypted the callback will be called with `$key_name` as found in the ticket. It should return (`$key`,`$current_key_name`) where `$key` is the key associated with the given `$key_name` and `$current_key_name` the name associated with the currently active key. If `$current_key_name` is different from the given `$key_name` the callback will be called again to re-encrypt the ticket with the currently active key.

  If no key can be found which matches the given `$key_name` then this function should return nothing (empty list).

  This mechanism should be used to limit the life time for each key encrypting the ticket. Compromise of a ticket encryption key might lead to decryption of SSL sessions which used session tickets protected by this key.

Example:

```
Net::SSLeay::RAND_bytes(my $oldkey,32);
Net::SSLeay::RAND_bytes(my $newkey,32);
my $oldkey_name = pack("a16",'oldsecret');
my $newkey_name = pack("a16",'newsecret');

my @keys = (
   [ $newkey_name, $newkey ], # current active key
   [ $oldkey_name, $oldkey ], # already expired
);

my $keycb = [ sub {
   my ($mykeys,$name) = @_;

   # return (current_key, current_key_name) if no name given
   return ($mykeys->[0][1],$mykeys->[0][0]) if ! $name;

   # return (matching_key, current_key_name) if we find a
 ↪key matching
   # the given name
   for(my $i = 0; $i<@$mykeys; $i++) {
       next if $name ne $mykeys->[$i][0];
       return ($mykeys->[$i][1],$mykeys->[0][0]);
   }

   # no matching key found
   return;
},\@keys ];

my $srv = IO::Socket::SSL->new(..., SSL_ticket_keycb =>
 ↪$keycb);
```

– SSL_on_peer_shutdown CODE

  This defines a callback which gets called on receiving SSL shutdown (close notify) from the peer. This can be used to override the default behavior, see `stop_SSL` for more.

– SSL_mode_release_buffers 1|0

  This enables or disables the SSL_MODE_RELEASE_BUFFERS option on the SSL object. With this option the read buffer will be released after each SSL_read but will need to be reallocated for each new SSL_read. If memory usage is a concern this might save lots of memory in the mean time though, about 34k per idle SSL connection according to the documentation in SSL_CTX_set_mode(3ssl).

– SSL_usebio 1|0

  If this option is used with `start_SSL` (not supported somewhere else) it will not upgrade the given IO handle but instead wrap it inside a new IO object and use openssl memory BIO's to manage exchange between TLS layer and socket layer. While this will add some overhead it makes it possible to stack TLS layers, i.e. first create an outer TLS connection (without or with SSL_use-

bio) and then create the inner TLS layer with SSL_usebio, using the outer TLS connection as the IO handle in `start_SSL`.

Example:

```
# create TLS connection to TLS enabled http proxy
my $outer = IO::Socket::SSL->new('proxy.test:3128');

# send proxy request
print $outer "CONNECT real-target.test:443 HTTP/1.0\r\n\r\n";
... # read and check response from proxy on $outer

# create an inner TLS connection to the server inside $outer
my $inner = IO::Socket::SSL->start_SSL($outer,
   SSL_usebio => 1,
   SSL_hostname => 'real-target.test', # SNI
);

# send request to server through proxy using $inner
print $inner "GET / HTTP/1.0\r\nHost: real-target.test\r\n\r\
 ↪n";
... # read response from server on $inner
```

- **accept**

  This behaves similar to the accept function of the underlying socket class, but additionally does the initial SSL handshake. But because the underlying socket class does return a blocking file handle even when accept is called on a non-blocking socket, the SSL handshake on the new file object will be done in a blocking way. Please see the section about non-blocking I/O for details. If you don't like this behavior you should do accept on the TCP socket and then upgrade it with `start_SSL` later.

- **connect(...)**

  This behaves similar to the connect function but also does an SSL handshake. Because you cannot give SSL specific arguments to this function, you should better either use `new` to create a connect SSL socket or `start_SSL` to upgrade an established TCP socket to SSL.

- **close(...)**

  Contrary to a close for a simple INET socket a close in SSL also mandates a proper shutdown of the SSL part. This is done by sending a close notify message by both peers.

  A naive implementation would thus wait until it receives the close notify message from the peer - which conflicts with the commonly expected semantic that a close will not block. The default behavior is thus to only send a close notify but not wait for the close notify of the peer. If it should instead wait for a close notify from the peer than `SSL_fast_shutdown` must be explicitly set to false:

  ```
  $self->close(SSL_fast_shutdown => 0);
  ```

  There are also cases where a SSL shutdown should not be done at all. This is true

for example when forking to let a child deal with the socket and closing the socket in the parent process. A naive explicit `close` or an implicit close when destroying the socket in the parent would send a close notify to the peer which would make the SSL socket in the client process unusable. In this case an explicit `close` with `SSL_no_shutdown` set to true should be done in the parent process:

```
$self->close(SSL_no_shutdown => 1);
```

For more details and other arguments see `stop_SSL` which gets called from `close` to shutdown the SSL state of the socket.

- **sysread(BUF, LEN, [OFFSET])**

  This function behaves from the outside the same as **sysread** in other *IO::Socket* objects, e.g. it returns at most LEN bytes of data. But in reality it reads not only LEN bytes from the underlying socket, but at a single SSL frame. It then returns up to LEN bytes it decrypted from this SSL frame. If the frame contained more data than requested it will return only LEN data, buffer the rest and return it on further read calls. This means, that it might be possible to read data, even if the underlying socket is not readable, so using poll or select might not be sufficient.

  sysread will only return data from a single SSL frame, e.g. either the pending data from the already buffered frame or it will read a frame from the underlying socket and return the decrypted data. It will not return data spanning several SSL frames in a single call.

  Also, calls to sysread might fail, because it must first finish an SSL handshake.

  To understand these behaviors is essential, if you write applications which use event loops and/or non-blocking sockets. Please read the specific sections in this documentation.

- **syswrite(BUF, [LEN, [OFFSET]])**

  This functions behaves from the outside the same as **syswrite** in other *IO::Socket* objects, e.g. it will write at most LEN bytes to the socket, but there is no guarantee, that all LEN bytes are written. It will return the number of bytes written. Because it basically just calls SSL_write from OpenSSL syswrite will write at most a single SSL frame. This means, that no more than 16.384 bytes, which is the maximum size of an SSL frame, will be written at once.

  For non-blocking sockets SSL specific behavior applies. Please read the specific section in this documentation.

- **peek(BUF, LEN, [OFFSET])**

  This function has exactly the same syntax as **sysread**, and performs nearly the same task but will not advance the read position so that successive calls to peek() with the same arguments will return the same results.

- **pending()**

  This function gives you the number of bytes available without reading from the underlying socket object. This function is essential if you work with event loops, please see the section about polling SSL sockets.

- **get_fingerprint([algo,certificate,pubkey])**

This methods returns the fingerprint of the given certificate in the form `algo$digest_hex`, where `algo` is the used algorithm, default 'sha256'. If no certificate is given the peer certificate of the connection is used. If `pubkey` is true it will not return the fingerprint of the certificate but instead the fingerprint of the pubkey inside the certificate as `algo$pub$digest_hex`.

- **get_fingerprint_bin([algo,certificate,pubkey])**

  This methods returns the binary fingerprint of the given certificate by using the algorithm `algo`, default 'sha256'. If no certificate is given the peer certificate of the connection is used. If `pubkey` is true it will not return the fingerprint of the certificate but instead the fingerprint of the pubkey inside the certificate.

- **get_cipher()**

  Returns the string form of the cipher that the IO::Socket::SSL object is using.

- **get_sslversion()**

  Returns the string representation of the SSL version of an established connection.

- **get_sslversion_int()**

  Returns the integer representation of the SSL version of an established connection.

- **get_session_reused()**

  This returns true if the session got reused and false otherwise. Note that with a reused session no certificates are send within the handshake and no ciphers are offered and thus functions which rely on this might not work.

- **dump_peer_certificate()**

  Returns a parsable string with select fields from the peer SSL certificate. This method directly returns the result of the dump_peer_certificate() method of Net::SSLeay.

- **peer_certificate($field;[$refresh])**

  If a peer certificate exists, this function can retrieve values from it. If no field is given the internal representation of certificate (X509 handle) from Net::SSLeay is returned, with reference owned internally (i.e. should not be freed). If refresh is true it will not used a cached version, but check again in case the certificate of the connection has changed due to renegotiation.

  The following fields can be queried:

  – authority (alias issuer)

    The certificate authority which signed the certificate.

  – owner (alias subject)

    The owner of the certificate.

  – commonName (alias cn) - only for Net::SSLeay version >=1.30

    The common name, usually the server name for SSL certificates.

  – subjectAltNames - only for Net::SSLeay version >=1.33

    Alternative names for the subject, usually different names for the same server, like example.org, example.com, *.example.com.

It returns a list of (typ,value) with typ GEN_DNS, GEN_IPADD etc (these constants are exported from IO::Socket::SSL). See Net::SSLeay::X509_get_subjectAltNames.

- **sock_certificate($field)**

  This is similar to `peer_certificate` but will return the sites own certificate. The same arguments for **$field** can be used. If no **$field** is given the X509 handle from the underlying OpenSSL will be returned. This handle is owned internally and will only be valid as long as the SSL connection exists. If used afterwards it might result in strange crashes of the application.

- **peer_certificates**

  This returns all the certificates send by the peer as X509 handles, e.g. first the peers own certificate and then the rest of the chain. You might use **CERT_asHash** from *IO::Socket::SSL::Utils* to inspect each of the certificates.

  Like with **sock_certificate** and **peer_certificates** the handles are owned internally and will only be valid as long as the SSL connection exists.

  This function depends on a version of Net::SSLeay >= 1.58 .

- **get_servername**

  This gives the name requested by the client if Server Name Indication (SNI) was used.

- **verify_hostname($hostname,$scheme,$publicsuffix)**

  This verifies the given hostname against the peer certificate using the given scheme. Hostname is usually what you specify within the PeerAddr. See the `SSL_verifycn_publicsuffix` parameter for an explanation of suffix checking and for the possible values.

  Verification of hostname against a certificate is different between various applications and RFCs. Some scheme allow wildcards for hostnames, some only in subjectAltNames, and even their different wildcard schemes are possible. RFC 6125 provides a good overview.

  To ease the verification the following schemes are predefined (both protocol name and rfcXXXX name can be used):

  – rfc2818, xmpp (rfc3920), ftp (rfc4217)

    Extended wildcards in subjectAltNames and common name are possible, e.g. *.example.org or even www*.example.org. The common name will be only checked if no DNS names are given in subjectAltNames.

  – http (alias www)

    While name checking is defined in rfc2818 the current browsers usually accept also an IP address (w/o wildcards) within the common name as long as no subjectAltNames are defined. Thus this is rfc2818 extended with this feature.

  – smtp (rfc2595), imap, pop3, acap (rfc4642), netconf (rfc5538), syslog (rfc5425), snmp (rfc5953)

    Simple wildcards in subjectAltNames are possible, e.g. *.example.org matches www.example.org but not lala.www.example.org. If nothing from subjectAlt-

Names match it checks against the common name, where wildcards are also allowed to match the full leftmost label.

- ldap (rfc4513)

  Simple wildcards are allowed in subjectAltNames, but not in common name. Common name will be checked even if subjectAltNames exist.

- sip (rfc5922)

  No wildcards are allowed and common name is checked even if subjectAltNames exist.

- gist (rfc5971)

  Simple wildcards are allowed in subjectAltNames and common name, but common name will only be checked if there are no DNS names in subjectAltNames.

- default

  This is a superset of all the rules and is automatically used if no scheme is given but a hostname (instead of IP) is known. Extended wildcards are allowed in subjectAltNames and common name and common name is checked always.

- none

  No verification will be done. Actually is does not make any sense to call verify_hostname in this case.

The scheme can be given either by specifying the name for one of the above predefined schemes, or by using a hash which can have the following keys and values:

- check_cn: 0|'always'|'when_only'

  Determines if the common name gets checked. If 'always' it will always be checked (like in ldap), if 'when_only' it will only be checked if no names are given in subjectAltNames (like in http), for any other values the common name will not be checked.

- wildcards_in_alt: 0|'full_label'|'anywhere'

  Determines if and where wildcards in subjectAltNames are possible. If 'full_label' only cases like *.example.org will be possible (like in ldap), for 'anywhere' www*.example.org is possible too (like http), dangerous things like but www.*.org or even '*' will not be allowed. For compatibility with older versions 'leftmost' can be given instead of 'full_label'.

- wildcards_in_cn: 0|'full_label'|'anywhere'

  Similar to wildcards_in_alt, but checks the common name. There is no predefined scheme which allows wildcards in common names.

- ip_in_cn: 0|1|4|6

  Determines if an IP address is allowed in the common name (no wildcards are allowed). If set to 4 or 6 it only allows IPv4 or IPv6 addresses, any other true value allows both.

- callback: \&coderef

If you give a subroutine for verification it will be called with the arguments ($hostname,$commonName,@subjectAltNames), where hostname is the name given for verification, commonName is the result from peer_certificate('cn') and subjectAltNames is the result from peer_certificate('subjectAltNames').

All other arguments for the verification scheme will be ignored in this case.

- **next_proto_negotiated()**

  This method returns the name of negotiated protocol - e.g. 'http/1.1'. It works for both client and server side of SSL connection.

  NPN support is available with Net::SSLeay 1.46+ and openssl-1.0.1+. To check support you might call `IO::Socket::SSL->can_npn()`.

- **alpn_selected()**

  Returns the protocol negotiated via ALPN as a string, e.g. 'http/1.1', 'http/2.0' or 'spdy/3.1'.

  ALPN support is available with Net::SSLeay 1.56+ and openssl-1.0.2+. To check support, use `IO::Socket::SSL->can_alpn()`.

- **errstr()**

  Returns the last error (in string form) that occurred. If you do not have a real object to perform this method on, call IO::Socket::SSL::errstr() instead.

  For read and write errors on non-blocking sockets, this method may include the string `SSL wants a read first!` or `SSL wants a write first!` meaning that the other side is expecting to read from or write to the socket and wants to be satisfied before you get to do anything. But with version 0.98 you are better comparing the global exported variable $SSL_ERROR against the exported symbols SSL_WANT_READ and SSL_WANT_WRITE.

- **opened()**

  This returns false if the socket could not be opened, 1 if the socket could be opened and the SSL handshake was successful done and -1 if the underlying IO::Handle is open, but the SSL handshake failed.

- **is_SSL()**

  This returns undefined if there is no SSL object associated with the socket. It will return `rw` if the state of the SSL object has SSL active for both directions, `r` if there was a locally initiated SSL shutdown (close notify), `w` if a remote SSL shutdown was received and `` (empty string but defined) if a bidirectional SSL shutdown was done.

- **IO::Socket::SSL->start_SSL($socket, ...)**

  This will convert a glob reference or a socket that you provide to an IO::Socket::SSL object. You may also pass parameters to specify context or connection options as with a call to new(). If you are using this function on an accept()ed socket, you must set the parameter "SSL_server" to 1, i.e. IO::Socket::SSL->start_SSL($socket, SSL_server => 1). If you have a class that inherits from IO::Socket::SSL and you want the $socket to be blessed into your own class instead, use MyClass->start_SSL($socket) to achieve the desired effect.

Note that if start_SSL() fails (wrong arguments, error in SSL negotiation ...), it will try to bless the socket back to its original class. This will fail if the given socket was not blessed, since there is no builtin "unbless". Therefore better use IO::Socket (or other IO::Handle) objects as `$socket` instead of creating unblessed sockets with `socket` or similar.

For non-blocking sockets you better just upgrade the socket to IO::Socket::SSL and call accept_SSL or connect_SSL and the upgraded object. To just upgrade the socket set **SSL_startHandshake** explicitly to 0. If you call start_SSL w/o this parameter it will revert to blocking behavior for accept_SSL and connect_SSL.

If given the parameter "Timeout" it will stop if after the timeout no SSL connection was established. This parameter is only used for blocking sockets, if it is not given the default Timeout from the underlying IO::Socket will be used.

- **stop_SSL(...)**

  This is the opposite of start_SSL(), connect_SSL() and accept_SSL(), e.g. it will shutdown the SSL connection and return to the class before start_SSL(). It gets the same arguments as close(), in fact close() calls stop_SSL() (but without downgrading the class).

  Will return true if it succeeded and undef if failed. This might be the case for non-blocking sockets. In this case $! is set to EWOULDBLOCK and the ssl error to SSL_WANT_READ or SSL_WANT_WRITE. In this case the call should be retried again with the same arguments once the socket is ready.

  For calling from `stop_SSL` `SSL_fast_shutdown` default to false, e.g. it waits for the close_notify of the peer. This is necessary in case you want to downgrade the socket and continue to use it as a plain socket. If `SSL_fast_shutdown` is set to true only the write direction will be shutdown for SSL and there can still be SSL data read. Read direction will then be closed either automatically when receiving a shutdown from the peer during read or by explicitly calling `stop_SSL` with `SSL_fast_shutdown` false.

  After a complete stop_SSL the socket can again be used to exchange plain data. Even if the SSL shutdown is only done for writing (`SSL_fast_shutdown`) plain data can be immediately written.

  If the SSL shutdown was initiated by the peer it will lead to an implicit SSL shutdown for reading and sysread will return with 0 in this case. It will also SSL shutdown the local side. After this the socket is ready to receive plain data from the peer and locally send data will also be transmitted in plain. To prevent accidental sending/receiving plain data after such implicit SSL shutdown, the read and write functions will return undef with $ERRNO set to EPERM until an explicit stop_SSL was done. If the peer directly closes the TCP connection ater the SSL shutdown without sending plain data this will still be detected though.

  If this default behavior is not wanted a callback can be setup using `SSL_on_peer_shutdown` which gets `$self` as first argument. The result from the callback will then be returned instead of 0 and no local SSL shutdown will be automatically initiated. Similar no explicit stop_SSL is needed before receiving plain data since the callback is considered explicit action enough.

- **connect_SSL, accept_SSL**

These functions should be used to do the relevant handshake, if the socket got created with `new` or upgraded with `start_SSL` and `SSL_startHandshake` was set to false. They will return undef until the handshake succeeded or an error got thrown. As long as the function returns undef and $! is set to EWOULDBLOCK one could retry the call after the socket got readable (SSL_WANT_READ) or writeable (SSL_WANT_WRITE).

- **set_msg_callback**

  This will add/remove a user defined callback for each message, internally using openssl SSL_set_msg_callback API. To make sure that the callback is active before the handshake starts, combine it with `SSL_startHandshake => 0` in the preceding setup of the SSL object. To remove callback explicitly call it with an empty callback function.

  Example:

```
$sock = IO::Socket::SSL->new( .... , SSL_startHandshake => 0);
# set callback
$sock->set_msg_callback(\&cb, $cbarg1, $cbarg2);
$sock->connect_SSL();

sub cb {
    my ($sock,
        # see SSL_set_msg_callback for the following args
        $direction, $ssl_ver, $content_type, $buf, $len, $ssl,
        $cbarg1, $cbarg2)
        = @_;
    ...
    if (no_longer_need_cb) {
        # disable callback
        $sock->set_msg_callback(undef);
    }
}
```

- **ocsp_resolver**

  This will create an OCSP resolver object, which can be used to create OCSP requests for the certificates of the SSL connection. Which certificates are verified depends on the setting of `SSL_ocsp_mode`: by default only the leaf certificate will be checked, but with SSL_OCSP_FULL_CHAIN all chain certificates will be checked.

  Because to create an OCSP request the certificate and its issuer certificate need to be known it is not possible to check certificates when the trust chain is incomplete or if the certificate is self-signed.

  The OCSP resolver gets created by calling `$ssl->ocsp_resolver` and provides the following methods:

  – hard_error

    This returns the hard error when checking the OCSP response. Hard errors are certificate revocations. With the `SSL_ocsp_mode` of SSL_OCSP_FAIL_HARD any soft error (e.g. failures to get signed information about the certificates) will be considered a hard error too.

The OCSP resolving will stop on the first hard error.

The method will return undef as long as no hard errors occurred and still requests to be resolved. If all requests got resolved and no hard errors occurred the method will return `''`.

– soft_error

This returns the soft error(s) which occurred when asking the OCSP responders.

– requests

This will return a hash consisting of `(url,request)`-tuples, e.g. which contain the OCSP request string and the URL where it should be sent too. The usual way to send such a request is as HTTP POST request with a content-type of `application/ocsp-request` or as a GET request with the base64 and url-encoded request is added to the path of the URL.

After you've handled all these requests and added the response with `add_response` you should better call this method again to make sure, that no more requests are outstanding. IO::Socket::SSL will combine multiple OCSP requests for the same server inside a single request, but some server don't give a response to all these requests, so that one has to ask again with the remaining requests.

– add_response($uri,$response)

This method takes the HTTP body of the response which got received when sending the OCSP request to `$uri`. If no response was received or an error occurred one should either retry or consider `$response` as empty which will trigger a soft error.

The method returns the current value of `hard_error`, e.g. a defined value when no more requests need to be done.

– resolve_blocking(%args)

This combines `requests` and `add_response` which *HTTP::Tiny* to do all necessary requests in a blocking way. `%args` will be given to *HTTP::Tiny* so that you can put proxy settings etc here. *HTTP::Tiny* will be called with `verify_SSL` of false, because the OCSP responses have their own signatures so no extra SSL verification is needed.

If you don't want to use blocking requests you need to roll your own user agent with `requests` and `add_response`.

- **IO::Socket::SSL->new_from_fd($fd, [mode], %sslargs)**

  This will convert a socket identified via a file descriptor into an SSL socket. Note that the argument list does not include a "MODE" argument; if you supply one, it will be thoughtfully ignored (for compatibility with IO::Socket::INET). Instead, a mode of '+<' is assumed, and the file descriptor passed must be able to handle such I/O because the initial SSL handshake requires bidirectional communication.

  Internally the given $fd will be upgraded to a socket object using the `new_from_fd` method of the super class (*IO::Socket::INET* or similar) and then `start_SSL` will be called using the given `%sslargs`. If `$fd` is already an IO::Socket object you should better call `start_SSL` directly.

- **IO::Socket::SSL::default_ca([path|dir| SSL_ca_file** = ..., SSL_ca_path => ...])>

  Determines or sets the default CA path. If existing path or dir or a hash is given it will set the default CA path to this value and never try to detect it automatically. If `undef` is given it will forget any stored defaults and continue with detection of system defaults. If no arguments are given it will start detection of system defaults, unless it has already stored user-set or previously detected values.

  The detection of system defaults works similar to OpenSSL, e.g. it will check the directory specified in environment variable SSL_CERT_DIR or the path OPENSSLDIR/certs (SSLCERTS: on VMS) and the file specified in environment variable SSL_CERT_FILE or the path OPENSSLDIR/cert.pem (SSLCERTS:cert.pem on VMS). Contrary to OpenSSL it will check if the SSL_ca_path contains PEM files with the hash as file name and if the SSL_ca_file looks like PEM. If no usable system default can be found it will try to load and use Mozilla::CA and if not available give up detection. The result of the detection will be saved to speed up future calls.

  The function returns the saved default CA as hash with SSL_ca_file and SSL_ca_path.

- **IO::Socket::SSL::set_default_context(...)**

  You may use this to make IO::Socket::SSL automatically re-use a given context (unless specifically overridden in a call to new()). It accepts one argument, which should be either an IO::Socket::SSL object or an IO::Socket::SSL::SSL_Context object. See the SSL_reuse_ctx option of new() for more details. Note that this sets the default context globally, so use with caution (esp. in mod_perl scripts).

- **IO::Socket::SSL::set_default_session_cache(...)**

  You may use this to make IO::Socket::SSL automatically re-use a given session cache (unless specifically overridden in a call to new()). It accepts one argument, which should be an IO::Socket::SSL::Session_Cache object or similar (e.g. something which implements get_session, add_session and del_session like IO::Socket::SSL::Session_Cache does). See the SSL_session_cache option of new() for more details. Note that this sets the default cache globally, so use with caution.

- **IO::Socket::SSL::set_defaults(%args)**

  With this function one can set defaults for all SSL_* parameter used for creation of the context, like the SSL_verify* parameter. Any SSL_* parameter can be given or the following short versions:

  - mode - SSL_verify_mode
  - callback - SSL_verify_callback
  - scheme - SSL_verifycn_scheme
  - name - SSL_verifycn_name

- **IO::Socket::SSL::set_client_defaults(%args)**

  Similar to `set_defaults`, but only sets the defaults for client mode.

- **IO::Socket::SSL::set_server_defaults(%args)**

  Similar to `set_defaults`, but only sets the defaults for server mode.

- **IO::Socket::SSL::set_args_filter_hack(\&code|'use_defaults')**

Sometimes one has to use code which uses unwanted or invalid arguments for SSL, typically disabling SSL verification or setting wrong ciphers or SSL versions. With this hack it is possible to override these settings and restore sanity. Example:

```
IO::Socket::SSL::set_args_filter_hack( sub {
    my ($is_server,$args) = @_;
    if ( ! $is_server ) {
        # client settings - enable verification with default CA
        # and fallback hostname verification etc
        delete @{$args}{qw(
            SSL_verify_mode
            SSL_ca_file
            SSL_ca_path
            SSL_verifycn_scheme
            SSL_version
        )};
        # and add some fingerprints for known certs which are␣
↪signed by
        # unknown CAs or are self-signed
        $args->{SSL_fingerprint} = ...
    }
});
```

With the short setting `set_args_filter_hack('use_defaults')` it will prefer the default settings in all cases. These default settings can be modified with `set_defaults`, `set_client_defaults` and `set_server_defaults`.

The following methods are unsupported (not to mention futile!) and IO::Socket::SSL will emit a large CROAK() if you are silly enough to use them:

- truncate
- stat
- ungetc
- setbuf
- setvbuf
- fdopen
- send/recv

  Note that send() and recv() cannot be reliably trapped by a tied filehandle (such as that used by IO::Socket::SSL) and so may send unencrypted data over the socket. Object-oriented calls to these functions will fail, telling you to use the print/printf/syswrite and read/sysread families instead.

ACCESS TO INTERNALS

`IO::Socket::SSL` uses internally `Net::SSLeay` which provides a lower level API to OpenSSL and provides functions which work directly with OpenSSL SSL and CTX handles. Sometimes it might be necessary to get direct access to these handles from `IO::Socket::SSL` in order to apply functionality available only with `Net::SSLeay`.

For this `_get_ssl_object` can be used on the `IO::Socket::SSL` object to get access to the OpenSSL SSL handle and `_get_ctx_object` can be used on `IO::Socket::SSL` and `IO::Socket::SSL_Context` objects to access the OpenSSL CTX handle. Use very carefully since OpenSSL memory/reference management must be done manually with `Net::SSLeay`.

DEPRECATIONS

The following functions are deprecated and are only retained for compatibility:

- context_init()

  use the SSL_reuse_ctx option if you want to re-use a context

- socketToSSL() and socket_to_SSL()

  use IO::Socket::SSL->start_SSL() instead

- kill_socket()

  use close() instead

- get_peer_certificate()

  use the peer_certificate() function instead. Used to return X509_Certificate with methods subject_name and issuer_name. Now simply returns $self which has these methods (although deprecated).

- issuer_name()

  use peer_certificate('issuer') instead

- subject_name()

  use peer_certificate('subject') instead

EXAMPLES

See the 'example' directory, the tests in 't' and also the tools in 'util'.

BUGS

If you use IO::Socket::SSL together with threads you should load it (e.g. use or require) inside the main thread before creating any other threads which use it. This way it is much faster because it will be initialized only once. Also there are reports that it might crash the other way.

Creating an IO::Socket::SSL object in one thread and closing it in another thread will not work.

IO::Socket::SSL does not work together with Storable::fd_retrieve/fd_store. See BUGS file for more information and how to work around the problem.

If you have a server and it looks like you have a memory leak you might check the size of your session cache. Default for Net::SSLeay seems to be 20480, see the example for SSL_create_ctx_callback for how to limit it.

TLS 1.3 support regarding session reuse is incomplete.

SEE ALSO

IO::Socket::INET, IO::Socket::INET6, IO::Socket::IP, Net::SSLeay.

THANKS

Many thanks to all who added patches or reported bugs or helped IO::Socket::SSL another way. Please keep reporting bugs and help with patches, even if they just fix the documentation.

Special thanks to the team of Net::SSLeay for the good cooperation.

AUTHORS

Steffen Ullrich, <sullr at cpan.org> is the current maintainer.

Peter Behroozi, <behrooz at fas.harvard.edu> (Note the lack of an "i" at the end of "behrooz")

Marko Asplund, <marko.asplund at kronodoc.fi>, was the original author of IO::Socket::SSL.

Patches incorporated from various people, see file Changes.

COPYRIGHT

The original versions of this module are Copyright (C) 1999-2002 Marko Asplund.

The rewrite of this module is Copyright (C) 2002-2005 Peter Behroozi.

Versions 0.98 and newer are Copyright (C) 2006-2014 Steffen Ullrich.

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

POD ERRORS

Hey! **The above document had some coding errors, which are explained below:**

- Around line 972:

  Non-ASCII character seen before =encoding in '—'. Assuming UTF-8

Modules

- `IO::Socket::SSL::Intercept` - SSL interception (man in the middle)
- `IO::Socket::SSL::PublicSuffix` - provide access to Mozilla's list of effective TLD names
- `IO::Socket::SSL::Utils` - loading, storing, creating certificates and keys

IO::Socket::SSL::Intercept

SSL interception (man in the middle)

SYNOPSIS

```
use IO::Socket::SSL::Intercept;
# create interceptor with proxy certificates
my $mitm = IO::Socket::SSL::Intercept->new(
    proxy_cert_file => 'proxy_cert.pem',
    proxy_key_file  => 'proxy_key.pem',
    ...
);
my $listen = IO::Socket::INET->new( LocalAddr => .., Listen => .. );
while (1) {
    # TCP accept new client
    my $client = $listen->accept or next;
    # SSL connect to server
    my $server = IO::Socket::SSL->new(
        PeerAddr => ..,
        SSL_verify_mode => ...,
        ...
    ) or die "ssl connect failed: $!,$SSL_ERROR";
    # clone server certificate
    my ($cert,$key) = $mitm->clone_cert( $server->peer_certificate
↪);
    # and upgrade client side to SSL with cloned certificate
    IO::Socket::SSL->start_SSL($client,
        SSL_server => 1,
        SSL_cert => $cert,
        SSL_key => $key
    ) or die "upgrade failed: $SSL_ERROR";
    # now transfer data between $client and $server and analyze
    # the unencrypted data
    ...
}
```

DESCRIPTION

This module provides functionality to clone certificates and sign them with a proxy certificate, thus making it easy to intercept SSL connections (man in the middle). It also manages a cache of the generated certificates.

How Intercepting SSL Works

Intercepting SSL connections is useful for analyzing encrypted traffic for security reasons or for testing. It does not break the end-to-end security of SSL, e.g. a properly written client will notice the interception unless you explicitly configure the client to trust your interceptor. Intercepting SSL works the following way:

- Create a new CA certificate, which will be used to sign the cloned certificates. This proxy CA certificate should be trusted by the client, or (a properly written client) will throw error messages or deny the connections because it detected a man in the middle attack. Due to the way the interception works there no support for client side certificates is possible.

  Using openssl such a proxy CA certificate and private key can be created with:

```
openssl genrsa -out proxy_key.pem 1024
openssl req -new -x509 -extensions v3_ca -key proxy_key.pem -
 ↪out proxy_cert.pem
# export as PKCS12 for import into browser
openssl pkcs12 -export -in proxy_cert.pem -inkey proxy_key.pem -
 ↪out proxy_cert.p12
```

- Configure client to connect to use intercepting proxy or somehow redirect connections from client to the proxy (e.g. packet filter redirects, ARP or DNS spoofing etc).
- Accept the TCP connection from the client, e.g. don't do any SSL handshakes with the client yet.
- Establish the SSL connection to the server and verify the servers certificate as usually. Then create a new certificate based on the original servers certificate, but signed by your proxy CA. This is the step where IO::Socket::SSL::Intercept helps.
- Upgrade the TCP connection to the client to SSL using the cloned certificate from the server. If the client trusts your proxy CA it will accept the upgrade to SSL.
- Transfer data between client and server. While the connections to client and server are both encrypted with SSL you will read/write the unencrypted data in your proxy application.

METHODS

IO::Socket::SSL::Intercept helps creating the cloned certificate with the following methods:

- **$mitm = IO::Socket::SSL::Intercept->new(%args)**

  This creates a new interceptor object. `%args` should be

  – proxy_cert X509 | proxy_cert_file filename

    This is the proxy certificate. It can be either given by an X509 object from *Net::SSLeay*s internal representation, or using a file in PEM format.

  – proxy_key EVP_PKEY | proxy_key_file filename

    This is the key for the proxy certificate. It can be either given by an EVP_PKEY object from *Net::SSLeay*s internal representation, or using a file in PEM format. The key should not have a passphrase.

  – cert_key EVP_PKEY | cert_key_file filename

    This optional argument specifies the public key used for the cloned certificate. It can be either given by an EVP_PKEY object from *Net::SSLeay*s internal representation, or using a file in PEM format. If not given it will create a new public key on each call of new.

- serial INTEGER|CODE

  This optional argument gives the starting point for the serial numbers of the newly created certificates. If not set the serial number will be created based on the digest of the original certificate. If the value is code it will be called with `serial(original_cert,CERT_asHash(original_cert))` and should return the new serial number.

- cache HASH | SUBROUTINE

  This optional argument gives a way to cache created certificates, so that they don't get recreated on future accesses to the same host. If the argument ist not given an internal HASH ist used.

  If the argument is a hash it will store for each generated certificate a hash reference with `cert` and `atime` in the hash, where `atime` is the time of last access (to expire unused entries) and `cert` is the certificate. Please note, that the certificate is in *Net::SSLeay*s internal X509 format and can thus not be simply dumped and restored. The key for the hash is an `ident` either given to `clone_cert` or generated from the original certificate.

  If the argument is a subroutine it will be called as `$cache->(ident,sub)`. This call should return either an existing (cached) `(cert,key)` or call `sub` without arguments to create a new `(cert,key)`, store it and return it. If called with `$cache->('type')` the function should just return 1 to signal that it supports the current type of cache. If it returns nothing instead the older cache interface is assumed for compatibility reasons.

- **($clone_cert,$key) = $mitm->clone_cert($original_cert,[$ident])**

  This clones the given certificate. An ident as the key into the cache can be given (like `host:port`), if not it will be created from the properties of the original certificate. It returns the cloned certificate and its key (which is the same for alle created certificates).

- **$string = $mitm->serialize**

  This creates a serialized version of the object (e.g. a string) which can then be used to persistently store created certificates over restarts of the application. The cache will only be serialized if it is a HASH. To work together with *Storable* the `STORABLE_freeze` function is defined to call `serialize`.

- **$mitm = IO::Socket::SSL::Intercept->unserialize($string)**

  This restores an Intercept object from a serialized string. To work together with *Storable* the `STORABLE_thaw` function is defined to call `unserialize`.

AUTHOR

Steffen Ullrich

IO::Socket::SSL::PublicSuffix

provide access to Mozilla's list of effective TLD names

SYNOPSIS

```
   # use builtin default
   use IO::Socket::SSL::PublicSuffix;
   $ps = IO::Socket::SSL::PublicSuffix->default;

   # load from string
   $ps = IO::Socket::SSL::PublicSuffix->from_string("*.uk\n*");

   # load from file or file handle
   $ps = IO::Socket::SSL::PublicSuffix->from_file($filename);
   $ps = IO::Socket::SSL::PublicSuffix->from_file(\*STDIN);


   # --- string in -> string out
   # $rest -> whatever.host
   # $tld  -> co.uk
   my ($rest,$tld) = $ps->public_suffix('whatever.host.co.uk');
   my $tld = $ps->public_suffix('whatever.host.co.uk');

   # $root_domain -> host.co.uk
   my $root_domain = $ps->public_suffix('whatever.host.co.uk', 1);

   # --- array in -> array out
   # $rest -> [qw(whatever host)]
   # $tld  -> [qw(co uk)]
   my ($rest,$tld) = $ps->public_suffix([qw(whatever host co uk)]);

----

   # To update this file with the current list:
   perl -MIO::Socket::SSL::PublicSuffix -e
↪'IO::Socket::SSL::PublicSuffix::update_self_from_url()'
```

DESCRIPTION

This module uses the list of effective top level domain names from the mozilla project to determine the public top level domain for a given hostname.

Method

- class->default(%args)

  Returns object with builtin default. `min_suffix` can be given in `%args` to specify the minimal suffix, default is 1.

- class->from_string(string,%args)

  Returns object with configuration from string. See method `default` for `%args`.

- class->from_file(file name | file handle, %args)

Returns object with configuration from file or file handle. See method `default` for `%args`.

- $self->public_suffix($host|\@host, [$add])

  In array context the function returns the non-tld part and the tld part of the given hostname, in scalar context only the tld part. It adds `$add` parts of the non-tld part to the tld, e.g. with `$add=1` it will return the root domain.

  If there were no explicit matches against the public suffix configuration it will fall back to a suffix of length 1.

  The function accepts a string or an array-ref (e.g. host split by `.`). In the first case it will return string(s), in the latter case array-ref(s).

  International hostnames or labels can be in ASCII (IDNA form starting with `xn--`) or unicode. In the latter case an IDNA handling library needs to be available. URI is preferred, but Net::IDN:::Encode, Net::LibIDN are still supported.

- ($self|class)->can_idn

  Returns true if IDN support is available.

FILES

http://publicsuffix.org/list/effective_tld_names.dat

SEE ALSO

Domain::PublicSuffix, Mozilla::PublicSuffix

BUGS

```
Q: Why yet another module, we already have L<Domain::PublicSuffix>␣
↪and
   L<Mozilla::PublicSuffix>.
A: Because the public suffix data change more often than these␣
↪modules do,
   IO::Socket::SSL needs this list and it is more easy this way to␣
↪keep it
   up-to-date.
```

AUTHOR

Steffen Ullrich

IO::Socket::SSL::Utils

loading, storing, creating certificates and keys

SYNOPSIS

```
use IO::Socket::SSL::Utils;

$cert = PEM_file2cert('cert.pem');      # load certificate from file
my $hash = CERT_asHash($cert);          # get details from␣
↪certificate
PEM_cert2file($cert,'cert.pem');        # write certificate to file
CERT_free($cert);                       # free memory within OpenSSL

@certs = PEM_file2certs('chain.pem');   # load multiple certificates␣
↪from file
PEM_certs2file('chain.pem', @certs);    # write multiple certificates␣
↪to file
CERT_free(@certs);                      # free memory for all within␣
↪OpenSSL

my $cert = PEM_string2cert($pem);       # load certificate from PEM␣
↪string
$pem = PEM_cert2string($cert);          # convert certificate to PEM␣
↪string

$key = KEY_create_rsa(2048);            # create new 2048-bit RSA key
PEM_key2file($key,"key.pem");           # and write it to file
KEY_free($key);                         # free memory within OpenSSL
```

DESCRIPTION

This module provides various utility functions to work with certificates and private keys, shielding some of the complexity of the underlying Net::SSLeay and OpenSSL.

FUNCTIONS

- Functions converting between string or file and certificates and keys. They croak if the operation cannot be completed.
  - PEM_file2cert(file) -> cert
  - PEM_cert2file(cert,file)
  - PEM_file2certs(file) -> @certs
  - PEM_certs2file(file,@certs)
  - PEM_string2cert(string) -> cert
  - PEM_cert2string(cert) -> string
  - PEM_file2key(file) -> key
  - PEM_key2file(key,file)
  - PEM_string2key(string) -> key
  - PEM_key2string(key) -> string

- Functions for cleaning up. Each loaded or created cert and key must be freed to not leak memory.
  - CERT_free(@certs)
  - KEY_free(@keys)
- KEY_create_rsa(bits) -> key

  Creates an RSA key pair, bits defaults to 2048.
- KEY_create_ec(curve) -> key

  Creates an EC key, curve defaults to `prime256v1`.
- CERT_asHash(cert,[digest_algo]) -> hash

  Extracts the information from the certificate into a hash and uses the given digest_algo (default: SHA-256) to determine digest of pubkey and cert. The resulting hash contains:
  - subject

    Hash with the parts of the subject, e.g. commonName, countryName, organizationName, stateOrProvinceName, localityName. If there are multiple values for any of these parts the hash value will be an array ref with the values in order instead of just a scalar.
  - subjectAltNames

    Array with list of alternative names. Each entry in the list is of `[type,value]`, where `type` can be OTHERNAME, EMAIL, DNS, X400, DIRNAME, EDIPARTY, URI, IP or RID.
  - issuer

    Hash with the parts of the issuer, e.g. commonName, countryName, organizationName, stateOrProvinceName, localityName. If there are multiple values for any of these parts the hash value will be an array ref with the values in order instead of just a scalar.
  - not_before, not_after

    The time frame, where the certificate is valid, as time_t, e.g. can be converted with localtime or similar functions.
  - serial

    The serial number
  - crl_uri

    List of URIs for CRL distribution.
  - ocsp_uri

    List of URIs for revocation checking using OCSP.
  - keyusage

    List of keyUsage information in the certificate.

- extkeyusage

  List of extended key usage information from the certificate. Each entry in this list consists of a hash with oid, nid, ln and sn.

- pubkey_digest_xxx

  Binary digest of the pubkey using the given digest algorithm, e.g. pubkey_digest_sha256 if (the default) SHA-256 was used.

- x509_digest_xxx

  Binary digest of the X.509 certificate using the given digest algorithm, e.g. x509_digest_sha256 if (the default) SHA-256 was used.

- fingerprint_xxx

  Fingerprint of the certificate using the given digest algorithm, e.g. fingerprint_sha256 if (the default) SHA-256 was used. Contrary to digest_* this is an ASCII string with a list if hexadecimal numbers, e.g. "73:59:75:5C:6D...".

- signature_alg

  Algorithm used to sign certificate, e.g. `sha256WithRSAEncryption`.

- ext

  List of extensions. Each entry in the list is a hash with oid, nid, sn, critical flag (boolean) and data (string representation given by X509V3_EXT_print).

- version

  Certificate version, usually 2 (x509v3)

- CERT_create(hash) -> (cert,key)

  Creates a certificate based on the given hash. If the issuer is not specified the certificate will be self-signed. The following keys can be given:

  - subject

    Hash with the parts of the subject, e.g. commonName, countryName, ... as described in `CERT_asHash`. Default points to IO::Socket::SSL.

  - not_before

    A time_t value when the certificate starts to be valid. Defaults to current time.

  - not_after

    A time_t value when the certificate ends to be valid. Defaults to current time plus one 365 days.

  - serial

    The serial number. If not given a cryptographically random 32-bit value is generated using `Net::SSLeay::RAND_bytes`.

  - version

    The version of the certificate, default 2 (x509v3).

  - CA true|false

    If true declare certificate as CA, defaults to false.

– purpose string|array|hash

Set the purpose of the certificate. The different purposes can be given as a string separated by non-word character, as array or hash. With string or array each purpose can be prefixed with '+' (enable) or '-' (disable) and same can be done with the value when given as a hash. By default enabling the purpose is assumed.

If the CA option is given and true the defaults "ca,sslca,emailca,objca" are assumed, but can be overridden with explicit purpose. If the CA option is given and false the defaults "server,client" are assumed. If no CA option and no purpose is given it defaults to "server,client".

Purpose affects basicConstraints, keyUsage, extKeyUsage and netscapeCertType. The following purposes are defined (case is not important):

```
client
server
email
objsign

CA
sslCA
emailCA
objCA

emailProtection
codeSigning
timeStamping

digitalSignature
nonRepudiation
keyEncipherment
dataEncipherment
keyAgreement
keyCertSign
cRLSign
encipherOnly
decipherOnly
```

Examples:

```
# root-CA for SSL certificates
purpose => 'sslCA'   # or CA => 1

# server certificate and CA (typically self-signed)
purpose => 'sslCA,server'

# client certificate
purpose => 'client',
```

– ext [{ sn => .., data => ... }, ...]

List of extensions. The type of the extension can be specified as name with sn

or as NID with `nid` and the data with `data`. These data must be in the same syntax as expected within openssl.cnf, e.g. something like `OCSP;URI=http://...`. Additionally the critical flag can be set with `critical = 1`>.

– key key

use given key as key for certificate, otherwise a new one will be generated and returned

– issuer_cert cert

set issuer for new certificate

– issuer_key key

sign new certificate with given key

– issuer [cert, key]

Instead of giving issuer_key and issuer_cert as separate arguments they can be given both together.

– digest algorithm

specify the algorithm used to sign the certificate, default SHA-256.

– ignore_invalid_args

ignore any unknown arguments which might be in the argument list (which might be in the arguments for example as result from CERT_asHash)

AUTHOR

Steffen Ullrich

IO::Socket::UNIX

Object interface for AF_UNIX domain sockets

SYNOPSIS

```
use IO::Socket::UNIX;

my $SOCK_PATH = "$ENV{HOME}/unix-domain-socket-test.sock";

# Server:
my $server = IO::Socket::UNIX->new(
    Type => SOCK_STREAM(),
    Local => $SOCK_PATH,
    Listen => 1,
);

my $count = 1;
while (my $conn = $server->accept()) {
    $conn->print("Hello " . ($count++) . "\n");
}
```

(continues on next page)

(continued from previous page)

```
# Client:
my $client = IO::Socket::UNIX->new(
    Type => SOCK_STREAM(),
    Peer => $SOCK_PATH,
);

# Now read and write from $client
```

DESCRIPTION

`IO::Socket::UNIX` provides an object interface to creating and using sockets in the AF_UNIX domain. It is built upon the *IO::Socket* interface and inherits all the methods defined by *IO::Socket*.

CONSTRUCTOR

- new ([ARGS])

  Creates an `IO::Socket::UNIX` object, which is a reference to a newly created symbol (see the *Symbol* package). new optionally takes arguments, these arguments are in key-value pairs.

  In addition to the key-value pairs accepted by *IO::Socket*, `IO::Socket::UNIX` provides.

  ```
  Type        Type of socket (eg SOCK_STREAM or SOCK_DGRAM)
  Local       Path to local fifo
  Peer        Path to peer fifo
  Listen      Queue size for listen
  ```

  If the constructor is only passed a single argument, it is assumed to be a `Peer` specification.

  If the `Listen` argument is given, but false, the queue size will be set to 5.

  If the constructor fails it will return `undef` and set the `$IO::Socket::errstr` package variable to contain an error message.

  ```
  $sock = IO::Socket::UNIX->new(...)
      or die "Cannot create socket - $IO::Socket::errstr\n";
  ```

  For legacy reasons the error message is also set into the global `$@` variable, and you may still find older code which looks here instead.

  ```
  $sock = IO::Socket::UNIX->new(...)
      or die "Cannot create socket - $@\n";
  ```

METHODS

- hostpath()

  Returns the pathname to the fifo at the local end

- peerpath()

  Returns the pathanme to the fifo at the peer end

SEE ALSO

Socket, *IO::Socket*

AUTHOR

Graham Barr. Currently maintained by the Perl Porters. Please report all bugs at https://github.com/Perl/perl5/issues.

COPYRIGHT

Copyright (c) 1996-8 Graham Barr gbarr@pobox.com. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

IO::Tty

Pseudo-terminals and the terminal constants: the compiled part under `IO::Pty` (a pty master and its slave as handles) and `IO::Tty`.

Synopsis

```
use IO::Pty;
my $pty = IO::Pty->new;                 # IO::Pty::pty_allocate
my $slave = $pty->slave;                # IO::Tty::_open_tty
print $pty->ttyname, "\n";              # e.g. /dev/pts/3

use IO::Tty qw(TIOCSWINSZ);             # IO::Tty::Constant
my $ws = IO::Tty::pack_winsize(24, 80);
```

How it works

`IO::Pty::pty_allocate` returns a pty's master and slave descriptors and the slave's name, or an empty list. It tries `posix_openpt`, then `getpt`, `openpty` and `/dev/ptmx`, then the BSD-style `/dev/pty??` names; for each master it grants and unlocks the slave, names it with `ptsname_r` or `ptsname`, opens it, and moves both descriptors above 2. With `$IO::Tty::DEBUG` true it says on stderr what it tries; with `use warnings` (category `io`) each method that fails warns. `IO::Tty::_open_tty` opens a terminal by name, `ttyname` names a handle's terminal, and `pack_winsize`/`unpack_winsize` convert a `struct winsize`. Loading defines the constants of `IO::Tty::Constant`, undef for the ones this system lacks, and `$IO::Tty::CONFIG`.

This is a native implementation of `Tty.xs` of IO-Tty 1.31 as its Makefile.PL configures it on Linux, with the `xssubs.c` it writes; `IO/Tty.pm`, `IO/Pty.pm` and the `IO/Tty/Constant.pm` it writes are used unchanged.

Differences from upstream

One, where perl 5.44 with Tty.xs crashes or reports memory it never wrote (t/81-xs-native/IO-Tty/150-or-better.t):

- `unpack_winsize` of a value with no string - undef, a number, a reference - dies "Bad arg length - got 0, expected 8". perl5 reads the length of a string that is not there: undef segfaults, and the others report a number from memory ("got 94515720394056").

Modules

- `IO::Tty::Constant` - Terminal Constants (autogenerated)

Functions

Pseudo-terminals

`ttyname`

The name of the terminal a handle is open on, or undef.

`pack_winsize`

A `struct winsize` as a string, for the TIOCSWINSZ `ioctl`.

`unpack_winsize`

The four numbers of a `struct winsize` string.

Synopsis

```
my ($rows, $cols, $xpixel, $ypixel) = IO::Tty::unpack_winsize($ws);
```

What you get back

Four numbers; a string that is not eight bytes long dies "IO::Tty::unpack_winsize(): Bad arg length - got N, expected 8".

IO::Tty::Constant

Terminal Constants (autogenerated)

SYNOPSIS

```
use IO::Tty::Constant qw(TIOCNOTTY);
...
```

DESCRIPTION

This package defines constants usually found in <termio.h> or <termios.h> (and their #include hierarchy). Find below an autogenerated alphabetic list of all known constants and whether they are defined on your system (prefixed with '+') and have compilation problems ('o'). Undefined or problematic constants are set to 'undef'.

DEFINED CONSTANTS

- –

  B0
- –

  B110
- –

  B115200
- –

  B1200
- –

  B134
- –

  B150
- –

  B153600
- –

  B1800
- –

  B19200
- –

  B200
- –

  B230400
- –

  B2400
- –

  B300
- –

  B307200

- –
  B38400
- –
  B460800
- –
  B4800
- –
  B50
- –
  B57600
- –
  B600
- –
  B75
- –
  B76800
- –
  B9600
- –
  BRKINT
- –
  BS0
- –
  BS1
- –
  BSDLY
- –
  CBAUD
- –
  CBAUDEXT
- –
  CBRK
- –
  CCTS_OFLOW

- –

  CDEL
- –

  CDSUSP
- –

  CEOF
- –

  CEOL
- –

  CEOL2
- –

  CEOT
- –

  CERASE
- –

  CESC
- –

  CFLUSH
- –

  CIBAUD
- –

  CIBAUDEXT
- –

  CINTR
- –

  CKILL
- –

  CLNEXT
- –

  CLOCAL
- –

  CNSWTCH
- –

  CNUL

- –
  CQUIT
- –
  CR0
- –
  CR1
- –
  CR2
- –
  CR3
- –
  CRDLY
- –
  CREAD
- –
  CRPRNT
- –
  CRTSCTS
- –
  CRTSXOFF
- –
  CRTS_IFLOW
- –
  CS5
- –
  CS6
- –
  CS7
- –
  CS8
- –
  CSIZE
- –
  CSTART

- –
  CSTOP
- –
  CSTOPB
- –
  CSUSP
- –
  CSWTCH
- –
  CWERASE
- –
  DEFECHO
- –
  DIOC
- –
  DIOCGETP
- –
  DIOCSETP
- –
  DOSMODE
- –
  ECHO
- –
  ECHOCTL
- –
  ECHOE
- –
  ECHOK
- –
  ECHOKE
- –
  ECHONL
- –
  ECHOPRT

- –

  EXTA
- –

  EXTB
- –

  FF0
- –

  FF1
- –

  FFDLY
- –

  FIORDCHK
- –

  FLUSHO
- –

  HUPCL
- –

  ICANON
- –

  ICRNL
- –

  IEXTEN
- –

  IGNBRK
- –

  IGNCR
- –

  IGNPAR
- –

  IMAXBEL
- –

  INLCR
- –

  INPCK

- –
  ISIG
- –
  ISTRIP
- –
  IUCLC
- –
  IXANY
- –
  IXOFF
- –
  IXON
- –
  KBENABLED
- –
  LDCHG
- –
  LDCLOSE
- –
  LDDMAP
- –
  LDEMAP
- –
  LDGETT
- –
  LDGMAP
- –
  LDIOC
- –
  LDNMAP
- –
  LDOPEN
- –
  LDSETT

- –
  LDSMAP
- –
  LOBLK
- –
  NCCS
- –
  NL0
- –
  NL1
- –
  NLDLY
- –
  NOFLSH
- –
  OCRNL
- –
  OFDEL
- –
  OFILL
- –
  OLCUC
- –
  ONLCR
- –
  ONLRET
- –
  ONOCR
- –
  OPOST
- –
  PAGEOUT
- –
  PARENB

- –
  PAREXT
- –
  PARMRK
- –
  PARODD
- –
  PENDIN
- –
  RCV1EN
- –
  RTS_TOG
- –
  TAB0
- –
  TAB1
- –
  TAB2
- –
  TAB3
- –
  TABDLY
- –
  TCDSET
- –
  TCFLSH
- –
  TCGETA
- –
  TCGETS
- –
  TCIFLUSH
- –
  TCIOFF

- –
  TCIOFLUSH
- –
  TCION
- –
  TCOFLUSH
- –
  TCOOFF
- –
  TCOON
- –
  TCSADRAIN
- –
  TCSAFLUSH
- –
  TCSANOW
- –
  TCSBRK
- –
  TCSETA
- –
  TCSETAF
- –
  TCSETAW
- –
  TCSETCTTY
- –
  TCSETS
- –
  TCSETSF
- –
  TCSETSW
- –
  TCXONC

- –
  TERM_D40
- –
  TERM_D42
- –
  TERM_H45
- –
  TERM_NONE
- –
  TERM_TEC
- –
  TERM_TEX
- –
  TERM_V10
- –
  TERM_V61
- –
  TIOCCBRK
- –
  TIOCCDTR
- –
  TIOCCONS
- –
  TIOCEXCL
- –
  TIOCFLUSH
- –
  TIOCGETD
- –
  TIOCGETC
- –
  TIOCGETP
- –
  TIOCGLTC

- –

  TIOCSETC
- –

  TIOCSETN
- –

  TIOCSETP
- –

  TIOCSLTC
- –

  TIOCGPGRP
- –

  TIOCGSID
- –

  TIOCGSOFTCAR
- –

  TIOCGWINSZ
- –

  TIOCHPCL
- –

  TIOCKBOF
- –

  TIOCKBON
- –

  TIOCLBIC
- –

  TIOCLBIS
- –

  TIOCLGET
- –

  TIOCLSET
- –

  TIOCMBIC
- –

  TIOCMBIS

- –
  TIOCMGET
- –
  TIOCMSET
- –
  TIOCM_CAR
- –
  TIOCM_CD
- –
  TIOCM_CTS
- –
  TIOCM_DSR
- –
  TIOCM_DTR
- –
  TIOCM_LE
- –
  TIOCM_RI
- –
  TIOCM_RNG
- –
  TIOCM_RTS
- –
  TIOCM_SR
- –
  TIOCM_ST
- –
  TIOCNOTTY
- –
  TIOCNXCL
- –
  TIOCOUTQ
- –
  TIOCREMOTE

- –

  TIOCSBRK
- –

  TIOCSCTTY
- –

  TIOCSDTR
- –

  TIOCSETD
- –

  TIOCSIGNAL
- –

  TIOCSPGRP
- –

  TIOCSSID
- –

  TIOCSSOFTCAR
- –

  TIOCSTART
- –

  TIOCSTI
- –

  TIOCSTOP
- –

  TIOCSWINSZ
- –

  TM_ANL
- –

  TM_CECHO
- –

  TM_CINVIS
- –

  TM_LCF
- –

  TM_NONE

- –
  TM_SET
- –
  TM_SNL
- –
  TOSTOP
- –
  VCEOF
- –
  VCEOL
- –
  VDISCARD
- –
  VDSUSP
- –
  VEOF
- –
  VEOL
- –
  VEOL2
- –
  VERASE
- –
  VINTR
- –
  VKILL
- –
  VLNEXT
- –
  VMIN
- –
  VQUIT
- –
  VREPRINT

- –

  VSTART
- –

  VSTOP
- –

  VSUSP
- –

  VSWTCH
- –

  VT0
- –

  VT1
- –

  VTDLY
- –

  VTIME
- –

  VWERASE
- –

  WRAP
- –

  XCASE
- –

  XCLUDE
- –

  XMT1EN
- –

  XTABS

FOR MORE INFO SEE

IO::Tty

ttyname

The name of the terminal a handle is open on, or undef.

Synopsis

```
my $name = IO::Tty::ttyname(\*STDIN);    # e.g. /dev/pts/3
```

What you get back

The name, or undef with `$!` set when the handle is not open on a terminal (or not open at all, EINVAL). Something that is not a handle dies “Bad filehandle”.

pack_winsize

A `struct winsize` as a string, for the TIOCSWINSZ `ioctl`.

Synopsis

```
my $ws = IO::Tty::pack_winsize($rows, $cols, $xpixel, $ypixel);
ioctl($tty, IO::Tty::Constant::TIOCSWINSZ(), $ws);
```

What you get back

Eight bytes, four unsigned shorts in native order; each number is taken modulo 65536, and the pixel sizes default to 0.

IO::Uncompress

Namespace containing the following modules:

- `IO::Uncompress::AnyInflate` - Uncompress zlib-based (zip, gzip) file/buffer
- `IO::Uncompress::AnyUncompress` - Uncompress gzip, zip, bzip2, zstd, xz, lzma, lzip, lzf or lzop file/buffer
- `IO::Uncompress::Base` - Base Class for IO::Uncompress modules
- `IO::Uncompress::Bunzip2` - Read bzip2 files/buffers
- `IO::Uncompress::Gunzip` - Read RFC 1952 files/buffers
- `IO::Uncompress::Inflate` - Read RFC 1950 files/buffers
- `IO::Uncompress::RawInflate` - Read RFC 1951 files/buffers
- `IO::Uncompress::Unzip` - Read zip files/buffers

IO::Uncompress::AnyInflate

Uncompress zlib-based (zip, gzip) file/buffer

SYNOPSIS

```
use IO::Uncompress::AnyInflate qw(anyinflate $AnyInflateError) ;

my $status = anyinflate $input => $output [,OPTS]
    or die "anyinflate failed: $AnyInflateError\n";

my $z = IO::Uncompress::AnyInflate->new( $input [OPTS] )
    or die "anyinflate failed: $AnyInflateError\n";

$status = $z->read($buffer)
$status = $z->read($buffer, $length)
$status = $z->read($buffer, $length, $offset)
$line = $z->getline()
$char = $z->getc()
$char = $z->ungetc()
$char = $z->opened()

$status = $z->inflateSync()

$data = $z->trailingData()
$status = $z->nextStream()
$data = $z->getHeaderInfo()
$z->tell()
$z->seek($position, $whence)
$z->binmode()
$z->fileno()
$z->eof()
$z->close()

$AnyInflateError ;

# IO::File mode

<$z>
read($z, $buffer);
read($z, $buffer, $length);
read($z, $buffer, $length, $offset);
tell($z)
seek($z, $position, $whence)
binmode($z)
fileno($z)
eof($z)
close($z)
```

DESCRIPTION

This module provides a Perl interface that allows the reading of files/buffers that have been compressed in a number of formats that use the zlib compression library.

The formats supported are

- RFC 1950
- RFC 1951 (optionally)
- gzip (RFC 1952)
- zip

The module will auto-detect which, if any, of the supported compression formats is being used.

Functional Interface

A top-level function, `anyinflate`, is provided to carry out "one-shot" uncompression between buffers and/or files. For finer control over the uncompression process, see the "OO Interface" section.

```
use IO::Uncompress::AnyInflate qw(anyinflate $AnyInflateError) ;

anyinflate $input_filename_or_reference => $output_filename_or_
 ↪reference [,OPTS]
    or die "anyinflate failed: $AnyInflateError\n";
```

The functional interface needs Perl5.005 or better.

anyinflate $input_filename_or_reference => $output_filename_or_reference [, OPTS]

`anyinflate` expects at least two parameters, `$input_filename_or_reference` and `$output_filename_or_reference` and zero or more optional parameters (see "Optional Parameters")

The `$input_filename_or_reference` parameter

The parameter, `$input_filename_or_reference`, is used to define the source of the compressed data.

It can take one of the following forms:

- A filename

  If the `$input_filename_or_reference` parameter is a simple scalar, it is assumed to be a filename. This file will be opened for reading and the input data will be read from it.

- A filehandle

  If the `$input_filename_or_reference` parameter is a filehandle, the input data will be read from it. The string '-' can be used as an alias for standard input.

- A scalar reference

  If `$input_filename_or_reference` is a scalar reference, the input data will be read from `$$input_filename_or_reference`.

- An array reference

  If `$input_filename_or_reference` is an array reference, each element in the array must be a filename.

  The input data will be read from each file in turn.

  The complete array will be walked to ensure that it only contains valid filenames before any data is uncompressed.

- An Input FileGlob string

  If `$input_filename_or_reference` is a string that is delimited by the characters "<" and ">" `anyinflate` will assume that it is an *input fileglob string*. The input is the list of files that match the fileglob.

  See *File::GlobMapper* for more details.

If the `$input_filename_or_reference` parameter is any other type, `undef` will be returned.

The `$output_filename_or_reference` parameter

The parameter `$output_filename_or_reference` is used to control the destination of the uncompressed data. This parameter can take one of these forms.

- A filename

  If the `$output_filename_or_reference` parameter is a simple scalar, it is assumed to be a filename. This file will be opened for writing and the uncompressed data will be written to it.

- A filehandle

  If the `$output_filename_or_reference` parameter is a filehandle, the uncompressed data will be written to it. The string '-' can be used as an alias for standard output.

- A scalar reference

  If `$output_filename_or_reference` is a scalar reference, the uncompressed data will be stored in `$$output_filename_or_reference`.

- An Array Reference

  If `$output_filename_or_reference` is an array reference, the uncompressed data will be pushed onto the array.

- An Output FileGlob

  If `$output_filename_or_reference` is a string that is delimited by the characters "<" and ">" `anyinflate` will assume that it is an *output fileglob string*. The output is the list of files that match the fileglob.

  When `$output_filename_or_reference` is an fileglob string, `$input_filename_or_reference` must also be a fileglob string. Anything else is an error.

See *File::GlobMapper* for more details.

If the `$output_filename_or_reference` parameter is any other type, `undef` will be returned.

Notes

When `$input_filename_or_reference` maps to multiple compressed files/buffers and `$output_filename_or_reference` is a single file/buffer, after uncompression `$output_filename_or_reference` will contain a concatenation of all the uncompressed data from each of the input files/buffers.

Optional Parameters

The optional parameters for the one-shot function `anyinflate` are (for the most part) identical to those used with the OO interface defined in the "Constructor Options" section. The exceptions are listed below

- `AutoClose => 0|1`

  This option applies to any input or output data streams to `anyinflate` that are filehandles.

  If `AutoClose` is specified, and the value is true, it will result in all input and/or output filehandles being closed once `anyinflate` has completed.

  This parameter defaults to 0.

- `BinModeOut => 0|1`

  This option is now a no-op. All files will be written in binmode.

- `Append => 0|1`

  The behaviour of this option is dependent on the type of output data stream.

  - A Buffer

    If Append is enabled, all uncompressed data will be append to the end of the output buffer. Otherwise the output buffer will be cleared before any uncompressed data is written to it.

  - A Filename

    If Append is enabled, the file will be opened in append mode. Otherwise the contents of the file, if any, will be truncated before any uncompressed data is written to it.

  - A Filehandle

    If Append is enabled, the filehandle will be positioned to the end of the file via a call to `seek` before any uncompressed data is written to it. Otherwise the file pointer will not be moved.

  When Append is specified, and set to true, it will *append* all uncompressed data to the output data stream.

  So when the output is a filehandle it will carry out a seek to the eof before writing any uncompressed data. If the output is a filename, it will be opened for appending.

If the output is a buffer, all uncompressed data will be appended to the existing buffer.

Conversely when Append is not specified, or it is present and is set to false, it will operate as follows.

When the output is a filename, it will truncate the contents of the file before writing any uncompressed data. If the output is a filehandle its position will not be changed. If the output is a buffer, it will be wiped before any uncompressed data is output.

Defaults to 0.

- `MultiStream => 0|1`

  If the input file/buffer contains multiple compressed data streams, this option will uncompress the whole lot as a single data stream.

  Defaults to 0.

- `TrailingData => $scalar`

  Returns the data, if any, that is present immediately after the compressed data stream once uncompression is complete.

  This option can be used when there is useful information immediately following the compressed data stream, and you don't know the length of the compressed data stream.

  If the input is a buffer, `trailingData` will return everything from the end of the compressed data stream to the end of the buffer.

  If the input is a filehandle, `trailingData` will return the data that is left in the filehandle input buffer once the end of the compressed data stream has been reached. You can then use the filehandle to read the rest of the input file.

  Don't bother using `trailingData` if the input is a filename.

  If you know the length of the compressed data stream before you start uncompressing, you can avoid having to use `trailingData` by setting the `InputLength` option.

OneShot Examples

To read the contents of the file `file1.txt.Compressed` and write the uncompressed data to the file `file1.txt`.

```
use strict ;
use warnings ;
use IO::Uncompress::AnyInflate qw(anyinflate $AnyInflateError) ;

my $input = "file1.txt.Compressed";
my $output = "file1.txt";
anyinflate $input => $output
    or die "anyinflate failed: $AnyInflateError\n";
```

To read from an existing Perl filehandle, `$input`, and write the uncompressed data to a buffer, `$buffer`.

```
use strict ;
use warnings ;
use IO::Uncompress::AnyInflate qw(anyinflate $AnyInflateError) ;
use IO::File ;

my $input = IO::File->new( "<file1.txt.Compressed" )
    or die "Cannot open 'file1.txt.Compressed': $!\n" ;
my $buffer ;
anyinflate $input => \$buffer
    or die "anyinflate failed: $AnyInflateError\n";
```

To uncompress all files in the directory "/my/home" that match "*.txt.Compressed" and store the compressed data in the same directory

```
use strict ;
use warnings ;
use IO::Uncompress::AnyInflate qw(anyinflate $AnyInflateError) ;

anyinflate '</my/home/*.txt.Compressed>' => '</my/home/#1.txt>'
    or die "anyinflate failed: $AnyInflateError\n";
```

and if you want to compress each file one at a time, this will do the trick

```
use strict ;
use warnings ;
use IO::Uncompress::AnyInflate qw(anyinflate $AnyInflateError) ;

for my $input ( glob "/my/home/*.txt.Compressed" )
{
    my $output = $input;
    $output =~ s/.Compressed// ;
    anyinflate $input => $output
        or die "Error compressing '$input': $AnyInflateError\n";
}
```

OO Interface

Constructor

The format of the constructor for IO::Uncompress::AnyInflate is shown below

```
my $z = IO::Uncompress::AnyInflate->new( $input [OPTS] )
    or die "IO::Uncompress::AnyInflate failed: $AnyInflateError\n";
```

The constructor takes one mandatory parameter, `$input`, defined below, and zero or more `OPTS`, defined in "Constructor Options".

Returns an `IO::Uncompress::AnyInflate` object on success and undef on failure. The variable `$AnyInflateError` will contain an error message on failure.

If you are running Perl 5.005 or better the object, `$z`, returned from IO::Uncompress::AnyInflate can be used exactly like an *IO::File* filehandle. This means that all

normal input file operations can be carried out with `$z`. For example, to read a line from a compressed file/buffer you can use either of these forms

```
$line = $z->getline();
$line = <$z>;
```

Below is a simple exaple of using the OO interface to read the compressed file `myfile.Compressed` and write its contents to stdout.

```
my $filename = "myfile.Compressed";
my $z = IO::Uncompress::AnyInflate->new($filename)
    or die "IO::Uncompress::AnyInflate failed: $AnyInflateError\n";

while (<$z>) {
    print $_;
}
$z->close();
```

See “EXAMPLES” for further examples

The mandatory parameter `$input` is used to determine the source of the compressed data. This parameter can take one of three forms.

- A filename

  If the `$input` parameter is a scalar, it is assumed to be a filename. This file will be opened for reading and the compressed data will be read from it.

- A filehandle

  If the `$input` parameter is a filehandle, the compressed data will be read from it. The string ‘-’ can be used as an alias for standard input.

- A scalar reference

  If `$input` is a scalar reference, the compressed data will be read from `$$input`.

Constructor Options

The option names defined below are case insensitive and can be optionally prefixed by a ‘-’. So all of the following are valid

```
-AutoClose
-autoclose
AUTOCLOSE
autoclose
```

OPTS is a combination of the following options:

- `AutoClose => 0|1`

  This option is only valid when the `$input` parameter is a filehandle. If specified, and the value is true, it will result in the file being closed once either the `close` method is called or the IO::Uncompress::AnyInflate object is destroyed.

  This parameter defaults to 0.

- `MultiStream => 0|1`

  Allows multiple concatenated compressed streams to be treated as a single compressed stream. Decompression will stop once either the end of the file/buffer is reached, an error is encountered (premature eof, corrupt compressed data) or the end of a stream is not immediately followed by the start of another stream.

  This parameter defaults to 0.

- `Prime => $string`

  This option will uncompress the contents of `$string` before processing the input file/buffer.

  This option can be useful when the compressed data is embedded in another file/data structure and it is not possible to work out where the compressed data begins without having to read the first few bytes. If this is the case, the uncompression can be *primed* with these bytes using this option.

- `Transparent => 0|1`

  If this option is set and the input file/buffer is not compressed data, the module will allow reading of it anyway.

  In addition, if the input file/buffer does contain compressed data and there is non-compressed data immediately following it, setting this option will make this module treat the whole file/buffer as a single data stream.

  This option defaults to 1.

- `BlockSize => $num`

  When reading the compressed input data, IO::Uncompress::AnyInflate will read it in blocks of `$num` bytes.

  This option defaults to 4096.

- `InputLength => $size`

  When present this option will limit the number of compressed bytes read from the input file/buffer to `$size`. This option can be used in the situation where there is useful data directly after the compressed data stream and you know beforehand the exact length of the compressed data stream.

  This option is mostly used when reading from a filehandle, in which case the file pointer will be left pointing to the first byte directly after the compressed data stream.

  This option defaults to off.

- `Append => 0|1`

  This option controls what the `read` method does with uncompressed data.

  If set to 1, all uncompressed data will be appended to the output parameter of the `read` method.

  If set to 0, the contents of the output parameter of the `read` method will be overwritten by the uncompressed data.

  Defaults to 0.

- `Strict => 0|1`

  This option controls whether the extra checks defined below are used when carrying out the decompression. When Strict is on, the extra tests are carried out, when Strict is off they are not.

  The default for this option is off.

  If the input is an RFC 1950 data stream, the following will be checked:

  1. The ADLER32 checksum field must be present.
  2. The value of the ADLER32 field read must match the adler32 value of the uncompressed data actually contained in the file.

  If the input is a gzip (RFC 1952) data stream, the following will be checked:

  1. If the FHCRC bit is set in the gzip FLG header byte, the CRC16 bytes in the header must match the crc16 value of the gzip header actually read.
  2. If the gzip header contains a name field (FNAME) it consists solely of ISO 8859-1 characters.
  3. If the gzip header contains a comment field (FCOMMENT) it consists solely of ISO 8859-1 characters plus line-feed.
  4. If the gzip FEXTRA header field is present it must conform to the sub-field structure as defined in RFC 1952.
  5. The CRC32 and ISIZE trailer fields must be present.
  6. The value of the CRC32 field read must match the crc32 value of the uncompressed data actually contained in the gzip file.
  7. The value of the ISIZE fields read must match the length of the uncompressed data actually read from the file.

- `RawInflate => 0|1`

  When auto-detecting the compressed format, try to test for raw-deflate (RFC 1951) content using the `IO::Uncompress::RawInflate` module.

  The reason this is not default behaviour is because RFC 1951 content can only be detected by attempting to uncompress it. This process is error prone and can result is false positives.

  Defaults to 0.

- `ParseExtra => 0|1` If the gzip FEXTRA header field is present and this option is set, it will force the module to check that it conforms to the sub-field structure as defined in RFC 1952.

  If the `Strict` is on it will automatically enable this option.

  Defaults to 0.

Methods

read

Usage is

```
$status = $z->read($buffer)
```

Reads a block of compressed data (the size of the compressed block is determined by the `Buffer` option in the constructor), uncompresses it and writes any uncompressed data into `$buffer`. If the `Append` parameter is set in the constructor, the uncompressed data will be appended to the `$buffer` parameter. Otherwise `$buffer` will be overwritten.

Returns the number of uncompressed bytes written to `$buffer`, zero if eof or a negative number on error.

read

Usage is

```
$status = $z->read($buffer, $length)
$status = $z->read($buffer, $length, $offset)

$status = read($z, $buffer, $length)
$status = read($z, $buffer, $length, $offset)
```

Attempt to read `$length` bytes of uncompressed data into `$buffer`.

The main difference between this form of the `read` method and the previous one, is that this one will attempt to return *exactly* `$length` bytes. The only circumstances that this function will not is if end-of-file or an IO error is encountered.

Returns the number of uncompressed bytes written to `$buffer`, zero if eof or a negative number on error.

getline

Usage is

```
$line = $z->getline()
$line = <$z>
```

Reads a single line.

This method fully supports the use of the variable `$/` (or `$INPUT_RECORD_SEPARATOR` or `$RS` when `English` is in use) to determine what constitutes an end of line. Paragraph mode, record mode and file slurp mode are all supported.

getc

Usage is

```
$char = $z->getc()
```

Read a single character.

ungetc

Usage is

```
$char = $z->ungetc($string)
```

inflateSync

Usage is

```
$status = $z->inflateSync()
```

TODO

getHeaderInfo

Usage is

```
$hdr  = $z->getHeaderInfo();
@hdrs = $z->getHeaderInfo();
```

This method returns either a hash reference (in scalar context) or a list or hash references (in array context) that contains information about each of the header fields in the compressed data stream(s).

tell

Usage is

```
$z->tell()
tell $z
```

Returns the uncompressed file offset.

eof

Usage is

```
$z->eof();
eof($z);
```

Returns true if the end of the compressed input stream has been reached.

seek

```
$z->seek($position, $whence);
seek($z, $position, $whence);
```

Provides a sub-set of the `seek` functionality, with the restriction that it is only legal to seek forward in the input file/buffer. It is a fatal error to attempt to seek backward.

Note that the implementation of `seek` in this module does not provide true random access to a compressed file/buffer. It works by uncompressing data from the current offset in the file/buffer until it reaches the uncompressed offset specified in the parameters to `seek`. For very small files this may be acceptable behaviour. For large files it may cause an unacceptable delay.

The `$whence` parameter takes one the usual values, namely SEEK_SET, SEEK_CUR or SEEK_END.

Returns 1 on success, 0 on failure.

binmode

Usage is

```
$z->binmode
binmode $z ;
```

This is a noop provided for completeness.

opened

```
$z->opened()
```

Returns true if the object currently refers to a opened file/buffer.

autoflush

```
my $prev = $z->autoflush()
my $prev = $z->autoflush(EXPR)
```

If the `$z` object is associated with a file or a filehandle, this method returns the current autoflush setting for the underlying filehandle. If `EXPR` is present, and is non-zero, it will enable flushing after every write/print operation.

If `$z` is associated with a buffer, this method has no effect and always returns `undef`.

Note that the special variable `$|` **cannot** be used to set or retrieve the autoflush setting.

input_line_number

```
$z->input_line_number()
$z->input_line_number(EXPR)
```

Returns the current uncompressed line number. If `EXPR` is present it has the effect of setting the line number. Note that setting the line number does not change the current position within the file/buffer being read.

The contents of `$/` are used to determine what constitutes a line terminator.

fileno

```
$z->fileno()
fileno($z)
```

If the `$z` object is associated with a file or a filehandle, `fileno` will return the underlying file descriptor. Once the `close` method is called `fileno` will return `undef`.

If the `$z` object is associated with a buffer, this method will return `undef`.

close

```
$z->close() ;
close $z ;
```

Closes the output file/buffer.

For most versions of Perl this method will be automatically invoked if the IO::Uncompress::AnyInflate object is destroyed (either explicitly or by the variable with the reference to the object going out of scope). The exceptions are Perl versions 5.005 through 5.00504 and 5.8.0. In these cases, the `close` method will be called automatically, but not until global destruction of all live objects when the program is terminating.

Therefore, if you want your scripts to be able to run on all versions of Perl, you should call `close` explicitly and not rely on automatic closing.

Returns true on success, otherwise 0.

If the `AutoClose` option has been enabled when the IO::Uncompress::AnyInflate object was created, and the object is associated with a file, the underlying file will also be closed.

nextStream

Usage is

```
my $status = $z->nextStream();
```

Skips to the next compressed data stream in the input file/buffer. If a new compressed data stream is found, the eof marker will be cleared and `$.` will be reset to 0.

Returns 1 if a new stream was found, 0 if none was found, and -1 if an error was encountered.

trailingData

Usage is

```
my $data = $z->trailingData();
```

Returns the data, if any, that is present immediately after the compressed data stream once uncompression is complete. It only makes sense to call this method once the end of the compressed data stream has been encountered.

This option can be used when there is useful information immediately following the compressed data stream, and you don't know the length of the compressed data stream.

If the input is a buffer, `trailingData` will return everything from the end of the compressed data stream to the end of the buffer.

If the input is a filehandle, `trailingData` will return the data that is left in the filehandle input buffer once the end of the compressed data stream has been reached. You can then use the filehandle to read the rest of the input file.

Don't bother using `trailingData` if the input is a filename.

If you know the length of the compressed data stream before you start uncompressing, you can avoid having to use `trailingData` by setting the `InputLength` option in the constructor.

Importing

No symbolic constants are required by IO::Uncompress::AnyInflate at present.

- :all

  Imports `anyinflate` and `$AnyInflateError`. Same as doing this

```
use IO::Uncompress::AnyInflate qw(anyinflate $AnyInflateError) ;
```

EXAMPLES

Working with Net::FTP

See IO::Compress::FAQ

SUPPORT

General feedback/questions/bug reports should be sent to https://github.com/pmqs/IO-Compress/issues (preferred) or https://rt.cpan.org/Public/Dist/Display.html?Name=IO-Compress.

SEE ALSO

Compress::Zlib, *IO::Compress::Gzip*, *IO::Uncompress::Gunzip*, *IO::Compress::Deflate*, *IO::Uncompress::Inflate*, *IO::Compress::RawDeflate*, *IO::Uncompress::RawInflate*, *IO::Compress::Bzip2*, *IO::Uncompress::Bunzip2*, IO::Compress::Lzma, IO::Uncompress::UnLzma, IO::Compress::Xz, IO::Uncompress::UnXz, IO::Compress::Lzip, IO::Uncompress::UnLzip, IO::Compress::Lzop, IO::Uncompress::UnLzop, IO::Compress::Lzf, IO::Uncompress::UnLzf, IO::Compress::Zstd, IO::Uncompress::UnZstd, *IO::Uncompress::AnyUncompress*

IO::Compress::FAQ

File::GlobMapper, Archive::Zip, *Archive::Tar*, *IO::Zlib*

For RFC 1950, 1951 and 1952 see https://datatracker.ietf.org/doc/html/rfc1950, https://datatracker.ietf.org/doc/html/rfc1951 and https://datatracker.ietf.org/doc/html/rfc1952

The *zlib* compression library was written by Jean-loup Gailly `gzip@prep.ai.mit.edu` and Mark Adler `madler@alumni.caltech.edu`.

The primary site for the *zlib* compression library is http://www.zlib.org.

The primary site for the *zlib-ng* compression library is https://github.com/zlib-ng/zlib-ng.

The primary site for gzip is http://www.gzip.org.

AUTHOR

This module was written by Paul Marquess, `pmqs@cpan.org`.

MODIFICATION HISTORY

See the Changes file.

COPYRIGHT AND LICENSE

Copyright (c) 2005-2026 Paul Marquess. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

IO::Uncompress::AnyUncompress

Uncompress gzip, zip, bzip2, zstd, xz, lzma, lzip, lzf or lzop file/buffer

SYNOPSIS

```
use IO::Uncompress::AnyUncompress qw(anyuncompress
 ↪$AnyUncompressError) ;

my $status = anyuncompress $input => $output [,OPTS]
    or die "anyuncompress failed: $AnyUncompressError\n";

my $z = IO::Uncompress::AnyUncompress->new( $input [OPTS] )
    or die "anyuncompress failed: $AnyUncompressError\n";

$status = $z->read($buffer)
$status = $z->read($buffer, $length)
$status = $z->read($buffer, $length, $offset)
$line = $z->getline()
$char = $z->getc()
$char = $z->ungetc()
$char = $z->opened()

$data = $z->trailingData()
$status = $z->nextStream()
$data = $z->getHeaderInfo()
$z->tell()
$z->seek($position, $whence)
$z->binmode()
$z->fileno()
```

(continues on next page)

(continued from previous page)

```
$z->eof()
$z->close()

$AnyUncompressError ;

# IO::File mode

<$z>
read($z, $buffer);
read($z, $buffer, $length);
read($z, $buffer, $length, $offset);
tell($z)
seek($z, $position, $whence)
binmode($z)
fileno($z)
eof($z)
close($z)
```

DESCRIPTION

This module provides a Perl interface that allows the reading of files/buffers that have been compressed with a variety of compression libraries.

The formats supported are:

- RFC 1950
- RFC 1951 (optionally)
- gzip (RFC 1952)
- zip
- zstd (Zstandard)
- bzip2
- lzop
- lzf
- lzma
- lzip
- xz

The module will auto-detect which, if any, of the supported compression formats is being used.

Functional Interface

A top-level function, `anyuncompress`, is provided to carry out "one-shot" uncompression between buffers and/or files. For finer control over the uncompression process, see the "OO Interface" section.

```
use IO::Uncompress::AnyUncompress qw(anyuncompress
 ↪$AnyUncompressError) ;

anyuncompress $input_filename_or_reference => $output_filename_or_
 ↪reference [,OPTS]
    or die "anyuncompress failed: $AnyUncompressError\n";
```

The functional interface needs Perl5.005 or better.

anyuncompress $input_filename_or_reference => $output_filename_or_reference [, OPTS]

anyuncompress expects at least two parameters, `$input_filename_or_reference` and `$output_filename_or_reference` and zero or more optional parameters (see "Optional Parameters")

The `$input_filename_or_reference` parameter

The parameter, `$input_filename_or_reference`, is used to define the source of the compressed data.

It can take one of the following forms:

- A filename

  If the `$input_filename_or_reference` parameter is a simple scalar, it is assumed to be a filename. This file will be opened for reading and the input data will be read from it.

- A filehandle

  If the `$input_filename_or_reference` parameter is a filehandle, the input data will be read from it. The string '-' can be used as an alias for standard input.

- A scalar reference

  If `$input_filename_or_reference` is a scalar reference, the input data will be read from `$$input_filename_or_reference`.

- An array reference

  If `$input_filename_or_reference` is an array reference, each element in the array must be a filename.

  The input data will be read from each file in turn.

  The complete array will be walked to ensure that it only contains valid filenames before any data is uncompressed.

- An Input FileGlob string

  If `$input_filename_or_reference` is a string that is delimited by the characters "<" and ">" anyuncompress will assume that it is an *input fileglob string*. The input is the list of files that match the fileglob.

  See *File::GlobMapper* for more details.

If the `$input_filename_or_reference` parameter is any other type, `undef` will be returned.

The `$output_filename_or_reference` parameter

The parameter `$output_filename_or_reference` is used to control the destination of the uncompressed data. This parameter can take one of these forms.

- A filename

  If the `$output_filename_or_reference` parameter is a simple scalar, it is assumed to be a filename. This file will be opened for writing and the uncompressed data will be written to it.

- A filehandle

  If the `$output_filename_or_reference` parameter is a filehandle, the uncompressed data will be written to it. The string '-' can be used as an alias for standard output.

- A scalar reference

  If `$output_filename_or_reference` is a scalar reference, the uncompressed data will be stored in `$$output_filename_or_reference`.

- An Array Reference

  If `$output_filename_or_reference` is an array reference, the uncompressed data will be pushed onto the array.

- An Output FileGlob

  If `$output_filename_or_reference` is a string that is delimited by the characters "<" and ">" `anyuncompress` will assume that it is an *output fileglob string*. The output is the list of files that match the fileglob.

  When `$output_filename_or_reference` is an fileglob string, `$input_filename_or_reference` must also be a fileglob string. Anything else is an error.

  See *File::GlobMapper* for more details.

If the `$output_filename_or_reference` parameter is any other type, `undef` will be returned.

Notes

When `$input_filename_or_reference` maps to multiple compressed files/buffers and `$output_filename_or_reference` is a single file/buffer, after uncompression `$output_filename_or_reference` will contain a concatenation of all the uncompressed data from each of the input files/buffers.

Optional Parameters

The optional parameters for the one-shot function `anyuncompress` are (for the most part) identical to those used with the OO interface defined in the "Constructor Options" section. The exceptions are listed below

- `AutoClose => 0|1`

  This option applies to any input or output data streams to anyuncompress that are filehandles.

  If `AutoClose` is specified, and the value is true, it will result in all input and/or output filehandles being closed once anyuncompress has completed.

  This parameter defaults to 0.

- `BinModeOut => 0|1`

  This option is now a no-op. All files will be written in binmode.

- `Append => 0|1`

  The behaviour of this option is dependent on the type of output data stream.

  - A Buffer

    If Append is enabled, all uncompressed data will be append to the end of the output buffer. Otherwise the output buffer will be cleared before any uncompressed data is written to it.

  - A Filename

    If Append is enabled, the file will be opened in append mode. Otherwise the contents of the file, if any, will be truncated before any uncompressed data is written to it.

  - A Filehandle

    If Append is enabled, the filehandle will be positioned to the end of the file via a call to `seek` before any uncompressed data is written to it. Otherwise the file pointer will not be moved.

  When Append is specified, and set to true, it will *append* all uncompressed data to the output data stream.

  So when the output is a filehandle it will carry out a seek to the eof before writing any uncompressed data. If the output is a filename, it will be opened for appending. If the output is a buffer, all uncompressed data will be appended to the existing buffer.

  Conversely when Append is not specified, or it is present and is set to false, it will operate as follows.

  When the output is a filename, it will truncate the contents of the file before writing any uncompressed data. If the output is a filehandle its position will not be changed. If the output is a buffer, it will be wiped before any uncompressed data is output.

  Defaults to 0.

- `MultiStream => 0|1`

  If the input file/buffer contains multiple compressed data streams, this option will uncompress the whole lot as a single data stream.

  Defaults to 0.

- `TrailingData => $scalar`

  Returns the data, if any, that is present immediately after the compressed data stream once uncompression is complete.

  This option can be used when there is useful information immediately following the compressed data stream, and you don't know the length of the compressed data stream.

  If the input is a buffer, `trailingData` will return everything from the end of the compressed data stream to the end of the buffer.

  If the input is a filehandle, `trailingData` will return the data that is left in the filehandle input buffer once the end of the compressed data stream has been reached. You can then use the filehandle to read the rest of the input file.

  Don't bother using `trailingData` if the input is a filename.

  If you know the length of the compressed data stream before you start uncompressing, you can avoid having to use `trailingData` by setting the `InputLength` option.

OneShot Examples

To read the contents of the file `file1.txt.Compressed` and write the uncompressed data to the file `file1.txt`.

```
use strict ;
use warnings ;
use IO::Uncompress::AnyUncompress qw(anyuncompress
 ↪$AnyUncompressError) ;

my $input = "file1.txt.Compressed";
my $output = "file1.txt";
anyuncompress $input => $output
    or die "anyuncompress failed: $AnyUncompressError\n";
```

To read from an existing Perl filehandle, `$input`, and write the uncompressed data to a buffer, `$buffer`.

```
use strict ;
use warnings ;
use IO::Uncompress::AnyUncompress qw(anyuncompress
 ↪$AnyUncompressError) ;
use IO::File ;

my $input = IO::File->new( "<file1.txt.Compressed" )
    or die "Cannot open 'file1.txt.Compressed': $!\n" ;
my $buffer ;
anyuncompress $input => \$buffer
    or die "anyuncompress failed: $AnyUncompressError\n";
```

To uncompress all files in the directory "/my/home" that match "*.txt.Compressed" and store the compressed data in the same directory

```
use strict ;
use warnings ;
use IO::Uncompress::AnyUncompress qw(anyuncompress
 ↪$AnyUncompressError) ;

anyuncompress '</my/home/*.txt.Compressed>' => '</my/home/#1.txt>'
    or die "anyuncompress failed: $AnyUncompressError\n";
```

and if you want to compress each file one at a time, this will do the trick

```
use strict ;
use warnings ;
use IO::Uncompress::AnyUncompress qw(anyuncompress
 ↪$AnyUncompressError) ;

for my $input ( glob "/my/home/*.txt.Compressed" )
{
    my $output = $input;
    $output =~ s/.Compressed// ;
    anyuncompress $input => $output
        or die "Error compressing '$input': $AnyUncompressError\n";
}
```

OO Interface

Constructor

The format of the constructor for IO::Uncompress::AnyUncompress is shown below

```
my $z = IO::Uncompress::AnyUncompress->new( $input [OPTS] )
    or die "IO::Uncompress::AnyUncompress failed:
 ↪$AnyUncompressError\n";
```

The constructor takes one mandatory parameter, `$input`, defined below, and zero or more `OPTS`, defined in “Constructor Options”.

Returns an `IO::Uncompress::AnyUncompress` object on success and undef on failure. The variable `$AnyUncompressError` will contain an error message on failure.

If you are running Perl 5.005 or better the object, `$z`, returned from IO::Uncompress::AnyUncompress can be used exactly like an *IO::File* filehandle. This means that all normal input file operations can be carried out with `$z`. For example, to read a line from a compressed file/buffer you can use either of these forms

```
$line = $z->getline();
$line = <$z>;
```

Below is a simple exaple of using the OO interface to read the compressed file `myfile.Compressed` and write its contents to stdout.

```
my $filename = "myfile.Compressed";
my $z = IO::Uncompress::AnyUncompress->new($filename)
    or die "IO::Uncompress::AnyUncompress failed:
 ↪$AnyUncompressError\n";

while (<$z>) {
    print $_;
}
$z->close();
```

See “EXAMPLES” for further examples

The mandatory parameter `$input` is used to determine the source of the compressed data. This parameter can take one of three forms.

- A filename

  If the `$input` parameter is a scalar, it is assumed to be a filename. This file will be opened for reading and the compressed data will be read from it.

- A filehandle

  If the `$input` parameter is a filehandle, the compressed data will be read from it. The string ‘-’ can be used as an alias for standard input.

- A scalar reference

  If `$input` is a scalar reference, the compressed data will be read from `$$input`.

Constructor Options

The option names defined below are case insensitive and can be optionally prefixed by a ‘-’. So all of the following are valid

```
-AutoClose
-autoclose
AUTOCLOSE
autoclose
```

OPTS is a combination of the following options:

- `AutoClose => 0|1`

  This option is only valid when the `$input` parameter is a filehandle. If specified, and the value is true, it will result in the file being closed once either the `close` method is called or the IO::Uncompress::AnyUncompress object is destroyed.

  This parameter defaults to 0.

- `MultiStream => 0|1`

  Allows multiple concatenated compressed streams to be treated as a single compressed stream. Decompression will stop once either the end of the file/buffer is reached, an error is encountered (premature eof, corrupt compressed data) or the end of a stream is not immediately followed by the start of another stream.

  This parameter defaults to 0.

- `Prime => $string`

  This option will uncompress the contents of `$string` before processing the input file/buffer.

  This option can be useful when the compressed data is embedded in another file/data structure and it is not possible to work out where the compressed data begins without having to read the first few bytes. If this is the case, the uncompression can be *primed* with these bytes using this option.

- `Transparent => 0|1`

  If this option is set and the input file/buffer is not compressed data, the module will allow reading of it anyway.

  In addition, if the input file/buffer does contain compressed data and there is non-compressed data immediately following it, setting this option will make this module treat the whole file/buffer as a single data stream.

  This option defaults to 1.

- `BlockSize => $num`

  When reading the compressed input data, IO::Uncompress::AnyUncompress will read it in blocks of `$num` bytes.

  This option defaults to 4096.

- `InputLength => $size`

  When present this option will limit the number of compressed bytes read from the input file/buffer to `$size`. This option can be used in the situation where there is useful data directly after the compressed data stream and you know beforehand the exact length of the compressed data stream.

  This option is mostly used when reading from a filehandle, in which case the file pointer will be left pointing to the first byte directly after the compressed data stream.

  This option defaults to off.

- `Append => 0|1`

  This option controls what the `read` method does with uncompressed data.

  If set to 1, all uncompressed data will be appended to the output parameter of the `read` method.

  If set to 0, the contents of the output parameter of the `read` method will be overwritten by the uncompressed data.

  Defaults to 0.

- `Strict => 0|1`

  This option controls whether the extra checks defined below are used when carrying out the decompression. When Strict is on, the extra tests are carried out, when Strict is off they are not.

  The default for this option is off.

- `RawInflate => 0|1`

  When auto-detecting the compressed format, try to test for raw-deflate (RFC 1951) content using the `IO::Uncompress::RawInflate` module.

  The reason this is not default behaviour is because RFC 1951 content can only be detected by attempting to uncompress it. This process is error prone and can result is false positives.

  Defaults to 0.

- `UnLzma => 0|1`

  When auto-detecting the compressed format, try to test for lzma_alone content using the `IO::Uncompress::UnLzma` module.

  The reason this is not default behaviour is because lzma_alone content can only be detected by attempting to uncompress it. This process is error prone and can result is false positives.

  Defaults to 0.

Methods

read

Usage is

```
$status = $z->read($buffer)
```

Reads a block of compressed data (the size of the compressed block is determined by the `Buffer` option in the constructor), uncompresses it and writes any uncompressed data into `$buffer`. If the Append parameter is set in the constructor, the uncompressed data will be appended to the `$buffer` parameter. Otherwise `$buffer` will be overwritten.

Returns the number of uncompressed bytes written to `$buffer`, zero if eof or a negative number on error.

read

Usage is

```
$status = $z->read($buffer, $length)
$status = $z->read($buffer, $length, $offset)

$status = read($z, $buffer, $length)
$status = read($z, $buffer, $length, $offset)
```

Attempt to read `$length` bytes of uncompressed data into `$buffer`.

The main difference between this form of the `read` method and the previous one, is that this one will attempt to return *exactly* `$length` bytes. The only circumstances that this function will not is if end-of-file or an IO error is encountered.

Returns the number of uncompressed bytes written to `$buffer`, zero if eof or a negative number on error.

getline

Usage is

```
$line = $z->getline()
$line = <$z>
```

Reads a single line.

This method fully supports the use of the variable `$/` (or `$INPUT_RECORD_SEPARATOR` or `$RS` when `English` is in use) to determine what constitutes an end of line. Paragraph mode, record mode and file slurp mode are all supported.

getc

Usage is

```
$char = $z->getc()
```

Read a single character.

ungetc

Usage is

```
$char = $z->ungetc($string)
```

getHeaderInfo

Usage is

```
$hdr  = $z->getHeaderInfo();
@hdrs = $z->getHeaderInfo();
```

This method returns either a hash reference (in scalar context) or a list or hash references (in array context) that contains information about each of the header fields in the compressed data stream(s).

tell

Usage is

```
$z->tell()
tell $z
```

Returns the uncompressed file offset.

eof

Usage is

```
$z->eof();
eof($z);
```

Returns true if the end of the compressed input stream has been reached.

seek

```
$z->seek($position, $whence);
seek($z, $position, $whence);
```

Provides a sub-set of the `seek` functionality, with the restriction that it is only legal to seek forward in the input file/buffer. It is a fatal error to attempt to seek backward.

Note that the implementation of `seek` in this module does not provide true random access to a compressed file/buffer. It works by uncompressing data from the current offset in the file/buffer until it reaches the uncompressed offset specified in the parameters to `seek`. For very small files this may be acceptable behaviour. For large files it may cause an unacceptable delay.

The `$whence` parameter takes one the usual values, namely SEEK_SET, SEEK_CUR or SEEK_END.

Returns 1 on success, 0 on failure.

binmode

Usage is

```
$z->binmode
binmode $z ;
```

This is a noop provided for completeness.

opened

```
$z->opened()
```

Returns true if the object currently refers to a opened file/buffer.

autoflush

```
my $prev = $z->autoflush()
my $prev = $z->autoflush(EXPR)
```

If the `$z` object is associated with a file or a filehandle, this method returns the current autoflush setting for the underlying filehandle. If `EXPR` is present, and is non-zero, it will enable flushing after every write/print operation.

If `$z` is associated with a buffer, this method has no effect and always returns `undef`.

Note that the special variable `$|` **cannot** be used to set or retrieve the autoflush setting.

input_line_number

```
$z->input_line_number()
$z->input_line_number(EXPR)
```

Returns the current uncompressed line number. If `EXPR` is present it has the effect of setting the line number. Note that setting the line number does not change the current position within the file/buffer being read.

The contents of `$/` are used to determine what constitutes a line terminator.

fileno

```
$z->fileno()
fileno($z)
```

If the `$z` object is associated with a file or a filehandle, `fileno` will return the underlying file descriptor. Once the `close` method is called `fileno` will return `undef`.

If the `$z` object is associated with a buffer, this method will return `undef`.

close

```
$z->close() ;
close $z ;
```

Closes the output file/buffer.

For most versions of Perl this method will be automatically invoked if the IO::Uncompress::AnyUncompress object is destroyed (either explicitly or by the variable with the reference to the object going out of scope). The exceptions are Perl versions 5.005 through 5.00504 and 5.8.0. In these cases, the `close` method will be called automatically, but not until global destruction of all live objects when the program is terminating.

Therefore, if you want your scripts to be able to run on all versions of Perl, you should call `close` explicitly and not rely on automatic closing.

Returns true on success, otherwise 0.

If the `AutoClose` option has been enabled when the IO::Uncompress::AnyUncompress object was created, and the object is associated with a file, the underlying file will also be closed.

nextStream

Usage is

```
my $status = $z->nextStream();
```

Skips to the next compressed data stream in the input file/buffer. If a new compressed data stream is found, the eof marker will be cleared and `$.` will be reset to 0.

Returns 1 if a new stream was found, 0 if none was found, and -1 if an error was encountered.

trailingData

Usage is

```
my $data = $z->trailingData();
```

Returns the data, if any, that is present immediately after the compressed data stream once uncompression is complete. It only makes sense to call this method once the end of the compressed data stream has been encountered.

This option can be used when there is useful information immediately following the compressed data stream, and you don't know the length of the compressed data stream.

If the input is a buffer, `trailingData` will return everything from the end of the compressed data stream to the end of the buffer.

If the input is a filehandle, `trailingData` will return the data that is left in the filehandle input buffer once the end of the compressed data stream has been reached. You can then use the filehandle to read the rest of the input file.

Don't bother using `trailingData` if the input is a filename.

If you know the length of the compressed data stream before you start uncompressing, you can avoid having to use `trailingData` by setting the `InputLength` option in the constructor.

Importing

No symbolic constants are required by IO::Uncompress::AnyUncompress at present.

- :all

  Imports `anyuncompress` and `$AnyUncompressError`. Same as doing this

  ```
  use IO::Uncompress::AnyUncompress qw(anyuncompress
  ↪$AnyUncompressError) ;
  ```

EXAMPLES

SUPPORT

General feedback/questions/bug reports should be sent to https://github.com/pmqs/IO-Compress/issues (preferred) or https://rt.cpan.org/Public/Dist/Display.html?Name=IO-Compress.

SEE ALSO

Compress::Zlib, *IO::Compress::Gzip*, *IO::Uncompress::Gunzip*, *IO::Compress::Deflate*, *IO::Uncompress::Inflate*, *IO::Compress::RawDeflate*, *IO::Uncompress::RawInflate*, *IO::Compress::Bzip2*, *IO::Uncompress::Bunzip2*, IO::Compress::Lzma, IO::Uncompress::UnLzma, IO::Compress::Xz, IO::Uncompress::UnXz, IO::Compress::Lzip, IO::Uncompress::UnLzip, IO::Compress::Lzop, IO::Uncompress::UnLzop, IO::Compress::Lzf, IO::Uncompress::UnLzf, IO::Compress::Zstd, IO::Uncompress::UnZstd, *IO::Uncompress::AnyInflate*

IO::Compress::FAQ

File::GlobMapper, Archive::Zip, *Archive::Tar*, *IO::Zlib*

AUTHOR

This module was written by Paul Marquess, `pmqs@cpan.org`.

MODIFICATION HISTORY

See the Changes file.

COPYRIGHT AND LICENSE

Copyright (c) 2005-2026 Paul Marquess. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

IO::Uncompress::Base

Base Class for IO::Uncompress modules

SYNOPSIS

```
use IO::Uncompress::Base ;
```

DESCRIPTION

This module is not intended for direct use in application code. Its sole purpose is to be sub-classed by IO::Uncompress modules.

SUPPORT

General feedback/questions/bug reports should be sent to https://github.com/pmqs/IO-Compress/issues (preferred) or https://rt.cpan.org/Public/Dist/Display.html?Name=IO-Compress.

SEE ALSO

Compress::Zlib, *IO::Compress::Gzip*, *IO::Uncompress::Gunzip*, *IO::Compress::Deflate*, *IO::Uncompress::Inflate*, *IO::Compress::RawDeflate*, *IO::Uncompress::RawInflate*, *IO::Compress::Bzip2*, *IO::Uncompress::Bunzip2*, IO::Compress::Lzma, IO::Uncompress::UnLzma, IO::Compress::Xz, IO::Uncompress::UnXz, IO::Compress::Lzip, IO::Uncompress::UnLzip, IO::Compress::Lzop, IO::Uncompress::UnLzop, IO::Compress::Lzf, IO::Uncompress::UnLzf, IO::Compress::Zstd, IO::Uncompress::UnZstd, *IO::Uncompress::AnyInflate*, *IO::Uncompress::AnyUncompress*

IO::Compress::FAQ

File::GlobMapper, Archive::Zip, *Archive::Tar*, *IO::Zlib*

AUTHOR

This module was written by Paul Marquess, `pmqs@cpan.org`.

MODIFICATION HISTORY

See the Changes file.

COPYRIGHT AND LICENSE

Copyright (c) 2005-2026 Paul Marquess. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

IO::Uncompress::Bunzip2

Read bzip2 files/buffers

SYNOPSIS

```
use IO::Uncompress::Bunzip2 qw(bunzip2 $Bunzip2Error) ;

my $status = bunzip2 $input => $output [,OPTS]
    or die "bunzip2 failed: $Bunzip2Error\n";

my $z = IO::Uncompress::Bunzip2->new( $input [OPTS] )
    or die "bunzip2 failed: $Bunzip2Error\n";

$status = $z->read($buffer)
$status = $z->read($buffer, $length)
$status = $z->read($buffer, $length, $offset)
$line = $z->getline()
$char = $z->getc()
$char = $z->ungetc()
$char = $z->opened()
```

(continues on next page)

(continued from previous page)

```
$data = $z->trailingData()
$status = $z->nextStream()
$data = $z->getHeaderInfo()
$z->tell()
$z->seek($position, $whence)
$z->binmode()
$z->fileno()
$z->eof()
$z->close()

$Bunzip2Error ;

# IO::File mode

<$z>
read($z, $buffer);
read($z, $buffer, $length);
read($z, $buffer, $length, $offset);
tell($z)
seek($z, $position, $whence)
binmode($z)
fileno($z)
eof($z)
close($z)
```

DESCRIPTION

This module provides a Perl interface that allows the reading of bzip2 files/buffers.

For writing bzip2 files/buffers, see the companion module IO::Compress::Bzip2.

Functional Interface

A top-level function, bunzip2, is provided to carry out "one-shot" uncompression between buffers and/or files. For finer control over the uncompression process, see the "OO Interface" section.

```
use IO::Uncompress::Bunzip2 qw(bunzip2 $Bunzip2Error) ;

bunzip2 $input_filename_or_reference => $output_filename_or_
 ↪reference [,OPTS]
    or die "bunzip2 failed: $Bunzip2Error\n";
```

The functional interface needs Perl5.005 or better.

bunzip2 $input_filename_or_reference => $output_filename_or_reference [, OPTS]

bunzip2 expects at least two parameters, `$input_filename_or_reference` and `$output_filename_or_reference` and zero or more optional parameters (see "Optional Parameters")

The `$input_filename_or_reference` parameter

The parameter, `$input_filename_or_reference`, is used to define the source of the compressed data.

It can take one of the following forms:

- A filename

  If the `$input_filename_or_reference` parameter is a simple scalar, it is assumed to be a filename. This file will be opened for reading and the input data will be read from it.

- A filehandle

  If the `$input_filename_or_reference` parameter is a filehandle, the input data will be read from it. The string '-' can be used as an alias for standard input.

- A scalar reference

  If `$input_filename_or_reference` is a scalar reference, the input data will be read from `$$input_filename_or_reference`.

- An array reference

  If `$input_filename_or_reference` is an array reference, each element in the array must be a filename.

  The input data will be read from each file in turn.

  The complete array will be walked to ensure that it only contains valid filenames before any data is uncompressed.

- An Input FileGlob string

  If `$input_filename_or_reference` is a string that is delimited by the characters "<" and ">" bunzip2 will assume that it is an *input fileglob string*. The input is the list of files that match the fileglob.

  See *File::GlobMapper* for more details.

If the `$input_filename_or_reference` parameter is any other type, `undef` will be returned.

The `$output_filename_or_reference` parameter

The parameter `$output_filename_or_reference` is used to control the destination of the uncompressed data. This parameter can take one of these forms.

- A filename

  If the `$output_filename_or_reference` parameter is a simple scalar, it is assumed to be a filename. This file will be opened for writing and the uncompressed data will be written to it.

- A filehandle

  If the `$output_filename_or_reference` parameter is a filehandle, the uncompressed data will be written to it. The string '-' can be used as an alias for standard output.

- A scalar reference

  If `$output_filename_or_reference` is a scalar reference, the uncompressed data will be stored in `$$output_filename_or_reference`.

- An Array Reference

  If `$output_filename_or_reference` is an array reference, the uncompressed data will be pushed onto the array.

- An Output FileGlob

  If `$output_filename_or_reference` is a string that is delimited by the characters "<" and ">" `bunzip2` will assume that it is an *output fileglob string*. The output is the list of files that match the fileglob.

  When `$output_filename_or_reference` is an fileglob string, `$input_filename_or_reference` must also be a fileglob string. Anything else is an error.

  See *File::GlobMapper* for more details.

If the `$output_filename_or_reference` parameter is any other type, `undef` will be returned.

Notes

When `$input_filename_or_reference` maps to multiple compressed files/buffers and `$output_filename_or_reference` is a single file/buffer, after uncompression `$output_filename_or_reference` will contain a concatenation of all the uncompressed data from each of the input files/buffers.

Optional Parameters

The optional parameters for the one-shot function `bunzip2` are (for the most part) identical to those used with the OO interface defined in the "Constructor Options" section. The exceptions are listed below

- `AutoClose => 0|1`

  This option applies to any input or output data streams to `bunzip2` that are filehandles.

  If `AutoClose` is specified, and the value is true, it will result in all input and/or output filehandles being closed once `bunzip2` has completed.

  This parameter defaults to 0.

- `BinModeOut => 0|1`

  This option is now a no-op. All files will be written in binmode.

- `Append => 0|1`

  The behaviour of this option is dependent on the type of output data stream.

- A Buffer

  If Append is enabled, all uncompressed data will be append to the end of the output buffer. Otherwise the output buffer will be cleared before any uncompressed data is written to it.

- A Filename

  If Append is enabled, the file will be opened in append mode. Otherwise the contents of the file, if any, will be truncated before any uncompressed data is written to it.

- A Filehandle

  If Append is enabled, the filehandle will be positioned to the end of the file via a call to `seek` before any uncompressed data is written to it. Otherwise the file pointer will not be moved.

When Append is specified, and set to true, it will *append* all uncompressed data to the output data stream.

So when the output is a filehandle it will carry out a seek to the eof before writing any uncompressed data. If the output is a filename, it will be opened for appending. If the output is a buffer, all uncompressed data will be appended to the existing buffer.

Conversely when Append is not specified, or it is present and is set to false, it will operate as follows.

When the output is a filename, it will truncate the contents of the file before writing any uncompressed data. If the output is a filehandle its position will not be changed. If the output is a buffer, it will be wiped before any uncompressed data is output.

Defaults to 0.

- `MultiStream => 0|1`

  If the input file/buffer contains multiple compressed data streams, this option will uncompress the whole lot as a single data stream.

  Defaults to 0.

- `TrailingData => $scalar`

  Returns the data, if any, that is present immediately after the compressed data stream once uncompression is complete.

  This option can be used when there is useful information immediately following the compressed data stream, and you don't know the length of the compressed data stream.

  If the input is a buffer, `trailingData` will return everything from the end of the compressed data stream to the end of the buffer.

  If the input is a filehandle, `trailingData` will return the data that is left in the filehandle input buffer once the end of the compressed data stream has been reached. You can then use the filehandle to read the rest of the input file.

  Don't bother using `trailingData` if the input is a filename.

If you know the length of the compressed data stream before you start uncompressing, you can avoid having to use `trailingData` by setting the `InputLength` option.

OneShot Examples

To read the contents of the file `file1.txt.bz2` and write the uncompressed data to the file `file1.txt`.

```
use strict ;
use warnings ;
use IO::Uncompress::Bunzip2 qw(bunzip2 $Bunzip2Error) ;

my $input = "file1.txt.bz2";
my $output = "file1.txt";
bunzip2 $input => $output
    or die "bunzip2 failed: $Bunzip2Error\n";
```

To read from an existing Perl filehandle, `$input`, and write the uncompressed data to a buffer, `$buffer`.

```
use strict ;
use warnings ;
use IO::Uncompress::Bunzip2 qw(bunzip2 $Bunzip2Error) ;
use IO::File ;

my $input = IO::File->new( "<file1.txt.bz2" )
    or die "Cannot open 'file1.txt.bz2': $!\n" ;
my $buffer ;
bunzip2 $input => \$buffer
    or die "bunzip2 failed: $Bunzip2Error\n";
```

To uncompress all files in the directory "/my/home" that match "*.txt.bz2" and store the compressed data in the same directory

```
use strict ;
use warnings ;
use IO::Uncompress::Bunzip2 qw(bunzip2 $Bunzip2Error) ;

bunzip2 '</my/home/*.txt.bz2>' => '</my/home/#1.txt>'
    or die "bunzip2 failed: $Bunzip2Error\n";
```

and if you want to compress each file one at a time, this will do the trick

```
use strict ;
use warnings ;
use IO::Uncompress::Bunzip2 qw(bunzip2 $Bunzip2Error) ;

for my $input ( glob "/my/home/*.txt.bz2" )
{
    my $output = $input;
```

(continues on next page)

(continued from previous page)

```
    $output =~ s/.bz2// ;
    bunzip2 $input => $output
        or die "Error compressing '$input': $Bunzip2Error\n";
}
```

OO Interface

Constructor

The format of the constructor for IO::Uncompress::Bunzip2 is shown below

```
my $z = IO::Uncompress::Bunzip2->new( $input [OPTS] )
    or die "IO::Uncompress::Bunzip2 failed: $Bunzip2Error\n";
```

The constructor takes one mandatory parameter, `$input`, defined below, and zero or more `OPTS`, defined in "Constructor Options".

Returns an `IO::Uncompress::Bunzip2` object on success and undef on failure. The variable `$Bunzip2Error` will contain an error message on failure.

If you are running Perl 5.005 or better the object, `$z`, returned from IO::Uncompress::Bunzip2 can be used exactly like an *IO::File* filehandle. This means that all normal input file operations can be carried out with `$z`. For example, to read a line from a compressed file/buffer you can use either of these forms

```
$line = $z->getline();
$line = <$z>;
```

Below is a simple exaple of using the OO interface to read the compressed file `myfile.bz2` and write its contents to stdout.

```
my $filename = "myfile.bz2";
my $z = IO::Uncompress::Bunzip2->new($filename)
    or die "IO::Uncompress::Bunzip2 failed: $Bunzip2Error\n";

while (<$z>) {
    print $_;
}
$z->close();
```

See "EXAMPLES" for further examples

The mandatory parameter `$input` is used to determine the source of the compressed data. This parameter can take one of three forms.

- A filename

  If the `$input` parameter is a scalar, it is assumed to be a filename. This file will be opened for reading and the compressed data will be read from it.

- A filehandle

  If the `$input` parameter is a filehandle, the compressed data will be read from it. The string '-' can be used as an alias for standard input.

- A scalar reference

  If `$input` is a scalar reference, the compressed data will be read from `$$input`.

Constructor Options

The option names defined below are case insensitive and can be optionally prefixed by a '-'. So all of the following are valid

```
-AutoClose
-autoclose
AUTOCLOSE
autoclose
```

OPTS is a combination of the following options:

- `AutoClose => 0|1`

  This option is only valid when the `$input` parameter is a filehandle. If specified, and the value is true, it will result in the file being closed once either the `close` method is called or the IO::Uncompress::Bunzip2 object is destroyed.

  This parameter defaults to 0.

- `MultiStream => 0|1`

  Allows multiple concatenated compressed streams to be treated as a single compressed stream. Decompression will stop once either the end of the file/buffer is reached, an error is encountered (premature eof, corrupt compressed data) or the end of a stream is not immediately followed by the start of another stream.

  This parameter defaults to 0.

- `Prime => $string`

  This option will uncompress the contents of `$string` before processing the input file/buffer.

  This option can be useful when the compressed data is embedded in another file/data structure and it is not possible to work out where the compressed data begins without having to read the first few bytes. If this is the case, the uncompression can be *primed* with these bytes using this option.

- `Transparent => 0|1`

  If this option is set and the input file/buffer is not compressed data, the module will allow reading of it anyway.

  In addition, if the input file/buffer does contain compressed data and there is non-compressed data immediately following it, setting this option will make this module treat the whole file/buffer as a single data stream.

  This option defaults to 1.

- `BlockSize => $num`

  When reading the compressed input data, IO::Uncompress::Bunzip2 will read it in blocks of `$num` bytes.

  This option defaults to 4096.

- `InputLength => $size`

  When present this option will limit the number of compressed bytes read from the input file/buffer to `$size`. This option can be used in the situation where there is useful data directly after the compressed data stream and you know beforehand the exact length of the compressed data stream.

  This option is mostly used when reading from a filehandle, in which case the file pointer will be left pointing to the first byte directly after the compressed data stream.

  This option defaults to off.

- `Append => 0|1`

  This option controls what the `read` method does with uncompressed data.

  If set to 1, all uncompressed data will be appended to the output parameter of the `read` method.

  If set to 0, the contents of the output parameter of the `read` method will be overwritten by the uncompressed data.

  Defaults to 0.

- `Strict => 0|1`

  This option is a no-op.

- `Small => 0|1`

  When non-zero this options will make bzip2 use a decompression algorithm that uses less memory at the expense of increasing the amount of time taken for decompression.

  Default is 0.

Methods

read

Usage is

```
$status = $z->read($buffer)
```

Reads a block of compressed data (the size of the compressed block is determined by the `Buffer` option in the constructor), uncompresses it and writes any uncompressed data into `$buffer`. If the Append parameter is set in the constructor, the uncompressed data will be appended to the `$buffer` parameter. Otherwise `$buffer` will be overwritten.

Returns the number of uncompressed bytes written to `$buffer`, zero if eof or a negative number on error.

read

Usage is

```
$status = $z->read($buffer, $length)
$status = $z->read($buffer, $length, $offset)

$status = read($z, $buffer, $length)
$status = read($z, $buffer, $length, $offset)
```

Attempt to read `$length` bytes of uncompressed data into `$buffer`.

The main difference between this form of the `read` method and the previous one, is that this one will attempt to return *exactly* `$length` bytes. The only circumstances that this function will not is if end-of-file or an IO error is encountered.

Returns the number of uncompressed bytes written to `$buffer`, zero if eof or a negative number on error.

getline

Usage is

```
$line = $z->getline()
$line = <$z>
```

Reads a single line.

This method fully supports the use of the variable `$/` (or `$INPUT_RECORD_SEPARATOR` or `$RS` when `English` is in use) to determine what constitutes an end of line. Paragraph mode, record mode and file slurp mode are all supported.

getc

Usage is

```
$char = $z->getc()
```

Read a single character.

ungetc

Usage is

```
$char = $z->ungetc($string)
```

getHeaderInfo

Usage is

```
$hdr  = $z->getHeaderInfo();
@hdrs = $z->getHeaderInfo();
```

This method returns either a hash reference (in scalar context) or a list or hash references (in array context) that contains information about each of the header fields in the compressed data stream(s).

tell

Usage is

```
$z->tell()
tell $z
```

Returns the uncompressed file offset.

eof

Usage is

```
$z->eof();
eof($z);
```

Returns true if the end of the compressed input stream has been reached.

seek

```
$z->seek($position, $whence);
seek($z, $position, $whence);
```

Provides a sub-set of the `seek` functionality, with the restriction that it is only legal to seek forward in the input file/buffer. It is a fatal error to attempt to seek backward.

Note that the implementation of `seek` in this module does not provide true random access to a compressed file/buffer. It works by uncompressing data from the current offset in the file/buffer until it reaches the uncompressed offset specified in the parameters to `seek`. For very small files this may be acceptable behaviour. For large files it may cause an unacceptable delay.

The `$whence` parameter takes one the usual values, namely SEEK_SET, SEEK_CUR or SEEK_END.

Returns 1 on success, 0 on failure.

binmode

Usage is

```
$z->binmode
binmode $z ;
```

This is a noop provided for completeness.

opened

```
$z->opened()
```

Returns true if the object currently refers to a opened file/buffer.

autoflush

```
my $prev = $z->autoflush()
my $prev = $z->autoflush(EXPR)
```

If the `$z` object is associated with a file or a filehandle, this method returns the current autoflush setting for the underlying filehandle. If `EXPR` is present, and is non-zero, it will enable flushing after every write/print operation.

If `$z` is associated with a buffer, this method has no effect and always returns `undef`.

Note that the special variable `$|` **cannot** be used to set or retrieve the autoflush setting.

input_line_number

```
$z->input_line_number()
$z->input_line_number(EXPR)
```

Returns the current uncompressed line number. If `EXPR` is present it has the effect of setting the line number. Note that setting the line number does not change the current position within the file/buffer being read.

The contents of `$/` are used to determine what constitutes a line terminator.

fileno

```
$z->fileno()
fileno($z)
```

If the `$z` object is associated with a file or a filehandle, `fileno` will return the underlying file descriptor. Once the `close` method is called `fileno` will return `undef`.

If the `$z` object is associated with a buffer, this method will return `undef`.

close

```
$z->close() ;
close $z ;
```

Closes the output file/buffer.

For most versions of Perl this method will be automatically invoked if the IO::Uncompress::Bunzip2 object is destroyed (either explicitly or by the variable with the reference to the object going out of scope). The exceptions are Perl versions 5.005 through 5.00504 and 5.8.0. In these cases, the `close` method will be called automatically, but not until global destruction of all live objects when the program is terminating.

Therefore, if you want your scripts to be able to run on all versions of Perl, you should call `close` explicitly and not rely on automatic closing.

Returns true on success, otherwise 0.

If the `AutoClose` option has been enabled when the IO::Uncompress::Bunzip2 object was created, and the object is associated with a file, the underlying file will also be closed.

nextStream

Usage is

```
my $status = $z->nextStream();
```

Skips to the next compressed data stream in the input file/buffer. If a new compressed data stream is found, the eof marker will be cleared and `$.` will be reset to 0.

Returns 1 if a new stream was found, 0 if none was found, and -1 if an error was encountered.

trailingData

Usage is

```
my $data = $z->trailingData();
```

Returns the data, if any, that is present immediately after the compressed data stream once uncompression is complete. It only makes sense to call this method once the end of the compressed data stream has been encountered.

This option can be used when there is useful information immediately following the compressed data stream, and you don't know the length of the compressed data stream.

If the input is a buffer, `trailingData` will return everything from the end of the compressed data stream to the end of the buffer.

If the input is a filehandle, `trailingData` will return the data that is left in the filehandle input buffer once the end of the compressed data stream has been reached. You can then use the filehandle to read the rest of the input file.

Don't bother using `trailingData` if the input is a filename.

If you know the length of the compressed data stream before you start uncompressing, you can avoid having to use `trailingData` by setting the `InputLength` option in the constructor.

Importing

No symbolic constants are required by IO::Uncompress::Bunzip2 at present.

- :all

  Imports `bunzip2` and `$Bunzip2Error`. Same as doing this

  ```
  use IO::Uncompress::Bunzip2 qw(bunzip2 $Bunzip2Error) ;
  ```

EXAMPLES

Working with Net::FTP

See IO::Compress::FAQ

SUPPORT

General feedback/questions/bug reports should be sent to https://github.com/pmqs/IO-Compress/issues (preferred) or https://rt.cpan.org/Public/Dist/Display.html?Name=IO-Compress.

SEE ALSO

Compress::Zlib, *IO::Compress::Gzip*, *IO::Uncompress::Gunzip*, *IO::Compress::Deflate*, *IO::Uncompress::Inflate*, *IO::Compress::RawDeflate*, *IO::Uncompress::RawInflate*, *IO::Compress::Bzip2*, IO::Compress::Lzma, IO::Uncompress::UnLzma, IO::Compress::Xz, IO::Uncompress::UnXz, IO::Compress::Lzip, IO::Uncompress::UnLzip, IO::Compress::Lzop, IO::Uncompress::UnLzop, IO::Compress::Lzf, IO::Uncompress::UnLzf, IO::Compress::Zstd, IO::Uncompress::UnZstd, *IO::Uncompress::AnyInflate*, *IO::Uncompress::AnyUncompress*

IO::Compress::FAQ

File::GlobMapper, Archive::Zip, *Archive::Tar*, *IO::Zlib*

The primary site for the bzip2 program is https://sourceware.org/bzip2/.

See the module Compress::Bzip2

AUTHOR

This module was written by Paul Marquess, `pmqs@cpan.org`.

MODIFICATION HISTORY

See the Changes file.

COPYRIGHT AND LICENSE

Copyright (c) 2005-2026 Paul Marquess. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

IO::Uncompress::Gunzip

Read RFC 1952 files/buffers

SYNOPSIS

```
use IO::Uncompress::Gunzip qw(gunzip $GunzipError) ;

my $status = gunzip $input => $output [,OPTS]
    or die "gunzip failed: $GunzipError\n";

my $z = IO::Uncompress::Gunzip->new( $input [OPTS] )
    or die "gunzip failed: $GunzipError\n";
```

(continues on next page)

(continued from previous page)

```
$status = $z->read($buffer)
$status = $z->read($buffer, $length)
$status = $z->read($buffer, $length, $offset)
$line = $z->getline()
$char = $z->getc()
$char = $z->ungetc()
$char = $z->opened()

$status = $z->inflateSync()

$data = $z->trailingData()
$status = $z->nextStream()
$data = $z->getHeaderInfo()
$z->tell()
$z->seek($position, $whence)
$z->binmode()
$z->fileno()
$z->eof()
$z->close()

$GunzipError ;

# IO::File mode

<$z>
read($z, $buffer);
read($z, $buffer, $length);
read($z, $buffer, $length, $offset);
tell($z)
seek($z, $position, $whence)
binmode($z)
fileno($z)
eof($z)
close($z)
```

DESCRIPTION

This module provides a Perl interface that allows the reading of files/buffers that conform to RFC 1952.

For writing RFC 1952 files/buffers, see the companion module IO::Compress::Gzip.

Functional Interface

A top-level function, gunzip, is provided to carry out “one-shot” uncompression between buffers and/or files. For finer control over the uncompression process, see the “OO Interface” section.

```
use IO::Uncompress::Gunzip qw(gunzip $GunzipError) ;

gunzip $input_filename_or_reference => $output_filename_or_reference␣
↪[,OPTS]
    or die "gunzip failed: $GunzipError\n";
```

The functional interface needs Perl5.005 or better.

gunzip $input_filename_or_reference => $output_filename_or_reference [, OPTS]

gunzip expects at least two parameters, `$input_filename_or_reference` and `$output_filename_or_reference` and zero or more optional parameters (see "Optional Parameters")

The $input_filename_or_reference parameter

The parameter, `$input_filename_or_reference`, is used to define the source of the compressed data.

It can take one of the following forms:

- A filename

  If the `$input_filename_or_reference` parameter is a simple scalar, it is assumed to be a filename. This file will be opened for reading and the input data will be read from it.

- A filehandle

  If the `$input_filename_or_reference` parameter is a filehandle, the input data will be read from it. The string '-' can be used as an alias for standard input.

- A scalar reference

  If `$input_filename_or_reference` is a scalar reference, the input data will be read from `$$input_filename_or_reference`.

- An array reference

  If `$input_filename_or_reference` is an array reference, each element in the array must be a filename.

  The input data will be read from each file in turn.

  The complete array will be walked to ensure that it only contains valid filenames before any data is uncompressed.

- An Input FileGlob string

  If `$input_filename_or_reference` is a string that is delimited by the characters "<" and ">" gunzip will assume that it is an *input fileglob string*. The input is the list of files that match the fileglob.

  See *File::GlobMapper* for more details.

If the `$input_filename_or_reference` parameter is any other type, `undef` will be returned.

The $output_filename_or_reference parameter

The parameter $output_filename_or_reference is used to control the destination of the uncompressed data. This parameter can take one of these forms.

- A filename

  If the $output_filename_or_reference parameter is a simple scalar, it is assumed to be a filename. This file will be opened for writing and the uncompressed data will be written to it.

- A filehandle

  If the $output_filename_or_reference parameter is a filehandle, the uncompressed data will be written to it. The string '-' can be used as an alias for standard output.

- A scalar reference

  If $output_filename_or_reference is a scalar reference, the uncompressed data will be stored in $$output_filename_or_reference.

- An Array Reference

  If $output_filename_or_reference is an array reference, the uncompressed data will be pushed onto the array.

- An Output FileGlob

  If $output_filename_or_reference is a string that is delimited by the characters "<" and ">" gunzip will assume that it is an *output fileglob string*. The output is the list of files that match the fileglob.

  When $output_filename_or_reference is an fileglob string, $input_filename_or_reference must also be a fileglob string. Anything else is an error.

  See *File::GlobMapper* for more details.

If the $output_filename_or_reference parameter is any other type, undef will be returned.

Notes

When $input_filename_or_reference maps to multiple compressed files/buffers and $output_filename_or_reference is a single file/buffer, after uncompression $output_filename_or_reference will contain a concatenation of all the uncompressed data from each of the input files/buffers.

Optional Parameters

The optional parameters for the one-shot function gunzip are (for the most part) identical to those used with the OO interface defined in the "Constructor Options" section. The exceptions are listed below

- AutoClose => 0|1

  This option applies to any input or output data streams to gunzip that are filehandles.

If `AutoClose` is specified, and the value is true, it will result in all input and/or output filehandles being closed once `gunzip` has completed.

This parameter defaults to 0.

- `BinModeOut => 0|1`

  This option is now a no-op. All files will be written in binmode.

- `Append => 0|1`

  The behaviour of this option is dependent on the type of output data stream.

  - A Buffer

    If Append is enabled, all uncompressed data will be append to the end of the output buffer. Otherwise the output buffer will be cleared before any uncompressed data is written to it.

  - A Filename

    If Append is enabled, the file will be opened in append mode. Otherwise the contents of the file, if any, will be truncated before any uncompressed data is written to it.

  - A Filehandle

    If Append is enabled, the filehandle will be positioned to the end of the file via a call to `seek` before any uncompressed data is written to it. Otherwise the file pointer will not be moved.

  When Append is specified, and set to true, it will *append* all uncompressed data to the output data stream.

  So when the output is a filehandle it will carry out a seek to the eof before writing any uncompressed data. If the output is a filename, it will be opened for appending. If the output is a buffer, all uncompressed data will be appended to the existing buffer.

  Conversely when Append is not specified, or it is present and is set to false, it will operate as follows.

  When the output is a filename, it will truncate the contents of the file before writing any uncompressed data. If the output is a filehandle its position will not be changed. If the output is a buffer, it will be wiped before any uncompressed data is output.

  Defaults to 0.

- `MultiStream => 0|1`

  If the input file/buffer contains multiple compressed data streams, this option will uncompress the whole lot as a single data stream.

  Defaults to 0.

- `TrailingData => $scalar`

  Returns the data, if any, that is present immediately after the compressed data stream once uncompression is complete.

This option can be used when there is useful information immediately following the compressed data stream, and you don't know the length of the compressed data stream.

If the input is a buffer, `trailingData` will return everything from the end of the compressed data stream to the end of the buffer.

If the input is a filehandle, `trailingData` will return the data that is left in the filehandle input buffer once the end of the compressed data stream has been reached. You can then use the filehandle to read the rest of the input file.

Don't bother using `trailingData` if the input is a filename.

If you know the length of the compressed data stream before you start uncompressing, you can avoid having to use `trailingData` by setting the `InputLength` option.

OneShot Examples

To read the contents of the file `file1.txt.gz` and write the uncompressed data to the file `file1.txt`.

```
use strict ;
use warnings ;
use IO::Uncompress::Gunzip qw(gunzip $GunzipError) ;

my $input = "file1.txt.gz";
my $output = "file1.txt";
gunzip $input => $output
    or die "gunzip failed: $GunzipError\n";
```

To read from an existing Perl filehandle, `$input`, and write the uncompressed data to a buffer, `$buffer`.

```
use strict ;
use warnings ;
use IO::Uncompress::Gunzip qw(gunzip $GunzipError) ;
use IO::File ;

my $input = IO::File->new( "<file1.txt.gz" )
    or die "Cannot open 'file1.txt.gz': $!\n" ;
my $buffer ;
gunzip $input => \$buffer
    or die "gunzip failed: $GunzipError\n";
```

To uncompress all files in the directory "/my/home" that match "*.txt.gz" and store the compressed data in the same directory

```
use strict ;
use warnings ;
use IO::Uncompress::Gunzip qw(gunzip $GunzipError) ;
```

(continues on next page)

(continued from previous page)

```
gunzip '</my/home/*.txt.gz>' => '</my/home/#1.txt>'
    or die "gunzip failed: $GunzipError\n";
```

and if you want to compress each file one at a time, this will do the trick

```
use strict ;
use warnings ;
use IO::Uncompress::Gunzip qw(gunzip $GunzipError) ;

for my $input ( glob "/my/home/*.txt.gz" )
{
    my $output = $input;
    $output =~ s/.gz// ;
    gunzip $input => $output
        or die "Error compressing '$input': $GunzipError\n";
}
```

OO Interface

Constructor

The format of the constructor for IO::Uncompress::Gunzip is shown below

```
my $z = IO::Uncompress::Gunzip->new( $input [OPTS] )
    or die "IO::Uncompress::Gunzip failed: $GunzipError\n";
```

The constructor takes one mandatory parameter, `$input`, defined below, and zero or more `OPTS`, defined in “Constructor Options”.

Returns an `IO::Uncompress::Gunzip` object on success and undef on failure. The variable `$GunzipError` will contain an error message on failure.

If you are running Perl 5.005 or better the object, `$z`, returned from IO::Uncompress::Gunzip can be used exactly like an *IO::File* filehandle. This means that all normal input file operations can be carried out with `$z`. For example, to read a line from a compressed file/buffer you can use either of these forms

```
$line = $z->getline();
$line = <$z>;
```

Below is a simple exaple of using the OO interface to read the compressed file `myfile.gz` and write its contents to stdout.

```
my $filename = "myfile.gz";
my $z = IO::Uncompress::Gunzip->new($filename)
    or die "IO::Uncompress::Gunzip failed: $GunzipError\n";

while (<$z>) {
    print $_;
}
$z->close();
```

See “EXAMPLES” for further examples

The mandatory parameter `$input` is used to determine the source of the compressed data. This parameter can take one of three forms.

- A filename

  If the `$input` parameter is a scalar, it is assumed to be a filename. This file will be opened for reading and the compressed data will be read from it.

- A filehandle

  If the `$input` parameter is a filehandle, the compressed data will be read from it. The string ‘-’ can be used as an alias for standard input.

- A scalar reference

  If `$input` is a scalar reference, the compressed data will be read from `$$input`.

Constructor Options

The option names defined below are case insensitive and can be optionally prefixed by a ‘-’. So all of the following are valid

```
-AutoClose
-autoclose
AUTOCLOSE
autoclose
```

OPTS is a combination of the following options:

- `AutoClose => 0|1`

  This option is only valid when the `$input` parameter is a filehandle. If specified, and the value is true, it will result in the file being closed once either the `close` method is called or the IO::Uncompress::Gunzip object is destroyed.

  This parameter defaults to 0.

- `MultiStream => 0|1`

  Allows multiple concatenated compressed streams to be treated as a single compressed stream. Decompression will stop once either the end of the file/buffer is reached, an error is encountered (premature eof, corrupt compressed data) or the end of a stream is not immediately followed by the start of another stream.

  This parameter defaults to 0.

- `Prime => $string`

  This option will uncompress the contents of `$string` before processing the input file/buffer.

  This option can be useful when the compressed data is embedded in another file/data structure and it is not possible to work out where the compressed data begins without having to read the first few bytes. If this is the case, the uncompression can be *primed* with these bytes using this option.

- `Transparent => 0|1`

If this option is set and the input file/buffer is not compressed data, the module will allow reading of it anyway.

In addition, if the input file/buffer does contain compressed data and there is non-compressed data immediately following it, setting this option will make this module treat the whole file/buffer as a single data stream.

This option defaults to 1.

- `BlockSize => $num`

  When reading the compressed input data, IO::Uncompress::Gunzip will read it in blocks of `$num` bytes.

  This option defaults to 4096.

- `InputLength => $size`

  When present this option will limit the number of compressed bytes read from the input file/buffer to `$size`. This option can be used in the situation where there is useful data directly after the compressed data stream and you know beforehand the exact length of the compressed data stream.

  This option is mostly used when reading from a filehandle, in which case the file pointer will be left pointing to the first byte directly after the compressed data stream.

  This option defaults to off.

- `Append => 0|1`

  This option controls what the `read` method does with uncompressed data.

  If set to 1, all uncompressed data will be appended to the output parameter of the `read` method.

  If set to 0, the contents of the output parameter of the `read` method will be overwritten by the uncompressed data.

  Defaults to 0.

- `Strict => 0|1`

  This option controls whether the extra checks defined below are used when carrying out the decompression. When Strict is on, the extra tests are carried out, when Strict is off they are not.

  The default for this option is off.

  1. If the FHCRC bit is set in the gzip FLG header byte, the CRC16 bytes in the header must match the crc16 value of the gzip header actually read.
  2. If the gzip header contains a name field (FNAME) it consists solely of ISO 8859-1 characters.
  3. If the gzip header contains a comment field (FCOMMENT) it consists solely of ISO 8859-1 characters plus line-feed.
  4. If the gzip FEXTRA header field is present it must conform to the sub-field structure as defined in RFC 1952.
  5. The CRC32 and ISIZE trailer fields must be present.

6. The value of the CRC32 field read must match the crc32 value of the uncompressed data actually contained in the gzip file.
7. The value of the ISIZE fields read must match the length of the uncompressed data actually read from the file.

- `ParseExtra => 0|1` If the gzip FEXTRA header field is present and this option is set, it will force the module to check that it conforms to the sub-field structure as defined in RFC 1952.

  If the `Strict` is on it will automatically enable this option.

  Defaults to 0.

Methods

read

Usage is

```
$status = $z->read($buffer)
```

Reads a block of compressed data (the size of the compressed block is determined by the `Buffer` option in the constructor), uncompresses it and writes any uncompressed data into `$buffer`. If the `Append` parameter is set in the constructor, the uncompressed data will be appended to the `$buffer` parameter. Otherwise `$buffer` will be overwritten.

Returns the number of uncompressed bytes written to `$buffer`, zero if eof or a negative number on error.

read

Usage is

```
$status = $z->read($buffer, $length)
$status = $z->read($buffer, $length, $offset)

$status = read($z, $buffer, $length)
$status = read($z, $buffer, $length, $offset)
```

Attempt to read `$length` bytes of uncompressed data into `$buffer`.

The main difference between this form of the `read` method and the previous one, is that this one will attempt to return *exactly* `$length` bytes. The only circumstances that this function will not is if end-of-file or an IO error is encountered.

Returns the number of uncompressed bytes written to `$buffer`, zero if eof or a negative number on error.

getline

Usage is

```
$line = $z->getline()
$line = <$z>
```

Reads a single line.

This method fully supports the use of the variable `$/` (or `$INPUT_RECORD_SEPARATOR` or `$RS` when `English` is in use) to determine what constitutes an end of line. Paragraph mode, record mode and file slurp mode are all supported.

getc

Usage is

```
$char = $z->getc()
```

Read a single character.

ungetc

Usage is

```
$char = $z->ungetc($string)
```

inflateSync

Usage is

```
$status = $z->inflateSync()
```

TODO

getHeaderInfo

Usage is

```
$hdr  = $z->getHeaderInfo();
@hdrs = $z->getHeaderInfo();
```

This method returns either a hash reference (in scalar context) or a list or hash references (in array context) that contains information about each of the header fields in the compressed data stream(s).

- Name

  The contents of the Name header field, if present. If no name is present, the value will be undef. Note this is different from a zero length name, which will return an empty string.

- Comment

  The contents of the Comment header field, if present. If no comment is present, the value will be undef. Note this is different from a zero length comment, which will return an empty string.

tell

Usage is

```
$z->tell()
tell $z
```

Returns the uncompressed file offset.

eof

Usage is

```
$z->eof();
eof($z);
```

Returns true if the end of the compressed input stream has been reached.

seek

```
$z->seek($position, $whence);
seek($z, $position, $whence);
```

Provides a sub-set of the `seek` functionality, with the restriction that it is only legal to seek forward in the input file/buffer. It is a fatal error to attempt to seek backward.

Note that the implementation of `seek` in this module does not provide true random access to a compressed file/buffer. It works by uncompressing data from the current offset in the file/buffer until it reaches the uncompressed offset specified in the parameters to `seek`. For very small files this may be acceptable behaviour. For large files it may cause an unacceptable delay.

The `$whence` parameter takes one the usual values, namely SEEK_SET, SEEK_CUR or SEEK_END.

Returns 1 on success, 0 on failure.

binmode

Usage is

```
$z->binmode
binmode $z ;
```

This is a noop provided for completeness.

opened

```
$z->opened()
```

Returns true if the object currently refers to a opened file/buffer.

autoflush

```
my $prev = $z->autoflush()
my $prev = $z->autoflush(EXPR)
```

If the `$z` object is associated with a file or a filehandle, this method returns the current autoflush setting for the underlying filehandle. If `EXPR` is present, and is non-zero, it will enable flushing after every write/print operation.

If `$z` is associated with a buffer, this method has no effect and always returns `undef`.

Note that the special variable `$|` **cannot** be used to set or retrieve the autoflush setting.

input_line_number

```
$z->input_line_number()
$z->input_line_number(EXPR)
```

Returns the current uncompressed line number. If `EXPR` is present it has the effect of setting the line number. Note that setting the line number does not change the current position within the file/buffer being read.

The contents of `$/` are used to determine what constitutes a line terminator.

fileno

```
$z->fileno()
fileno($z)
```

If the `$z` object is associated with a file or a filehandle, `fileno` will return the underlying file descriptor. Once the `close` method is called `fileno` will return `undef`.

If the `$z` object is associated with a buffer, this method will return `undef`.

close

```
$z->close() ;
close $z ;
```

Closes the output file/buffer.

For most versions of Perl this method will be automatically invoked if the IO::Uncompress::Gunzip object is destroyed (either explicitly or by the variable with the reference to the object going out of scope). The exceptions are Perl versions 5.005 through 5.00504 and 5.8.0. In these cases, the `close` method will be called automatically, but not until global destruction of all live objects when the program is terminating.

Therefore, if you want your scripts to be able to run on all versions of Perl, you should call `close` explicitly and not rely on automatic closing.

Returns true on success, otherwise 0.

If the `AutoClose` option has been enabled when the IO::Uncompress::Gunzip object was created, and the object is associated with a file, the underlying file will also be closed.

nextStream

Usage is

```
my $status = $z->nextStream();
```

Skips to the next compressed data stream in the input file/buffer. If a new compressed data stream is found, the eof marker will be cleared and `$.` will be reset to 0.

Returns 1 if a new stream was found, 0 if none was found, and -1 if an error was encountered.

trailingData

Usage is

```
my $data = $z->trailingData();
```

Returns the data, if any, that is present immediately after the compressed data stream once uncompression is complete. It only makes sense to call this method once the end of the compressed data stream has been encountered.

This option can be used when there is useful information immediately following the compressed data stream, and you don't know the length of the compressed data stream.

If the input is a buffer, `trailingData` will return everything from the end of the compressed data stream to the end of the buffer.

If the input is a filehandle, `trailingData` will return the data that is left in the filehandle input buffer once the end of the compressed data stream has been reached. You can then use the filehandle to read the rest of the input file.

Don't bother using `trailingData` if the input is a filename.

If you know the length of the compressed data stream before you start uncompressing, you can avoid having to use `trailingData` by setting the `InputLength` option in the constructor.

Importing

No symbolic constants are required by IO::Uncompress::Gunzip at present.

- :all

  Imports `gunzip` and `$GunzipError`. Same as doing this

  ```
  use IO::Uncompress::Gunzip qw(gunzip $GunzipError) ;
  ```

EXAMPLES

Working with Net::FTP

See IO::Compress::FAQ

SUPPORT

General feedback/questions/bug reports should be sent to https://github.com/pmqs/IO-Compress/issues (preferred) or https://rt.cpan.org/Public/Dist/Display.html?Name=IO-Compress.

SEE ALSO

Compress::Zlib, *IO::Compress::Gzip*, *IO::Compress::Deflate*, *IO::Uncompress::Inflate*, *IO::Compress::RawDeflate*, *IO::Uncompress::RawInflate*, *IO::Compress::Bzip2*, *IO::Uncompress::Bunzip2*, IO::Compress::Lzma, IO::Uncompress::UnLzma, IO::Compress::Xz, IO::Uncompress::UnXz, IO::Compress::Lzip, IO::Uncompress::UnLzip, IO::Compress::Lzop, IO::Uncompress::UnLzop, IO::Compress::Lzf, IO::Uncompress::UnLzf, IO::Compress::Zstd, IO::Uncompress::UnZstd, *IO::Uncompress::AnyInflate*, *IO::Uncompress::AnyUncompress*

IO::Compress::FAQ

File::GlobMapper, Archive::Zip, *Archive::Tar*, *IO::Zlib*

For RFC 1950, 1951 and 1952 see https://datatracker.ietf.org/doc/html/rfc1950, https://datatracker.ietf.org/doc/html/rfc1951 and https://datatracker.ietf.org/doc/html/rfc1952

The *zlib* compression library was written by Jean-loup Gailly `gzip@prep.ai.mit.edu` and Mark Adler `madler@alumni.caltech.edu`.

The primary site for the *zlib* compression library is http://www.zlib.org.

The primary site for the *zlib-ng* compression library is https://github.com/zlib-ng/zlib-ng.

The primary site for gzip is http://www.gzip.org.

AUTHOR

This module was written by Paul Marquess, `pmqs@cpan.org`.

MODIFICATION HISTORY

See the Changes file.

COPYRIGHT AND LICENSE

Copyright (c) 2005-2026 Paul Marquess. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

IO::Uncompress::Inflate

Read RFC 1950 files/buffers

SYNOPSIS

```
use IO::Uncompress::Inflate qw(inflate $InflateError) ;

my $status = inflate $input => $output [,OPTS]
    or die "inflate failed: $InflateError\n";

my $z = IO::Uncompress::Inflate->new( $input [OPTS] )
    or die "inflate failed: $InflateError\n";

$status = $z->read($buffer)
$status = $z->read($buffer, $length)
$status = $z->read($buffer, $length, $offset)
$line = $z->getline()
$char = $z->getc()
$char = $z->ungetc()
$char = $z->opened()

$status = $z->inflateSync()

$data = $z->trailingData()
$status = $z->nextStream()
$data = $z->getHeaderInfo()
$z->tell()
$z->seek($position, $whence)
$z->binmode()
$z->fileno()
$z->eof()
$z->close()

$InflateError ;

# IO::File mode

<$z>
read($z, $buffer);
read($z, $buffer, $length);
read($z, $buffer, $length, $offset);
tell($z)
seek($z, $position, $whence)
binmode($z)
fileno($z)
eof($z)
close($z)
```

DESCRIPTION

This module provides a Perl interface that allows the reading of files/buffers that conform to RFC 1950.

For writing RFC 1950 files/buffers, see the companion module IO::Compress::Deflate.

Functional Interface

A top-level function, `inflate`, is provided to carry out "one-shot" uncompression between buffers and/or files. For finer control over the uncompression process, see the "OO Interface" section.

```
use IO::Uncompress::Inflate qw(inflate $InflateError) ;

inflate $input_filename_or_reference => $output_filename_or_
 ↪reference [,OPTS]
    or die "inflate failed: $InflateError\n";
```

The functional interface needs Perl5.005 or better.

inflate $input_filename_or_reference => $output_filename_or_reference [, OPTS]

`inflate` expects at least two parameters, `$input_filename_or_reference` and `$output_filename_or_reference` and zero or more optional parameters (see "Optional Parameters")

The `$input_filename_or_reference` parameter

The parameter, `$input_filename_or_reference`, is used to define the source of the compressed data.

It can take one of the following forms:

- A filename

  If the `$input_filename_or_reference` parameter is a simple scalar, it is assumed to be a filename. This file will be opened for reading and the input data will be read from it.

- A filehandle

  If the `$input_filename_or_reference` parameter is a filehandle, the input data will be read from it. The string '-' can be used as an alias for standard input.

- A scalar reference

  If `$input_filename_or_reference` is a scalar reference, the input data will be read from `$$input_filename_or_reference`.

- An array reference

  If `$input_filename_or_reference` is an array reference, each element in the array must be a filename.

  The input data will be read from each file in turn.

  The complete array will be walked to ensure that it only contains valid filenames before any data is uncompressed.

- An Input FileGlob string

  If `$input_filename_or_reference` is a string that is delimited by the characters "<" and ">" `inflate` will assume that it is an *input fileglob string*. The input is the list of files that match the fileglob.

See *File::GlobMapper* for more details.

If the `$input_filename_or_reference` parameter is any other type, `undef` will be returned.

The `$output_filename_or_reference` parameter

The parameter `$output_filename_or_reference` is used to control the destination of the uncompressed data. This parameter can take one of these forms.

- A filename

  If the `$output_filename_or_reference` parameter is a simple scalar, it is assumed to be a filename. This file will be opened for writing and the uncompressed data will be written to it.

- A filehandle

  If the `$output_filename_or_reference` parameter is a filehandle, the uncompressed data will be written to it. The string '-' can be used as an alias for standard output.

- A scalar reference

  If `$output_filename_or_reference` is a scalar reference, the uncompressed data will be stored in `$$output_filename_or_reference`.

- An Array Reference

  If `$output_filename_or_reference` is an array reference, the uncompressed data will be pushed onto the array.

- An Output FileGlob

  If `$output_filename_or_reference` is a string that is delimited by the characters "<" and ">" `inflate` will assume that it is an *output fileglob string*. The output is the list of files that match the fileglob.

  When `$output_filename_or_reference` is an fileglob string, `$input_filename_or_reference` must also be a fileglob string. Anything else is an error.

  See *File::GlobMapper* for more details.

If the `$output_filename_or_reference` parameter is any other type, `undef` will be returned.

Notes

When `$input_filename_or_reference` maps to multiple compressed files/buffers and `$output_filename_or_reference` is a single file/buffer, after uncompression `$output_filename_or_reference` will contain a concatenation of all the uncompressed data from each of the input files/buffers.

Optional Parameters

The optional parameters for the one-shot function `inflate` are (for the most part) identical to those used with the OO interface defined in the "Constructor Options" section. The exceptions are listed below

- `AutoClose => 0|1`

  This option applies to any input or output data streams to `inflate` that are filehandles.

  If `AutoClose` is specified, and the value is true, it will result in all input and/or output filehandles being closed once `inflate` has completed.

  This parameter defaults to 0.

- `BinModeOut => 0|1`

  This option is now a no-op. All files will be written in binmode.

- `Append => 0|1`

  The behaviour of this option is dependent on the type of output data stream.

  - A Buffer

    If Append is enabled, all uncompressed data will be append to the end of the output buffer. Otherwise the output buffer will be cleared before any uncompressed data is written to it.

  - A Filename

    If Append is enabled, the file will be opened in append mode. Otherwise the contents of the file, if any, will be truncated before any uncompressed data is written to it.

  - A Filehandle

    If Append is enabled, the filehandle will be positioned to the end of the file via a call to `seek` before any uncompressed data is written to it. Otherwise the file pointer will not be moved.

  When Append is specified, and set to true, it will *append* all uncompressed data to the output data stream.

  So when the output is a filehandle it will carry out a seek to the eof before writing any uncompressed data. If the output is a filename, it will be opened for appending. If the output is a buffer, all uncompressed data will be appended to the existing buffer.

  Conversely when Append is not specified, or it is present and is set to false, it will operate as follows.

  When the output is a filename, it will truncate the contents of the file before writing any uncompressed data. If the output is a filehandle its position will not be changed. If the output is a buffer, it will be wiped before any uncompressed data is output.

  Defaults to 0.

- `MultiStream => 0|1`

If the input file/buffer contains multiple compressed data streams, this option will uncompress the whole lot as a single data stream.

Defaults to 0.

- `TrailingData => $scalar`

  Returns the data, if any, that is present immediately after the compressed data stream once uncompression is complete.

  This option can be used when there is useful information immediately following the compressed data stream, and you don't know the length of the compressed data stream.

  If the input is a buffer, `trailingData` will return everything from the end of the compressed data stream to the end of the buffer.

  If the input is a filehandle, `trailingData` will return the data that is left in the filehandle input buffer once the end of the compressed data stream has been reached. You can then use the filehandle to read the rest of the input file.

  Don't bother using `trailingData` if the input is a filename.

  If you know the length of the compressed data stream before you start uncompressing, you can avoid having to use `trailingData` by setting the `InputLength` option.

OneShot Examples

To read the contents of the file `file1.txt.1950` and write the uncompressed data to the file `file1.txt`.

```
use strict ;
use warnings ;
use IO::Uncompress::Inflate qw(inflate $InflateError) ;

my $input = "file1.txt.1950";
my $output = "file1.txt";
inflate $input => $output
    or die "inflate failed: $InflateError\n";
```

To read from an existing Perl filehandle, `$input`, and write the uncompressed data to a buffer, `$buffer`.

```
use strict ;
use warnings ;
use IO::Uncompress::Inflate qw(inflate $InflateError) ;
use IO::File ;

my $input = IO::File->new( "<file1.txt.1950" )
    or die "Cannot open 'file1.txt.1950': $!\n" ;
my $buffer ;
inflate $input => \$buffer
    or die "inflate failed: $InflateError\n";
```

To uncompress all files in the directory "/my/home" that match "*.txt.1950" and store the compressed data in the same directory

```
use strict ;
use warnings ;
use IO::Uncompress::Inflate qw(inflate $InflateError) ;

inflate '</my/home/*.txt.1950>' => '</my/home/#1.txt>'
    or die "inflate failed: $InflateError\n";
```

and if you want to compress each file one at a time, this will do the trick

```
use strict ;
use warnings ;
use IO::Uncompress::Inflate qw(inflate $InflateError) ;

for my $input ( glob "/my/home/*.txt.1950" )
{
    my $output = $input;
    $output =~ s/.1950// ;
    inflate $input => $output
        or die "Error compressing '$input': $InflateError\n";
}
```

OO Interface

Constructor

The format of the constructor for IO::Uncompress::Inflate is shown below

```
my $z = IO::Uncompress::Inflate->new( $input [OPTS] )
    or die "IO::Uncompress::Inflate failed: $InflateError\n";
```

The constructor takes one mandatory parameter, `$input`, defined below, and zero or more `OPTS`, defined in "Constructor Options".

Returns an `IO::Uncompress::Inflate` object on success and undef on failure. The variable `$InflateError` will contain an error message on failure.

If you are running Perl 5.005 or better the object, `$z`, returned from IO::Uncompress::Inflate can be used exactly like an *IO::File* filehandle. This means that all normal input file operations can be carried out with `$z`. For example, to read a line from a compressed file/buffer you can use either of these forms

```
$line = $z->getline();
$line = <$z>;
```

Below is a simple exaple of using the OO interface to read the compressed file `myfile.1950` and write its contents to stdout.

```
my $filename = "myfile.1950";
my $z = IO::Uncompress::Inflate->new($filename)
```

(continues on next page)

(continued from previous page)

```
    or die "IO::Uncompress::Inflate failed: $InflateError\n";

while (<$z>) {
    print $_;
}
$z->close();
```

See “EXAMPLES” for further examples

The mandatory parameter `$input` is used to determine the source of the compressed data. This parameter can take one of three forms.

- A filename

  If the `$input` parameter is a scalar, it is assumed to be a filename. This file will be opened for reading and the compressed data will be read from it.

- A filehandle

  If the `$input` parameter is a filehandle, the compressed data will be read from it. The string ‘-’ can be used as an alias for standard input.

- A scalar reference

  If `$input` is a scalar reference, the compressed data will be read from `$$input`.

Constructor Options

The option names defined below are case insensitive and can be optionally prefixed by a ‘-’. So all of the following are valid

```
-AutoClose
-autoclose
AUTOCLOSE
autoclose
```

OPTS is a combination of the following options:

- `AutoClose => 0|1`

  This option is only valid when the `$input` parameter is a filehandle. If specified, and the value is true, it will result in the file being closed once either the `close` method is called or the IO::Uncompress::Inflate object is destroyed.

  This parameter defaults to 0.

- `MultiStream => 0|1`

  Allows multiple concatenated compressed streams to be treated as a single compressed stream. Decompression will stop once either the end of the file/buffer is reached, an error is encountered (premature eof, corrupt compressed data) or the end of a stream is not immediately followed by the start of another stream.

  This parameter defaults to 0.

- `Prime => $string`

This option will uncompress the contents of `$string` before processing the input file/buffer.

This option can be useful when the compressed data is embedded in another file/data structure and it is not possible to work out where the compressed data begins without having to read the first few bytes. If this is the case, the uncompression can be *primed* with these bytes using this option.

- `Transparent => 0|1`

  If this option is set and the input file/buffer is not compressed data, the module will allow reading of it anyway.

  In addition, if the input file/buffer does contain compressed data and there is non-compressed data immediately following it, setting this option will make this module treat the whole file/buffer as a single data stream.

  This option defaults to 1.

- `BlockSize => $num`

  When reading the compressed input data, IO::Uncompress::Inflate will read it in blocks of `$num` bytes.

  This option defaults to 4096.

- `InputLength => $size`

  When present this option will limit the number of compressed bytes read from the input file/buffer to `$size`. This option can be used in the situation where there is useful data directly after the compressed data stream and you know beforehand the exact length of the compressed data stream.

  This option is mostly used when reading from a filehandle, in which case the file pointer will be left pointing to the first byte directly after the compressed data stream.

  This option defaults to off.

- `Append => 0|1`

  This option controls what the `read` method does with uncompressed data.

  If set to 1, all uncompressed data will be appended to the output parameter of the `read` method.

  If set to 0, the contents of the output parameter of the `read` method will be overwritten by the uncompressed data.

  Defaults to 0.

- `Strict => 0|1`

  This option controls whether the extra checks defined below are used when carrying out the decompression. When Strict is on, the extra tests are carried out, when Strict is off they are not.

  The default for this option is off.

  1. The ADLER32 checksum field must be present.

2. The value of the ADLER32 field read must match the adler32 value of the uncompressed data actually contained in the file.

Methods

read

Usage is

```
$status = $z->read($buffer)
```

Reads a block of compressed data (the size of the compressed block is determined by the `Buffer` option in the constructor), uncompresses it and writes any uncompressed data into `$buffer`. If the `Append` parameter is set in the constructor, the uncompressed data will be appended to the `$buffer` parameter. Otherwise `$buffer` will be overwritten.

Returns the number of uncompressed bytes written to `$buffer`, zero if eof or a negative number on error.

read

Usage is

```
$status = $z->read($buffer, $length)
$status = $z->read($buffer, $length, $offset)

$status = read($z, $buffer, $length)
$status = read($z, $buffer, $length, $offset)
```

Attempt to read `$length` bytes of uncompressed data into `$buffer`.

The main difference between this form of the `read` method and the previous one, is that this one will attempt to return *exactly* `$length` bytes. The only circumstances that this function will not is if end-of-file or an IO error is encountered.

Returns the number of uncompressed bytes written to `$buffer`, zero if eof or a negative number on error.

getline

Usage is

```
$line = $z->getline()
$line = <$z>
```

Reads a single line.

This method fully supports the use of the variable `$/` (or `$INPUT_RECORD_SEPARATOR` or `$RS` when `English` is in use) to determine what constitutes an end of line. Paragraph mode, record mode and file slurp mode are all supported.

getc

Usage is

```
$char = $z->getc()
```

Read a single character.

ungetc

Usage is

```
$char = $z->ungetc($string)
```

inflateSync

Usage is

```
$status = $z->inflateSync()
```

TODO

getHeaderInfo

Usage is

```
$hdr  = $z->getHeaderInfo();
@hdrs = $z->getHeaderInfo();
```

This method returns either a hash reference (in scalar context) or a list or hash references (in array context) that contains information about each of the header fields in the compressed data stream(s).

tell

Usage is

```
$z->tell()
tell $z
```

Returns the uncompressed file offset.

eof

Usage is

```
$z->eof();
eof($z);
```

Returns true if the end of the compressed input stream has been reached.

seek

```
$z->seek($position, $whence);
seek($z, $position, $whence);
```

Provides a sub-set of the `seek` functionality, with the restriction that it is only legal to seek forward in the input file/buffer. It is a fatal error to attempt to seek backward.

Note that the implementation of `seek` in this module does not provide true random access to a compressed file/buffer. It works by uncompressing data from the current offset in the file/buffer until it reaches the uncompressed offset specified in the parameters to `seek`. For very small files this may be acceptable behaviour. For large files it may cause an unacceptable delay.

The `$whence` parameter takes one the usual values, namely SEEK_SET, SEEK_CUR or SEEK_END.

Returns 1 on success, 0 on failure.

binmode

Usage is

```
$z->binmode
binmode $z ;
```

This is a noop provided for completeness.

opened

```
$z->opened()
```

Returns true if the object currently refers to a opened file/buffer.

autoflush

```
my $prev = $z->autoflush()
my $prev = $z->autoflush(EXPR)
```

If the `$z` object is associated with a file or a filehandle, this method returns the current autoflush setting for the underlying filehandle. If `EXPR` is present, and is non-zero, it will enable flushing after every write/print operation.

If `$z` is associated with a buffer, this method has no effect and always returns `undef`.

Note that the special variable `$|` **cannot** be used to set or retrieve the autoflush setting.

input_line_number

```
$z->input_line_number()
$z->input_line_number(EXPR)
```

Returns the current uncompressed line number. If EXPR is present it has the effect of setting the line number. Note that setting the line number does not change the current position within the file/buffer being read.

The contents of `$/` are used to determine what constitutes a line terminator.

fileno

```
$z->fileno()
fileno($z)
```

If the `$z` object is associated with a file or a filehandle, `fileno` will return the underlying file descriptor. Once the `close` method is called `fileno` will return `undef`.

If the `$z` object is associated with a buffer, this method will return `undef`.

close

```
$z->close() ;
close $z ;
```

Closes the output file/buffer.

For most versions of Perl this method will be automatically invoked if the IO::Uncompress::Inflate object is destroyed (either explicitly or by the variable with the reference to the object going out of scope). The exceptions are Perl versions 5.005 through 5.00504 and 5.8.0. In these cases, the `close` method will be called automatically, but not until global destruction of all live objects when the program is terminating.

Therefore, if you want your scripts to be able to run on all versions of Perl, you should call `close` explicitly and not rely on automatic closing.

Returns true on success, otherwise 0.

If the `AutoClose` option has been enabled when the IO::Uncompress::Inflate object was created, and the object is associated with a file, the underlying file will also be closed.

nextStream

Usage is

```
my $status = $z->nextStream();
```

Skips to the next compressed data stream in the input file/buffer. If a new compressed data stream is found, the eof marker will be cleared and `$.` will be reset to 0.

Returns 1 if a new stream was found, 0 if none was found, and -1 if an error was encountered.

trailingData

Usage is

```
my $data = $z->trailingData();
```

Returns the data, if any, that is present immediately after the compressed data stream once uncompression is complete. It only makes sense to call this method once the end of the compressed data stream has been encountered.

This option can be used when there is useful information immediately following the compressed data stream, and you don't know the length of the compressed data stream.

If the input is a buffer, `trailingData` will return everything from the end of the compressed data stream to the end of the buffer.

If the input is a filehandle, `trailingData` will return the data that is left in the filehandle input buffer once the end of the compressed data stream has been reached. You can then use the filehandle to read the rest of the input file.

Don't bother using `trailingData` if the input is a filename.

If you know the length of the compressed data stream before you start uncompressing, you can avoid having to use `trailingData` by setting the `InputLength` option in the constructor.

Importing

No symbolic constants are required by IO::Uncompress::Inflate at present.

- :all

  Imports `inflate` and `$InflateError`. Same as doing this

  ```
  use IO::Uncompress::Inflate qw(inflate $InflateError) ;
  ```

EXAMPLES

Working with Net::FTP

See IO::Compress::FAQ

SUPPORT

General feedback/questions/bug reports should be sent to https://github.com/pmqs/IO-Compress/issues (preferred) or https://rt.cpan.org/Public/Dist/Display.html?Name=IO-Compress.

SEE ALSO

Compress::Zlib, *IO::Compress::Gzip*, *IO::Uncompress::Gunzip*, *IO::Compress::Deflate*, *IO::Compress::RawDeflate*, *IO::Uncompress::RawInflate*, *IO::Compress::Bzip2*, *IO::Uncompress::Bunzip2*, IO::Compress::Lzma, IO::Uncompress::UnLzma, IO::Compress::Xz,

IO::Uncompress::UnXz, IO::Compress::Lzip, IO::Uncompress::UnLzip, IO::Compress::Lzop, IO::Uncompress::UnLzop, IO::Compress::Lzf, IO::Uncompress::UnLzf, IO::Compress::Zstd, IO::Uncompress::UnZstd, *IO::Uncompress::AnyInflate*, *IO::Uncompress::AnyUncompress*

IO::Compress::FAQ

File::GlobMapper, Archive::Zip, *Archive::Tar*, *IO::Zlib*

For RFC 1950, 1951 and 1952 see https://datatracker.ietf.org/doc/html/rfc1950, https://datatracker.ietf.org/doc/html/rfc1951 and https://datatracker.ietf.org/doc/html/rfc1952

The *zlib* compression library was written by Jean-loup Gailly `gzip@prep.ai.mit.edu` and Mark Adler `madler@alumni.caltech.edu`.

The primary site for the *zlib* compression library is http://www.zlib.org.

The primary site for the *zlib-ng* compression library is https://github.com/zlib-ng/zlib-ng.

The primary site for gzip is http://www.gzip.org.

AUTHOR

This module was written by Paul Marquess, `pmqs@cpan.org`.

MODIFICATION HISTORY

See the Changes file.

COPYRIGHT AND LICENSE

Copyright (c) 2005-2026 Paul Marquess. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

IO::Uncompress::RawInflate

Read RFC 1951 files/buffers

SYNOPSIS

```
use IO::Uncompress::RawInflate qw(rawinflate $RawInflateError) ;

my $status = rawinflate $input => $output [,OPTS]
    or die "rawinflate failed: $RawInflateError\n";

my $z = IO::Uncompress::RawInflate->new( $input [OPTS] )
    or die "rawinflate failed: $RawInflateError\n";

$status = $z->read($buffer)
$status = $z->read($buffer, $length)
$status = $z->read($buffer, $length, $offset)
```

(continues on next page)

(continued from previous page)

```
$line = $z->getline()
$char = $z->getc()
$char = $z->ungetc()
$char = $z->opened()

$status = $z->inflateSync()

$data = $z->trailingData()
$status = $z->nextStream()
$data = $z->getHeaderInfo()
$z->tell()
$z->seek($position, $whence)
$z->binmode()
$z->fileno()
$z->eof()
$z->close()

$RawInflateError ;

# IO::File mode

<$z>
read($z, $buffer);
read($z, $buffer, $length);
read($z, $buffer, $length, $offset);
tell($z)
seek($z, $position, $whence)
binmode($z)
fileno($z)
eof($z)
close($z)
```

DESCRIPTION

This module provides a Perl interface that allows the reading of files/buffers that conform to RFC 1951.

For writing RFC 1951 files/buffers, see the companion module IO::Compress::RawDeflate.

Functional Interface

A top-level function, `rawinflate`, is provided to carry out "one-shot" uncompression between buffers and/or files. For finer control over the uncompression process, see the "OO Interface" section.

```
use IO::Uncompress::RawInflate qw(rawinflate $RawInflateError) ;

rawinflate $input_filename_or_reference => $output_filename_or_
 ↪reference [,OPTS]
```

(continues on next page)

(continued from previous page)

```
    or die "rawinflate failed: $RawInflateError\n";
```

The functional interface needs Perl5.005 or better.

rawinflate $input_filename_or_reference => $output_filename_or_reference [, OPTS]

`rawinflate` expects at least two parameters, `$input_filename_or_reference` and `$output_filename_or_reference` and zero or more optional parameters (see "Optional Parameters")

The `$input_filename_or_reference` parameter

The parameter, `$input_filename_or_reference`, is used to define the source of the compressed data.

It can take one of the following forms:

- A filename

  If the `$input_filename_or_reference` parameter is a simple scalar, it is assumed to be a filename. This file will be opened for reading and the input data will be read from it.

- A filehandle

  If the `$input_filename_or_reference` parameter is a filehandle, the input data will be read from it. The string '-' can be used as an alias for standard input.

- A scalar reference

  If `$input_filename_or_reference` is a scalar reference, the input data will be read from `$$input_filename_or_reference`.

- An array reference

  If `$input_filename_or_reference` is an array reference, each element in the array must be a filename.

  The input data will be read from each file in turn.

  The complete array will be walked to ensure that it only contains valid filenames before any data is uncompressed.

- An Input FileGlob string

  If `$input_filename_or_reference` is a string that is delimited by the characters "<" and ">" `rawinflate` will assume that it is an *input fileglob string*. The input is the list of files that match the fileglob.

  See *File::GlobMapper* for more details.

If the `$input_filename_or_reference` parameter is any other type, `undef` will be returned.

The $output_filename_or_reference parameter

The parameter $output_filename_or_reference is used to control the destination of the uncompressed data. This parameter can take one of these forms.

- A filename

  If the $output_filename_or_reference parameter is a simple scalar, it is assumed to be a filename. This file will be opened for writing and the uncompressed data will be written to it.

- A filehandle

  If the $output_filename_or_reference parameter is a filehandle, the uncompressed data will be written to it. The string '-' can be used as an alias for standard output.

- A scalar reference

  If $output_filename_or_reference is a scalar reference, the uncompressed data will be stored in $$output_filename_or_reference.

- An Array Reference

  If $output_filename_or_reference is an array reference, the uncompressed data will be pushed onto the array.

- An Output FileGlob

  If $output_filename_or_reference is a string that is delimited by the characters "<" and ">" rawinflate will assume that it is an *output fileglob string*. The output is the list of files that match the fileglob.

  When $output_filename_or_reference is an fileglob string, $input_filename_or_reference must also be a fileglob string. Anything else is an error.

  See *File::GlobMapper* for more details.

If the $output_filename_or_reference parameter is any other type, undef will be returned.

Notes

When $input_filename_or_reference maps to multiple compressed files/buffers and $output_filename_or_reference is a single file/buffer, after uncompression $output_filename_or_reference will contain a concatenation of all the uncompressed data from each of the input files/buffers.

Optional Parameters

The optional parameters for the one-shot function rawinflate are (for the most part) identical to those used with the OO interface defined in the "Constructor Options" section. The exceptions are listed below

- AutoClose => 0|1

  This option applies to any input or output data streams to rawinflate that are filehandles.

If `AutoClose` is specified, and the value is true, it will result in all input and/or output filehandles being closed once `rawinflate` has completed.

This parameter defaults to 0.

- `BinModeOut => 0|1`

  This option is now a no-op. All files will be written in binmode.

- `Append => 0|1`

  The behaviour of this option is dependent on the type of output data stream.

  - A Buffer

    If Append is enabled, all uncompressed data will be append to the end of the output buffer. Otherwise the output buffer will be cleared before any uncompressed data is written to it.

  - A Filename

    If Append is enabled, the file will be opened in append mode. Otherwise the contents of the file, if any, will be truncated before any uncompressed data is written to it.

  - A Filehandle

    If Append is enabled, the filehandle will be positioned to the end of the file via a call to `seek` before any uncompressed data is written to it. Otherwise the file pointer will not be moved.

  When Append is specified, and set to true, it will *append* all uncompressed data to the output data stream.

  So when the output is a filehandle it will carry out a seek to the eof before writing any uncompressed data. If the output is a filename, it will be opened for appending. If the output is a buffer, all uncompressed data will be appended to the existing buffer.

  Conversely when Append is not specified, or it is present and is set to false, it will operate as follows.

  When the output is a filename, it will truncate the contents of the file before writing any uncompressed data. If the output is a filehandle its position will not be changed. If the output is a buffer, it will be wiped before any uncompressed data is output.

  Defaults to 0.

- `MultiStream => 0|1`

  This option is a no-op.

- `TrailingData => $scalar`

  Returns the data, if any, that is present immediately after the compressed data stream once uncompression is complete.

  This option can be used when there is useful information immediately following the compressed data stream, and you don't know the length of the compressed data stream.

If the input is a buffer, `trailingData` will return everything from the end of the compressed data stream to the end of the buffer.

If the input is a filehandle, `trailingData` will return the data that is left in the filehandle input buffer once the end of the compressed data stream has been reached. You can then use the filehandle to read the rest of the input file.

Don't bother using `trailingData` if the input is a filename.

If you know the length of the compressed data stream before you start uncompressing, you can avoid having to use `trailingData` by setting the `InputLength` option.

OneShot Examples

To read the contents of the file `file1.txt.1951` and write the uncompressed data to the file `file1.txt`.

```
use strict ;
use warnings ;
use IO::Uncompress::RawInflate qw(rawinflate $RawInflateError) ;

my $input = "file1.txt.1951";
my $output = "file1.txt";
rawinflate $input => $output
    or die "rawinflate failed: $RawInflateError\n";
```

To read from an existing Perl filehandle, `$input`, and write the uncompressed data to a buffer, `$buffer`.

```
use strict ;
use warnings ;
use IO::Uncompress::RawInflate qw(rawinflate $RawInflateError) ;
use IO::File ;

my $input = IO::File->new( "<file1.txt.1951" )
    or die "Cannot open 'file1.txt.1951': $!\n" ;
my $buffer ;
rawinflate $input => \$buffer
    or die "rawinflate failed: $RawInflateError\n";
```

To uncompress all files in the directory "/my/home" that match "*.txt.1951" and store the compressed data in the same directory

```
use strict ;
use warnings ;
use IO::Uncompress::RawInflate qw(rawinflate $RawInflateError) ;

rawinflate '</my/home/*.txt.1951>' => '</my/home/#1.txt>'
    or die "rawinflate failed: $RawInflateError\n";
```

and if you want to compress each file one at a time, this will do the trick

```
use strict ;
use warnings ;
use IO::Uncompress::RawInflate qw(rawinflate $RawInflateError) ;

for my $input ( glob "/my/home/*.txt.1951" )
{
    my $output = $input;
    $output =~ s/.1951// ;
    rawinflate $input => $output
        or die "Error compressing '$input': $RawInflateError\n";
}
```

OO Interface

Constructor

The format of the constructor for IO::Uncompress::RawInflate is shown below

```
my $z = IO::Uncompress::RawInflate->new( $input [OPTS] )
    or die "IO::Uncompress::RawInflate failed: $RawInflateError\n";
```

The constructor takes one mandatory parameter, `$input`, defined below, and zero or more `OPTS`, defined in "Constructor Options".

Returns an `IO::Uncompress::RawInflate` object on success and undef on failure. The variable `$RawInflateError` will contain an error message on failure.

If you are running Perl 5.005 or better the object, `$z`, returned from IO::Uncompress::RawInflate can be used exactly like an *IO::File* filehandle. This means that all normal input file operations can be carried out with `$z`. For example, to read a line from a compressed file/buffer you can use either of these forms

```
$line = $z->getline();
$line = <$z>;
```

Below is a simple exaple of using the OO interface to read the compressed file `myfile.1951` and write its contents to stdout.

```
my $filename = "myfile.1951";
my $z = IO::Uncompress::RawInflate->new($filename)
    or die "IO::Uncompress::RawInflate failed: $RawInflateError\n";

while (<$z>) {
    print $_;
}
$z->close();
```

See "EXAMPLES" for further examples

The mandatory parameter `$input` is used to determine the source of the compressed data. This parameter can take one of three forms.

- A filename

If the `$input` parameter is a scalar, it is assumed to be a filename. This file will be opened for reading and the compressed data will be read from it.

- A filehandle

  If the `$input` parameter is a filehandle, the compressed data will be read from it. The string '-' can be used as an alias for standard input.

- A scalar reference

  If `$input` is a scalar reference, the compressed data will be read from `$$input`.

Constructor Options

The option names defined below are case insensitive and can be optionally prefixed by a '-'. So all of the following are valid

```
-AutoClose
-autoclose
AUTOCLOSE
autoclose
```

OPTS is a combination of the following options:

- `AutoClose => 0|1`

  This option is only valid when the `$input` parameter is a filehandle. If specified, and the value is true, it will result in the file being closed once either the `close` method is called or the IO::Uncompress::RawInflate object is destroyed.

  This parameter defaults to 0.

- `MultiStream => 0|1`

  Allows multiple concatenated compressed streams to be treated as a single compressed stream. Decompression will stop once either the end of the file/buffer is reached, an error is encountered (premature eof, corrupt compressed data) or the end of a stream is not immediately followed by the start of another stream.

  This parameter defaults to 0.

- `Prime => $string`

  This option will uncompress the contents of `$string` before processing the input file/buffer.

  This option can be useful when the compressed data is embedded in another file/data structure and it is not possible to work out where the compressed data begins without having to read the first few bytes. If this is the case, the uncompression can be *primed* with these bytes using this option.

- `Transparent => 0|1`

  If this option is set and the input file/buffer is not compressed data, the module will allow reading of it anyway.

  In addition, if the input file/buffer does contain compressed data and there is non-compressed data immediately following it, setting this option will make this module treat the whole file/buffer as a single data stream.

This option defaults to 1.

- `BlockSize => $num`

  When reading the compressed input data, IO::Uncompress::RawInflate will read it in blocks of `$num` bytes.

  This option defaults to 4096.

- `InputLength => $size`

  When present this option will limit the number of compressed bytes read from the input file/buffer to `$size`. This option can be used in the situation where there is useful data directly after the compressed data stream and you know beforehand the exact length of the compressed data stream.

  This option is mostly used when reading from a filehandle, in which case the file pointer will be left pointing to the first byte directly after the compressed data stream.

  This option defaults to off.

- `Append => 0|1`

  This option controls what the `read` method does with uncompressed data.

  If set to 1, all uncompressed data will be appended to the output parameter of the `read` method.

  If set to 0, the contents of the output parameter of the `read` method will be overwritten by the uncompressed data.

  Defaults to 0.

- `Strict => 0|1`

  This option is a no-op.

Methods

read

Usage is

```
$status = $z->read($buffer)
```

Reads a block of compressed data (the size of the compressed block is determined by the `Buffer` option in the constructor), uncompresses it and writes any uncompressed data into `$buffer`. If the Append parameter is set in the constructor, the uncompressed data will be appended to the `$buffer` parameter. Otherwise `$buffer` will be overwritten.

Returns the number of uncompressed bytes written to `$buffer`, zero if eof or a negative number on error.

read

Usage is

```
$status = $z->read($buffer, $length)
$status = $z->read($buffer, $length, $offset)

$status = read($z, $buffer, $length)
$status = read($z, $buffer, $length, $offset)
```

Attempt to read `$length` bytes of uncompressed data into `$buffer`.

The main difference between this form of the `read` method and the previous one, is that this one will attempt to return *exactly* `$length` bytes. The only circumstances that this function will not is if end-of-file or an IO error is encountered.

Returns the number of uncompressed bytes written to `$buffer`, zero if eof or a negative number on error.

getline

Usage is

```
$line = $z->getline()
$line = <$z>
```

Reads a single line.

This method fully supports the use of the variable `$/` (or `$INPUT_RECORD_SEPARATOR` or `$RS` when `English` is in use) to determine what constitutes an end of line. Paragraph mode, record mode and file slurp mode are all supported.

getc

Usage is

```
$char = $z->getc()
```

Read a single character.

ungetc

Usage is

```
$char = $z->ungetc($string)
```

inflateSync

Usage is

```
$status = $z->inflateSync()
```

TODO

getHeaderInfo

Usage is

```
$hdr  = $z->getHeaderInfo();
@hdrs = $z->getHeaderInfo();
```

This method returns either a hash reference (in scalar context) or a list or hash references (in array context) that contains information about each of the header fields in the compressed data stream(s).

tell

Usage is

```
$z->tell()
tell $z
```

Returns the uncompressed file offset.

eof

Usage is

```
$z->eof();
eof($z);
```

Returns true if the end of the compressed input stream has been reached.

seek

```
$z->seek($position, $whence);
seek($z, $position, $whence);
```

Provides a sub-set of the `seek` functionality, with the restriction that it is only legal to seek forward in the input file/buffer. It is a fatal error to attempt to seek backward.

Note that the implementation of `seek` in this module does not provide true random access to a compressed file/buffer. It works by uncompressing data from the current offset in the file/buffer until it reaches the uncompressed offset specified in the parameters to `seek`. For very small files this may be acceptable behaviour. For large files it may cause an unacceptable delay.

The `$whence` parameter takes one the usual values, namely SEEK_SET, SEEK_CUR or SEEK_END.

Returns 1 on success, 0 on failure.

binmode

Usage is

```
$z->binmode
binmode $z ;
```

This is a noop provided for completeness.

opened

```
$z->opened()
```

Returns true if the object currently refers to a opened file/buffer.

autoflush

```
my $prev = $z->autoflush()
my $prev = $z->autoflush(EXPR)
```

If the `$z` object is associated with a file or a filehandle, this method returns the current autoflush setting for the underlying filehandle. If `EXPR` is present, and is non-zero, it will enable flushing after every write/print operation.

If `$z` is associated with a buffer, this method has no effect and always returns `undef`.

Note that the special variable `$|` **cannot** be used to set or retrieve the autoflush setting.

input_line_number

```
$z->input_line_number()
$z->input_line_number(EXPR)
```

Returns the current uncompressed line number. If `EXPR` is present it has the effect of setting the line number. Note that setting the line number does not change the current position within the file/buffer being read.

The contents of `$/` are used to determine what constitutes a line terminator.

fileno

```
$z->fileno()
fileno($z)
```

If the `$z` object is associated with a file or a filehandle, `fileno` will return the underlying file descriptor. Once the `close` method is called `fileno` will return `undef`.

If the `$z` object is associated with a buffer, this method will return `undef`.

close

```
$z->close() ;
close $z ;
```

Closes the output file/buffer.

For most versions of Perl this method will be automatically invoked if the IO::Uncompress::RawInflate object is destroyed (either explicitly or by the variable with the reference to the object going out of scope). The exceptions are Perl versions 5.005 through 5.00504 and 5.8.0. In these cases, the `close` method will be called automatically, but not until global destruction of all live objects when the program is terminating.

Therefore, if you want your scripts to be able to run on all versions of Perl, you should call `close` explicitly and not rely on automatic closing.

Returns true on success, otherwise 0.

If the `AutoClose` option has been enabled when the IO::Uncompress::RawInflate object was created, and the object is associated with a file, the underlying file will also be closed.

nextStream

Usage is

```
my $status = $z->nextStream();
```

Skips to the next compressed data stream in the input file/buffer. If a new compressed data stream is found, the eof marker will be cleared and `$.` will be reset to 0.

Returns 1 if a new stream was found, 0 if none was found, and -1 if an error was encountered.

trailingData

Usage is

```
my $data = $z->trailingData();
```

Returns the data, if any, that is present immediately after the compressed data stream once uncompression is complete. It only makes sense to call this method once the end of the compressed data stream has been encountered.

This option can be used when there is useful information immediately following the compressed data stream, and you don't know the length of the compressed data stream.

If the input is a buffer, `trailingData` will return everything from the end of the compressed data stream to the end of the buffer.

If the input is a filehandle, `trailingData` will return the data that is left in the filehandle input buffer once the end of the compressed data stream has been reached. You can then use the filehandle to read the rest of the input file.

Don't bother using `trailingData` if the input is a filename.

If you know the length of the compressed data stream before you start uncompressing, you can avoid having to use `trailingData` by setting the `InputLength` option in the constructor.

Importing

No symbolic constants are required by IO::Uncompress::RawInflate at present.

- :all

  Imports `rawinflate` and `$RawInflateError`. Same as doing this

```
use IO::Uncompress::RawInflate qw(rawinflate $RawInflateError) ;
```

EXAMPLES

Working with Net::FTP

See IO::Compress::FAQ

SUPPORT

General feedback/questions/bug reports should be sent to https://github.com/pmqs/IO-Compress/issues (preferred) or https://rt.cpan.org/Public/Dist/Display.html?Name=IO-Compress.

SEE ALSO

Compress::Zlib, *IO::Compress::Gzip*, *IO::Uncompress::Gunzip*, *IO::Compress::Deflate*, *IO::Uncompress::Inflate*, *IO::Compress::RawDeflate*, *IO::Compress::Bzip2*, *IO::Uncompress::Bunzip2*, IO::Compress::Lzma, IO::Uncompress::UnLzma, IO::Compress::Xz, IO::Uncompress::UnXz, IO::Compress::Lzip, IO::Uncompress::UnLzip, IO::Compress::Lzop, IO::Uncompress::UnLzop, IO::Compress::Lzf, IO::Uncompress::UnLzf, IO::Compress::Zstd, IO::Uncompress::UnZstd, *IO::Uncompress::AnyInflate*, *IO::Uncompress::AnyUncompress*

IO::Compress::FAQ

File::GlobMapper, Archive::Zip, *Archive::Tar*, *IO::Zlib*

For RFC 1950, 1951 and 1952 see https://datatracker.ietf.org/doc/html/rfc1950, https://datatracker.ietf.org/doc/html/rfc1951 and https://datatracker.ietf.org/doc/html/rfc1952

The *zlib* compression library was written by Jean-loup Gailly `gzip@prep.ai.mit.edu` and Mark Adler `madler@alumni.caltech.edu`.

The primary site for the *zlib* compression library is http://www.zlib.org.

The primary site for the *zlib-ng* compression library is https://github.com/zlib-ng/zlib-ng.

The primary site for gzip is http://www.gzip.org.

AUTHOR

This module was written by Paul Marquess, pmqs@cpan.org.

MODIFICATION HISTORY

See the Changes file.

COPYRIGHT AND LICENSE

Copyright (c) 2005-2026 Paul Marquess. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

IO::Uncompress::Unzip

Read zip files/buffers

SYNOPSIS

```
use IO::Uncompress::Unzip qw(unzip $UnzipError) ;

my $status = unzip $input => $output [,OPTS]
    or die "unzip failed: $UnzipError\n";

my $z = IO::Uncompress::Unzip->new( $input [OPTS] )
    or die "unzip failed: $UnzipError\n";

$status = $z->read($buffer)
$status = $z->read($buffer, $length)
$status = $z->read($buffer, $length, $offset)
$line = $z->getline()
$char = $z->getc()
$char = $z->ungetc()
$char = $z->opened()

$status = $z->inflateSync()

$data = $z->trailingData()
$status = $z->nextStream()
$data = $z->getHeaderInfo()
$z->tell()
$z->seek($position, $whence)
$z->binmode()
$z->fileno()
$z->eof()
$z->close()

$UnzipError ;
```

(continues on next page)

(continued from previous page)

```
# IO::File mode

<$z>
read($z, $buffer);
read($z, $buffer, $length);
read($z, $buffer, $length, $offset);
tell($z)
seek($z, $position, $whence)
binmode($z)
fileno($z)
eof($z)
close($z)
```

DESCRIPTION

This module provides a Perl interface that allows the reading of zlib files/buffers.

For writing zip files/buffers, see the companion module IO::Compress::Zip.

The primary purpose of this module is to provide *streaming* read access to zip files and buffers.

At present the following compression methods are supported by IO::Uncompress::Unzip

- Store (0)
- Deflate (8)
- Bzip2 (12)

  To read Bzip2 content, the module `IO::Uncompress::Bunzip2` must be installed.
- Lzma (14)

  To read LZMA content, the module `IO::Uncompress::UnLzma` must be installed.
- Xz (95)

  To read Xz content, the module `IO::Uncompress::UnXz` must be installed.
- Zstandard (93)

  To read Zstandard content, the module `IO::Uncompress::UnZstd` must be installed.

Functional Interface

A top-level function, `unzip`, is provided to carry out "one-shot" uncompression between buffers and/or files. For finer control over the uncompression process, see the "OO Interface" section.

```
use IO::Uncompress::Unzip qw(unzip $UnzipError) ;

unzip $input_filename_or_reference => $output_filename_or_reference
```

(continues on next page)

(continued from previous page)

```
↪[,OPTS]
    or die "unzip failed: $UnzipError\n";
```

The functional interface needs Perl5.005 or better.

unzip $input_filename_or_reference => $output_filename_or_reference [, OPTS]

`unzip` expects at least two parameters, `$input_filename_or_reference` and `$output_filename_or_reference` and zero or more optional parameters (see "Optional Parameters")

The `$input_filename_or_reference` parameter

The parameter, `$input_filename_or_reference`, is used to define the source of the compressed data.

It can take one of the following forms:

- A filename

  If the `$input_filename_or_reference` parameter is a simple scalar, it is assumed to be a filename. This file will be opened for reading and the input data will be read from it.

- A filehandle

  If the `$input_filename_or_reference` parameter is a filehandle, the input data will be read from it. The string '-' can be used as an alias for standard input.

- A scalar reference

  If `$input_filename_or_reference` is a scalar reference, the input data will be read from `$$input_filename_or_reference`.

- An array reference

  If `$input_filename_or_reference` is an array reference, each element in the array must be a filename.

  The input data will be read from each file in turn.

  The complete array will be walked to ensure that it only contains valid filenames before any data is uncompressed.

- An Input FileGlob string

  If `$input_filename_or_reference` is a string that is delimited by the characters "<" and ">" `unzip` will assume that it is an *input fileglob string*. The input is the list of files that match the fileglob.

  See *File::GlobMapper* for more details.

If the `$input_filename_or_reference` parameter is any other type, `undef` will be returned.

The $output_filename_or_reference parameter

The parameter $output_filename_or_reference is used to control the destination of the uncompressed data. This parameter can take one of these forms.

- A filename

  If the $output_filename_or_reference parameter is a simple scalar, it is assumed to be a filename. This file will be opened for writing and the uncompressed data will be written to it.

- A filehandle

  If the $output_filename_or_reference parameter is a filehandle, the uncompressed data will be written to it. The string '-' can be used as an alias for standard output.

- A scalar reference

  If $output_filename_or_reference is a scalar reference, the uncompressed data will be stored in $$output_filename_or_reference.

- An Array Reference

  If $output_filename_or_reference is an array reference, the uncompressed data will be pushed onto the array.

- An Output FileGlob

  If $output_filename_or_reference is a string that is delimited by the characters "<" and ">" unzip will assume that it is an *output fileglob string*. The output is the list of files that match the fileglob.

  When $output_filename_or_reference is an fileglob string, $input_filename_or_reference must also be a fileglob string. Anything else is an error.

  See *File::GlobMapper* for more details.

If the $output_filename_or_reference parameter is any other type, undef will be returned.

Notes

When $input_filename_or_reference maps to multiple compressed files/buffers and $output_filename_or_reference is a single file/buffer, after uncompression $output_filename_or_reference will contain a concatenation of all the uncompressed data from each of the input files/buffers.

Optional Parameters

The optional parameters for the one-shot function unzip are (for the most part) identical to those used with the OO interface defined in the "Constructor Options" section. The exceptions are listed below

- AutoClose => 0|1

  This option applies to any input or output data streams to unzip that are filehandles.

If `AutoClose` is specified, and the value is true, it will result in all input and/or output filehandles being closed once `unzip` has completed.

This parameter defaults to 0.

- `BinModeOut => 0|1`

  This option is now a no-op. All files will be written in binmode.

- `Append => 0|1`

  The behaviour of this option is dependent on the type of output data stream.

  - A Buffer

    If Append is enabled, all uncompressed data will be append to the end of the output buffer. Otherwise the output buffer will be cleared before any uncompressed data is written to it.

  - A Filename

    If Append is enabled, the file will be opened in append mode. Otherwise the contents of the file, if any, will be truncated before any uncompressed data is written to it.

  - A Filehandle

    If Append is enabled, the filehandle will be positioned to the end of the file via a call to `seek` before any uncompressed data is written to it. Otherwise the file pointer will not be moved.

  When Append is specified, and set to true, it will *append* all uncompressed data to the output data stream.

  So when the output is a filehandle it will carry out a seek to the eof before writing any uncompressed data. If the output is a filename, it will be opened for appending. If the output is a buffer, all uncompressed data will be appended to the existing buffer.

  Conversely when Append is not specified, or it is present and is set to false, it will operate as follows.

  When the output is a filename, it will truncate the contents of the file before writing any uncompressed data. If the output is a filehandle its position will not be changed. If the output is a buffer, it will be wiped before any uncompressed data is output.

  Defaults to 0.

- `MultiStream => 0|1`

  If the input file/buffer contains multiple compressed data streams, this option will uncompress the whole lot as a single data stream.

  Defaults to 0.

- `TrailingData => $scalar`

  Returns the data, if any, that is present immediately after the compressed data stream once uncompression is complete.

This option can be used when there is useful information immediately following the compressed data stream, and you don't know the length of the compressed data stream.

If the input is a buffer, `trailingData` will return everything from the end of the compressed data stream to the end of the buffer.

If the input is a filehandle, `trailingData` will return the data that is left in the filehandle input buffer once the end of the compressed data stream has been reached. You can then use the filehandle to read the rest of the input file.

Don't bother using `trailingData` if the input is a filename.

If you know the length of the compressed data stream before you start uncompressing, you can avoid having to use `trailingData` by setting the `InputLength` option.

OneShot Examples

Say you have a zip file, `file1.zip`, that only contains a single member, you can read it and write the uncompressed data to the file `file1.txt` like this.

```
use strict ;
use warnings ;
use IO::Uncompress::Unzip qw(unzip $UnzipError) ;

my $input = "file1.zip";
my $output = "file1.txt";
unzip $input => $output
    or die "unzip failed: $UnzipError\n";
```

If you have a zip file that contains multiple members and want to read a specific member from the file, say `"data1"`, use the `Name` option

```
use strict ;
use warnings ;
use IO::Uncompress::Unzip qw(unzip $UnzipError) ;

my $input = "file1.zip";
my $output = "file1.txt";
unzip $input => $output, Name => "data1"
    or die "unzip failed: $UnzipError\n";
```

Alternatively, if you want to read the `"data1"` member into memory, use a scalar reference for the `output` parameter.

```
use strict ;
use warnings ;
use IO::Uncompress::Unzip qw(unzip $UnzipError) ;

my $input = "file1.zip";
my $output ;
unzip $input => \$output, Name => "data1"
```

(continues on next page)

(continued from previous page)

```
    or die "unzip failed: $UnzipError\n";
# $output now contains the uncompressed data
```

To read from an existing Perl filehandle, `$input`, and write the uncompressed data to a buffer, `$buffer`.

```
use strict ;
use warnings ;
use IO::Uncompress::Unzip qw(unzip $UnzipError) ;
use IO::File ;

my $input = IO::File->new( "<file1.zip" )
    or die "Cannot open 'file1.zip': $!\n" ;
my $buffer ;
unzip $input => \$buffer
    or die "unzip failed: $UnzipError\n";
```

OO Interface

Constructor

The format of the constructor for IO::Uncompress::Unzip is shown below

```
my $z = IO::Uncompress::Unzip->new( $input [OPTS] )
    or die "IO::Uncompress::Unzip failed: $UnzipError\n";
```

The constructor takes one mandatory parameter, `$input`, defined below, and zero or more `OPTS`, defined in “Constructor Options”.

Returns an `IO::Uncompress::Unzip` object on success and undef on failure. The variable `$UnzipError` will contain an error message on failure.

If you are running Perl 5.005 or better the object, `$z`, returned from IO::Uncompress::Unzip can be used exactly like an *IO::File* filehandle. This means that all normal input file operations can be carried out with `$z`. For example, to read a line from a compressed file/buffer you can use either of these forms

```
$line = $z->getline();
$line = <$z>;
```

Below is a simple exaple of using the OO interface to read the compressed file `myfile.zip` and write its contents to stdout.

```
my $filename = "myfile.zip";
my $z = IO::Uncompress::Unzip->new($filename)
    or die "IO::Uncompress::Unzip failed: $UnzipError\n";

while (<$z>) {
    print $_;
}
$z->close();
```

See “EXAMPLES” for further examples

The mandatory parameter `$input` is used to determine the source of the compressed data. This parameter can take one of three forms.

- A filename

  If the `$input` parameter is a scalar, it is assumed to be a filename. This file will be opened for reading and the compressed data will be read from it.

- A filehandle

  If the `$input` parameter is a filehandle, the compressed data will be read from it. The string ‘-’ can be used as an alias for standard input.

- A scalar reference

  If `$input` is a scalar reference, the compressed data will be read from `$$input`.

Constructor Options

The option names defined below are case insensitive and can be optionally prefixed by a ‘-’. So all of the following are valid

```
-AutoClose
-autoclose
AUTOCLOSE
autoclose
```

OPTS is a combination of the following options:

- `Name => "membername"`

  Open “membername” from the zip file for reading.

- `Efs => 0| 1`

  When this option is set to true AND the zip archive being read has the “Language Encoding Flag” (EFS) set, the member name is assumed to be encoded in UTF-8.

  If the member name in the zip archive is not valid UTF-8 when this optionn is true, the script will die with an error message.

  Note that this option only works with Perl 5.8.4 or better.

  This option defaults to **false**.

- `AutoClose => 0|1`

  This option is only valid when the `$input` parameter is a filehandle. If specified, and the value is true, it will result in the file being closed once either the `close` method is called or the IO::Uncompress::Unzip object is destroyed.

  This parameter defaults to 0.

- `MultiStream => 0|1`

  Treats the complete zip file/buffer as a single compressed data stream. When reading in multi-stream mode each member of the zip file/buffer will be uncompressed in turn until the end of the file/buffer is encountered.

This parameter defaults to 0.

- `Prime => $string`

  This option will uncompress the contents of `$string` before processing the input file/buffer.

  This option can be useful when the compressed data is embedded in another file/data structure and it is not possible to work out where the compressed data begins without having to read the first few bytes. If this is the case, the uncompression can be *primed* with these bytes using this option.

- `Transparent => 0|1`

  If this option is set and the input file/buffer is not compressed data, the module will allow reading of it anyway.

  In addition, if the input file/buffer does contain compressed data and there is non-compressed data immediately following it, setting this option will make this module treat the whole file/buffer as a single data stream.

  This option defaults to 1.

- `BlockSize => $num`

  When reading the compressed input data, IO::Uncompress::Unzip will read it in blocks of `$num` bytes.

  This option defaults to 4096.

- `InputLength => $size`

  When present this option will limit the number of compressed bytes read from the input file/buffer to `$size`. This option can be used in the situation where there is useful data directly after the compressed data stream and you know beforehand the exact length of the compressed data stream.

  This option is mostly used when reading from a filehandle, in which case the file pointer will be left pointing to the first byte directly after the compressed data stream.

  This option defaults to off.

- `Append => 0|1`

  This option controls what the `read` method does with uncompressed data.

  If set to 1, all uncompressed data will be appended to the output parameter of the `read` method.

  If set to 0, the contents of the output parameter of the `read` method will be overwritten by the uncompressed data.

  Defaults to 0.

- `Strict => 0|1`

  This option controls whether the extra checks defined below are used when carrying out the decompression. When Strict is on, the extra tests are carried out, when Strict is off they are not.

  The default for this option is off.

Methods

read

Usage is

```
$status = $z->read($buffer)
```

Reads a block of compressed data (the size of the compressed block is determined by the `Buffer` option in the constructor), uncompresses it and writes any uncompressed data into `$buffer`. If the `Append` parameter is set in the constructor, the uncompressed data will be appended to the `$buffer` parameter. Otherwise `$buffer` will be overwritten.

Returns the number of uncompressed bytes written to `$buffer`, zero if eof or a negative number on error.

read

Usage is

```
$status = $z->read($buffer, $length)
$status = $z->read($buffer, $length, $offset)

$status = read($z, $buffer, $length)
$status = read($z, $buffer, $length, $offset)
```

Attempt to read `$length` bytes of uncompressed data into `$buffer`.

The main difference between this form of the `read` method and the previous one, is that this one will attempt to return *exactly* `$length` bytes. The only circumstances that this function will not is if end-of-file or an IO error is encountered.

Returns the number of uncompressed bytes written to `$buffer`, zero if eof or a negative number on error.

getline

Usage is

```
$line = $z->getline()
$line = <$z>
```

Reads a single line.

This method fully supports the use of the variable `$/` (or `$INPUT_RECORD_SEPARATOR` or `$RS` when `English` is in use) to determine what constitutes an end of line. Paragraph mode, record mode and file slurp mode are all supported.

getc

Usage is

```
$char = $z->getc()
```

Read a single character.

ungetc

Usage is

```
$char = $z->ungetc($string)
```

inflateSync

Usage is

```
$status = $z->inflateSync()
```

TODO

getHeaderInfo

Usage is

```
$hdr  = $z->getHeaderInfo();
@hdrs = $z->getHeaderInfo();
```

This method returns either a hash reference (in scalar context) or a list or hash references (in array context) that contains information about each of the header fields in the compressed data stream(s).

tell

Usage is

```
$z->tell()
tell $z
```

Returns the uncompressed file offset.

eof

Usage is

```
$z->eof();
eof($z);
```

Returns true if the end of the compressed input stream has been reached.

seek

```
$z->seek($position, $whence);
seek($z, $position, $whence);
```

Provides a sub-set of the `seek` functionality, with the restriction that it is only legal to seek forward in the input file/buffer. It is a fatal error to attempt to seek backward.

Note that the implementation of `seek` in this module does not provide true random access to a compressed file/buffer. It works by uncompressing data from the current offset in the file/buffer until it reaches the uncompressed offset specified in the parameters to `seek`. For very small files this may be acceptable behaviour. For large files it may cause an unacceptable delay.

The `$whence` parameter takes one the usual values, namely SEEK_SET, SEEK_CUR or SEEK_END.

Returns 1 on success, 0 on failure.

binmode

Usage is

```
$z->binmode
binmode $z ;
```

This is a noop provided for completeness.

opened

```
$z->opened()
```

Returns true if the object currently refers to a opened file/buffer.

autoflush

```
my $prev = $z->autoflush()
my $prev = $z->autoflush(EXPR)
```

If the `$z` object is associated with a file or a filehandle, this method returns the current autoflush setting for the underlying filehandle. If `EXPR` is present, and is non-zero, it will enable flushing after every write/print operation.

If `$z` is associated with a buffer, this method has no effect and always returns `undef`.

Note that the special variable `$|` **cannot** be used to set or retrieve the autoflush setting.

input_line_number

```
$z->input_line_number()
$z->input_line_number(EXPR)
```

Returns the current uncompressed line number. If `EXPR` is present it has the effect of setting the line number. Note that setting the line number does not change the current position within the file/buffer being read.

The contents of `$/` are used to determine what constitutes a line terminator.

fileno

```
$z->fileno()
fileno($z)
```

If the `$z` object is associated with a file or a filehandle, `fileno` will return the underlying file descriptor. Once the `close` method is called `fileno` will return `undef`.

If the `$z` object is associated with a buffer, this method will return `undef`.

close

```
$z->close() ;
close $z ;
```

Closes the output file/buffer.

For most versions of Perl this method will be automatically invoked if the IO::Uncompress::Unzip object is destroyed (either explicitly or by the variable with the reference to the object going out of scope). The exceptions are Perl versions 5.005 through 5.00504 and 5.8.0. In these cases, the `close` method will be called automatically, but not until global destruction of all live objects when the program is terminating.

Therefore, if you want your scripts to be able to run on all versions of Perl, you should call `close` explicitly and not rely on automatic closing.

Returns true on success, otherwise 0.

If the `AutoClose` option has been enabled when the IO::Uncompress::Unzip object was created, and the object is associated with a file, the underlying file will also be closed.

nextStream

Usage is

```
my $status = $z->nextStream();
```

Skips to the next compressed data stream in the input file/buffer. If a new compressed data stream is found, the eof marker will be cleared and `$.` will be reset to 0.

If trailing data is present immediately after the zip archive and the `Transparent` option is enabled, this method will consider that trailing data to be another member of the zip archive.

Returns 1 if a new stream was found, 0 if none was found, and -1 if an error was encountered.

trailingData

Usage is

```
my $data = $z->trailingData();
```

Returns the data, if any, that is present immediately after the compressed data stream once uncompression is complete. It only makes sense to call this method once the end of the compressed data stream has been encountered.

This option can be used when there is useful information immediately following the compressed data stream, and you don't know the length of the compressed data stream.

If the input is a buffer, `trailingData` will return everything from the end of the compressed data stream to the end of the buffer.

If the input is a filehandle, `trailingData` will return the data that is left in the filehandle input buffer once the end of the compressed data stream has been reached. You can then use the filehandle to read the rest of the input file.

Don't bother using `trailingData` if the input is a filename.

If you know the length of the compressed data stream before you start uncompressing, you can avoid having to use `trailingData` by setting the `InputLength` option in the constructor.

Importing

No symbolic constants are required by IO::Uncompress::Unzip at present.

- :all

  Imports `unzip` and `$UnzipError`. Same as doing this

```
use IO::Uncompress::Unzip qw(unzip $UnzipError) ;
```

EXAMPLES

Simple Read

Say you have a zip file, `file1.zip`, that only contains a single member, you can read it and write the uncompressed data to the file `file1.txt` like this.

```
use strict ;
use warnings ;
use IO::Uncompress::Unzip qw(unzip $UnzipError) ;

my $filename = "file1.zip";
my $z = IO::Uncompress::Unzip->new($filename)
    or die "IO::Uncompress::Unzip failed: $UnzipError\n";
open my $out, ">", "file1.txt";

while (<$z>) {
    print $out $_;
}
$z->close();
```

If you have a zip file that contains multiple members and want to read a specific member from the file, say `"data1"`, use the `Name` option when constructing the

```
use strict ;
use warnings ;
use IO::Uncompress::Unzip qw(unzip $UnzipError) ;

my $filename = "file1.zip";
my $z = IO::Uncompress::Unzip->new($filename, Name => "data1")
    or die "IO::Uncompress::Unzip failed: $UnzipError\n";
```

Walking through a zip file

The code below can be used to traverse a zip file, one compressed data stream at a time.

```
use IO::Uncompress::Unzip qw($UnzipError);

my $zipfile = "somefile.zip";
my $u = IO::Uncompress::Unzip->new( $zipfile )
    or die "Cannot open $zipfile: $UnzipError";

my $status;
for ($status = 1; $status > 0; $status = $u->nextStream())
{

    my $name = $u->getHeaderInfo()->{Name};
    warn "Processing member $name\n" ;

    my $buff;
    while (($status = $u->read($buff)) > 0) {
        # Do something here
    }

    last if $status < 0;
}

die "Error processing $zipfile: $!\n"
    if $status < 0 ;
```

Each individual compressed data stream is read until the logical end-of-file is reached. Then `nextStream` is called. This will skip to the start of the next compressed data stream and clear the end-of-file flag.

It is also worth noting that `nextStream` can be called at any time – you don't have to wait until you have exhausted a compressed data stream before skipping to the next one.

Unzipping a complete zip file to disk

Daniel S. Sterling has written a script that uses `IO::Uncompress::UnZip` to read a zip file and unzip its contents to disk.

The script is available from https://gist.github.com/eqhmcow/5389877

Working with Net::FTP

See IO::Compress::FAQ

SUPPORT

General feedback/questions/bug reports should be sent to https://github.com/pmqs/IO-Compress/issues (preferred) or https://rt.cpan.org/Public/Dist/Display.html?Name=IO-Compress.

SEE ALSO

Compress::Zlib, *IO::Compress::Gzip*, *IO::Uncompress::Gunzip*, *IO::Compress::Deflate*, *IO::Uncompress::Inflate*, *IO::Compress::RawDeflate*, *IO::Uncompress::RawInflate*, *IO::Compress::Bzip2*, *IO::Uncompress::Bunzip2*, IO::Compress::Lzma, IO::Uncompress::UnLzma, IO::Compress::Xz, IO::Uncompress::UnXz, IO::Compress::Lzip, IO::Uncompress::UnLzip, IO::Compress::Lzop, IO::Uncompress::UnLzop, IO::Compress::Lzf, IO::Uncompress::UnLzf, IO::Compress::Zstd, IO::Uncompress::UnZstd, *IO::Uncompress::AnyInflate*, *IO::Uncompress::AnyUncompress*

IO::Compress::FAQ

File::GlobMapper, Archive::Zip, *Archive::Tar*, *IO::Zlib*

For RFC 1950, 1951 and 1952 see https://datatracker.ietf.org/doc/html/rfc1950, https://datatracker.ietf.org/doc/html/rfc1951 and https://datatracker.ietf.org/doc/html/rfc1952

The *zlib* compression library was written by Jean-loup Gailly `gzip@prep.ai.mit.edu` and Mark Adler `madler@alumni.caltech.edu`.

The primary site for the *zlib* compression library is http://www.zlib.org.

The primary site for the *zlib-ng* compression library is https://github.com/zlib-ng/zlib-ng.

The primary site for gzip is http://www.gzip.org.

AUTHOR

This module was written by Paul Marquess, `pmqs@cpan.org`.

MODIFICATION HISTORY

See the Changes file.

COPYRIGHT AND LICENSE

Copyright (c) 2005-2026 Paul Marquess. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

IO::Zlib

IO:: style interface to *Compress::Zlib*

SYNOPSIS

With any version of Perl 5 you can use the basic OO interface:

```
use IO::Zlib;

$fh = new IO::Zlib;
if ($fh->open("file.gz", "rb")) {
    print <$fh>;
    $fh->close;
}

$fh = IO::Zlib->new("file.gz", "wb9");
if (defined $fh) {
    print $fh "bar\n";
    $fh->close;
}

$fh = IO::Zlib->new("file.gz", "rb");
if (defined $fh) {
    print <$fh>;
    undef $fh;       # automatically closes the file
}
```

With Perl 5.004 you can also use the TIEHANDLE interface to access compressed files just like ordinary files:

```
use IO::Zlib;

tie *FILE, 'IO::Zlib', "file.gz", "wb";
print FILE "line 1\nline2\n";

tie *FILE, 'IO::Zlib', "file.gz", "rb";
while (<FILE>) { print "LINE: ", $_ };
```

DESCRIPTION

`IO::Zlib` provides an IO:: style interface to *Compress::Zlib* and hence to gzip/zlib compressed files. It provides many of the same methods as the *IO::Handle* interface.

Starting from IO::Zlib version 1.02, IO::Zlib can also use an external `gzip` command. The default behaviour is to try to use an external `gzip` if no `Compress::Zlib` can be loaded, unless explicitly disabled by

```
use IO::Zlib qw(:gzip_external 0);
```

If explicitly enabled by

```
use IO::Zlib qw(:gzip_external 1);
```

then the external `gzip` is used **instead** of `Compress::Zlib`.

CONSTRUCTOR

- new ([ARGS])

  Creates an `IO::Zlib` object. If it receives any parameters, they are passed to the method open; if the open fails, the object is destroyed. Otherwise, it is returned to the caller.

OBJECT METHODS

- open (FILENAME, MODE)

  open takes two arguments. The first is the name of the file to open and the second is the open mode. The mode can be anything acceptable to *Compress::Zlib* and by extension anything acceptable to *zlib* (that basically means POSIX fopen() style mode strings plus an optional number to indicate the compression level).

- opened

  Returns true if the object currently refers to a opened file.

- close

  Close the file associated with the object and disassociate the file from the handle. Done automatically on destroy.

- getc

  Return the next character from the file, or undef if none remain.

- getline

  Return the next line from the file, or undef on end of string. Can safely be called in an array context. Currently ignores $/ ($INPUT_RECORD_SEPARATOR or $RS when *English* is in use) and treats lines as delimited by "\n".

- getlines

  Get all remaining lines from the file. It will croak() if accidentally called in a scalar context.

- print (ARGS...)

  Print ARGS to the file.

- read (BUF, NBYTES, [OFFSET])

  Read some bytes from the file. Returns the number of bytes actually read, 0 on end-of-file, undef on error.

- eof

  Returns true if the handle is currently positioned at end of file?

- seek (OFFSET, WHENCE)

  Seek to a given position in the stream. Not yet supported.

- tell

  Return the current position in the stream, as a numeric offset. Not yet supported.

- setpos (POS)

  Set the current position, using the opaque value returned by `getpos()`. Not yet supported.

- getpos (POS)

  Return the current position in the string, as an opaque object. Not yet supported.

USING THE EXTERNAL GZIP

If the external `gzip` is used, the following opens are used:

```
open(FH, "gzip -dc $filename |")  # for read opens
open(FH, " | gzip > $filename")   # for write opens
```

You can modify the 'commands' for example to hardwire an absolute path by e.g.

```
use IO::Zlib ':gzip_read_open'  => '/some/where/gunzip -c %s |';
use IO::Zlib ':gzip_write_open' => '| /some/where/gzip.exe > %s';
```

The `%s` is expanded to be the filename (`sprintf` is used, so be careful to escape any other % signs). The 'commands' are checked for sanity - they must contain the `%s`, and the read open must end with the pipe sign, and the write open must begin with the pipe sign.

CLASS METHODS

- has_Compress_Zlib

  Returns true if `Compress::Zlib` is available. Note that this does not mean that `Compress::Zlib` is being used: see "gzip_external" and gzip_used.

- gzip_external

  Undef if an external `gzip` **can** be used if `Compress::Zlib` is not available (see "has_Compress_Zlib"), true if an external `gzip` is explicitly used, false if an external `gzip` must not be used. See "gzip_used".

- gzip_used

  True if an external `gzip` is being used, false if not.

- gzip_read_open

  Return the 'command' being used for opening a file for reading using an external `gzip`.

- gzip_write_open

  Return the 'command' being used for opening a file for writing using an external `gzip`.

DIAGNOSTICS

- IO::Zlib::getlines: must be called in list context

  If you want read lines, you must read in list context.

- IO::Zlib::gzopen_external: mode '...' is illegal

  Use only modes 'rb' or 'wb' or /wb[1-9]/.

- IO::Zlib::import: '...' is illegal

  The known import symbols are the `:gzip_external`, `:gzip_read_open`, and `:gzip_write_open`. Anything else is not recognized.

- IO::Zlib::import: ':gzip_external' requires an argument

  The `:gzip_external` requires one boolean argument.

- IO::Zlib::import: 'gzip_read_open' requires an argument

  The `:gzip_external` requires one string argument.

- IO::Zlib::import: 'gzip_read' '...' is illegal

  The `:gzip_read_open` argument must end with the pipe sign (`|`) and have the `%s` for the filename. See "USING THE EXTERNAL GZIP".

- IO::Zlib::import: 'gzip_write_open' requires an argument

  The `:gzip_external` requires one string argument.

- IO::Zlib::import: 'gzip_write_open' '...' is illegal

  The `:gzip_write_open` argument must begin with the pipe sign (`|`) and have the `%s` for the filename. An output redirect (>) is also often a good idea, depending on your operating system shell syntax. See "USING THE EXTERNAL GZIP".

- IO::Zlib::import: no Compress::Zlib and no external gzip

  Given that we failed to load `Compress::Zlib` and that the use of an external `gzip` was disabled, IO::Zlib has not much chance of working.

- IO::Zlib::open: needs a filename

  No filename, no open.

- IO::Zlib::READ: NBYTES must be specified

  We must know how much to read.

- IO::Zlib::WRITE: too long LENGTH

  The LENGTH must be less than or equal to the buffer size.

SEE ALSO

perlfunc, *"I/O Operators" in perlop*, *IO::Handle*, *Compress::Zlib*

HISTORY

Created by Tom Hughes <tom@compton.nu>.

Support for external gzip added by Jarkko Hietaniemi <jhi@iki.fi>.

COPYRIGHT

Copyright (c) 1998-2004 Tom Hughes <tom@compton.nu>. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

5.2.58 IPC

Namespace containing the following modules:

- IPC::Cmd - finding and running system commands made easy
- IPC::Msg - SysV Msg IPC object class
- IPC::Open2 - open a process for both reading and writing using open2()
- IPC::Open3 - open a process for reading, writing, and error handling using open3()
- IPC::Semaphore - SysV Semaphore IPC object class
- IPC::SharedMem - SysV Shared Memory IPC object class
- IPC::SysV - System V IPC constants and the system calls Perl's builtins do not cover.

IPC::Cmd

finding and running system commands made easy

SYNOPSIS

```
use IPC::Cmd qw[can_run run run_forked];

my $full_path = can_run('wget') or warn 'wget is not installed!';

### commands can be arrayrefs or strings ###
my $cmd = "$full_path -b theregister.co.uk";
my $cmd = [$full_path, '-b', 'theregister.co.uk'];

### in scalar context ###
my $buffer;
if( scalar run( command => $cmd,
                verbose => 0,
                buffer  => \$buffer,
                timeout => 20 )
) {
    print "fetched webpage successfully: $buffer\n";
}
```

(continues on next page)

(continued from previous page)

```
### in list context ###
my( $success, $error_message, $full_buf, $stdout_buf, $stderr_buf )
 ↪=
        run( command => $cmd, verbose => 0 );

if( $success ) {
    print "this is what the command printed:\n";
    print join "", @$full_buf;
}

### run_forked example ###
my $result = run_forked("$full_path -q -O - theregister.co.uk", {
 ↪'timeout' => 20});
if ($result->{'exit_code'} eq 0 && !$result->{'timeout'}) {
    print "this is what wget returned:\n";
    print $result->{'stdout'};
}

### check for features
print "IPC::Open3 available: "   . IPC::Cmd->can_use_ipc_open3;
print "IPC::Run available: "     . IPC::Cmd->can_use_ipc_run;
print "Can capture buffer: "     . IPC::Cmd->can_capture_buffer;

### don't have IPC::Cmd be verbose, ie don't print to stdout or
### stderr when running commands -- default is '0'
$IPC::Cmd::VERBOSE = 0;
```

DESCRIPTION

IPC::Cmd allows you to run commands platform independently, interactively if desired, but have them still work.

The `can_run` function can tell you if a certain binary is installed and if so where, whereas the `run` function can actually execute any of the commands you give it and give you a clear return value, as well as adhere to your verbosity settings.

CLASS METHODS

$ipc_run_version = IPC::Cmd->can_use_ipc_run([VERBOSE])

Utility function that tells you if `IPC::Run` is available. If the `verbose` flag is passed, it will print diagnostic messages if IPC::Run can not be found or loaded.

$ipc_open3_version = IPC::Cmd->can_use_ipc_open3([VERBOSE])

Utility function that tells you if `IPC::Open3` is available. If the verbose flag is passed, it will print diagnostic messages if `IPC::Open3` can not be found or loaded.

$bool = IPC::Cmd->can_capture_buffer

Utility function that tells you if `IPC::Cmd` is capable of capturing buffers in it's current configuration.

$bool = IPC::Cmd->can_use_run_forked

Utility function that tells you if `IPC::Cmd` is capable of providing `run_forked` on the current platform.

FUNCTIONS

$path = can_run(PROGRAM);

`can_run` takes only one argument: the name of a binary you wish to locate. `can_run` works much like the unix binary `which` or the bash command `type`, which scans through your path, looking for the requested binary.

Unlike `which` and `type`, this function is platform independent and will also work on, for example, Win32.

If called in a scalar context it will return the full path to the binary you asked for if it was found, or `undef` if it was not.

If called in a list context and the global variable `$INSTANCES` is a true value, it will return a list of the full paths to instances of the binary where found in `PATH`, or an empty list if it was not found.

$ok | ($ok, $err, $full_buf, $stdout_buff, $stderr_buff) = run(command => COMMAND, [verbose => BOOL, buffer => \$SCALAR, timeout => DIGIT]);

`run` takes 4 arguments:

- command

  This is the command to execute. It may be either a string or an array reference. This is a required argument.

  See "Caveats" for remarks on how commands are parsed and their limitations.

- verbose

  This controls whether all output of a command should also be printed to STDOUT/STDERR or should only be trapped in buffers (NOTE: buffers require IPC::Run to be installed, or your system able to work with *IPC::Open3*).

  It will default to the global setting of `$IPC::Cmd::VERBOSE`, which by default is 0.

- buffer

  This will hold all the output of a command. It needs to be a reference to a scalar. Note that this will hold both the STDOUT and STDERR messages, and you have no

way of telling which is which. If you require this distinction, run the `run` command in list context and inspect the individual buffers.

Of course, this requires that the underlying call supports buffers. See the note on buffers above.

- timeout

  Sets the maximum time the command is allowed to run before aborting, using the built-in `alarm()` call. If the timeout is triggered, the `errorcode` in the return value will be set to an object of the `IPC::Cmd::TimeOut` class. See the "error message" section below for details.

  Defaults to `0`, meaning no timeout is set.

`run` will return a simple `true` or `false` when called in scalar context. In list context, you will be returned a list of the following items:

- success

  A simple boolean indicating if the command executed without errors or not.

- error message

  If the first element of the return value (`success`) was 0, then some error occurred. This second element is the error message the command you requested exited with, if available. This is generally a pretty printed value of `$?` or `$@`. See `perldoc perlvar` for details on what they can contain. If the error was a timeout, the `error message` will be prefixed with the string `IPC::Cmd::TimeOut`, the timeout class.

- full_buffer

  This is an array reference containing all the output the command generated. Note that buffers are only available if you have IPC::Run installed, or if your system is able to work with *IPC::Open3* – see below). Otherwise, this element will be `undef`.

- out_buffer

  This is an array reference containing all the output sent to STDOUT the command generated. The notes from "full_buffer" apply.

- error_buffer

  This is an arrayreference containing all the output sent to STDERR the command generated. The notes from "full_buffer" apply.

See the "HOW IT WORKS" section below to see how `IPC::Cmd` decides what modules or function calls to use when issuing a command.

$hashref = run_forked(COMMAND, { child_stdin => SCALAR, timeout => DIGIT, stdout_handler => CODEREF, stderr_handler => CODEREF});

`run_forked` is used to execute some program or a coderef, optionally feed it with some input, get its return code and output (both stdout and stderr into separate buffers). In addition, it allows to terminate the program if it takes too long to finish.

The important and distinguishing feature of run_forked is execution timeout which at first seems to be quite a simple task but if you think that the program which you're spawning might spawn some children itself (which in their turn could do the same and so on) it turns out to be not a simple issue.

`run_forked` is designed to survive and successfully terminate almost any long running task, even a fork bomb in case your system has the resources to survive during given timeout.

This is achieved by creating separate watchdog process which spawns the specified program in a separate process session and supervises it: optionally feeds it with input, stores its exit code, stdout and stderr, terminates it in case it runs longer than specified.

Invocation requires the command to be executed or a coderef and optionally a hashref of options:

- `timeout`

  Specify in seconds how long to run the command before it is killed with SIG_KILL (9), which effectively terminates it and all of its children (direct or indirect).

- `child_stdin`

  Specify some text that will be passed into the `STDIN` of the executed program.

- `stdout_handler`

  Coderef of a subroutine to call when a portion of data is received on STDOUT from the executing program.

- `stderr_handler`

  Coderef of a subroutine to call when a portion of data is received on STDERR from the executing program.

- `wait_loop_callback`

  Coderef of a subroutine to call inside of the main waiting loop (while `run_forked` waits for the external to finish or fail). It is useful to stop running external process before it ends by itself, e.g.

```
my $r = run_forked("some external command", {
        'wait_loop_callback' => sub {
        if (condition) {
            kill(1, $$);
        }
        },
        'terminate_on_signal' => 'HUP',
        });
```

  Combined with `stdout_handler` and `stderr_handler` allows terminating external command based on its output. Could also be used as a timer without engaging with alarm (signals).

  Remember that this code could be called every millisecond (depending on the output which external command generates), so try to make it as lightweight as possible.

- `discard_output`

  Discards the buffering of the standard output and standard errors for return by run_forked(). With this option you have to use the std*_handlers to read what the command outputs. Useful for commands that send a lot of output.

- `terminate_on_parent_sudden_death`

  Enable this option if you wish all spawned processes to be killed if the initially spawned process (the parent) is killed or dies without waiting for child processes.

`run_forked` will return a HASHREF with the following keys:

- `exit_code`

  The exit code of the executed program.

- `timeout`

  The number of seconds the program ran for before being terminated, or 0 if no timeout occurred.

- `stdout`

  Holds the standard output of the executed command (or empty string if there was no STDOUT output or if `discard_output` was used; it's always defined!)

- `stderr`

  Holds the standard error of the executed command (or empty string if there was no STDERR output or if `discard_output` was used; it's always defined!)

- `merged`

  Holds the standard output and error of the executed command merged into one stream (or empty string if there was no output at all or if `discard_output` was used; it's always defined!)

- `err_msg`

  Holds some explanation in the case of an error.

$q = QUOTE

Returns the character used for quoting strings on this platform. This is usually a `'` (single quote) on most systems, but some systems use different quotes. For example, `Win32` uses `"` (double quote).

You can use it as follows:

```
use IPC::Cmd qw[run QUOTE];
my $cmd = q[echo ] . QUOTE . q[foo bar] . QUOTE;
```

This makes sure that `foo bar` is treated as a string, rather than two separate arguments to the `echo` function.

HOW IT WORKS

`run` will try to execute your command using the following logic:

- If you have `IPC::Run` installed, and the variable `$IPC::Cmd::USE_IPC_RUN` is set to true (See the "Global Variables" section) use that to execute the command. You will have the full output available in buffers, interactive commands are sure to work and you are guaranteed to have your verbosity settings honored cleanly.

- Otherwise, if the variable `$IPC::Cmd::USE_IPC_OPEN3` is set to true (See the "Global Variables" section), try to execute the command using *IPC::Open3*. Buffers will be available on all platforms, interactive commands will still execute cleanly, and also your verbosity settings will be adhered to nicely;
- Otherwise, if you have the `verbose` argument set to true, we fall back to a simple `system()` call. We cannot capture any buffers, but interactive commands will still work.
- Otherwise we will try and temporarily redirect STDERR and STDOUT, do a `system()` call with your command and then re-open STDERR and STDOUT. This is the method of last resort and will still allow you to execute your commands cleanly. However, no buffers will be available.

Global Variables

The behaviour of IPC::Cmd can be altered by changing the following global variables:

$IPC::Cmd::VERBOSE

This controls whether IPC::Cmd will print any output from the commands to the screen or not. The default is 0.

$IPC::Cmd::USE_IPC_RUN

This variable controls whether IPC::Cmd will try to use IPC::Run when available and suitable.

$IPC::Cmd::USE_IPC_OPEN3

This variable controls whether IPC::Cmd will try to use *IPC::Open3* when available and suitable. Defaults to true.

$IPC::Cmd::WARN

This variable controls whether run-time warnings should be issued, like the failure to load an `IPC::*` module you explicitly requested.

Defaults to true. Turn this off at your own risk.

$IPC::Cmd::INSTANCES

This variable controls whether `can_run` will return all instances of the binary it finds in the `PATH` when called in a list context.

Defaults to false, set to true to enable the described behaviour.

$IPC::Cmd::ALLOW_NULL_ARGS

This variable controls whether `run` will remove any empty/null arguments it finds in command arguments.

Defaults to false, so it will remove null arguments. Set to true to allow them.

Caveats

- Whitespace and IPC::Open3 / system()

  When using `IPC::Open3` or `system`, if you provide a string as the `command` argument, it is assumed to be appropriately escaped. You can use the `QUOTE` constant to use as a portable quote character (see above). However, if you provide an array reference, special rules apply:

  If your command contains **special characters** (< > | &), it will be internally stringified before executing the command, to avoid that these special characters are escaped and passed as arguments instead of retaining their special meaning.

  However, if the command contained arguments that contained whitespace, stringifying the command would lose the significance of the whitespace. Therefore, `IPC::Cmd` will quote any arguments containing whitespace in your command if the command is passed as an arrayref and contains special characters.

- Whitespace and IPC::Run

  When using `IPC::Run`, if you provide a string as the `command` argument, the string will be split on whitespace to determine the individual elements of your command. Although this will usually just Do What You Mean, it may break if you have files or commands with whitespace in them.

  If you do not wish this to happen, you should provide an array reference, where all parts of your command are already separated out. Note however, if there are extra or spurious whitespaces in these parts, the parser or underlying code may not interpret it correctly, and cause an error.

  Example: The following code

```
gzip -cdf foo.tar.gz | tar -xf -
```

  should either be passed as

```
"gzip -cdf foo.tar.gz | tar -xf -"
```

  or as

```
['gzip', '-cdf', 'foo.tar.gz', '|', 'tar', '-xf', '-']
```

  But take care not to pass it as, for example

```
['gzip -cdf foo.tar.gz', '|', 'tar -xf -']
```

  Since this will lead to issues as described above.

- IO Redirect

  Currently it is too complicated to parse your command for IO redirections. For capturing STDOUT or STDERR there is a work around however, since you can just inspect your buffers for the contents.

- Interleaving STDOUT/STDERR

  Neither IPC::Run nor IPC::Open3 can interleave STDOUT and STDERR. For short bursts of output from a program, e.g. this sample,

```
for ( 1..4 ) {
    $_ % 2 ? print STDOUT $_ : print STDERR $_;
}
```

IPC::[Run|Open3] will first read all of STDOUT, then all of STDERR, meaning the output looks like '13' on STDOUT and '24' on STDERR, instead of

```
1
2
3
4
```

This has been recorded in rt.cpan.org as bug #37532: Unable to interleave STDOUT and STDERR.

See Also

IPC::Run, *IPC::Open3*

ACKNOWLEDGEMENTS

Thanks to James Mastros and Martijn van der Streek for their help in getting *IPC::Open3* to behave nicely.

Thanks to Petya Kohts for the `run_forked` code.

BUG REPORTS

Please report bugs or other issues to bug-ipc-cmd@rt.cpan.org.

AUTHOR

Original author: Jos Boumans kane@cpan.org. Current maintainer: Chris Williams bingos@cpan.org.

COPYRIGHT

This library is free software; you may redistribute and/or modify it under the same terms as Perl itself.

IPC::Msg

SysV Msg IPC object class

SYNOPSIS

```
use IPC::SysV qw(IPC_PRIVATE S_IRUSR S_IWUSR);
use IPC::Msg;
```

(continues on next page)

(continued from previous page)

```
$msg = IPC::Msg->new(IPC_PRIVATE, S_IRUSR | S_IWUSR);

$msg->snd($msgtype, $msgdata);

$msg->rcv($buf, 256);

$ds = $msg->stat;

$msg->remove;
```

DESCRIPTION

A class providing an object based interface to SysV IPC message queues.

METHODS

- new (KEY , FLAGS)

  Creates a new message queue associated with `KEY`. A new queue is created if

  - `KEY` is equal to `IPC_PRIVATE`
  - `KEY` does not already have a message queue associated with it, and `_FLAGS_ & IPC_CREAT` is true.

  On creation of a new message queue `FLAGS` is used to set the permissions. Be careful not to set any flags that the Sys V IPC implementation does not allow: in some systems setting execute bits makes the operations fail.

- id

  Returns the system message queue identifier.

- rcv (BUF, LEN [, TYPE [, FLAGS]])

  Read a message from the queue. Returns the type of the message read. See msgrcv(2). The BUF becomes tainted.

- remove

  Remove and destroy the message queue from the system.

- set (STAT)

- set (NAME => VALUE [, NAME => VALUE ...])

  `set` will set the following values of the `stat` structure associated with the message queue.

  ```
  uid
  gid
  mode (oly the permission bits)
  qbytes
  ```

  `set` accepts either a stat object, as returned by the `stat` method, or a list of *name-value* pairs.

- snd (TYPE, MSG [, FLAGS])

  Place a message on the queue with the data from `MSG` and with type `TYPE`. See msgsnd(2).

- stat

  Returns an object of type `IPC::Msg::stat` which is a sub-class of `Class::Struct`. It provides the following fields. For a description of these fields see you system documentation.

```
uid
gid
cuid
cgid
mode
qnum
qbytes
lspid
lrpid
stime
rtime
ctime
```

SEE ALSO

IPC::SysV, *Class::Struct*

AUTHORS

Graham Barr gbarr@pobox.com, Marcus Holland-Moritz mhx@cpan.org

COPYRIGHT

Version 2.x, Copyright (C) 2007-2013, Marcus Holland-Moritz.

Version 1.x, Copyright (c) 1997, Graham Barr.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

IPC::Open2

open a process for both reading and writing using open2()

SYNOPSIS

```
use IPC::Open2;

my $pid = open2(my $chld_out, my $chld_in,
  'some', 'cmd', 'and', 'args');
# or passing the command through the shell
```

(continues on next page)

(continued from previous page)

```
my $pid = open2(my $chld_out, my $chld_in, 'some cmd and args');

# read from parent STDIN and write to already open handle
open my $outfile, '>', 'outfile.txt' or die "open failed: $!";
my $pid = open2(['&', $outfile], ['&', *STDIN],
  'some', 'cmd', 'and', 'args');

# read from already open handle and write to parent STDOUT
open my $infile, '<', 'infile.txt' or die "open failed: $!";
my $pid = open2(['&', *STDOUT], ['&', $infile],
  'some', 'cmd', 'and', 'args');

# reap zombie and retrieve exit status
waitpid( $pid, 0 );
my $child_exit_status = $? >> 8;
```

DESCRIPTION

The `open2()` function runs the given command and connects `$chld_out` for reading and `$chld_in` for writing. It's what you think should work when you try

```
my $pid = open(my $fh, "|cmd args|");  # ERROR
```

but you have to write it as:

```
my $pid = open2($chld_out, $chld_in, @command_and_args);
```

The `$chld_in` filehandle will have autoflush turned on.

By default, the filehandles you pass in are used as output parameters. open2 internally creates two pipes. The write end of the first pipe and the read end of the second pipe are connected to the command's standard output and input, respectively. The corresponding read and write ends are placed in the first and second argument to open2.

The filehandle arguments can take the following forms:

- An uninitialized variable (technically, either `undef` or the empty string will work): open2 generates a fresh filehandle and assigns it to the argument, which must be a modifiable variable for this work (otherwise an exception will be raised).
- An existing handle in the form of a typeglob like `*STDIN` or `*FOO` or a reference to such: open2 places the filehandle in the `IO` slot of the typeglob, which means the corresponding bareword filehandle (like `STDIN` or `FOO`) can be used for I/O from/to the child process. (If the handle is already open, it is automatically closed first.)
- A string containing the name of a bareword handle (like `'STDIN'` or `'FOO'`): Such strings are resolved to typeglobs at runtime and then act like the case described above.

However, it is possible to make open2 use an existing handle directly (as an input argument) and skip the creation of a pipe. To do this, the filehandle argument must have one of the following two forms:

- An array reference like `['&', $fh]`, i.e. the first element is the string `'&'` and the second element is the existing handle to use in the child process.
- A string of the form `'<&FOO'` or `'>&FOO'`, i.e. a string starting with the two characters <& (for input) or >& (for output), followed by the name of a bareword filehandle. (The string form cannot be used with handles stored in variables.)

If you use this form for `$chld_in`, the filehandle will be closed in the parent process.

open2 returns the process ID of the child process. It doesn't return on failure: it just raises an exception matching `/^open2:/`. However, `exec` failures in the child are not detected. You'll have to trap SIGPIPE yourself.

open2 does not wait for and reap the child process after it exits. Except for short programs where it's acceptable to let the operating system take care of this, you need to do this yourself. This is normally as simple as calling `waitpid $pid, 0` when you're done with the process. Failing to do this can result in an accumulation of defunct or "zombie" processes. See *"waitpid" in perlfunc* for more information.

This whole affair is quite dangerous, as you may block forever. It assumes it's going to talk to something like bc(1), both writing to it and reading from it. This is presumably safe because you "know" that commands like bc(1) will read a line at a time and output a line at a time. Programs like sort(1) that read their entire input stream first, however, are quite apt to cause deadlock.

The big problem with this approach is that if you don't have control over source code being run in the child process, you can't control what it does with pipe buffering. Thus you can't just open a pipe to `cat -v` and continually read and write a line from it.

The *IO::Pty* and Expect modules from CPAN can help with this, as they provide a real tty (well, a pseudo-tty, actually), which gets you back to line buffering in the invoked command again.

WARNING

The order of arguments differs from that of open3 from *IPC::Open3*.

SEE ALSO

See *IPC::Open3* for an alternative that handles `STDERR` as well. This function is really just a wrapper around open3.

IPC::Open3

open a process for reading, writing, and error handling using open3()

SYNOPSIS

```
use Symbol 'gensym'; # vivify a separate handle for STDERR
my $pid = open3(my $chld_in, my $chld_out, my $chld_err = gensym,
                'some', 'cmd', 'and', 'args');
# or pass the command through the shell
my $pid = open3(my $chld_in, my $chld_out, my $chld_err = gensym,
```

(continues on next page)

(continued from previous page)

```
                'some cmd and args');

# read from parent STDIN
# send STDOUT and STDERR to already open handle
open my $outfile, '>>', 'output.txt' or die "open failed: $!";
my $pid = open3(['&', *STDIN], ['&', $outfile], undef,
                'some', 'cmd', 'and', 'args');

# write to parent STDOUT and STDERR
my $pid = open3(my $chld_in, ['&', *STDOUT], ['&', *STDERR],
                'some', 'cmd', 'and', 'args');

# reap zombie and retrieve exit status
waitpid( $pid, 0 );
my $child_exit_status = $? >> 8;
```

DESCRIPTION

Extremely similar to open2 from *IPC::Open2*, open3 spawns the given command and provides filehandles for interacting with the command's standard I/O streams.

```
my $pid = open3($chld_in, $chld_out, $chld_err, @command_and_args);
```

It connects `$chld_in` for writing to the child's standard input, `$chld_out` for reading from the child's standard output, and `$chld_err` for reading from the child's standard error stream. If `$chld_err` is false, or the same file descriptor as `$chld_out`, then `STDOUT` and `STDERR` of the child are on the same filehandle. This means that you cannot pass an uninitialized variable for `$chld_err` and have open3 auto-generate a filehandle for you, but gensym from *Symbol* can be used to vivify a new glob reference; see "SYNOPSIS". The `$chld_in` handle will have autoflush turned on.

By default, the filehandles you pass in are used as output parameters. open3 internally creates three pipes. The write end of the first pipe and the read ends of the other pipes are connected to the command's standard input/output/error, respectively. The corresponding read and write ends are placed in the first three argument to open3.

The filehandle arguments can take the following forms:

- An uninitialized variable (technically, either `undef` or the empty string will work): open3 generates a fresh filehandle and assigns it to the argument, which must be a modifiable variable for this work (otherwise an exception will be raised).

  This does not work for `$chld_err`, however: If the `$chld_err` argument is a false value, the child's error stream is automatically redirected to its standard output.

- An existing handle in the form of a typeglob like `*STDIN` or `*FOO` or a reference to such: open3 places the filehandle in the `IO` slot of the typeglob, which means the corresponding bareword filehandle (like `STDIN` or `FOO`) can be used for I/O from/to the child process. (If the handle is already open, it is automatically closed first.)

- A string containing the name of a bareword handle (like `'STDIN'` or `'FOO'`): Such strings are resolved to typeglobs at runtime and then act like the case described above.

However, it is possible to make open3 use an existing handle directly (as an input argument) and skip the creation of a pipe. To do this, the filehandle argument must have one of the following two forms:

- An array reference like `['&', $fh]`, i.e. the first element is the string `'&'` and the second element is the existing handle to use in the child process.
- A string of the form `'<&FOO'` or `'>&FOO'`, i.e. a string starting with the two characters <& (for input) or >& (for output), followed by the name of a bareword filehandle. (The string form cannot be used with handles stored in variables.)

If you use this form for `$chld_in`, the filehandle will be closed in the parent process.

The filehandles may also be integers, in which case they are understood as file descriptors.

open3 returns the process ID of the child process. It doesn't return on failure: it just raises an exception matching `/^open3:/`. However, `exec` failures in the child (such as no such file or permission denied), are just reported to `$chld_err` under Windows and OS/2, as it is not possible to trap them.

If the child process dies for any reason, the next write to `$chld_in` is likely to generate a SIGPIPE in the parent, which is fatal by default, So you may wish to handle this signal.

Note: if you specify `-` as the command, in an analogous fashion to `open(my $fh, "-|")` the child process will just be the forked Perl process rather than an external command. This feature isn't yet supported on Win32 platforms.

open3 does not wait for and reap the child process after it exits. Except for short programs where it's acceptable to let the operating system take care of this, you need to do this yourself. This is normally as simple as calling `waitpid $pid, 0` when you're done with the process. Failing to do this can result in an accumulation of defunct or "zombie" processes. See *"waitpid" in perlfunc* for more information.

If you try to read from the child's stdout writer and their stderr writer, you'll have problems with blocking, which means you'll want to use `select` or *IO::Select*, which means you'd best use `sysread` instead of `readline` for normal stuff.

This is very dangerous, as you may block forever. open3 assumes it's going to talk to something like bc(1), both writing to it and reading from it. This is presumably safe because you "know" that commands like bc(1) will read a line at a time and output a line at a time. Programs like sort(1) that read their entire input stream first, however, are quite apt to cause deadlock.

The big problem with this approach is that if you don't have control over source code being run in the child process, you can't control what it does with pipe buffering. Thus you can't just open a pipe to `cat -v` and continually read and write a line from it.

See Also

- *IPC::Open2*

  Like *IPC::Open3* but without STDERR capture.

- IPC::Run

  This is a CPAN module that has better error handling and more facilities than *IPC::Open3*.

WARNING

The order of arguments differs from that of open2.

IPC::Semaphore

SysV Semaphore IPC object class

SYNOPSIS

```
use IPC::SysV qw(IPC_PRIVATE S_IRUSR S_IWUSR IPC_CREAT);
use IPC::Semaphore;

$sem = IPC::Semaphore->new(IPC_PRIVATE, 10, S_IRUSR | S_IWUSR | IPC_
↪CREAT);

$sem->setall( (0) x 10);

@sem = $sem->getall;

$ncnt = $sem->getncnt;

$zcnt = $sem->getzcnt;

$ds = $sem->stat;

$sem->remove;
```

DESCRIPTION

A class providing an object based interface to SysV IPC semaphores.

METHODS

- new (KEY , NSEMS , FLAGS)

  Create a new semaphore set associated with KEY. NSEMS is the number of semaphores in the set. A new set is created if

  - KEY is equal to IPC_PRIVATE
  - KEY does not already have a semaphore identifier associated with it, and _FLAGS_ & IPC_CREAT is true.

  On creation of a new semaphore set FLAGS is used to set the permissions. Be careful not to set any flags that the Sys V IPC implementation does not allow: in some systems setting execute bits makes the operations fail.

- getall

  Returns the values of the semaphore set as an array.

- getncnt (SEM)

Returns the number of processes waiting for the semaphore SEM to become greater than its current value

- getpid (SEM)

  Returns the process id of the last process that performed an operation on the semaphore SEM.

- getval (SEM)

  Returns the current value of the semaphore SEM.

- getzcnt (SEM)

  Returns the number of processes waiting for the semaphore SEM to become zero.

- id

  Returns the system identifier for the semaphore set.

- op (OPLIST)

  OPLIST is a list of operations to pass to semop. OPLIST is a concatenation of smaller lists, each which has three values. The first is the semaphore number, the second is the operation and the last is a flags value. See semop(2) for more details. For example

```
$sem->op(
    0, -1, IPC_NOWAIT,
    1,  1, IPC_NOWAIT
);
```

- remove

  Remove and destroy the semaphore set from the system.

- set (STAT)

- set (NAME => VALUE [, NAME => VALUE ...])

  set will set the following values of the stat structure associated with the semaphore set.

```
uid
gid
mode (only the permission bits)
```

  set accepts either a stat object, as returned by the stat method, or a list of *name-value* pairs.

- setall (VALUES)

  Sets all values in the semaphore set to those given on the VALUES list. VALUES must contain the correct number of values.

- setval (N , VALUE)

  Set the Nth value in the semaphore set to VALUE

- stat

Returns an object of type `IPC::Semaphore::stat` which is a sub-class of `Class::Struct`. It provides the following fields. For a description of these fields see your system documentation.

```
uid
gid
cuid
cgid
mode
ctime
otime
nsems
```

SEE ALSO

IPC::SysV, *Class::Struct*, semget(2), semctl(2), semop(2)

AUTHORS

Graham Barr gbarr@pobox.com, Marcus Holland-Moritz mhx@cpan.org

COPYRIGHT

Version 2.x, Copyright (C) 2007-2013, Marcus Holland-Moritz.

Version 1.x, Copyright (c) 1997, Graham Barr.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

IPC::SharedMem

SysV Shared Memory IPC object class

SYNOPSIS

```
use IPC::SysV qw(IPC_PRIVATE S_IRUSR S_IWUSR);
use IPC::SharedMem;

$shm = IPC::SharedMem->new(IPC_PRIVATE, 8, S_IRWXU);

$shm->write(pack("S", 4711), 2, 2);

$data = $shm->read(0, 2);

$ds = $shm->stat;

$shm->remove;
```

DESCRIPTION

A class providing an object based interface to SysV IPC shared memory.

METHODS

- new (KEY , SIZE , FLAGS)

  Creates a new shared memory segment of `SIZE` bytes size associated with `KEY`. A new segment is created if

  – `KEY` is equal to `IPC_PRIVATE`

  – `KEY` does not already have a shared memory segment associated with it, and `_FLAGS_ & IPC_CREAT` is true.

  On creation of a new shared memory segment `FLAGS` is used to set the permissions. Be careful not to set any flags that the Sys V IPC implementation does not allow: in some systems setting execute bits makes the operations fail.

- id

  Returns the shared memory identifier.

- read (POS, SIZE)

  Read `SIZE` bytes from the shared memory segment at `POS`. Returns the string read, or `undef` if there was an error. The return value becomes tainted. See shmread.

- write (STRING, POS, SIZE)

  Write `SIZE` bytes to the shared memory segment at `POS`. Returns true if successful, or false if there is an error. See shmwrite.

- remove

  Remove the shared memory segment from the system or mark it as removed as long as any processes are still attached to it.

- is_removed

  Returns true if the shared memory segment has been removed or marked for removal.

- stat

  Returns an object of type `IPC::SharedMem::stat` which is a sub-class of `Class::Struct`. It provides the following fields. For a description of these fields see you system documentation.

```
uid
gid
cuid
cgid
mode
segsz
lpid
cpid
nattch
```

(continues on next page)

(continued from previous page)

```
atime
dtime
ctime
```

- attach ([FLAG])

  Permanently attach to the shared memory segment. When a `IPC::SharedMem` object is attached, it will use memread and memwrite instead of shmread and shmwrite for accessing the shared memory segment. Returns true if successful, or false on error. See shmat(2).

- detach

  Detach from the shared memory segment that previously has been attached to. Returns true if successful, or false on error. See shmdt(2).

- addr

  Returns the address of the shared memory that has been attached to in a format suitable for use with `pack('P')`. Returns `undef` if the shared memory has not been attached.

SEE ALSO

IPC::SysV, *Class::Struct*

AUTHORS

Marcus Holland-Moritz mhx@cpan.org

COPYRIGHT

Version 2.x, Copyright (C) 2007-2013, Marcus Holland-Moritz.

Version 1.x, Copyright (c) 1997, Graham Barr.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

IPC::SysV

System V IPC constants and the system calls Perl's builtins do not cover.

Perl's core already provides `msgget`/`msgctl`/`msgsnd`/`msgrcv`, `semget`/`semctl`/`semop` and `shmget`/`shmctl`/`shmread`/`shmwrite` as builtins. What `IPC::SysV` adds on top is the constant table (`IPC_CREAT`, `SEM_UNDO`, `SHM_RDONLY`, ...), the `ftok` key generator, shared-memory attach/detach, and the `pack`/`unpack` pair each of `IPC::Msg::stat`, `IPC::Semaphore::stat` and `IPC::SharedMem::stat` uses to move a `struct msqid_ds` / `semid_ds` / `shmid_ds` across the Perl boundary.

`IPC::Msg`, `IPC::Semaphore` and `IPC::SharedMem` are pure Perl on top of this module; `@EXPORT_OK`, `%EXPORT_TAGS` and the `AUTOLOAD` that resolves a constant name come from `IPC/SysV.pm`.

```
use IPC::SysV qw(IPC_PRIVATE IPC_CREAT IPC_RMID S_IRUSR S_IWUSR ftok␣
 ↪shmat shmdt memread memwrite);

my $key = ftok($0, 1);                        # a key derived from a␣
 ↪file
my $id = shmget(IPC_PRIVATE, 64, IPC_CREAT | S_IRUSR | S_IWUSR) //␣
 ↪die "shmget: $!";
my $addr = shmat($id, undef, 0) // die "shmat: $!";   # attach: an␣
 ↪address
memwrite($addr, "hello", 0, 5) or die;
memread($addr, my $buf, 0, 5) or die;
print "read back '$buf'\n";                   # read back 'hello'
shmdt($addr);
shmctl($id, IPC_RMID, 0);
```

This is a native implementation of the XS half of IPC::SysV 2.09; `IPC/SysV.pm` and the three object modules are used unchanged. Nothing is exported by default; ask for names, or for everything with `:all`.

Differences from upstream

None.

Functions

Keys

`ftok`

`ftok($path, $id = undef)` - derive a System V IPC key from an existing file and a project id.

Shared memory

`memread`

`memread($addr, $sv, $pos, $size)` - copy `$size` bytes out of an attached shared-memory segment into `$sv`.

`memwrite`

`memwrite($addr, $sv, $pos, $size)` - copy `$sv` into an attached shared-memory segment at byte offset `$pos`.

`shmat`

`shmat($id, $addr, $flag)` - attach the shared-memory segment `$id` and return an opaque address for it.

shmdt

`shmdt($addr)` - detach a shared-memory segment previously attached with `shmat`.

shmat

`shmat($id, $addr, $flag)` - attach the shared-memory segment `$id` and return an opaque address for it.

Pass `undef` as `$addr` to let the kernel choose where to attach; `$flag` takes `SHM_RDONLY`, `SHM_RND` and friends. The value that comes back is a string of raw address bytes, meaningful only to `memread`, `memwrite` and `shmdt` - do not print it or do arithmetic on it. Returns `undef` with `$!` set on failure; a negative `$id` sets `$!` to `EINVAL` without reaching the kernel.

5.2.59 Imager

Perl bindings for Imager, the raster graphics library.

This is the `Imager` module, version 1.037 of the CPAN API: the same packages, methods, arguments and return values that Perl code written against `Imager` already expects - for the part of it that is available so far (see below).

```
use Imager;

my $img = Imager->new(xsize => 4, ysize => 3, channels => 4) or die␣
↪Imager->errstr;
$img->setpixel(x => 1, y => 1, color => Imager::Color->new(255, 0,␣
↪0));
my $c = $img->getpixel(x => 1, y => 1);
printf "%dx%d, %d channels, pixel %s\n", $img->getwidth, $img->
↪getheight,
    $img->getchannels, join(",", $c->rgba);      # 4x3, 4 channels,␣
↪pixel 255,0,0,255
my @s = $img->getsamples(y => 1, x => 1, width => 1);   # 255 0 0␣
↪255
my $copy = $img->copy;
$img->settag(name => 'note', value => 'hello');
```

What is available

The image engine and the operations on it that are native so far: creating images of every kind below, `getpixel`/`setpixel`, `getsamples`/`setsamples`, `getscanline`/`setscanline`, `copy`, palettes (`addcolors`, `getcolors`, `colorcount` and the rest), tags, the channel mask, masked images, the colour classes, drawing (`box`, `line`, `arc`, `circle`, `polygon`, `polypolygon`, `flood_fill`, `polybezier`), fills, `paste`, `crop`, `rubthrough` and `compose`, `flip`, `convert`, `map` and `Imager->combine`, `to_paletted`, `Imager->make_palette`, `getcolorcount` and `getcolorusage`, `scale`, `scaleX` and `scaleY`, `rotate` and `matrix_transform`, `trim` and `trim_rect`, `transform` and `Imager::transform2`, the filters, `difference` and `rgb_difference`, the files below, and text through `Imager::Font::FT2`.

Not available: `Imager->load_plugin`, the loader for filter plugins compiled from C. It returns undef and sets `$Imager::ERRSTR` to "Couldn't load plugin '...'", as it does for a file that cannot be loaded. Imager's documentation has called that interface deprecated since 0.48 (2006).

Drawing

box, `line`, `arc`, `circle` and `polygon` draw in a colour or, where they fill an area, through a fill (below). `aa => 1` anti-aliases lines, arcs, circles and outlines; `filled => 0` gives the outline of a circle or an arc. A polygon is always anti-aliased, and `mode => 'nonzero'` fills it by the non-zero winding rule instead of the even-odd one; `polypolygon` takes several at once, so one can be a hole in another. `flood_fill` fills the region of one colour around a point, or with `border` everything up to that colour, and `polybezier` draws a Bezier curve through control points.

```
$img->circle(x => 50, y => 50, r => 30, color => 'red', aa => 1);
$img->arc(x => 50, y => 50, r => 40, d1 => 0, d2 => 90, filled => 0,␣
↪aa => 1);
$img->polygon(points => [ [ 10, 10 ], [ 90, 30 ], [ 40, 80 ] ],␣
↪color => 'green');
$img->flood_fill(x => 2, y => 2, color => 'blue') or die $img->
↪errstr;
```

Where Imager differs from the other raster modules

Imager is not a wrapper around one C library; its image engine is its own. That engine is what this module provides natively, and it is why Imager can do things the others cannot: an image may hold **8-bit, 16-bit or floating-point samples**, may be **paletted or direct**, and may have **one to four channels**, in any combination. A greyscale 16-bit image and an RGBA 8-bit image are both ordinary Imager images.

Every operation works on all of them, so code written for one kind of image works on another without change.

Four kinds of image

An image is described by three independent properties, and Imager supports the whole product of them:

Property	Values
sample size	8-bit, 16-bit, double
storage	direct, or paletted (up to 256 colours)
channels	1 (grey), 2 (grey+alpha), 3 (RGB), 4 (RGBA)

```
my $rgb   = Imager->new(xsize => 10, ysize => 10);
↪# 8-bit RGB
my $deep  = Imager->new(xsize => 10, ysize => 10, bits => 16);
my $float = Imager->new(xsize => 10, ysize => 10, bits => 'double');
my $pal   = Imager->new(xsize => 10, ysize => 10, type => 'paletted
```

(continues on next page)

(continued from previous page)

```
↪');
my $grey  = Imager->new(xsize => 10, ysize => 10, channels => 1);
```

Reading and writing a pixel works the same on all of them. `getpixel`/`setpixel` speak `Imager::Color` (bytes) or `Imager::Color::Float` (doubles); a 16-bit or floating-point image keeps its extra precision only if you use the float form.

Colours

A colour is four channels. `Imager::Color` holds them as bytes, 0 to 255, which is what most code wants; `Imager::Color::Float` holds them as floating-point numbers, nominally 0 to 1, which is what a 16-bit or floating-point image needs to keep its precision.

`Imager::Color->new` accepts the friendly spellings - a name, an HTML hex string, a channel list:

```
my $red   = Imager::Color->new('red');
my $red2  = Imager::Color->new('#FF0000');
my $red3  = Imager::Color->new(255, 0, 0);
my $trans = Imager::Color->new(255, 0, 0, 128);

my ($r, $g, $b, $a) = $red->rgba;
```

Anywhere a method takes `color =>`, an array reference works too and is the least ceremony for a one-off: `$img->setpixel(x => 0, y => 0, color => [0, 128, 255])`.

The channel mask

`$img->setmask(mask => 0x5)` makes subsequent writes touch only the channels whose bit is set - channel 0 and channel 2 here. Reads ignore it. It is how you draw into one channel of an image without disturbing the others.

Errors

Imager reports failure by returning false and leaving the reason in an error stack. `Imager->errstr` on the class gives the last error; `$img->errstr` gives that image's. Nothing croaks for an ordinary failure, so the idiom is to check and report:

```
my $img = Imager->new(xsize => 10, ysize => 10) or die Imager->
↪errstr;
```

Files

PNM files (PBM, PGM and PPM, ASCII and binary, 8 and 16 bit), Windows BMP files, Targa files and raw files (`type => 'raw'`, samples and nothing else) are read and written - the formats Imager has without any library. GIF, JPEG, PNG and TIFF are not available yet. BMP is read at 1, 4, 8, 16, 24 and 32 bits a pixel, run length encoded and with bit-field masks, and written uncompressed: 24 bits for a direct image, 1, 4 or 8 for a paletted one.

```
my $img = Imager->new(file => 'in.ppm') or die Imager->errstr;
$img->write(file => 'out.pgm', type => 'pnm') or die $img->errstr;
my @all = Imager->read_multi(file => 'frames.ppm');
```

Targa is read and written colour-mapped, direct and grey, run length encoded (`compress => 1`, the default) or not, with `wierdpack => 1` for 15 and 16 bits a pixel and `idstring` for the file's id.

Masked images

`$img->masked` gives a view of a rectangle of an image, optionally written through a mask: a pixel of the view is a pixel of the image, reads see all of it, and a write lands only where the mask's first channel is not zero. The view can be drawn into, copied and written to a file like any image.

```
my $mask = Imager->new(xsize => 100, ysize => 100, channels => 1);
$mask->box(filled => 1, color => 'white', xmin => 20, xmax => 79);
my $view = $img->masked(mask => $mask, left => 10, top => 10,
                        right => 110, bottom => 110);
$view->box(filled => 1, color => 'red');    # a red band in $img
```

Fills

box and `arc` take a `fill` instead of a colour: a hash of `Imager::Fill->new` parameters or such an object. A fill is a solid colour, one of the 32 named hatches or a custom 8x8 one (`hatch`, `fg`, `bg`, `dx`, `dy`), a tiled image (`image`, with `xoff`, `yoff` and a `matrix`), or another fill made more transparent (`type => 'opacity'`). `combine` says how it meets what is already there: `none`, `normal`, `multiply`, `dissolve`, `add`, `subtract`, `diff`, `lighten`, `darken`, `hue`, `saturation`, `value` or `color`. A fountain fill (`fountain => 'linear'`, `radial`, `conical` and the rest, with `segments` of colours as `Imager::Fountain` builds them) is a gradient between the points `xa`, `ya` and `xb`, `yb`; the `fountain` filter draws the same over a whole image.

```
$img->box(xmin => 10, ymin => 10, xmax => 90, ymax => 50,
          fill => { hatch => 'check2x2', fg => 'red', bg => 'white'␣
↪});
my $glass = Imager::Fill->new(solid => Imager::Color->new(0, 0, 255,␣
↪96),
                              combine => 'normal');
$img->arc(x => 50, y => 50, r => 30, fill => $glass);
```

One image onto another

`paste` copies a rectangle as it is. `rubthrough` lays the source over the target by the source's alpha channel. `compose` does the same with a combine mode (the names above) and an `opacity`, and with `mask` takes the coverage of each pixel from the first channel of a third image.

```
$img->rubthrough(src => $logo, tx => 10, ty => 10) or die $img->
↪errstr;
```

(continues on next page)

(continued from previous page)

```
$img->compose(src => $logo, tx => 10, ty => 10, combine => 'multiply
↪',
              opacity => 0.5, mask => $mask) or die $img->errstr;
```

`setsamples` writes samples of chosen channels, as 8-bit values, floating point (`type => 'float'`) or 16-bit (`type => '16bit'`, on a 16-bit image), and `getsamples` reads them in the same three forms.

An alpha channel is written over the colour of the `i_background` tag, black without one; `pnm_write_wide_data => 1` writes a 16-bit image with 16-bit samples. `Imager->set_file_limits` bounds the width, height and bytes of what `read` accepts, and `Imager->add_file_magic` teaches the format detection a new signature.

Channels

`convert` makes a new image whose channels are weighted sums of this one's, by a `matrix` or a `preset` (`grey`, `noalpha`, `rgb`, `addalpha`, `red`, `green`, `blue`, `alpha`); `Imager->combine` makes one from a channel each of several images; `map` runs the samples of chosen channels through lookup tables in place, and `flip` mirrors the image in place.

```
my $grey  = $img->convert(preset => 'grey') or die $img->errstr;
my $bgr   = $img->convert(matrix => [ [ 0, 0, 1 ], [ 0, 1, 0 ], [ 1,␣
↪0, 0 ] ]);
my $again = Imager->combine(src => [ $r, $g, $b ]) or die Imager->
↪errstr;
$img->map(red => [ reverse 0 .. 255 ]);
$img->flip(dir => 'h');
```

Scaling

`scale` returns a scaled copy: by `scalefactor`, or to `xpixels` and `ypixels`, where `type` says how the two combine (`max`, `min`, `nonprop`). `qtype` picks the method: `normal` (a Lanczos filter), `preview` (nearest neighbour) or `mixing` (the mean of the area each pixel covers). `normal` and `preview` give an 8-bit image whatever the source holds, as in Imager; `mixing` keeps the depth.

```
my $thumb = $img->scale(xpixels => 160, ypixels => 120, type => 'min
↪',
                        qtype => 'mixing') or die $img->errstr;
```

Rotation and matrix transforms

`rotate(right => 90)` (or 180, 270) turns the image exactly; `rotate(degrees => ...)` or `radians` rotates by any angle into an image large enough to hold the result, with `back` for the corners. `matrix_transform` maps the image through a 3 x 3 matrix, such as `Imager::Matrix2d` builds, with bilinear interpolation. A paletted image is not interpolated.

```
my $turned = $img->rotate(degrees => 30, back => 'white') or die
 ↪$img->errstr;
```

Expressions per pixel

`Imager::transform2` builds an image from an expression evaluated for every pixel, in postfix (`rpnexpr`) or, with Parse::RecDescent installed, infix (`expr`): arithmetic, comparisons, `getp1` to `getp3` for the pixels of up to three input images, `rgb`, `hsv` and their parts. `$img->transform` is the older form: two small stack programs give the source position of each pixel.

```
my $mirrored = Imager::transform2({ rpnexpr => 'w x - 1 - y getp1' }
 ↪, $img)
    or die Imager->errstr;
```

Filters

`$img->filter(type => ...)` changes the image in place, with every filter Imager has built in: `autolevels` and `autolevels_skew`, `bumpmap` and `bumpmap_complex`, `contrast`, `conv`, `fountain`, `gaussian` and `gaussian2`, `gradgen`, `hardinvert` and `hardinvertall`, `mosaic`, `nearest_color`, `noise`, `postlevels`, `radnoise`, `turbnoise`, `unsharpmask` and `watermark`. Many work through 8-bit samples, as in Imager, so a 16-bit or double image keeps 8 bits of precision afterwards. `$img->difference` and `rgb_difference` make a new image of where two images differ.

```
$img->filter(type => 'autolevels') or die $img->errstr;
$img->filter(type => 'mosaic', size => 8);
```

Paletted images from direct ones

`to_paletted` quantises an image. `make_colors` says where the palette comes from: `mediancut` (the default), `addi`, a fixed map (`webmap`, `mono`, `gray`, `gray4`, `gray16`) or none for the `colors` you give; `max_colors` bounds it. `translate` says how the pixels are matched to it: `closest`, `perturb`, or `errdiff` with an `errdiff` of `floyd`, `jarvis`, `stucki` or a custom map. `Imager->make_palette` makes one palette for several images, and `getcolorcount` and `getcolorusage` count the colours of an image.

```
my $pal = $img->to_paletted(make_colors => 'mediancut', max_colors
 ↪=> 16,
                            translate => 'errdiff') or die $img->
 ↪errstr;
my $n   = $img->getcolorcount(maxcolors => 256);   # undef if more
```

The `addi` palette and the `perturb` translation use the C library's random numbers, which Imager never seeds: the same image gives the same result every run.

Differences from upstream

Beyond `load_plugin` (above), eighteen:

- Where perl 5.44 with XS Imager reads freed memory: an `Imager::ImgRaw` or `Imager::IO` used after its `DESTROY` croaks "not a live Imager::ImgRaw" (or "Imager::IO") (t/64-auto-ffi/414-imager-or-better.t).
- `Imager::io_slurp` of an I/O object that is not a buffer chain croaks "io_slurp: called on a source that is not from a bufchain". Imager 1.037 ends the whole program there, silently and with exit status 0.
- Reading a row of a paletted image as floating point colours (`getscanline(type => 'float')`) returns its colours and leaves the image alone. Imager 1.037 writes where it should read there (image.c `i_glinf_fp` calls `i_plin` on an uninitialised buffer): the row comes back as whatever was in memory, and the image is overwritten with it and turned into a direct colour image.
- A grey image with an alpha channel written as BMP has its grey applied over the background by its own alpha. Imager 1.037 reads the wrong samples there (paste.im `i_gsamp_bg` asks for 2 a pixel and reads 4): every pixel takes its neighbour's grey as alpha and half the row is whatever was in memory.
- A masked image keeps its target and its mask alive, an image fill its image and an opacity fill its base fill. In Imager 1.037 that is done by `$img->masked` and `Imager::Fill->new`, which store them in the result; the objects themselves hold bare pointers, so one made with `Imager::i_img_masked_new` or `Imager::i_new_fill_image` directly reads freed memory once the other is gone.
- `paste` with a source rectangle that starts left of or above the source image writes black for the part outside it. Imager 1.037 writes whatever was in the memory it had just allocated.
- The hue combine mode leaves a grey pixel of an image without an alpha channel as it is. Imager 1.037 changes it to a colour that differs from run to run (it reads an alpha the image does not have).
- A custom hatch given as a string shorter than 8 bytes has empty rows for the missing bytes; Imager 1.037 takes them from whatever follows the string in memory.
- An image fill with a `matrix` drawn into a 16-bit or double image from a source without an alpha channel draws the source. In Imager 1.037 that fill comes out black.
- A polygon with a coordinate beyond 134 million pixels keeps its shape. Imager 1.037 converts such a coordinate to an `int` it does not fit in; on x86-64 the vertex lands far to the left of the image instead of far to the right.
- `compose` with a rectangle the target clips and a negative source or mask offset larger than what is left draws nothing and succeeds. Imager 1.037 prints "Out of memory" and aborts the program there.
- The fountain filter called as `Imager::i_fountain` with a super-sampling number it does not know draws without super-sampling, with a negative sample count draws nothing, and with segments that are not an array croaks. Imager 1.037 crashes in all three. A fountain fill none of whose samples has a colour draws transparent black where Imager draws what was in memory.

- `getsamples(type => '16bit')` of a negative sample in a double image gives 0. Imager 1.037 converts the negative number to an unsigned one, which on x86-64 gives a value near 4294967295.
- `to_paletted` with `make_colors => 'addi'` makes opaque colours. Imager 1.037 sets only their red, green and blue, so the palette of a four channel image has an alpha that is whatever was in memory.
- `transform` and `transform2` start from black and from empty registers, end a program that reaches outside its registers or its code, clamp `rgb()` of a number beyond the int range, and refuse a missing input image or a channel count outside 1 to 4. Imager 1.037 computes from memory it never set in the first cases and crashes in the last two.
- The `postlevels` filter with `levels => 0`, `autolevels_skew` on a channel whose range comes out empty, and `mosaic` with a `size` of 0 or less leave the image as it is. Imager 1.037 dies of a division by zero in the first two and never returns from the third. A negative bump map channel is refused; Imager reads outside the colour.
- `Imager::i_gradgen` with a distance measure other than 0, 1 or 2 croaks. Imager 1.037 ends the program with exit status 3.
- `Imager->set_out_of_memory_handler($code)` stores and returns the handler, but it is never called. An image whose pixel storage cannot be allocated fails with "Out of memory allocating image surface", as upstream; running out of memory for anything smaller ends the process, as upstream does when no handler is set.

Modules

- `Imager::Color` - Color handling for Imager.
- `Imager::Expr` - implements expression parsing and compilation for the expression evaluation engine used by Imager::transform2()
- `Imager::Fill` - general fill types
- `Imager::Font` - Font handling for Imager.
- `Imager::Fountain` - a class for building fountain fills suitable for use by the fountain filter.
- `Imager::Matrix2d` - simple wrapper for matrix construction
- `Imager::Regops` - generated information about the register based virtual machine
- `Imager::Test` - common functions used in testing Imager
- `Imager::Transform` - a library of register machine image transformations
- `Imager::TrimColorList` - represent a list of color ranges for Imager's trim() method.

Functions

Library information

i_list_formats

The image file formats this build can read and write, as a list of short names such as "png" and "jpeg".

Errors

i_clear_error

Discard any pending error.

i_push_error

Add a message to the error stack.

i_errors

The pending errors, newest first, each as an array reference [$message, $code]. Imager->errstr is built from this.

Imager::Color

Color handling for Imager.

SYNOPSIS

```
use Imager;

$color = Imager::Color->new($red, $green, $blue);
$color = Imager::Color->new($red, $green, $blue, $alpha);
$color = Imager::Color->new("#C0C0FF"); # html color specification

$color->set($red, $green, $blue);
$color->set($red, $green, $blue, $alpha);
$color->set("#C0C0FF"); # html color specification

($red, $green, $blue, $alpha) = $color->rgba();
@hsv = $color->hsv();

$color->info();

if ($color->equals(other=>$other_color)) {
  ...
}
```

DESCRIPTION

This module handles creating color objects used by Imager. The idea is that in the future this module will be able to handle color space calculations as well.

An Imager color consists of up to four components, each in the range 0 to 255. Unfortunately the meaning of the components can change depending on the type of image you're dealing with:

- for 3 or 4 channel images the color components are red, green, blue, alpha.
- for 1 or 2 channel images the color components are gray, alpha, with the other two components ignored.

An alpha value of zero is fully transparent, an alpha value of 255 is fully opaque.

METHODS

- new

  This creates a color object to pass to functions that need a color argument.

- set

  This changes an already defined color. Note that this does not affect any places where the color has been used previously.

- rgba()

  This returns the red, green, blue and alpha channels of the color the object contains.

- info

  Calling info merely dumps the relevant color to the log.

- equals(other=>$other_color)

- equals(other=>$other_color, ignore_alpha=>1)

  Compares $self and color $other_color returning true if the color components are the same.

  Compares all four channels unless `ignore_alpha` is set. If `ignore_alpha` is set only the first three channels are compared.

You can specify colors in several different ways, you can just supply simple values:

- an Imager::Color object
- an Imager::Color::Float object, the ranges of samples are translated from 0.0...1.0 to 0...255.
- simple numeric parameters - if you supply 3 or 4 numeric arguments, you get a color made up of those RGB (and possibly A) components.
- a six hex digit web color, either RRGGBB or #RRGGBB
- an eight hex digit web color, either RRGGBBAA or #RRGGBBAA.
- a CSS rgb() color, based on CSS Color 4. The none keyword is not supported and numbers must be simple decimals without exponents. eg.

```
rgb(50% 50% 100%)
rgb(0, 0, 255)
rgb(0.5 0.5 1.0 / 0.8)
rgb(50%, 50%, 100%, 80%)
```

Samples from percentages or decimals are rounded up per CSS Color 3 and 4.

This accepts some colors not accepted by the CSS rgb() specification, this may change.

- a 3 hex digit web color, `#RGB` - a value of F becomes 255.
- a color name, from whichever of the gimp `Named_Colors` file or X `rgb.txt` is found first. The same as using the `name` keyword.

You can supply named parameters:

- 'red', 'green' and 'blue', optionally shortened to 'r', 'g' and 'b'. The color components in the range 0 to 255.

```
# all of the following are equivalent
my $c1 = Imager::Color->new(red=>100, blue=>255, green=>0);
my $c2 = Imager::Color->new(r=>100, b=>255, g=>0);
my $c3 = Imager::Color->new(r=>100, blue=>255, g=>0);
```

- hue, `saturation` and `value`, optionally shortened to `h`, `s` and `v`, to specify a HSV color. 0 <= hue < 360, 0 <= s <= 1 and 0 <= v <= 1.

```
# the same as RGB(127,255,127)
my $c1 = Imager::Color->new(hue=>120, v=>1, s=>0.5);
my $c1 = Imager::Color->new(hue=>120, value=>1, saturation=>0.
↪5);
```

- `web`, which can specify a 6 or 3 hex digit web color, in any of the forms `#RRGGBB`, `#RGB`, `RRGGBB` or `RGB`.

```
my $c1 = Imager::Color->new(web=>'#FFC0C0'); # pale red
```

- `gray` or `grey` which specifies a single channel, from 0 to 255.

```
# exactly the same
my $c1 = Imager::Color->new(gray=>128);
my $c1 = Imager::Color->new(grey=>128);
```

- `rgb` which takes a 3 member arrayref, containing each of the red, green and blue values.

```
# the same
my $c1 = Imager::Color->new(rgb=>[255, 100, 0]);
my $c1 = Imager::Color->new(r=>255, g=>100, b=>0);
```

- `hsv` which takes a 3 member arrayref, containing each of hue, saturation and value.

```
# the same
my $c1 = Imager::Color->new(hsv=>[120, 0.5, 1]);
my $c1 = Imager::Color->new(hue=>120, v=>1, s=>0.5);
```

- gimp which specifies a color from a GIMP palette file. You can specify the file name of the palette file with the 'palette' parameter, or let Imager::Color look in various places, typically `$HOME/gimp-1.x/palettes/Named_Colors` with and without the version number, and in `/usr/share/gimp/palettes/`. The palette file must have color names.

```
my $c1 = Imager::Color->new(gimp=>'snow');
my $c1 = Imager::Color->new(gimp=>'snow', palette=>'testimg/
↪test_gimp_pal);
```

- xname which specifies a color from an X11 `rgb.txt` file. You can specify the file name of the `rgb.txt` file with the `palette` parameter, or let Imager::Color look in various places, typically `/usr/lib/X11/rgb.txt`.

```
my $c1 = Imager::Color->new(xname=>'blue') # usually RGB(0, 0,␣
↪255)
```

- builtin which specifies a color from the built-in color table in Imager::Color::Table. The colors in this module are the same as the default X11 `rgb.txt` file.

```
my $c1 = Imager::Color->new(builtin=>'black') # always RGB(0, 0,
↪ 0)
```

- name which specifies a name from either a GIMP palette, an X `rgb.txt` file or the built-in color table, whichever is found first.
- 'channel0', 'channel1', etc, each of which specifies a single channel. These can be abbreviated to 'c0', 'c1' etc.
- 'channels' which takes an arrayref of the channel values.

Optionally you can add an alpha channel to a color with the 'alpha' or 'a' parameter.

These color specifications can be used for both constructing new colors with the new() method and modifying existing colors with the set() method.

METHODS

- hsv()

```
my($h, $s, $v, $alpha) = $color->hsv();
```

  Returns the color as a Hue/Saturation/Value/Alpha tuple.

- red
- green
- blue
- alpha

Returns the respective component as an integer from 0 to 255.

- as_float

  Returns the color as a *Imager::Color::Float* object.

- as_css_rgb

  Returns the color as a CSS rgb() format color. This is always returned in the byte form, eg. rgb(255 128 64).

  If the alpha is not full coverage (255) it will be rounded if the result of converting the color back to an 8 bit color would return the same alpha, eg. if the color alpha is 128, it will be formatted as 0.5, not as the more precise 50.2%.

AUTHOR

Arnar M. Hrafnkelsson, addi@umich.edu And a great deal of help from others - see the `README` for a complete list.

SEE ALSO

Imager(3), Imager::Color http://imager.perl.org/

Modules

- `Imager::Color::Float` - Rough floating point sample color handling
- `Imager::Color::Table` - built-in Imager color table

Imager::Color::Float

Rough floating point sample color handling

SYNOPSIS

```
$color = Imager::Color->new($red, $green, $blue);
$color = Imager::Color->new($red, $green, $blue, $alpha);
$color = Imager::Color->new("#C0C0FF"); # html color specification

$color->set($red, $green, $blue);
$color->set($red, $green, $blue, $alpha);
$color->set("#C0C0FF"); # html color specification

($red, $green, $blue, $alpha) = $color->rgba();
@hsv = $color->hsv(); # not implemented but proposed
my $c8 = $color->as_8bit;

$color->info();
```

DESCRIPTION

This module handles creating color objects used by Imager. The idea is that in the future this module will be able to handle color space calculations as well.

A floating point Imager color consists of up to four components, each in the range 0.0 to 1.0. Unfortunately the meaning of the components can change depending on the type of image you're dealing with:

- for 3 or 4 channel images the color components are red, green, blue, alpha.
- for 1 or 2 channel images the color components are gray, alpha, with the other two components ignored.

An alpha value of zero is fully transparent, an alpha value of 1.0 is fully opaque.

METHODS

- new

  This creates a color object to pass to functions that need a color argument.

- set

  This changes an already defined color. Note that this does not affect any places where the color has been used previously.

- rgba()

  This returns the red, green, blue and alpha channels of the color the object contains.

- info

  Calling info merely dumps the relevant color to the log.

- red
- green
- blue
- alpha

  Returns the respective component as a floating point value typically from 0 to 1.0.

- as_8bit

  Returns the color as the roughly equivalent 8-bit Imager::Color object. Samples below zero or above 1.0 are clipped.

- as_css_rgb

  Formats the color as a CSS rgb() style color. If the color is closely representable as byte style syntax, eg rgb(255, 128, 128), it will be returned in that form, otherwise the samples are formatted as percentages with up to 2 decimal places.

Setting colors

The new() and set() methods can accept the following parameters:

- an Imager::Color::Float object
- an Imager::Color object, the ranges of samples are translated from 0...255 to 0.0...1.0.
- 3 values, treated as red, green, blue
- 4 values, treated as red, green, blue, alpha
- an 8 character hex color, optionally preceded by #.
- a 6 character hex color, optionally preceded by #.
- a CSS rgb() color, based on CSS Color 4. The none keyword is not supported and numbers must be simple decimals without exponents. eg.

```
rgb(50% 50% 100%)
rgb(0, 0, 255)
rgb(0.5 0.5 1.0 / 0.8)
rgb(50%, 50%, 100%, 80%)
```

  This accepts some colors not accepted by the CSS rgb() specification, this may change.

AUTHOR

Arnar M. Hrafnkelsson, addi@umich.edu And a great deal of help from others - see the `README` for a complete list.

SEE ALSO

Imager(3), Imager::Color.

http://imager.perl.org/

Imager::Color::Table

built-in Imager color table

SYNOPSIS

```
use Imager::Color::Table;
my @rgb = Imager::Color::Table->get($name)
  or die "Color $name not found";
```

DESCRIPTION

This class provides a base color table for use in resolving color names.

The table contains the standard X11 `rgb.txt` colors.

This table is not included as part of Imager::Color itself since it's moderately large.

There is only one method:

- get

```
my @rgb = Imager::Color::Table->get('red')
  or die "No red found";
```

  Retrieves a color from Imager::Color::Tables built-in color table.

AUTHOR

Tony Cook tony@develop-help.com

Imager::Expr

implements expression parsing and compilation for the expression evaluation engine used by Imager::transform2()

SYNOPSIS

my $code = Imager::Expr->new({rpnexpr=>$someexpr}) or die "Cannot compile $someexpr: ",Imager::Expr::error();

DESCRIPTION

This module is used internally by the Imager::transform2() function. You shouldn't have much need to use it directly, but you may want to extend it.

To create a new Imager::Expr object, call:

```
my %options;
my $expr = Imager::Expr->new(\%options)
  or die Imager::Expr::error();
```

You will need to set an expression value and you may set any of the following:

- constants

  A hashref defining extra constants for expression parsing. The names of the constants must be valid identifiers (/[^\W\d]\w*/) and the values must be valid numeric constants (that Perl recognizes in scalars).

  Imager::Expr may define it's own constants (currently just pi.)

- variables

  A reference to an array of variable names. These are allocated numeric registers starting from register zero.

By default you can define a `rpnexpr` key (which emulates RPN) or `expr` (an infix expression). It's also possible to write other expression parsers that will use other keys. Only one expression key should be defined.

Instance methods

The Imager::Expr::error() method is used to retrieve the error if the expression object cannot be created.

Methods

Imager::Expr provides only a few simple methods meant for external use:

- Imager::Expr->type_registered($keyword)

  Returns true if the given expression type is available. The parameter is the key supplied to the new() method.

  ```
  if (Imager::Expr->type_registered('expr')) {
    # use infix expressions
  }
  ```

- $expr->code()

  Returns the compiled code.

- $expr->nregs()

  Returns a reference to the array of numeric registers.

- $expr->cregs()

  Returns a reference to the array of color registers.

- $expr->dumpops()

  Returns a string with the generated VM “machine code”.

- $expr->dumpcode()

  Returns a string with the disassembled VM “machine code”.

Creating a new parser

I’ll write this one day.

Methods used by parsers:

- compile

  This is the main method you’ll need to implement in a parser. See the existing parsers for a guide.

  It’s supplied the following parameters:

  – $expr - the expression to be parsed

  – $options - the options hash supplied to transform2.

  Return an array ref of array refs containing opcodes and operands.

- @vars = $self->_variables()

  A list (not a reference) of the input variables. This should be used to allocate as many registers as there are variable as input registers.

- $self->error($message)

  Set the return value of Imager::Expr::error()

- @ops = $self->stack_to_reg(@stack_ops)

  Converts marginally parsed RPN to register code.

- assemble()

  Called to convert op codes into byte code.

- numre()

  Returns a regular expression that matches floating point numbers.

- optimize()

  Optimizes the assembly code, including attempting common subexpression elimination and strength reducing division by a constant into multiplication by a constant.

- register_type()

  Called by a new expression parser implementation to register itself, call as:

```
YourClassName->register_type('type code');
```

  where type code is the parameter that will accept the expression.

Future compatibility

Try to avoid doing your own optimization beyond literal folding - if we add some sort of jump, the existing optimizer will need to be rewritten, and any optimization you perform may well be broken too (well, your code generation will probably be broken anyway <sigh>).

AUTHOR

Tony Cook tonyc@cpan.org, Arnar M. Hrafnkelsson

Modules

- `Imager::Expr::Assem` - an assembler for producing code for the Imager register machine

Imager::Expr::Assem

an assembler for producing code for the Imager register machine

SYNOPSIS

```
use Imager::Expr::Assem;
my $expr = Imager::Expr->new(assem=>'...', ...)
```

DESCRIPTION

This module is a simple Imager::Expr compiler that compiles a low-level language that has a nearly 1-to-1 relationship to the internal representation used for compiled register machine code.

Syntax

Each line can contain multiple statements separated by semi-colons.

Anything after '#' in a line is ignored.

Types of statements:

- variable definition

  ```
  `var` _name_:_type_
  ```

  defines variable *name* to have *type*, which can be any of `n` or `num` for a numeric type or `pixel`, `p` or `c` for a pixel or color type.

  Variable names cannot include white-space.

- operators

  Operators can be split into 3 basic types, those that have a result value, those that don't and the null operator, eg. jump has no value.

  The format for operators that return a value is typically:

  ```
  _result_ = _operator_ _operand_ ...
  ```

  and for those that don't return a value:

  ```
  _operator_ _operand_
  ```

  where operator is any valid register machine operator, result is any variable defined with `var`, and operands are variables, constants or literals, or for jump operators, labels.

  The set operator can be simplified to:

  ```
  _result_ = _operator_
  ```

  All operators maybe preceded by a label, which is any non-white-space text immediately followed by a colon (':').

BUGS

Note that the current optimizer may produce incorrect optimization for your code, fortunately the optimizer will disable itself if you include any jump operator in your code. A single jump to anywhere after your final `ret` operator can be used to disable the optimizer without slowing down your code.

There's currently no high-level code generation that can generate code with loops or real conditions.

SEE ALSO

Imager(3), `transform.perl`, `regmach.c`

AUTHOR

Tony Cook tony@develop-help.com

Imager::Fill

general fill types

SYNOPSIS

```
use Imager;
use Imager::Fill;

my $fill1 = Imager::Fill->new(solid=>$color, combine=>$combine);
my $fill2 = Imager::Fill->new(hatch=>'vline2', fg=>$color1, bg=>
 ↪$color2,
                               dx=>$dx, dy=>$dy);
my $fill3 = Imager::Fill->new(fountain=>$type, ...);
my $fill4 = Imager::Fill->new(image=>$img, ...);
my $fill5 = Imager::Fill->new(type => "opacity", other => $fill,
                               opacity => ...);
```

DESCRIPTION

Creates fill objects for use by most filled area drawing functions.

All fills are created with the new method.

- new

  ```
  my $fill = Imager::Fill->new(...);
  ```

  The parameters depend on the type of fill being created. See below for details.

The currently available fills are:

- solid
- hatch
- fountain (similar to gradients in paint software)
- image - fill with an image, possibly transformed
- opacity - a lower opacity version of some other fill

Common options

- combine

  The way in which the fill data is combined with the underlying image. See "Combine Types" in Imager::Draw.

In general colors can be specified as *Imager::Color* or *Imager::Color::Float* objects. The fill object will typically store both types and convert from one to the other. If a fill takes 2 color objects they should have the same type.

Solid fills

```
my $fill = Imager::Fill->new(solid=>$color, combine =>$combine)
```

Creates a solid fill, the only required parameter is `solid` which should be the color to fill with.

A translucent red fill:

```
my $red = Imager::Fill->new(solid => "FF000080", combine => "normal
↪");
```

Hatched fills

```
my $fill = Imager::Fill->new(hatch=>$type, fg=>$fgcolor, bg=>
↪$bgcolor,
                             dx=>$dx, $dy=>$dy);
```

Creates a hatched fill. You can specify the following keywords:

- `hatch` - The type of hatch to perform, this can either be the numeric index of the hatch (not recommended), the symbolic name of the hatch, or an array of 8 integers which specify the pattern of the hatch.

  Hatches are represented as cells 8x8 arrays of bits, which limits their complexity.

  Current hatch names are:

  - `check1x1`, `check2x2`, `check4x4` - checkerboards at various sizes
  - `vline1`, `vline2`, `vline4` - 1, 2, or 4 vertical lines per cell
  - `hline1`, `hline2`, `hline4` - 1, 2, or 4 horizontal lines per cell
  - `slash1`, `slash2` - 1 or 2 / lines per cell.
  - `slosh1`, `slosh2` - 1 or 2 \ lines per cell
  - `grid1`, `grid2`, `grid4` - 1, 2, or 4 vertical and horizontal lines per cell
  - `dots1`, `dots4`, `dots16` - 1, 4 or 16 dots per cell
  - `stipple`, `stipple2` - see the samples
  - `weave` - I hope this one is obvious.
  - `cross1`, `cross2` - 2 densities of crosshatch

  - `vlozenge`, `hlozenge` - something like lozenge tiles
  - `scalesdown`, `scalesup`, `scalesleft`, `scalesright` - Vaguely like fish scales in each direction.
  - `tile_L` - L-shaped tiles

- `fg`, bg - The `fg` color is rendered where bits are set in the hatch, and the bg where they are clear. If you use a transparent `fg` or bg, and set combine, you can overlay the hatch onto an existing image.

  `fg` defaults to black, bg to white.

- dx, dy - An offset into the hatch cell. Both default to zero.

A blue and white 4-pixel check pattern:

```
my $fill = Imager::Fill->new(hatch => "check2x2", fg => "blue");
```

You can call Imager::Fill->hatches for a list of hatch names.

Fountain fills

```
my $fill = Imager::Fill->new(fountain=>$ftype,
     xa=>$xa, ya=>$ya, xb=>$xb, yb=>$yb,
     segments=>$segments, repeat=>$repeat, combine=>$combine,
     super_sample=>$super_sample, ssample_param=>$ssample_param);
```

This fills the given region with a fountain fill. This is exactly the same fill as the `fountain` filter, but is restricted to the shape you are drawing, and the fountain parameter supplies the fill type, and is required.

A radial fill from white to transparent centered on (50, 50) with a 50 pixel radius:

```
use Imager::Fountain;
my $segs = Imager::Fountain->simple(colors => [ "FFFFFF", "FFFFFF00"␣
↪],
                                    positions => [ 0, 1 ]);
my $fill = Imager::Fill->new(fountain => "radial", segments =>
↪$segs,
                             xa => 50, ya => 50, xb => 0, yb => 50,
                             combine => "normal");
```

Image Fills

```
my $fill = Imager::Fill->new(image=>$src, xoff=>$xoff, yoff=>$yoff,
                             matrix=>$matrix, combine => $combine);
```

Fills the given image with a tiled version of the given image. The first non-zero value of `xoff` or `yoff` will provide an offset along the given axis between rows or columns of tiles respectively.

The matrix parameter performs a co-ordinate transformation from the co-ordinates in the target image to the fill image co-ordinates. Linear interpolation is used to determine the fill pixel. You can use the *Imager::Matrix2d* class to create transformation matrices.

The matrix parameter will significantly slow down the fill.

```
# some image to act as a texture
my $txim = Imager->new(...);

# simple tiling
my $fill = Imager::Fill->new(image => $txim);

# tile with a vertical offset
my $fill = Imager::Fill->new(image => $txim, yoff => 10);

# tile with a horizontal offset
my $fill = Imager::Fill->new(image => $txim, xoff => 10);

# rotated
use Imager::Matrix2d;
my $fill = Imager::Fill->new(image => $txim,
              matrix => Imager::Matrix2d->rotate(degrees => 20));
```

Opacity modification fill

```
my $fill = Imager::Fill->new(type => "opacity",
    other => $fill, opacity => 0.25);
```

This can be used to make a fill that is a more translucent or opaque version of an existing fill. This is intended for use where you receive a fill object as a parameter and need to change the opacity.

Parameters:

- type => “opacity” - Required
- other - the fill to produce a modified version of. This must be an Imager::Fill object. Required.
- opacity - multiplier for the source fill opacity. Default: 0.5.

The source fills combine mode is used.

```
my $hatch = Imager::Fill->new(hatch => "check4x4", combine =>
 ↪"normal");
my $fill = Imager::Fill->new(type => "opacity", other => $hatch);
```

OTHER METHODS

- Imager::Fill->hatches

  A list of all defined hatch names.

- Imager::Fill->combines

  A list of all combine types.

FUTURE PLANS

I'm planning on adding the following types of fills:

- `checkerboard` - combines 2 other fills in a checkerboard
- `combine` - combines 2 other fills using the levels of an image
- `regmach` - uses the transform2() register machine to create fills

AUTHOR

Tony Cook tony@develop-help.com

SEE ALSO

Imager(3)

Imager::Font

Font handling for Imager.

SYNOPSIS

```
use Imager;

$t1font = Imager::Font->new(file => 'pathtofont.pfb');
$ttfont = Imager::Font->new(file => 'pathtofont.ttf');
$w32font = Imager::Font->new(face => 'Times New Roman');

$blue = Imager::Color->new("#0000FF");
$font = Imager::Font->new(file  => 'pathtofont.ttf',
                          color => $blue,
                          size  => 30);

($neg_width,
 $global_descent,
 $pos_width,
 $global_ascent,
 $descent,
 $ascent,
 $advance_width,
 $right_bearing) = $font->bounding_box(string=>"Foo");

my $bbox_object = $font->bounding_box(string=>"Foo");

# documented in Imager::Draw
$img->string(font  => $font,
           text  => "Model-XYZ",
           x     => 15,
           y     => 40,
```

(continues on next page)

(continued from previous page)

```
            size  => 40,
            color => $red,
            aa    => 1);
```

DESCRIPTION

This module manages, the font object returned by Imager::Font->new will typically be of a class derived from Imager::Font.

- new

  This creates a font object to pass to functions that take a font argument.

```
$font = Imager::Font->new(file  => 'denmark.ttf',
                          index => 0,
                          color => $blue,
                          size  => 30,
                          aa    => 1);
```

  This creates a font which is the TrueType font `denmark.ttf`. It's default color is $blue, default size is 30 pixels and it's rendered anti-aliased by default. Imager can see which type of font a file is by looking at the suffix of the file name for the font. A suffix of `ttf` is taken to mean a TrueType font while a suffix of `pfb` is taken to mean a Type 1 Postscript font. If Imager cannot tell which type a font is you can tell it explicitly by using the `type` parameter:

```
$t1font = Imager::Font->new(file => 'fruitcase', type => 't1');
$ttfont = Imager::Font->new(file => 'arglebarf', type => 'tt');
```

  The `index` parameter is used to select a single face from a font file containing more than one face, for example, from a Macintosh font suitcase or a `.dfont` file.

  If any of the `color`, `size` or `aa` parameters are omitted when calling `Imager::Font->new()` the they take the following values:

```
color => Imager::Color->new(255, 0, 0, 0);  # this default
↪should be changed
size  => 15
aa    => 0
index => 0
```

  To use Win32 fonts supply the face name of the font:

```
$font = Imager::Font->new(face=>'Arial Bold Italic');
```

  There isn't any access to other logical font attributes, but this typically isn't necessary for Win32 TrueType fonts, since you can construct the full name of the font as above.

  Other logical font attributes may be added if there is sufficient demand.

  Parameters:

- `file` - name of the file to load the font from.
-

```
 `face` - face name.  This is used only under Win32 to create␣
↪a GDI based
 font.  This is ignored if the `file` parameter is supplied.
```

- `type` - font driver to use. Currently the permitted values for this are:
  * `tt` - FreeType 1.x driver. Supports TrueType (`.ttf`) fonts.
  *

```
 `t1` - T1 Lib driver.  Supports Postscript Type 1 fonts. ␣
↪Allows for
 synthesis of underline, strikethrough and overline.
```

  * `ft2` - FreeType 2.x driver. Supports many different font formats. Also supports the transform() method.
- `color` - the default color used with this font. Default: red.
- `size` - the default size used with this font. Default: 15.
- `utf8` - if non-zero then text supplied to $img->string(...) and $font->bounding_box(...) is assumed to be UTF-8 encoded by default.
- `align` - the default value for the $img->string(...) `align` parameter. Default: 1.
- `vlayout` - the default value for the $img->string(...) `vlayout` parameter. Default: 0.
- `aa` - the default value for the $im->string(...) `aa` parameter. Default: 0.
- `index` - for font file containing multiple fonts this selects which font to use. This is useful for Macintosh `DFON` (`.dfont`) and suitcase font files.

  If you want to use a suitcase font you will need to tell Imager to use the FreeType 2.x driver by setting `type` to `'ft2'`:

```
my $font = Imager::Font->new(file=>$file, index => 1, type=>
↪'ft2')
  or die Imager->errstr;
```

Returns the new font object on success. Returns `undef` on failure and sets an error message readable with `Imager->errstr`.

- bounding_box()

Returns the bounding box for the specified string. Example:

```
my ($neg_width,
    $global_descent,
    $pos_width,
    $global_ascent,
    $descent,
```

(continues on next page)

(continued from previous page)

```
    $ascent,
    $advance_width,
    $right_bearing) = $font->bounding_box(string => "A Fool");

my $bbox_object = $font->bounding_box(string => "A Fool");
```

- `$neg_width`

  the relative start of a the string. In some cases this can be a negative number, in that case the first letter stretches to the left of the starting position that is specified in the string method of the Imager class

- `$global_descent`

  how far down the lowest letter of the entire font reaches below the baseline (this is often j).

- `$pos_width`

  how wide the string from the starting position is. The total width of the string is `$pos_width-$neg_width`.

- `$descent`

- `$ascent`

  the same as <$global_descent> and <$global_ascent> except that they are only for the characters that appear in the string.

- `$advance_width`

  the distance from the start point that the next string output should start at, this is often the same as `$pos_width`, but can be different if the final character overlaps the right side of its character cell.

- `$right_bearing`

  The distance from the right side of the final glyph to the end of the advance width. If the final glyph overflows the advance width this value is negative.

Obviously we can stuff all the results into an array just as well:

```
@metrics = $font->bounding_box(string => "testing 123");
```

Note that extra values may be added, so $metrics[-1] isn't supported. It's possible to translate the output by a passing coordinate to the bounding box method:

```
@metrics = $font->bounding_box(string => "testing 123", x=>45,␣
↪y=>34);
```

This gives the bounding box as if the string had been put down at `(x,y)` By giving bounding_box 'canon' as a true value it's possible to measure the space needed for the string:

```
@metrics = $font->bounding_box(string=>"testing",size=>15,
↪canon=>1);
```

This returns the same values in $metrics[0] and $metrics[1], but:

```
$bbox[2] - horizontal space taken by glyphs
$bbox[3] - vertical space taken by glyphs
```

Returns an *Imager::Font::BBox* object in scalar context, so you can avoid all those confusing indexes. This has methods as named above, with some extra convenience methods.

Parameters are:

- `string` - the string to calculate the bounding box for. Required.
- `size` - the font size to use. Default: value set in Imager::Font->new(), or 15.
- `sizew` - the font width to use. Default to the value of the `size` parameter.
- `utf8` - For drivers that support it, treat the string as UTF-8 encoded. For versions of perl that support Unicode (5.6 and later), this will be enabled automatically if the 'string' parameter is already a UTF-8 string. See "UTF-8" for more information. Default: the `utf8` value passed to Imager::Font->new(...) or 0.
- x, y - offsets applied to @box[0..3] to give you a adjusted bounding box. Ignored in scalar context.
- `canon` - if non-zero and the x, y parameters are not supplied, then $pos_width and $global_ascent values will returned as the width and height of the text instead.

On success returns either the list of bounds, or a bounding box object in scalar context. Returns an empty list or `undef` on failure and sets an error message readable with `Imager->errstr`.

The transformation matrix set by "transform()" has no effect on the result of this method - the bounds of the untransformed text is returned.

- string()

  The $img->string(...) method is now documented in "string()" in Imager::Draw

- align(string=>$text,size=>$size,x=>...,y=>...,valign => ...,halign=>...)

  Higher level text output - outputs the text aligned as specified around the given point (x,y).

```
# "Hello" centered at 100, 100 in the image.
my ($left, $top, $right, $bottom) =
  $font->align(string=>"Hello",
               x=>100, y=>100,
               halign=>'center', valign=>'center',
               image=>$image);
```

  Takes the same parameters as $font->draw(), and the following extra parameters:

  - `valign` - Possible values are:

    * `top`

      Point is at the top of the text.

        * `bottom`

          Point is at the bottom of the text.

        * `baseline`

          Point is on the baseline of the text (default.)

        * `center`

          Point is vertically centered within the text.

      - `halign`

        * `left` - the point is at the left of the text.

        * `start` - the point is at the start point of the text.

        * `center` - the point is horizontally centered within the text.

        * `right` - the point is at the right end of the text.

        * `end` - the point is at the end point of the text.

      - `image` - The image to draw to. Set to `undef` to avoid drawing but still calculate the bounding box.

    Returns a list specifying the bounds of the drawn text on success. Returns an empty list on failure, if an `image` parameter was supplied the error message can be read with `$image->errstr`, otherwise it's available as `Imager->errstr`.

- dpi()
- dpi(xdpi=>$xdpi, ydpi=>$ydpi)
- dpi(dpi=>$dpi)

  Set or retrieve the spatial resolution of the image in dots per inch. The default is 72 dpi.

  This isn't implemented for all font types yet.

  Possible parameters are:

  - `xdpi`, `ydpi` - set the horizontal and vertical resolution in dots per inch.
  - `dpi` - set both horizontal and vertical resolution to this value.

  Returns a list containing the previous `xdpi`, `ydpi` values on success. Returns an empty list on failure, with an error message returned in `Imager->errstr`.

- transform()

```
$font->transform(matrix=>$matrix);
```

  Applies a transformation to the font, where matrix is an array ref of numbers representing a 2 x 3 matrix:

```
[  $matrix->[0],  $matrix->[1],  $matrix->[2],
   $matrix->[3],  $matrix->[4],  $matrix->[5]   ]
```

  Not all font types support transformations, these will return false.

It's possible that a driver will disable hinting if you use a transformation, to prevent discontinuities in the transformations. See the end of the test script t/t38ft2font.t for an example.

Currently only the ft2 (FreeType 2.x) driver supports the transform() method.

See samples/slant_text.pl for a sample using this function.

Note that the transformation is done in font co-ordinates where y increases as you move up, not image co-ordinates where y decreases as you move up.

`transform()` has no effect on the results of "bounding_box()".

Returns true on success. Returns false on failure with the cause readable from `Imager->errstr`.

- has_chars(string=>$text)

  Checks if the characters in $text are defined by the font.

  In a list context returns a list of true or false value corresponding to the characters in $text, true if the character is defined, false if not. In scalar context returns a string of NUL or non-NUL characters. Supports UTF-8 where the font driver supports UTF-8.

  Not all fonts support this method (use $font->can("has_chars") to check.)

  On error, returns an empty list or undef in scalar context, and sets an error message readable with `Imager->errstr`.

  - `string` - string of characters to check for. Required. Must contain at least one character.
  - `utf8` - For drivers that support it, treat the string as UTF-8 encoded. For versions of perl that support Unicode (5.6 and later), this will be enabled automatically if the 'string' parameter is already a UTF-8 string. See "UTF-8" for more information. Default: the `utf8` value passed to Imager::Font->new(...) or 0.

- face_name()

  Returns the internal name of the face. Not all font types support this method yet, so you should check with `$font->can("face_name")` before calling `face_name`.

- glyph_names(string=>$string [, utf8=>$utf8][, reliable_only=>0]);

  Returns a list of glyph names for each of the characters in the string. If the character has no name then `undef` is returned for the character.

  Some font files do not include glyph names, in this case FreeType 2 will not return any names. FreeType 1 can return standard names even if there are no glyph names in the font.

  FreeType 2 has an API function that returns true only if the font has "reliable glyph names", unfortunately this always returns false for TrueType fonts. This can avoid the check of this API by supplying `reliable_only` as 0. The consequences of using this on an unknown font may be unpredictable, since the FreeType documentation doesn't say how those name tables are unreliable, or how FT2 handles them.

  Both FreeType 1.x and 2.x allow support for glyph names to not be included.

If the supplied `string` is marked as UTF-8 or the `utf8` parameter is true and the supplied string does not contain valid UTF-8, returns an empty string and set an error message readable from `Imager->errstr`,

- can_glyph_names()

  As a class method, returns true if the underlying library supports returning glyph names.

  As an object method, returns true if the supplied font supports returning glyph names.

- draw

  This is used by Imager's string() method to implement drawing text. See "string()" in Imager::Draw.

MULTIPLE MASTER FONTS

The FreeType 2 driver supports multiple master fonts:

- is_mm()

  Test if the font is a multiple master font.

- mm_axes()

  Returns a list of the axes that can be changes in the font. Each entry is an array reference which contains:

  1. Name of the axis.
  2. minimum value for this axis.
  3. maximum value for this axis

- set_mm_coords(coords=>\@values)

  Blends an interpolated design from the master fonts. @values must contain as many values as there are axes in the font.

For example, to select the minimum value in each axis:

```
my @axes = $font->mm_axes;
my @coords = map $_->[1], @axes;
$font->set_mm_coords(coords=>\@coords);
```

It's possible other drivers will support multiple master fonts in the future, check if your selected font object supports the is_mm() method using the can() method.

UTF-8

There are 2 ways of rendering Unicode characters with Imager:

- For versions of perl that support it, use perl's native UTF-8 strings. This is the simplest method.
- Hand build your own UTF-8 encoded strings. Only recommended if your version of perl has no UTF-8 support.

Imager won't construct characters for you, so if want to output Unicode character 00C3 "LATIN CAPITAL LETTER A WITH DIAERESIS", and your font doesn't support it, Imager will *not* build it from 0041 "LATIN CAPITAL LETTER A" and 0308 "COMBINING DIAERESIS".

To check if a driver supports UTF-8 call the utf8() method:

- utf8()

  Return true if the font supports UTF-8.

Native UTF-8 Support

If your version of perl supports UTF-8 and the driver supports UTF-8, just use the $im->string() method, and it should do the right thing.

Build your own

In this case you need to build your own UTF-8 encoded characters.

For example:

```
$x = pack("C*", 0xE2, 0x80, 0x90); # character code 0x2010 HYPHEN
```

You need to be careful with versions of perl that have UTF-8 support, since your string may end up doubly UTF-8 encoded.

For example:

```
$x = "A\xE2\x80\x90\x41\x{2010}";
substr($x, -1, 0) = "";
# at this point $x is has the UTF-8 flag set, but has 5 characters,
# none, of which is the constructed UTF-8 character
```

The test script t/t38ft2font.t has a small example of this after the comment:

```
# an attempt using emulation of UTF-8
```

DRIVER CONTROL

If you don't supply a 'type' parameter to Imager::Font->new(), but you do supply a 'file' parameter, Imager will attempt to guess which font driver to used based on the extension of the font file.

Since some formats can be handled by more than one driver, a priority list is used to choose which one should be used, if a given format can be handled by more than one driver.

- priorities

  The current priorities can be retrieved with:

  ```
  @drivers = Imager::Font->priorities();
  ```

  You can set new priorities and save the old priorities with:

```
@old = Imager::Font->priorities(@drivers);
```

If you supply driver names that are not currently supported, they will be ignored.

Note that by default the priority list no longer includes `tt` and `t1`, so typically you will need to have *Imager::Font::FT2* installed to create fonts with Imager.

```
my @old = Imager::Font->priorities(qw(tt ft2 t1));
```

- register

  Registers an extra font driver. Accepts the following parameters:

  - type - a brief identifier for the font driver. You can supply this value to `Imager::Font->new()` to create fonts of this type. Required.
  - class - the font class name. Imager will attempted to load this module by name. Required.
  - files - a regular expression to match against file names. If supplied this must be a valid perl regular expression. If not supplied you can only create fonts of this type by supplying the `type` parameter to `Imager::Font->new()`
  - description - a brief description of the font driver. Defaults to the value supplied in `class`.

AUTHOR

Arnar M. Hrafnkelsson, addi@umich.edu And a great deal of help from others - see the `README` for a complete list.

BUGS

The $pos_width member returned by the bounding_box() method has historically returned different values from different drivers. The FreeType 1.x and 2.x, and the Win32 drivers return the max of the advance width and the right edge of the right-most glyph. The Type 1 driver always returns the right edge of the right-most glyph.

The newer advance_width and right_bearing values allow access to any of the above.

REVISION

$Revision$

SEE ALSO

Imager(3), Imager::Font::FreeType2(3), Imager::Font::Type1(3), Imager::Font::Win32(3), Imager::Font::Truetype(3), Imager::Font::BBox(3)

```
http://imager.perl.org/
```

Modules

- `Imager::Font::BBox` - objects representing the bounding box of a string.
- `Imager::Font::FT2` - FreeType 2 fonts for Imager: TrueType, OpenType, Type 1 and the other outline and bitmap formats FreeType reads.
- `Imager::Font::FreeType2` - low-level functions for FreeType2 text output
- `Imager::Font::Test` - font driver producing consistent output for tests.
- `Imager::Font::Wrap` - simple wrapped text output

Imager::Font::BBox

objects representing the bounding box of a string.

SYNOPSIS

```
use Imager::Font;

# get the object
my $font = Imager::Font->new(...);
my $bbox = $font->bounding_box(string=>$text, size=>$size);

# methods
my $start = $bbox->start_offset;
my $left_bearing = $bbox->left_bearing;
my $right_bearing = $bbox->right_bearing;
my $end = $bbox->end_offset;
my $gdescent = $box->global_descent;
my $gascent = $bbox->global_ascent;
my $ascent = $bbox->ascent;
my $decent = $bbox->descent;
my $total_width = $bbox->total_width;
my $fheight = $bbox->font_height;
my $theight = $bbox->text_height;
my $display_width = $bbox->display_width;
```

DESCRIPTION

Objects of this class are returned by the Imager::Font bounding_box() method when it is called in scalar context.

This will hopefully make the information from this method more accessible.

METHODS

- start_offset()
- neg_width
- left_bearing

Returns the horizontal offset from the selected drawing location to the left edge of the first character drawn. If this is positive, the first glyph is to the right of the drawing location.

The alias neg_width() is present to match the bounding_box() documentation for list context.

The alias left_bearing() is present to match font terminology.

- advance_width()

  The advance width of the string, if the driver supports that, otherwise the same as end_offset.

- right_bearing

  The distance from the right of the last glyph to the end of the advance point.

  If the glyph overflows the right side of the advance width this value is negative.

- display_width

  The distance from the left-most pixel of the left-most glyph to the right-most pixel of the right-most glyph.

  Equals advance_width - left_bearing - right_bearing (and implemented that way.)

- global_descent()

  The lowest position relative to the font baseline that any character in the font reaches in the character cell. Normally negative.

  At least one font we've seen has reported a positive number for this.

- global_ascent()

  The highest position relative to the font baseline that any character in the font reaches in the character cell. Normally positive.

- descent()

  The lowest position relative to the font baseline that any character in the supplied string reaches. Negative when any character's glyph reaches below the baseline.

- ascent()

  The highest position relative to the font baseline that any character in the supplied string reaches. Positive if any character's glyph reaches above the baseline.

- font_height()

  The maximum displayed height of any string using this font.

- text_height()

  The displayed height of the supplied string.

OBSOLETE METHODS

These methods include bugs kept for backwards compatibility and shouldn't be used in new code.

- total_width()

  The total displayed width of the string.

  New code should use display_width().

  This depends on end_offset(), and is limited by its backward compatibility.

- end_offset

- pos_width

  The offset from the selected drawing location to the right edge of the last character drawn. Should always be positive.

  You can use the alias pos_width() if you are used to the bounding_box() documentation for list context.

  For backwards compatibility this method returns the maximum of the advance width and the offset of the right edge of the last glyph.

INTERNAL FUNCTIONS

- new(...)

  Called by Imager::Font->bounding_box() to create the object.

BUGS

Doesn't reproduce the functionality that you get using the x and y parameters to Imager::Font->bounding_box(). I considered:

```
my ($left, $top, $right, $bottom) = $box->offset(x=>$x, y=>$y)
```

but this is about as clumsy as the original.

AUTHOR

Tony Cook tony@develop-help.com

SEE ALSO

Imager(3), Imager::Font(3)

Imager::Font::FT2

FreeType 2 fonts for Imager: TrueType, OpenType, Type 1 and the other outline and bitmap formats FreeType reads.

`Imager::Font->new(file => ...)` picks this driver on its own when the FreeType library is installed; `type => 'ft2'` asks for it by name. Text is drawn with `$img->string`, measured with `$font->bounding_box`, and may be transformed (`$font->transform(matrix => ...)`), laid out vertically for fonts that carry vertical metrics, and drawn into a single channel with `channel =>`.

```
use Imager;

my $font = Imager::Font->new(file => 'DejaVuSans.ttf', size => 24)
    or die Imager->errstr;
$img->string(font => $font, text => 'Hello', x => 10, y => 40,
             color => 'black', aa => 1);
```

The FreeType library (`libfreetype.so.6`) is loaded when the module is; without it `require Imager::Font::FT2` fails and Imager reports the `ft2` format as unavailable, as a perl5 Imager built without FreeType does.

Drawing into a paletted image with a colour that is not in its palette goes through `Imager::Font::FT2` exactly as it does in perl5: the image becomes a direct RGB image. This is a native implementation of Imager::Font::FT2 1.005; its `.pm` is used unchanged.

Differences from upstream

`i_ft2_version(0)`, the version FreeType had when Imager was compiled, reports the version of the library actually loaded, because nothing here is compiled against FreeType.

Imager::Font::FreeType2

low-level functions for FreeType2 text output

DESCRIPTION

Imager::Font creates a Imager::Font::FreeType2 object when asked to.

See Imager::Font to see how to use this type.

This class provides low-level functions that require the caller to perform data validation.

This driver supports:

- transform()
- dpi()
- draw()

  The following parameters:

  - `utf8`
  - `vlayout`
  - `sizew`

Special behaviors

If you call transform() to set a transformation matrix, hinting will be switched off. This prevents sudden jumps in the size of the text caused by the hinting when the transformation is the identity matrix. If for some reason you want hinting enabled, use $font->hinting(hinting=>1) to re-enable hinting. This will need to be called after *each* call to transform().

AUTHOR

Addi, Tony

Imager::Font::Test

font driver producing consistent output for tests.

SYNOPSIS

```
my $font = Imager::Font::Test->new;

# use $font where you use other fonts
```

DESCRIPTION

Imager::Font::Test is intended to produce consistent output without being subject to the inconsistent output produced by different versions of font libraries.

The output is simple box for the whole string.

AUTHOR

Tony Cook tonyc@cpan.org

Imager::Font::Wrap

simple wrapped text output

SYNOPSIS

```
use Imager::Font::Wrap;

my $img = Imager->new(xsize=>$xsize, ysize=>$ysize);

my $font = Imager::Font->new(file=>$fontfile);

my $string = "..."; # text with or without newlines

Imager::Font::Wrap->wrap_text( image  => $img,
                               font   => $font,
                               string => $string,
                               x      => $left,
                               y      => $top,
                               width  => $width,
                               .... );
```

DESCRIPTION

This is a simple text wrapper with options to control the layout of text within the line.

You can control the position, width and height of the text with the `image`, `x`, `y`, `width` and `height` options.

You can simply calculate space usage by setting `image` to `undef`, or set `savepos` to see how much text can fit within the given `height`.

- wrap_text()

  Draw word-wrapped text.

  - `x`, `y` - The top-left corner of the rectangle the text is formatted into. Defaults to (0, 0).
  - `width` - The width of the formatted text in pixels. Defaults to the horizontal gap between the top-left corner and the right edge of the image. If no image is supplied then this is required.
  - `height` - The maximum height of the formatted text in pixels. Not required.
  - `savepos` - The amount of text consumed (as a count of characters) will be stored into the scalar this refers to.

```
my $pagenum = 1;
my $string = "...";
my $font = ...;
my $savepos;

while (length $string) {
  my $img = Imager->new(xsize=>$xsize, ysize=>$ysize);
  Imager::Font::Wrap->wrap_text(string=>$string, font=>$font,
↪
                                image=>$img, savepos => \
↪$savepos)
    or die $img->errstr;
  $savepos > 0
    or die "Could not fit any text on page\n";
  $string = substr($string, $savepos);
  $img->write(file=>"page$pagenum.ppm");
}
```

  - `image` - The image to render the text to. Can be supplied as `undef` or not provided to simply calculate the bounding box.
  - `font` - The font used to render the text. Required.
  - `size` - The size to render the font in. Defaults to the size stored in the font object. Required if it isn't stored in the font object.
  - `string` - The text to render. This can contain non-white-space, blanks (ASCII 0x20), and newlines.

    Newlines must match /(?:\x0A\x0D?|\x0D\x0A?)/. White-space other than blanks and newlines are completely ignored.

- justify

  The way text is formatted within each line. Possible values include:

  * left - left aligned against the left edge of the text box.
  * right - right aligned against the right edge of the text box.
  * center - centered horizontally in the text box.
  * fill - all but the final line of the paragraph has spaces expanded so that the line fills from the left to the right edge of the text box.

- linegap - Gap between lines of text in pixels. This is in addition to the size from $font->font_height. Can be positive or negative. Default 0.

Any other parameters are passed onto Imager::Font->draw().

Returns a list:

```
($left, $top, $right, $bottom)
```

which are the bounds of the space used to layout the text.

If height is set then this is the space used within that height.

You can use this to calculate the space required to format the text before doing it:

```
my ($left, $top, $right, $bottom) =
  Imager::Font::Wrap->wrap_text(string => $string,
                                font   => $font,
                                width  => $xsize);
my $img = Imager->new(xsize=>$xsize, ysize=>$bottom);
Imager::Font::Wrap->wrap_text(string => $string,
                              font   => $font,
                              width  => $xsize,
                              image  => $image);
```

BUGS

Imager::Font can handle UTF-8 encoded text itself, but this module doesn't support that (and probably won't). This could probably be done with regex magic.

Currently ignores the sizew parameter, if you supply one it will be supplied to the draw() function and the text will be too short or too long for the width.

Uses a simplistic text model, which is why there's no hyphenation, and no tabs.

AUTHOR

Tony Cook tony@develop-help.com

SEE ALSO

Imager(3), Imager::Font(3)

Imager::Fountain

a class for building fountain fills suitable for use by the fountain filter.

SYNOPSIS

```
use Imager::Fountain;
my $f1 = Imager::Fountain->read(gimp=>$filename);
$f->write(gimp=>$filename);
my $f1 = Imager::Fountain->new;
$f1->add(start=>0, middle=>0.5, end=>1.0,
         c0=>Imager::Color->new(...),
         c1=>Imager::Color->new(...),
         type=>$trans_type, color=>$color_trans_type);
```

DESCRIPTION

Provide an interface to build arrays suitable for use by the Imager fountain filter. These can be loaded from or saved to a GIMP gradient file or you can build them from scratch.

- read(gimp=>$filename)
- read(gimp=>$filename, name=>\$name)

  Loads a gradient from the given GIMP gradient file, and returns a new Imager::Fountain object.

  If the name parameter is supplied as a scalar reference then any name field from newer GIMP gradient files will be returned in it.

  ```
  my $gradient = Imager::Fountain->read(gimp=>'foo.ggr');
  my $name;
  my $gradient2 = Imager::Fountain->read(gimp=>'bar.ggr', name=>\
  ↪$name);
  ```

- write(gimp=>$filename)
- write(gimp=>$filename, name=>$name)

  Save the gradient to a GIMP gradient file.

  The second variant allows the gradient name to be set (for newer versions of the GIMP).

  ```
  $gradient->write(gimp=>'foo.ggr')
    or die Imager->errstr;
  $gradient->write(gimp=>'bar.ggr', name=>'the bar gradient')
    or die Imager->errstr;
  ```

- new

  Create an empty fountain fill description.

- add(start=>$start, middle=>$middle, end=>1.0, c0=>$start_color, c1=>$end_color, type=>$trans_type, color=>$color_trans_type)

  Adds a new segment to the fountain fill, the possible options are:

  - `start` - the start position in the gradient where this segment takes effect between 0 and 1. Default: 0.
  - `middle` - the mid-point of the transition between the 2 colors, between 0 and 1. Default: average of `start` and end.
  - end - the end of the gradient, from 0 to 1. Default: 1.
  - `c0` - the color of the fountain fill where the fill parameter is equal to *start*. Default: opaque black.
  - `c1` - the color of the fountain fill where the fill parameter is equal to *end*. Default: opaque black.
  - `type` - the type of segment, controls the way in which the fill parameter moves from 0 to 1. Default: linear.

    This can take any of the following values:

    * `linear`
    * `curved` - unimplemented so far.
    * `sine`
    * `sphereup`
    * `spheredown`

  - `color` - the way in which the color transitions between `c0` and `c1`. Default: direct.

    This can take any of the following values:

    * `direct` - each channel is simple scaled between c0 and c1.
    * hueup - the color is converted to a HSV value and the scaling is done such that the hue increases as the fill parameter increases.
    * huedown - the color is converted to a HSV value and the scaling is done such that the hue decreases as the fill parameter increases.

  In most cases you can ignore some of the arguments, eg.

```
# assuming $f is a new Imager::Fountain in each case here
use Imager ':handy';
# simple transition from red to blue
$f->add(c0=>NC('#FF0000'), c1=>NC('#0000FF'));
# simple 2 stages from red to green to blue
$f->add(end=>0.5, c0=>NC('#FF0000'), c1=>NC('#00FF00'))
$f->add(start=>0.5, c0=>NC('#00FF00'), c1=>NC('#0000FF'));
```

- simple(positions=>[...], colors=>[...])

  Creates a simple fountain fill object consisting of linear segments.

  The array references passed as positions and colors must have the same number of elements. They must have at least 2 elements each.

  colors must contain Imager::Color or Imager::Color::Float objects.

  eg.

```
my $f = Imager::Fountain->simple(positions=>[0, 0.2, 1.0],
                                 colors=>[ NC(255,0,0), NC(0,
↪255,0),
                                           NC(0,0,255) ]);
```

Implementation Functions

Documented for internal use.

- _load_gimp_gradient($class, $fh, $name)

  Does the work of loading a GIMP gradient file.

- _save_gimp_gradient($self, $fh, $name)

  Does the work of saving to a GIMP gradient file.

FILL PARAMETER

The add() documentation mentions a fill parameter in a few places, this is as good a place as any to discuss it.

The process of deciding the color produced by the gradient works through the following steps:

1. calculate the base value, which is typically a distance or an angle of some sort. This can be positive or occasionally negative, depending on the type of fill being performed (linear, radial, etc).
2. clamp or convert the base value to the range 0 through 1, how this is done depends on the repeat parameter. I'm calling this result the fill parameter.
3. the appropriate segment is found. This is currently done with a linear search, and the first matching segment is used. If there is no matching segment the pixel is not touched.
4. the fill parameter is scaled from 0 to 1 depending on the segment type.
5. the color produced, depending on the segment color type.

AUTHOR

Tony Cook tony@develop-help.com

SEE ALSO

Imager(3)

Imager::Matrix2d

simple wrapper for matrix construction

SYNOPSIS

```
use Imager::Matrix2d;
$m1 = Imager::Matrix2d->identity;
$m2 = Imager::Matrix2d->rotate(radians=>$angle, x=>$cx, y=>$cy);
$m3 = Imager::Matrix2d->translate(x=>$dx, y=>$dy);
$m4 = Imager::Matrix2d->shear(x=>$sx, y=>$sy);
$m5 = Imager::Matrix2d->reflect(axis=>$axis);
$m6 = Imager::Matrix2d->scale(x=>$xratio, y=>$yratio);
$m8 = Imager::Matric2d->matrix($v11, $v12, $v13,
                               $v21, $v22, $v23,
                               $v31, $v32, $v33);
$m6 = $m1 * $m2;
$m7 = $m1 + $m2;
use Imager::Matrix2d qw(:handy);
# various m2d_* functions imported
# where m2d_(.*) calls Imager::Matrix2d->$1()
```

DESCRIPTION

This class provides a simple wrapper around a reference to an array of 9 coefficients, treated as a matrix:

```
[ 0, 1, 2,
  3, 4, 5,
  6, 7, 8 ]
```

Most of the methods in this class are constructors. The others are overloaded operators.

Note that since Imager represents images with y increasing from top to bottom, rotation angles are clockwise, rather than counter-clockwise.

- identity()

  Returns the identity matrix.

- rotate(radians=>$angle)
- rotate(degrees=>$angle)

  Creates a matrix that rotates around the origin, or around the point (x,y) if the 'x' and 'y' parameters are provided.

- translate(x=>$dx, y=>$dy)
- translate(x=>$dx)

- translate(y=>$dy)

  Translates by the specify amounts.

- shear(x=>$sx, y=>$sy)
- shear(x=>$sx)
- shear(y=>$sy)

  Shear by the given amounts.

- reflect(axis=>$axis)

  Reflect around the given axis, either 'x' or 'y'.

- reflect(radians=>$angle)
- reflect(degrees=>$angle)

  Reflect around a line drawn at the given angle from the origin.

- scale(x=>$xratio, y=>$yratio)

  Scales at the given ratios.

  You can also specify a center for the scaling with the `cx` and `cy` parameters.

- matrix($v11, $v12, $v13, $v21, $v22, $v23, $v31, $v32, $v33)

  Create a matrix with custom coefficients.

- transform($x, $y)

  Transform a point the same way matrix_transform does.

- compose(matrix...)

  Compose several matrices together for use in transformation.

  For example, for three matrices:

```
my $out = Imager::Matrix2d->compose($m1, $m2, $m3);
```

  is equivalent to:

```
my $out = $m3 * $m2 * $m1;
```

  Returns the identity matrix if no parameters are supplied.

  May return the supplied matrix if only one matrix is supplied.

- _mult()

  Implements the overloaded '*' operator. Internal use.

  Currently both the left and right-hand sides of the operator must be an Imager::Matrix2d.

  When composing a matrix for transformation you should multiply the matrices in the reverse order of the transformations:

```
my $shear = Imager::Matrix2d->shear(x => 0.1);
my $rotate = Imager::Matrix2d->rotate(degrees => 45);
my $shear_then_rotate = $rotate * $shear;
```

or use the compose method:

```
my $shear_then_rotate = Imager::Matrix2d->compose($shear,
 ↪$rotate);
```

- _add()

  Implements the overloaded binary '+' operator.

  Currently both the left and right sides of the operator must be Imager::Matrix2d objects.

- _string()

  Implements the overloaded stringification operator.

  This returns a string containing 3 lines of text with no terminating newline.

  I tried to make it fairly nicely formatted. You might disagree :)

- _eq

  Implement the overloaded equality operator.

  Provided for older perls that don't handle magic auto generation of eq from "".

The following functions are shortcuts to the various constructors.

These are not methods.

You can import these methods with:

```
use Imager::Matrix2d ':handy';
```

- m2d_identity
- m2d_rotate()
- m2d_translate()
- m2d_shear()
- m2d_reflect()
- m2d_scale()

AUTHOR

Tony Cook tony@develop-help.com

BUGS

Needs a way to invert a matrix.

SEE ALSO

Imager(3), Imager::Font(3)

http://imager.perl.org/

Imager::Regops

generated information about the register based virtual machine

SYNOPSIS

```
use Imager::Regops;
$Imager::Regops::Attr{$opname}->{opcode} # opcode for given operator
$Imager::Regops::Attr{$opname}->{parms} # number of parameters
$Imager::Regops::Attr{$opname}->{types} # types of parameters
$Imager::Regops::Attr{$opname}->{func} # operator is a function
$Imager::Regops::Attr{$opname}->{result} # r for numeric, p for␣
↪pixel result
$Imager::Regops::MaxOperands; # maximum number of operands
```

DESCRIPTION

This module is generated automatically from `regmach.h` so we don't need to maintain the same information in at least one extra place.

At least that's the idea.

AUTHOR

Tony Cook, tony@develop-help.com

SEE ALSO

perl(1), Imager(3), http://imager.perl.org/

Imager::Test

common functions used in testing Imager

SYNOPSIS

```
use Imager::Test 'diff_text_with_nul';
diff_text_with_nul($test_name, $text1, $text2, @string_options);
```

DESCRIPTION

This is a repository of functions used in testing Imager.

Some functions will only be useful in testing Imager itself, while others should be useful in testing modules that use Imager.

No functions are exported by default.

FUNCTIONS

Test functions

- is_color1($color, $grey, $comment)

  Tests if the first channel of $color matches $grey.

- is_color3($color, $red, $green, $blue, $comment)

  Tests if $color matches the given ($red, $green, $blue)

- is_color4($color, $red, $green, $blue, $alpha, $comment)

  Tests if $color matches the given ($red, $green, $blue, $alpha)

- is_fcolor1($fcolor, $grey, $comment)

- is_fcolor1($fcolor, $grey, $epsilon, $comment)

  Tests if $fcolor's first channel is within $epsilon of ($grey). For the first form $epsilon is taken as 0.001.

- is_fcolor3($fcolor, $red, $green, $blue, $comment)

- is_fcolor3($fcolor, $red, $green, $blue, $epsilon, $comment)

  Tests if $fcolor's channels are within $epsilon of ($red, $green, $blue). For the first form $epsilon is taken as 0.001.

- is_fcolor4($fcolor, $red, $green, $blue, $alpha, $comment)

- is_fcolor4($fcolor, $red, $green, $blue, $alpha, $epsilon, $comment)

  Tests if $fcolor's channels are within $epsilon of ($red, $green, $blue, $alpha). For the first form $epsilon is taken as 0.001.

- is_image($im1, $im2, $comment)

  Tests if the 2 images have the same content. Both images must be defined, have the same width, height, channels and the same color in each pixel. The color comparison is done at 8-bits per pixel. The color representation such as direct vs paletted, bits per sample are not checked. Equivalent to is_image_similar($im1, $im2, 0, $comment).

- is_imaged($im, $im2, $comment)

- is_imaged($im, $im2, $epsilon, $comment)

  Tests if the two images have the same content at the double/sample level. `$epsilon` defaults to the platform DBL_EPSILON multiplied by four.

- is_image_similar($im1, $im2, $maxdiff, $comment)

  Tests if the 2 images have similar content. Both images must be defined, have the same width, height and channels. The sum of the squares of the differences of each sample are calculated and must be less than or equal to *$maxdiff* for the test to pass. The color comparison is done at 8-bits per pixel. The color representation such as direct vs paletted, bits per sample are not checked.

- isnt_image($im1, $im2, $comment)

  Tests that the two images are different. For regressions tests where something (like text output of "0") produced no change, but should have produced a change.

- test_colorf_gpix($im, $x, $y, $expected, $epsilon, $comment)

  Retrieves the pixel ($x,$y) from the low-level image $im and compares it to the floating point color $expected, with a tolerance of epsilon.

- test_color_gpix($im, $x, $y, $expected, $comment)

  Retrieves the pixel ($x,$y) from the low-level image $im and compares it to the floating point color $expected.

- test_colorf_glin($im, $x, $y, $pels, $comment)

  Retrieves the floating point pixels ($x, $y)-[$x+@$pels, $y] from the low level image $im and compares them against @$pels.

- is_color_close3($color, $red, $green, $blue, $tolerance, $comment)

  Tests if $color's first three channels are within $tolerance of ($red, $green, $blue).

Test suite functions

Functions that perform one or more tests, typically used to test various parts of Imager's implementation.

- std_image_tests(\%options)

  Performs various basic tests. Assumes you're doing modern `no_plan` testing.

- image_bounds_checks($im)

  Attempts to write to various pixel positions outside the edge of the image to ensure that it fails in those locations.

  Any new image type should pass these tests. Does 16 separate tests.

- mask_tests($im, $epsilon)

  Perform a standard set of mask tests on the OO image $im. Does 24 separate tests.

- diff_text_with_nul($test_name, $text1, $text2, @options)

  Creates 2 test images and writes $text1 to the first image and $text2 to the second image with the string() method. Each call adds 3 `ok`/`not ok` to the output of the test script.

  Extra options that should be supplied include the font and either a color or channel parameter.

  This was explicitly created for regression tests on #21770.

- std_font_tests({ font => $font })

  Perform standard font interface tests.

- std_font_test_count()

  The number of tests performed by std_font_tests().

Helper functions

- test_image_raw()

  Returns a 150x150x3 Imager::ImgRaw test image.

- test_image()

  Returns a 150x150x3 8-bit/sample OO test image. Name: `basic`.

- test_image_16()

  Returns a 150x150x3 16-bit/sample OO test image. Name: `basic16`

- test_image_double()

  Returns a 150x150x3 double/sample OO test image. Name: `basic_double`.

- test_image_gray()

  Returns a 150x150 single channel OO test image. Name: `gray`.

- test_image_gray_16()

  Returns a 150x150 16-bit/sample single channel OO test image. Name: `gray16`.

- test_image_mono()

  Returns a 150x150 bilevel image that passes the is_bilevel() test. Name: `mono`.

- test_image_named($name)

  Return one of the other test images above based on name.

- color_cmp($c1, $c2)

  Performs an ordering of 3-channel colors (like <=>).

- colorf_cmp($c1, $c2)

  Performs an ordering of 3-channel floating point colors (like <=>).

AUTHOR

Tony Cook tony@develop-help.com

Imager::Transform

a library of register machine image transformations

SYNOPSIS

```
# get a list of transformations
my @funcs = Imager::Transform->list;
# create a transformation object
my $tran = Imager::Transform->new($name);
# describe it
print $tran->describe;
# a list of constant names
my @constants = $tran->constants;
# information about some of the constants
my @info = $tran->constants(@constants);
```

DESCRIPTION

This module provides a library of transformations that use the Imager transform2() function.

The aim is to provide a place to collect these transformations.

At some point there might be an interface to add new functions, but there's not a whole lot of point to that.

The interface is a little sparse as yet.

METHODS

- my @names = Imager::Transform->list

  Returns a list of the transformations.

- my $desc = Imager::Transform->describe($name);

- my $desc = $tran->describe()

  Describes a transformation specified either by name (as a class method) or by reference (as an instance method).

  The class method returns undef if there is no such transformation.

- my $tran = Imager::Transform->new($name)

  Create a new transformation object. Returns undef if there is no such transformation.

- my @inputs = $tran->inputs;

- my $inputs = $tran->inputs;

  Returns a list of input image descriptions, or the number of them, depending on context.

  The list contains hash references, which currently contain only one member, `desc`, a description of the use of the input image.

- $tran->constants

  Returns a list of names of constants that can be set for the transformation.

- $tran->constants($name, $name, ...)

  Returns a hashref for each named constant, which contains the default in key `default` and a description in key `desc`.

- my $out = $tran->transform(\%opts, \%constants, @imgs)

  Perform the image transformation.

  Returns the new image on success, or undef on failure, in which case you can use $tran->errstr to get an error message.

- $tran->errstr

  The error message, if any from the last image transformation.

BUGS

Needs more transformations.

SEE ALSO

Imager(3), `transform.perl`

AUTHOR

Tony Cook tonyc@cpan.org

Imager::TrimColorList

represent a list of color ranges for Imager's trim() method.

SYNOPSIS

```
use Imager::TrimColorList;

# empty list
my $tcl = Imager::TrimColorList->new;

# add an entry in a variety of forms
$tcl->add($entry);

# add an explicit color object entry
$tcl->add_color($c1, $c2);

# add an explicit floating color object entry
$tcl->add_fcolor($cf1, $cf2);

# number of entries
my $count = $tcl->count;

# fetch an entry
```

(continues on next page)

(continued from previous page)

```
my $entry = $tcl->get($index);

# fetch all entries
my @all = $tcl->all;

# make a list and populate it
my $tcl = Imager::TrimColorList->new($entry1, $entry2, ...);

# dump contents of the list as a string
print $tcl->describe;
```

DESCRIPTION

An Imager::TrimColorList represents a list of color ranges to supply to the trim() method.

Each range can be either an 8-bit color range, ie. *Imager::Color* objects, or a floating point color range, ie. *Imager::Color::Float* objects, these can be mixed freely in a single list but each range must be 8-bit or floating point.

You can supply an entry in a small variety of forms:

- a simple color object of either type, or something convertible to a color object such as a color name such as `"red"`, a hex color such as `"#FFF"`. Any of the forms that Imager::Color supports should work here *except* for the array form. This becomes a range of only that color.

  ```
  $tcl->add("#000");
  $tcl->add(Imager::Color->new(0, 0, 0));
  $tcl->add(Imager::Color::Float->new(0, 0, 0));
  ```

- an arrayref containing a single color object, or something convertible to a color object. This becomes a range of only that color.

  ```
  $tcl->add([ "#000" ]);
  $tcl->add([ Imager::Color->new(0, 0, 0) ]);
  $tcl->add([ Imager::Color::Float->new(0, 0, 0) ]);
  ```

- an arrayref containing two color objects of the same type, ie. both Imager::Color objects or convertible to Imager::Color objects, or two Imager::Color::Float objects. This becomes a range between those two colors inclusive.

  ```
  $tcl->add([ "#000", "#002" ]);
  $tcl->add([ Imager::Color->new(0, 0, 0), Imager::Color->new(0,␣
  ↪0, 2) ]);
  $tcl->add([ Imager::Color::Float->new(0, 0, 0),␣
  ↪Imager::Color::Float->new(0, 0, 2/255) ]);
  ```

- an arrayref containing a color object of either type and a number representing the variance within the color space on either side of the specified color to include.

```
$tcl->add([ "#000", 0.01 ])
$tcl->add([ Imager::Color->new(0, 0, 0), 0.01 ]);
$tcl->add([ Imager::Color::Float->new(0, 0, 0), 0.01 ]);
```

A range specified this way with an 8-bit color clips at the top and bottom of the sample ranges, so the example 8-bit range above goes from (0, 0, 0) to (2, 2, 2) inclusive, while the floating point range isn't clipped and results in the floating color range (-0.01, -0.01, -0.01) to (0.01, 0.01, 0.01) inclusive.

METHODS

- new()
- new($entry1, ...)

  Class method. Create a new Imager::TrimColorList object and optionally add some color ranges to it.

  Returns an optionally populated Imager::TrimColorList object, or an empty list (or undef) on failure.

- add($entry)

  Add a single range entry. Note that this accepts a single value which can be a color or convertible to a color, or a reference to an array as described above.

  Returns a true value on success and a false value on failure.

- add_color($color1, $color2)

  Add a single 8-bit color range going from the `$color1` object to the `$color2` object inclusive. Both parameters must be Imager::Color objects or an exception is thrown.

- add_fcolor($fcolor1, $fcolor2)

  Add a single floating point color range going from the `$fcolor1` object to the `$fcolor2` object inclusive. Both parameters must be Imager::Color::Float objects or an exception is thrown.

- count()

  Fetch the number of color ranges stored in the object.

- get($index)

  Fetch the color range at the given index. This returns a reference to an array containing either two Imager::Color objects or two Imager::Color::Float objects.

  Returns undef if `$index` is out of range and does not set `Imager->errstr`.

- all()

  Fetch all ranges from the object.

- describe()

  Return a string describing the color range as code that can create the object.

- clone()

  Duplicate the object.

- auto()

  Automatically produce a trim color list based on an input image.

  This is used to implement 'auto' for image trim() and trim_rect() methods.

  Parameters:

  - `image` - the image to base the color list on. Required.
  - `auto` - the mechanism used to produce the color list, one of:
    * 1 - a "best" mechanism is selected, this is currently the `center` method, but it subject to change.
    * `center`, `centre` - the pixels at the center of each side of the image are used.

    Default: 1.
  - `tolerance` - used to control the range of pixel colors to be accepted as part of the color list. Default: 0.01.
  - `name` - used internally to attribute errors back to the original method. Default: `auto`.

If any method returns an error you can fetch a diagnostic from `Imager->errstr`.

AUTHOR

Tony Cook tonyc@cpan.org

HISTORY

Originally the range handling for this was going to be embedded in the trim() method, but this meant that every call that used color ranges would pay some cost as the range list was checked for names (vs color objects) and non-standard forms such as single colors and the color plus variance.

The object allows a simple test for the trim() `colors` parameter that doesn't pay that cost, and still allows a caller to use the explicit convention.

5.2.60 JSON

Namespace containing the following modules:

- `JSON::PP` - Encode Perl data as JSON text and decode JSON text into Perl data, with JSON::PP's options and answers.
- `JSON::XS` - Encode Perl data as JSON text and decode JSON text into Perl data, in compiled code.

JSON::PP

Encode Perl data as JSON text and decode JSON text into Perl data, with JSON::PP's options and answers.

`encode_json` and `decode_json` convert between a Perl structure and UTF-8 encoded JSON. An object made with `JSON::PP->new` takes JSON::XS's options and JSON::PP's own - `allow_singlequote`, `allow_barekey`, `allow_bignum`, `loose`, `escape_slash`, `indent_length`, `sort_by`, `as_nonblessed`, `core_bools` - and parses incrementally with `incr_parse`. JSON `true` and `false` decode to `$JSON::PP::true` and `$JSON::PP::false`.

This is JSON::PP 4.16 over pperl's one JSON engine, the one under JSON::XS: the object and its option methods are JSON::PP's own (`JSON/PP.pm`), encoding and decoding are the engine's, in JSON::PP's dialect.

Synopsis

```
use JSON::PP;

my $text = encode_json { name => "caf\x{e9}", list => [1, 2.5, "x"]␣
 ↪};
my $data = decode_json '{"a":[1,true,null]}';

my $json = JSON::PP->new->pretty->canonical->escape_slash;
print $json->encode({ url => "http://example.com/" });

my $loose = JSON::PP->new->allow_singlequote->allow_barekey;
my $hash = $loose->decode("{key:'value'}");
```

Differences from upstream

- With `allow_tags`, the values a FREEZE method returns are encoded as JSON; JSON::PP joins them as they are, which is not JSON and loses references (`("T")[3,x,ARRAY(0x...)]`). A FREEZE that returns the object itself among them dies, as it would recurse.
- `decode_prefix` on a character string returns the number of characters it decoded, as its documentation says; JSON::PP returns the number of bytes of their UTF-8 encoding, and a caller that cuts the text there loses characters. `incr_parse` keeps the rest of its buffer as characters too; JSON::PP turns it into bytes.
- `incr_parse` on character strings takes the text as it is; JSON::PP decodes it as UTF-8 first, which changes Latin-1 text that happens to form UTF-8 (`"\xc3\xa9"` becomes `"\xe9"`).
- A bare key with non-ASCII characters under `allow_barekey` keeps them; JSON::PP gives the bytes of their UTF-8 (`{café:1}` decodes to the key "cafÃ©").
- A tag with no value under `allow_tags` - a lone `(`, or `(null)` - is an error; JSON::PP returns nothing for it, which decodes to undef or fails later with an unrelated message.

- A high surrogate escape followed by another (`"\ud800\ud83d\ude00"`) is an error; JSON::PP drops the first one and its character.
- A character from U+110000 on dies when encoded, as under JSON::XS; JSON::PP writes two `\u` escapes that decode to other characters.
- Errors perl itself raises on a wrong invocant (`Not a HASH reference` and the like) name the caller's line; JSON::PP's name a line in its own file.

Modules

- `JSON::PP::Boolean` - dummy module providing JSON::PP::Boolean

Functions

Data serialisation

encode

Perl data to JSON text, with this object's options.

Also available as `JSON::PP::PP_encode_json`.

decode

JSON text to Perl data, with this object's options.

decode_prefix

Decode the JSON text at the start of a string.

encode_json

Perl data to UTF-8 encoded JSON text.

decode_json

UTF-8 encoded JSON text to Perl data.

to_json

The old name of `encode_json`; it only dies, saying so.

What you get back

Nothing: it croaks "JSON::PP::to_json has been renamed to encode_json.". `from_json` does the same for `decode_json`.

from_json

The old name of `decode_json`; it only dies, saying so.

new

A new encoder/decoder with the defaults: `allow_nonref` on, a maximum depth of 512, an indent of 3, everything else off.

ascii

The option setters: `ascii`, `latin1`, `utf8`, `indent`, `canonical`, `space_before`, `space_after`, `allow_nonref`, `shrink`, `allow_blessed`, `convert_blessed`, `relaxed`, `allow_unknown`, `allow_tags`, and JSON::PP's own `allow_singlequote`, `allow_barekey`, `allow_bignum`, `loose`, `escape_slash` and `as_nonblessed`.

get_ascii

The option getters: `get_ascii`, `get_utf8`, `get_allow_barekey` and the others of each setter.

Synopsis

```
print JSON::PP->new->get_allow_nonref ? "on\n" : "off\n";   # on
```

What you get back

1 when the option is on, the empty string otherwise.

pretty

Set `indent`, `space_before` and `space_after` together.

Synopsis

```
print JSON::PP->new->pretty->encode({ a => [1, 2] });
```

What you get back

The object; `pretty(0)` turns the three off.

max_depth

The deepest nesting of arrays and objects that encode and decode accept.

get_max_depth

The limit `max_depth` set, as it was given.

max_size

The largest JSON text, in bytes, that decode accepts.

Synopsis

```
my $json = JSON::PP->new->max_size(1 << 20);
```

What you get back

The object. 0 (the default, and without an argument) means no limit, and so does 1. `get_max_size` returns the value as it was set.

get_max_size

The limit `max_size` set, as it was given.

boolean_values

The values JSON `false` and `true` decode to.

get_boolean_values

The values set by `boolean_values`, false first.

What you get back

The two values, or the empty list when none are set.

core_bools

Decode `true` and `false` to Perl's core booleans, `!!1` and `!!0`.

get_core_bools

Whether `core_bools` is on.

unblessed_bool

The same as `core_bools`.

get_unblessed_bool

The same as `get_core_bools`.

filter_json_object

A sub that each decoded object is passed to; what it returns takes the object's place.

filter_json_single_key_object

A sub for objects of a single key, by that key.

indent_length

The number of spaces each level indents by under `indent`, 0 to 15.

Synopsis

```
print JSON::PP->new->pretty->indent_length(2)->encode({ a => [1] });
```

What you get back

The object. A value outside 0 to 15 warns and leaves the length as it was; the default is 3.

get_indent_length

The indent `indent_length` set.

sort_by

The order in which an object's keys are written.

allow_bigint

The old name of `allow_bignum`: it warns and turns `allow_bignum` on.

incr_parse

Parse JSON texts as they arrive, a piece at a time.

incr_skip

Drop the text the incremental parser has looked at, after an error.

Synopsis

```
eval { $json->incr_parse };
$json->incr_skip if $@;
```

incr_reset

Forget everything the incremental parser holds.

incr_text

The text the incremental parser holds, as an lvalue.

Synopsis

```
$json->incr_text = '[1,2]';
my $data = $json->incr_parse;
```

What you get back

The buffer itself. It dies once parsing has started on its text.

is_bool

Whether a value is a JSON boolean.

true

`$JSON::PP::true`, the value JSON `true` decodes to.

false

`$JSON::PP::false`, the value JSON `false` decodes to.

null

undef, the value JSON `null` decodes to.

JSON::PP::Boolean

dummy module providing JSON::PP::Boolean

SYNOPSIS

```
# do not "use" yourself
```

DESCRIPTION

This module exists only to provide overload resolution for Storable and similar modules. See *JSON::PP* for more info about this class.

AUTHOR

This idea is from *JSON::XS::Boolean* written by Marc Lehmann <schmorp[at]schmorp.de>

LICENSE

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

encode

Perl data to JSON text, with this object's options.

Also available as `JSON::PP::PP_encode_json`.

Synopsis

```
my $text = JSON::PP->new->canonical->encode({ b => 1, a => [undef, \
↪1] });
# {"a":[null,true],"b":1}
```

What you get back

The JSON text: characters, or UTF-8 bytes under `utf8`. Dies on what JSON cannot represent unless an option allows it: a blessed object, a code reference, a reference to a scalar other than 0 or 1, nesting deeper than `max_depth`.

decode

JSON text to Perl data, with this object's options.

Synopsis

```
my $data = JSON::PP->new->relaxed->decode('[1, 2, // a comment
                                           3,]');
```

What you get back

The Perl structure: `true` and `false` as `$JSON::PP::true` and `$JSON::PP::false`, `null` as undef, numbers as numbers. Dies with the reason and the offset on text that is not valid JSON.

decode_prefix

Decode the JSON text at the start of a string.

Synopsis

```
my ($data, $chars) = JSON::PP->new->decode_prefix('[1,2] trailing␣
↪text');
# ([1, 2], 5)
```

What you get back

The Perl structure and the number of characters its text took; what follows is not looked at.

encode_json

Perl data to UTF-8 encoded JSON text.

Synopsis

```
use JSON::PP;
my $bytes = encode_json { name => "caf\x{e9}" };   # {"name":"caf\
↪xc3\xa9"}
```

What you get back

The JSON text as UTF-8 bytes: `JSON::PP->new->utf8->encode($data)`. Exported by default.

decode_json

UTF-8 encoded JSON text to Perl data.

Synopsis

```
use JSON::PP;
my $data = decode_json '{"list":[1,true,null]}';
```

What you get back

The Perl structure: `JSON::PP->new->utf8->decode($bytes)`. `true` and `false` are `$JSON::PP::true` and `$JSON::PP::false`. Exported by default.

new

A new encoder/decoder with the defaults: `allow_nonref` on, a maximum depth of 512, an indent of 3, everything else off.

Synopsis

```
my $json = JSON::PP->new->utf8->canonical->escape_slash;
```

What you get back

A hash blessed into the class it was called on, as JSON::PP's objects always are: a subclass may keep its own keys in it. The option methods return it again, so that they chain.

ascii

The option setters: `ascii`, `latin1`, `utf8`, `indent`, `canonical`, `space_before`, `space_after`, `allow_nonref`, `shrink`, `allow_blessed`, `convert_blessed`, `relaxed`, `allow_unknown`, `allow_tags`, and JSON::PP's own `allow_singlequote`, `allow_barekey`, `allow_bignum`, `loose`, `escape_slash` and `as_nonblessed`.

Synopsis

```
my $json = JSON::PP->new->allow_singlequote->allow_barekey;
my $data = $json->decode(q({key: 'value'}));
```

What you get back

The object itself, with the option on (no argument, or a defined true one) or off.

- `allow_singlequote`: strings may also be quoted with `'`.
- `allow_barekey`: an object's keys may be bare words of ASCII word characters, `$` and non-ASCII characters.
- `allow_bignum`: numbers too large or too precise for Perl decode to Math::BigInt and Math::BigFloat objects, and such objects encode as numbers.
- `loose`: unknown backslash escapes and control characters are taken as they are.
- `escape_slash`: `/` is written as `\/`.
- `as_nonblessed`: with `allow_blessed`, a blessed hash or array is encoded as if it were not blessed.
- `shrink` is accepted and changes nothing.

max_depth

The deepest nesting of arrays and objects that encode and decode accept.

Synopsis

```
my $json = JSON::PP->new->max_depth(16);
```

What you get back

The object. Without an argument the limit is 2**31; deeper data dies "json text or perl structure exceeds maximum nesting level (max_depth set too low?)". `get_max_depth` returns the value as it was set.

boolean_values

The values JSON `false` and `true` decode to.

Synopsis

```
my $json = JSON::PP->new->boolean_values(0, 1);
my ($false, $true) = $json->get_boolean_values;
```

What you get back

The object. With no arguments the default booleans, `$JSON::PP::false` and `$JSON::PP::true`, come back. Two core booleans (`!!0`, `!!1`) also turn `core_bools` on.

core_bools

Decode `true` and `false` to Perl's core booleans, `!!1` and `!!0`.

Synopsis

```
my $data = JSON::PP->new->core_bools->decode('[true]');
builtin::is_bool($data->[0]);   # true
```

What you get back

The object; `core_bools(0)` goes back to `$JSON::PP::true` and `$JSON::PP::false`. `unblessed_bool` is the same method, and core booleans always encode as `true` and `false`.

filter_json_object

A sub that each decoded object is passed to; what it returns takes the object's place.

Synopsis

```
my $json = JSON::PP->new->filter_json_object(sub { bless $_[0],
↪'Point' });
```

What you get back

The object. A filter that returns nothing leaves the object as it is; one that returns more than one value dies. A non-code argument removes the filter.

filter_json_single_key_object

A sub for objects of a single key, by that key.

Synopsis

```
my $json = JSON::PP->new->filter_json_single_key_object(
    '$date' => sub { Time::Piece->strptime($_[0], '%F') });
```

What you get back

The object. The sub gets the key's value; what it returns replaces the object. When it returns nothing, the object stays, and the `filter_json_object` filter is not called for it.

sort_by

The order in which an object's keys are written.

Synopsis

```
my $json = JSON::PP->new->sort_by(sub { $JSON::PP::b cmp
↪$JSON::PP::a });
```

What you get back

The object. A code reference, or the name of a sub in JSON::PP, sorts the keys, comparing `$JSON::PP::a` with `$JSON::PP::b`; any other true value sorts them as `canonical` does.

incr_parse

Parse JSON texts as they arrive, a piece at a time.

Synopsis

```
my $json = JSON::PP->new;
$json->incr_parse('[1,');              # append, nothing complete␣
↪yet
my @texts = $json->incr_parse('2][3]');   # ([1, 2], [3])
```

What you get back

In void context nothing: the text is only appended. In scalar context the first complete JSON text decoded, or undef while none is complete; in list context every complete one.

is_bool

Whether a value is a JSON boolean.

Synopsis

```
JSON::PP::is_bool(JSON::PP::true);    # true
JSON::PP::is_bool(!!1);               # true: a core boolean
JSON::PP::is_bool(1);                 # false
```

What you get back

True for an object of JSON::PP::Boolean, Types::Serialiser::BooleanBase or JSON::XS::Boolean, and for a core boolean.

JSON::XS

Encode Perl data as JSON text and decode JSON text into Perl data, in compiled code.

`encode_json` and `decode_json` convert between a Perl structure and UTF-8 encoded JSON. An object made with `JSON::XS->new` takes options

- `utf8`, `pretty`, `canonical`, `relaxed`, `allow_nonref`, `convert_blessed`, `allow_tags`, `max_depth`, `max_size`, filter callbacks - and parses incrementally from a stream with `incr_parse`. JSON `true` and `false` decode to `Types::Serialiser`'s booleans.

This is a native implementation of JSON::XS 4.04's `XS.xs`; its `JSON/XS.pm` is used unchanged.

Synopsis

```
use JSON::XS;

my $text = encode_json { name => "caf\x{e9}", list => [1, 2.5, "x"]␣
↪};
my $data = decode_json '{"a":[1,true,null]}';

my $json = JSON::XS->new->pretty->canonical;
print $json->encode($data);

$json->incr_parse('[1,2');
$json->incr_parse(',3]');
my $array = $json->incr_parse;           # [1, 2, 3]
```

Differences from upstream

- A method called on something blessed into `JSON::XS` that `JSON::XS->new` did not make - a hash, an array, a short string - dies "object is not of type JSON::XS"; perl crashes.
- After `JSON::XS::CLONE` has been called by hand, `new` and the decoding of `true` and `false` work as they do in a perl built with threads; a perl without threads crashes.

Modules

- `JSON::XS::Boolean` - dummy module providing JSON::XS::Boolean

Functions

Data serialisation

new

A new encoder/decoder with the defaults: `allow_nonref` on, a maximum depth of 512, everything else off.

Synopsis

```
my $json = JSON::XS->new->utf8->canonical;
```

What you get back

An object blessed into the class it was called on, whose option methods return it again, so that they chain.

allow_blessed

The option setters: `ascii`, `latin1`, `utf8`, `indent`, `canonical`, `space_before`, `space_after`, `pretty`, `allow_nonref`, `shrink`, `allow_blessed`, `convert_blessed`, `relaxed`, `allow_unknown` and `allow_tags`.

Also available as `JSON::XS::allow_nonref`, `JSON::XS::allow_tags`, `JSON::XS::allow_unknown`, `JSON::XS::ascii`, `JSON::XS::canonical`, `JSON::XS::convert_blessed`, `JSON::XS::indent`, `JSON::XS::latin1`, `JSON::XS::pretty`, `JSON::XS::relaxed`, `JSON::XS::shrink`, `JSON::XS::space_after`, `JSON::XS::space_before`, `JSON::XS::utf8`.

get_allow_blessed

The option getters: `get_ascii`, `get_utf8`, `get_canonical` and the others of each setter but `pretty`.

Also available as `JSON::XS::get_allow_nonref`, `JSON::XS::get_allow_tags`, `JSON::XS::get_allow_unknown`, `JSON::XS::get_ascii`, `JSON::XS::get_canonical`, `JSON::XS::get_convert_blessed`, `JSON::XS::get_indent`, `JSON::XS::get_latin1`, `JSON::XS::get_relaxed`, `JSON::XS::get_shrink`, `JSON::XS::get_space_after`, `JSON::XS::get_space_before`, `JSON::XS::get_utf8`.

Synopsis

```
print JSON::XS->new->get_allow_nonref ? "on\n" : "off\n";   # on
```

What you get back

True when the option is on, false otherwise.

encode

Perl data to JSON text, with this object's options.

decode

JSON text to Perl data, with this object's options.

decode_prefix

Decode the JSON text at the start of a string.

incr_parse

Parse JSON texts as they arrive, a piece at a time.

encode_json

Perl data to UTF-8 encoded JSON text.

Synopsis

```
use JSON::XS;
my $bytes = encode_json { name => "caf\x{e9}" };   # {"name":"caf\
↪xc3\xa9"}
```

What you get back

The JSON text as UTF-8 bytes: `JSON::XS->new->utf8->encode($data)`.

decode_json

UTF-8 encoded JSON text to Perl data.

JSON::XS::Boolean

dummy module providing JSON::XS::Boolean

SYNOPSIS

```
# do not "use" yourself
```

DESCRIPTION

This module exists only to provide overload resolution for Storable and similar modules. It's only needed for compatibility with data serialised (by other modules such as Storable) that was decoded by JSON::XS versions before 3.0.

Since 3.0, JSON::PP::Boolean has replaced it. Support for JSON::XS::Boolean will be removed in a future release.

AUTHOR

```
Marc Lehmann <schmorp@schmorp.de>
http://home.schmorp.de/
```

allow_blessed

The option setters: `ascii`, `latin1`, `utf8`, `indent`, `canonical`, `space_before`, `space_after`, `pretty`, `allow_nonref`, `shrink`, `allow_blessed`, `convert_blessed`, `relaxed`, `allow_unknown` and `allow_tags`.

Also available as `JSON::XS::allow_nonref`, `JSON::XS::allow_tags`, `JSON::XS::allow_unknown`, `JSON::XS::ascii`, `JSON::XS::canonical`, `JSON::XS::convert_blessed`, `JSON::XS::indent`, `JSON::XS::latin1`, `JSON::XS::pretty`, `JSON::XS::relaxed`, `JSON::XS::shrink`, `JSON::XS::space_after`, `JSON::XS::space_before`, `JSON::XS::utf8`.

Synopsis

```
my $json = JSON::XS->new->utf8->pretty->canonical(1)->relaxed(0);
```

What you get back

The object itself, with the option on (no argument, or a true one) or off (a false one). `pretty` sets `indent`, `space_before` and `space_after` together.

encode

Perl data to JSON text, with this object's options.

Synopsis

```
my $text = JSON::XS->new->canonical->encode({ b => 1, a => [undef, \
 ↪1] });
# {"a":[null,true],"b":1}
```

What you get back

The JSON text: characters, or UTF-8 bytes under `utf8`. Dies on what JSON cannot represent unless an option allows it: a blessed object, a code reference, a reference to a scalar other than 0 or 1, nesting deeper than `max_depth`.

decode

JSON text to Perl data, with this object's options.

Synopsis

```
my $data = JSON::XS->new->relaxed->decode('[1, 2, # a comment
                                           3,]');
```

What you get back

The Perl structure: `true` and `false` as `Types::Serialiser`'s booleans, `null` as undef, numbers as numbers. Dies with the reason and the character offset on text that is not valid JSON.

decode_prefix

Decode the JSON text at the start of a string.

Synopsis

```
my ($data, $chars) = JSON::XS->new->decode_prefix('[1,2] trailing␣
 ↪text');
# ([1, 2], 5)
```

What you get back

The Perl structure and the number of characters its text took; what follows is not looked at.

incr_parse

Parse JSON texts as they arrive, a piece at a time.

Synopsis

```
my $json = JSON::XS->new;
$json->incr_parse('[1,');           # append, nothing complete yet
my @texts = $json->incr_parse('2][3]');   # ([1, 2], [3])
```

What you get back

In void context nothing: the text is only appended. In scalar context the first complete JSON text decoded, or nothing while none is complete; in list context every complete one.

decode_json

UTF-8 encoded JSON text to Perl data.

Synopsis

```
use JSON::XS;
my $data = decode_json '{"list":[1,true,null]}';
```

What you get back

The Perl structure: `JSON::XS->new->utf8->decode($bytes)`. Dies on text that is not valid JSON, and on characters above 255 (the text must be bytes).

5.2.61 LMDB_File

Tie a Perl hash to an LMDB database, or drive LMDB's environments, transactions and cursors directly.

LMDB_File is the CPAN interface to OpenLDAP's Lightning Memory-Mapped Database. An environment is a directory (or, with `MDB_NOSUBDIR`, a file) holding one unnamed database and, with `maxdbs`, named ones. Every read and write happens in a transaction: read transactions see a snapshot and never block, and one write transaction at a time commits atomically.

The simplest use is a tied hash, which opens the environment, begins a transaction and commits it when the hash is untied. The object interface gives the environment (`LMDB::Env`), transactions (`LMDB::Txn`), databases (`LMDB_File`) and cursors (`LMDB::Cursor`) explicitly.

Synopsis

```
use LMDB_File qw(:flags :cursor_op);

tie my %h, 'LMDB_File', 'data-dir';
$h{greeting} = 'hello';
untie %h;

my $env = LMDB::Env->new('data-dir', { mapsize => 1 << 30, maxdbs =>
 ↪4 });
my $txn = $env->BeginTxn;
my $db  = $txn->OpenDB({ dbname => 'users', flags => MDB_CREATE });
$db->put(alice => 'admin');
$txn->commit;
```

The tie interface

```
tie %h, 'LMDB_File', $path;                    # options as below
tie %h, 'LMDB_File', $path, \%options;
tie %h, 'LMDB_File', $path, $flags, $mode;     # as other DBM
 ↪modules
tie %h, 'LMDB_File', $env, \%dboptions;        # in an environment
tie %h, 'LMDB_File', $txn, \%dboptions;        # in a transaction
```

The forms with a path open (and create) the environment, begin a transaction and open the database in it; `%options` takes the environment's options and the database's (`dbname`, `flags`). A transaction the tie begins commits when the hash is untied. `$path` is a directory, which must exist, unless `MDB_NOSUBDIR` is among the flags.

LMDB::Env

`LMDB::Env->new($path, \%options)` creates (if need be) and opens an environment. The options: `mapsize` (the most data it can hold, default 1 MiB - choose it large, it reserves address space only), `maxdbs` (named databases allowed, default 0), `maxreaders` (default 126), `mode` (of files created, default `0600`) and `flags`, the `:envflags` constants such as `MDB_NOSUBDIR`, `MDB_RDONLY`, `MDB_NOSYNC`, `MDB_WRITEMAP`.

- `$env->BeginTxn($flags)` - a new `LMDB::Txn`; `MDB_RDONLY` for a read transaction (the environment's own `MDB_RDONLY` is passed on).
- `$env->copy($path)`, `$env->copyfd($fh)` - a consistent copy of the environment, to a directory or an open file.
- `$env->stat` - the unnamed database's statistics, a hash reference (`psize`, `depth`, `branch_pages`, `leaf_pages`, `overflow_pages`, `entries`); `$env->info` - the map's `mapaddr`, `mapsize`, `last_pgno`, `last_txnid`, `maxreaders`, `numreaders`.
- `$env->sync($force)` - flush to disk (needed with `MDB_NOSYNC`).
- `$env->set_flags($flags, $on)`, `$env->get_flags($var)` - change or read the flags that can change while open.
- `$env->get_path($var)`, `$env->get_maxreaders($var)`, `$env->get_maxkeysize` - the path, the reader slots, the longest key (511 bytes with the usual liblmdb).
- `$env->set_mapsize($size)` - grow the map while no transaction is live.
- `$env->id` - a number identifying the environment.

The environment closes when its last reference goes.

LMDB::Txn

A transaction is `$env->BeginTxn` or `LMDB::Txn->new($env, $flags)`. A read transaction sees a snapshot and never blocks; one write transaction at a time sees its own writes and makes them all durable at `commit`. A transaction that goes out of scope is aborted, unless `$txn->AutoCommit(1)` asks for a commit.

- `$txn->commit`, `$txn->abort` - end it; afterwards only `env` may be called.
- `$txn->reset`, `$txn->renew` - let a read transaction's snapshot go, and take a fresh one later.
- `$txn->SubTxn($flags)` - a nested transaction, whose writes join the parent's at its commit and vanish at its abort.
- `$txn->OpenDB(\%dboptions)` or `$txn->OpenDB($dbname, $flags)` - an `LMDB_File` database object; `$txn->open($dbname, $flags)` - the bare database number. The `:dbflags` constants: `MDB_CREATE`, `MDB_DUPSORT`, `MDB_INTEGERKEY` and the like.

- `$txn->put($dbi, $key, $value, $flags)`, `$txn->get($dbi, $key, $value)` - the fast path, on a database number.
- `$txn->env` - the environment; `$txn->id` - the transaction's ID.

LMDB_File: a database in a transaction

`$txn->OpenDB(...)`, `LMDB_File->open($txn, $dbname, $flags)` or `LMDB_File->new($txn, $dbi)` give an object that pairs a database with a transaction.

- `$db->get($key)` - the value or undef; `$db->get($key, $value)` puts it in `$value` and reports a missing key as an error.
- `$db->put($key, $value, $flags)` - store; the `:writeflags` constants `MDB_NOOVERWRITE`, `MDB_NODUPDATA`, `MDB_APPEND`, `MDB_APPENDDUP`, and `MDB_RESERVE` with a length to write the value in place afterwards.
- `$db->del($key, $value)` - delete the key, or in an `MDB_DUPSORT` database one of its values.
- `$db->set_compare(sub { $a cmp $b })` - a sort sub for the keys, called with `$a` and `$b` as `sort` calls one; set it before any other use, the same every time.
- `$db->ReadMode(1)` - `get` returns the value in place, a read-only scalar over the map, without copying it (writable with `MDB_WRITEMAP` in a write transaction); it is valid until the transaction ends or, in a write transaction, the next write. `$db->Rget($key)` does the same for one call.
- `$db->UTF8(1)` - store character strings as UTF-8 and decode them on the way out (`use bytes` skips the decoding).
- `$db->stat`, `$db->flags`, `$db->drop($remove)` (empty, or delete the database), `$db->Alive`, `$db->Txn` (assignable), `$db->dbi` (assignable), `$db->Cursor`.

LMDB::Cursor

`$db->Cursor` opens a cursor. `$cursor->get($key, $value, $op)` moves it by one of the `:cursor_op` operations (`MDB_FIRST`, `MDB_NEXT`, `MDB_SET_RANGE`, `MDB_NEXT_DUP` ...) and puts the key and value it lands on in `$key` and `$value`; `$cursor->put($key, $value, $flags)` stores and leaves the cursor on the item; `$cursor->del($flags)` deletes where it stands (`MDB_NODUPDATA`: all the key's values). `count`, `renew($txn)`, `txn`, `dbi` and `close` complete it.

Errors

By default an LMDB error dies with LMDB's message in `$@`. With `$LMDB_File::die_on_err` false a call returns the error code instead; either way `$LMDB_File::last_err` holds the last code. `LMDB_File::strerror($code)` gives a code's message.

Two more functions, not exported: `LMDB_File::version(my $major, my $minor, my $patch)` returns liblmdb's version string ("LMDB 0.9.35: (Jan 27, 2026)") and sets the three numbers; and `LMDB_File::reader_check($env, my $dead)` clears the reader slots left behind by processes that died with a read transaction open, setting `$dead` to how many it cleared.

Constants

`use LMDB_File qw(...)` imports by tag: `:envflags`, `:dbflags`, `:writeflags`, `:flags` (all three), `:cursor_op`, `:error` (the `MDB_*` error codes) and `:version` (`MDB_VERSION_STRING` and the like, the version of the liblmdb in use).

Differences from upstream

Four, each where perl 5.44 with XS LMDB_File fails (t/15-modules/LMDB_File/150-or-better.t):

- `LMDB::Cursor::_put` with `MDB_RESERVE` takes the length as its fifth argument. Upstream demands five arguments and then reads a sixth, past the end of them, and dies "MDB_RESERVE length must be > 0".
- A failed `LMDB::Cursor::open` leaves its argument undef, which `LMDB_File.pm` checks for; upstream fills it with a pointer to nothing.
- `$env->close` leaves the environment closed for good: closing it again, as its `DESTROY` does, is harmless, and the same database can be opened again. Upstream frees the environment twice and aborts.
- `$env->close` with a transaction still open ends the transaction first, as dropping the environment does; using it afterwards dies "Not an active transaction" or "Terminated transaction". Upstream leaves it pointing at freed memory and crashes.

liblmdb is the system's library, loaded when first needed. Everything else behaves as with upstream's LMDB_File, as its own test suite shows: `t/15-modules/LMDB_File/100-upstream-00-basic.t` to `140-upstream-04-bigone.t`.

5.2.62 List

Namespace containing the following modules:

- `List::MoreUtils`
- `List::Util` - The standard library of list reductions and scanners.

List::MoreUtils

Namespace containing the following modules:

- `List::MoreUtils::XS` - The compiled implementation of List::MoreUtils: the list functions List::Util does not have - junctions (`any`, `all`, `none`, `notall`, one and their `_u` variants that answer undef for an empty list), index and value searches from either end (`firstidx`, `lastval`, `onlyres`, ...), slicing by a predicate (`after`, `before_incl`), iterators (`each_array`, `natatime`, `slideatatime`), merging (`mesh`, `zip6`, `pairwise`, `arrayify`), counting (`uniq`, `singleton`, `duplicates`, `frequency`, `occurrences`, `mode`), binary searches over sorted lists (`bsearch`, `lower_bound`, `equal_range`, `binsert`, `bremove`) and an in-place `qsort`.

List::MoreUtils::XS

The compiled implementation of List::MoreUtils: the list functions List::Util does not have - junctions (`any`, `all`, `none`, `notall`, `one` and their `_u` variants that answer undef for an empty list), index and value searches from either end (`firstidx`, `lastval`, `onlyres`, ...), slicing by a predicate (`after`, `before_incl`), iterators (`each_array`, `natatime`, `slideatatime`), merging (`mesh`, `zip6`, `pairwise`, `arrayify`), counting (`uniq`, `singleton`, `duplicates`, `frequency`, `occurrences`, `mode`), binary searches over sorted lists (`bsearch`, `lower_bound`, `equal_range`, `binsert`, `bremove`) and an in-place `qsort`.

This is a native implementation of List::MoreUtils::XS 0.430's `XS.xs`; its `List/MoreUtils/XS.pm` is used unchanged, and List::MoreUtils, which picks it as its backend, comes from CPAN.

Synopsis

```
use List::MoreUtils qw(firstidx pairwise natatime uniq);

my $i = firstidx { $_ eq 'b' } qw(a b c);       # 1
my @s = pairwise { $a + $b } @x, @y;
my $it = natatime 3, 1 .. 10;
while (my @chunk = $it->()) { print "@chunk\n" }
my $n = uniq 1, 1, 2, undef, undef;              # 3
```

Differences from upstream

- A block that is a compiled sub (any `\&Scalar::Util::blessed, ...`) is called; perl jumps into it as if it were Perl code and crashes.
- `minmax` over objects that overload only other operators, with `fallback => 1`, compares their numeric values as `<=>` would; perl crashes. An overloaded `<=>` that returns a string or a fraction is read as a number; perl reads an unrelated field and answers wrongly (the maximum of 3, 1, 2 comes out as 2).
- `reduce_u`, `reduce_0` and `reduce_1` leave the caller's `$_` alone; perl overwrites it with the last index, which changes the current element of an enclosing `for (@array)` and dies inside `for (1)` with "Modification of a read-only value attempted".
- `frequency` no longer leaks a scalar per call when the list holds undef.
- An object that overloads `@{}` over a hash or a scalar is read through its overloading by `arrayify`, `mesh`, `zip6`, `listcmp`, `each_array`, `insert_after` and `insert_after_string`; perl reads the hash as an array and crashes.
- An array with holes (`$a[5] = 1` on an empty array) is read with undef in the holes by `arrayify`, `each_array`, `insert_after` and `insert_after_string`; perl crashes.
- Calling the iterator bodies `List::MoreUtils::XS::_array_iterator` and `_slideatatime_iterator` directly returns an empty list; perl crashes.

List::Util

The standard library of list reductions and scanners.

`List::Util` is the first stop when you need to collapse a list to a single answer (`sum`, `max`, `product`, `reduce`), find or test elements by predicate (`first`, `any`, `all`, `none`, `notall`), strip duplicates (`uniq`, `uniqnum`, `uniqint`, `uniqstr`), or reshape an even-sized key/value list (`pairs`, `pairkeys`, `pairmap`, `pairgrep`). It also covers slicing (`head`, `tail`), randomisation (`shuffle`, `sample`), and lockstep iteration over multiple arrays (`zip`, `mesh`, and their `_shortest` / `_longest` variants).

The block-taking reductions - `reduce`, `reductions`, `pairmap`, `pairgrep`, `pairfirst` - expose the current pair or accumulator through `$a` and `$b`. The per-element scanners - `first`, `any`, `all`, `none`, `notall` - expose the current element through `$_`.

`Sub::Util` (`subname`, `set_subname`, `set_prototype`) ships with the same distribution and becomes available once `List::Util` is loaded; import it from `Sub::Util` as usual.

This is List::Util from Scalar-List-Utils 1.70; `List/Util.pm` is upstream's. Nothing is exported by default.

Synopsis

```
use List::Util qw(sum max first any reduce uniq pairs pairmap head
 ↪shuffle zip);

my @n = (3, 9, 4, 1, 9);
print sum(@n), ' ', max(@n), "\n";                    # 26 9
print first { $_ > 3 } @n;                            # 9
print((any { $_ == 4 } @n) ? "has 4\n" : "no 4\n");   # has 4
print reduce { $a * $b } 1 .. 5;                      # 120
print join(',', uniq @n), "\n";                       # 3,9,4,1

my %age = (alice => 30, bob => 25);
for my $p (sort { $a->key cmp $b->key } pairs %age) {
    printf "%s is %d\n", $p->key, $p->value;          # alice is 30,
 ↪bob is 25
}
print join(',', pairmap { "$a=$b" } %age), "\n";      # alice=30,
 ↪bob=25 (hash order)
print join(',', head 2, @n), "\n";                    # 3,9
print join(' ', map { "[@$_]" } zip [1, 2], ['a', 'b']), "\n";   #
 ↪[1 a] [2 b]
my @deck = shuffle 1 .. 52;
```

`sum`, `product`, `min` and `max` of a long list of plain integers (65536 elements and more) run on several cores; the answer is the one the single-core loop gives, the same element for `min` and `max`. `--no-parallel` turns that off.

Differences from upstream

Each where perl 5.44 with ListUtil.xs fails (t/81-xs-native/List-Util/151-or-better.t):

- `unpairs 42` croaks "Not a reference"; perl5 crashes.
- `unpairs` takes an array reference held in a variable that once held a string (`my $x = "abc"; $x = [1, 2]`); perl5 refuses it.
- `mesh` reads a hole in an input array (`my @h; $h[2] = 5`) as undef; perl5 crashes.

Modules

- `List::Util::XS` - Require List::Util's compiled implementation

Functions

Reductions

sum

Add up every value in the list and return the numerical total.

sum0

Add up every value in the list, returning `0` for an empty list.

min

Return the numerically smallest value in the list.

max

Return the numerically largest value in the list.

product

Multiply every value in the list and return the numerical product.

minstr

Return the lexically smallest value in the list.

maxstr

Return the lexically largest value in the list.

`reduce`

Collapse a list to a single value by repeated pairwise combination.

`reductions`

Like `reduce`, but also return every intermediate accumulator value.

Scanners

`first`

Return the first list element for which a block returns true.

`any`

True if the block returns true for at least one element.

`all`

True if the block returns true for every element (including empty list).

`none`

True if the block returns true for no element (including empty list).

`notall`

True if the block returns false for at least one element.

Pair handling

`pairs`

Group an even-sized list into key/value pair objects.

`unpairs`

Flatten a list of two-element arrayrefs into a key/value list.

`pairkeys`

Extract just the keys from an even-sized key/value list.

`pairvalues`

Extract just the values from an even-sized key/value list.

pairmap

Run a block for each pair and concatenate the results.

pairgrep

Keep the pairs for which a block returns true.

pairfirst

Return the first pair for which a block returns true.

Selection

head

Return the first `$size` elements of a list.

tail

Return the last `$size` elements of a list.

uniq

Remove subsequent duplicates by string equality, preserving order.

uniqstr

Remove subsequent duplicates by string equality, treating `undef` as the empty string.

uniqnum

Remove subsequent duplicates by numerical equality, preserving order.

uniqint

Remove subsequent duplicates by integer equality, preserving order.

Misc

shuffle

Return the input list in a uniformly random order.

sample

Pick `$count` distinct elements from the list at random.

mesh

Interleave multiple arrays, padding shorter ones with undef.

mesh_longest

The same as mesh, by the name that says what it does with lists of different lengths: the elements run to the longest input, the shorter ones padded with undef.

mesh_shortest

Interleave multiple arrays, stopping at the shortest.

zip

Walk multiple arrays in lockstep, returning one arrayref per row.

zip_longest

The same as zip, by the name that says what it does with lists of different lengths: rows run to the longest input, the shorter ones padded with undef.

zip_shortest

Walk multiple arrays in lockstep, stopping at the shortest.

List::Util::XS

Require List::Util's compiled implementation

SYNOPSIS

```
use List::Util::XS 1.20;
```

DESCRIPTION

use List::Util::XS 1.20; is a dependency check: it fails unless List::Util runs its compiled implementation rather than a pure-Perl fallback. Under pperl List::Util is native, so the check always holds, and $List::Util::XS::VERSION is List::Util's version.

SEE ALSO

Scalar::Util, *List::Util*, *List::MoreUtils*

COPYRIGHT

Copyright (c) 2008 Graham Barr gbarr@pobox.com. All rights reserved. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

sum

Add up every value in the list and return the numerical total.

Synopsis

```
my $total = sum @values;
my $total = sum 1..10;           # 55
```

What you get back

A single number - an integer if every argument summed as an integer, a float otherwise. An integer total that would overflow continues as a float from the element that overflows. If any argument is an object with overloaded +, the total is built with that + and can be such an object. For an empty list, `sum` returns `undef` for backwards compatibility; if that is awkward use `sum0` instead.

Examples

```
my $n = sum 3, 9, 12;            # 24
my $n = sum @bar, @baz;          # concatenated lists
my $n = sum();                   # undef
my $n = sum map { length } @s;   # total length of all strings
```

Edge cases

- Empty list returns `undef`. Use `sum0` for a `0` default.
- Mixing ints and floats promotes the running total to a float.
- Non-numeric strings are coerced via the usual numification.

Differences from upstream

Fully compatible with upstream.

See also

- `sum0` - same as `sum`, but returns `0` for an empty list.
- `product` - multiplies rather than adds.
- `reduce` - the generic reduction when `sum` is not specific enough.

sum0

Add up every value in the list, returning `0` for an empty list.

Synopsis

```
my $total = sum0 @values;
my $total = sum0 ();              # 0, not undef
```

What you get back

The same number `sum` would produce, except that an empty list yields `0` instead of `undef`. This is almost always what callers want when adding up a list that may legitimately be empty.

Examples

```
my $n = sum0 1..10;               # 55
my $n = sum0 ();                  # 0
my $n = sum0 grep { $_ > 0 } @x; # total of positives, 0 if none
```

Differences from upstream

Fully compatible with upstream.

See also

- `sum` - the original; returns `undef` on empty input.
- `product` - identity element for multiplication (returns 1 on empty).

min

Return the numerically smallest value in the list.

Synopsis

```
my $lo = min @values;
my $lo = min 3, 9, 12;            # 3
```

What you get back

A single number. If every argument is an integer the result is returned as an integer; otherwise it is returned as a float. An empty list yields `undef`.

Examples

```
my $n = min 1..10;                # 1
my $n = min @bar, @baz;           # smallest across concatenated␣
↪lists
my $n = min();                    # undef
```

Edge cases

- Empty list returns `undef`.
- Strings are compared numerically (use `minstr` for lexical order).

Differences from upstream

Fully compatible with upstream.

See also

- `max` - the numerical maximum counterpart.
- `minstr` - string-wise minimum (`lt` semantics).
- `reduce` - custom minima (e.g. by a key function).

max

Return the numerically largest value in the list.

Synopsis

```
my $hi = max @values;
my $hi = max 3, 9, 12;          # 12
```

What you get back

A single number. If every argument is an integer the result is returned as an integer; otherwise it is returned as a float. An empty list yields `undef`.

Examples

```
my $n = max 1..10;              # 10
my $n = max @bar, @baz;         # largest across concatenated lists
my $n = max();                  # undef
```

Edge cases

- Empty list returns `undef`.
- Strings are compared numerically (use `maxstr` for lexical order).

Differences from upstream

Fully compatible with upstream.

See also

- `min` - the numerical minimum counterpart.
- `maxstr` - string-wise maximum (`gt` semantics).
- `reduce` - custom maxima (e.g. by a key function).

product

Multiply every value in the list and return the numerical product.

Synopsis

```
my $p = product @values;
my $p = product 1..10;          # 3628800
```

What you get back

A single number. Integer overflow is promoted to floating point transparently, so the result is an integer as long as the exact product fits in an `IV`. An empty list yields 1 (the multiplicative identity).

Examples

```
my $n = product 3, 9, 12;       # 324
my $n = product();              # 1
my $n = product map { $_->weight } @items;
```

Differences from upstream

Fully compatible with upstream.

See also

- `sum` - additive counterpart.
- `sum0` - additive counterpart that mirrors `product`'s identity-element behaviour on empty input.
- `reduce` - the generic reduction when `product` is not specific enough.

head

Return the first `$size` elements of a list.

Synopsis

```
my @first = head $size, @list;
my @first = head 2, qw( foo bar baz );   # ('foo', 'bar')
```

What you get back

A list of at most `$size` elements, taken from the start of the input. If `$size` is negative, head returns all but the last `-$size` elements - useful for dropping a tail.

Examples

```
my @r = head  2, qw( foo bar baz );   # ('foo', 'bar')
my @r = head -2, qw( foo bar baz );   # ('foo')
my @r = head  5, qw( a b c );         # ('a', 'b', 'c') - fewer than
 ↪$size
```

Edge cases

- `$size >= @list` returns the whole list.
- `$size == 0` or `-$size >= @list` returns the empty list.

Differences from upstream

Fully compatible with upstream.

See also

- `tail` - mirror operation, taking from the end.
- `sample` - random selection without positional bias.

tail

Return the last `$size` elements of a list.

Synopsis

```
my @last = tail $size, @list;
my @last = tail 2, qw( foo bar baz );    # ('bar', 'baz')
```

What you get back

A list of at most `$size` elements, taken from the end of the input. If `$size` is negative, `tail` returns all but the first `-$size` elements - useful for dropping a head.

Examples

```
my @r = tail  2, qw( foo bar baz );   # ('bar', 'baz')
my @r = tail -2, qw( foo bar baz );   # ('baz')
my @r = tail  5, qw( a b c );         # ('a', 'b', 'c') - fewer than
 ↪$size
```

Differences from upstream

Fully compatible with upstream.

See also

- `head` - mirror operation, taking from the start.
- `sample` - random selection without positional bias.

shuffle

Return the input list in a uniformly random order.

Synopsis

```
my @mixed = shuffle @values;
my @deck  = shuffle 0..51;       # a shuffled deck
```

What you get back

A new list with the same elements as the input, permuted by a Fisher-Yates shuffle. Each permutation is equally likely given a well-distributed random source. The source is the global `$List::Util::RAND`, or perl's built-in `rand()` when `$RAND` is undefined.

Examples

```
my @picked = shuffle @candidates;
my ($winner) = shuffle @entries;       # pick one at random
local $List::Util::RAND = sub { 0.5 }; # deterministic for tests
```

Differences from upstream

Fully compatible with upstream.

See also

- `sample` - take a fixed count without replacement.
- `head` - deterministic prefix selection.

sample

Pick `$count` distinct elements from the list at random.

Synopsis

```
my @picks = sample $count, @values;
my @three = sample 3, 1..100;
```

What you get back

A list of at most `$count` elements drawn without replacement. If `$count` is larger than the input, every element is returned in a random order - i.e. the function degrades to `shuffle`. The random source is `$List::Util::RAND` if set, otherwise perl's built-in `rand()`.

Examples

```
my @preview = sample 5, @articles;
my ($winner) = sample 1, @entries;
my @all     = sample 999, qw( a b c );   # ('c','a','b') say
```

Edge cases

- `$count <= 0` returns the empty list.
- Fewer inputs than `$count` returns all inputs, shuffled.

Differences from upstream

Fully compatible with upstream.

See also

- `shuffle` - permute the whole list.
- `head` - deterministic prefix (no randomness).

uniq

Remove subsequent duplicates by string equality, preserving order.

Elements are compared as strings: two values are the same when their stringifications are byte-for-byte identical. The first occurrence of any given value wins; later duplicates are dropped. `undef` is the one exception: it is distinct from the empty string, raises no warning, and is kept as `undef` in the result. That is the only difference from `uniqstr`.

Synopsis

```
my @subset = uniq @values;
my $count  = uniq @values;    # scalar context - count only
```

What you get back

In list context, the unique elements in first-seen order. In scalar context, the count of unique elements. `uniq undef, ''` returns both elements; later `undef`s are duplicates of the first.

Examples

```
my @u = uniq    qw( a b a c b );       # ('a', 'b', 'c')
my @u = uniqstr 'a', 'A';              # ('a', 'A') - case matters
my $n = uniq    qw( a b a c b );       # 3
```

Differences from upstream

Fully compatible with upstream.

See also

- `uniqnum` - compare as numbers rather than as strings.
- `uniqint` - compare as integers after truncation.

uniqstr

Remove subsequent duplicates by string equality, treating `undef` as the empty string.

Synopsis

```
my @subset = uniqstr @values;
my $count  = uniqstr @values;    # scalar context - count only
```

What you get back

In list context, the unique elements in first-seen order; in scalar context, their count. Unlike `uniq`, an `undef` element is indistinguishable from `""` and comes back as `""`.

See also

- `uniq` - identical, except `undef` is kept as `undef`.
- `uniqnum` / `uniqint` - numerical comparisons.

uniqnum

Remove subsequent duplicates by numerical equality, preserving order.

Synopsis

```
my @subset = uniqnum @values;
my $count  = uniqnum @values;    # scalar context - count only
```

What you get back

In list context, the unique numerical values in first-seen order. In scalar context, the count. `undef` is treated as `0` and coerced to `0` in the output (matching upstream's numerical handling). `+0.0` and `-0.0` are treated as equal. Every `NaN` is treated as a duplicate of every other `NaN`, despite `NaN != NaN`.

Examples

```
my @u = uniqnum  1, 1.0, '1', 2;   # (1, 2)
my @u = uniqnum  0, -0.0;          # (0)
my $n = uniqnum  1, 2, 3, 2, 1;    # 3
```

Differences from upstream

Fully compatible with upstream.

See also

- `uniq` - string-equality deduplication.
- `uniqint` - integer-equality deduplication (ignores fractions).

uniqint

Remove subsequent duplicates by integer equality, preserving order.

Synopsis

```
my @subset = uniqint @values;
my $count  = uniqint @values;    # scalar context - count only
```

What you get back

In list context, the unique integer values in first-seen order - every input is truncated to its integer part before comparison, and the returned list itself contains integers (not the original SVs). `undef` is treated as `0`.

Examples

```
my @u = uniqint 1.1, 1.9, 2.0;      # (1, 2)
my @u = uniqint '3', 3, 3.7;        # (3)
my $n = uniqint 1, 2, 2, 3;         # 3
```

Differences from upstream

Fully compatible with upstream.

See also

- `uniqnum` - full numerical (float) equality.
- `uniq` - string-equality deduplication.

minstr

Return the lexically smallest value in the list.

Synopsis

```
my $s = minstr @values;
my $s = minstr 'A'..'Z';          # 'A'
```

What you get back

The first (original) list element whose stringification compares less than every other element's stringification under `lt` (byte-wise comparison). Empty list yields `undef`.

Examples

```
my $s = minstr 'hello', 'world';    # 'hello'
my $s = minstr qw( foo bar baz );   # 'bar'
my $s = minstr ();                  # undef
```

Differences from upstream

Fully compatible with upstream.

See also

- `maxstr` - string-wise maximum.
- `min` - numerical minimum.

maxstr

Return the lexically largest value in the list.

Synopsis

```
my $s = maxstr @values;
my $s = maxstr 'A'..'Z';          # 'Z'
```

What you get back

The first (original) list element whose stringification compares greater than every other element's stringification under `gt` (byte-wise comparison). Empty list yields `undef`.

Examples

```
my $s = maxstr 'hello', 'world';    # 'world'
my $s = maxstr qw( foo bar baz );   # 'foo'
my $s = maxstr ();                  # undef
```

Differences from upstream

Fully compatible with upstream.

See also

- `minstr` - string-wise minimum.
- `max` - numerical maximum.

pairs

Group an even-sized list into key/value pair objects.

Synopsis

```
my @pairs = pairs @kvlist;
for my $p ( pairs %hash ) {
    my ($k, $v) = @$p;            # or $p->key / $p->value
}
```

What you get back

A list of two-element blessed array references (class `List::Util::_Pair`). Each pair responds to `->key`, `->value`, and `->TO_JSON`, and can also be dereferenced as a plain arrayref. For an odd-sized input, the final pair's value is `undef`.

Examples

```
for my $p ( pairs one => 1, two => 2 ) {
    print $p->key, '=', $p->value, "\n";
}
# one=1
# two=2
```

```
my @sorted = sort { $a->key cmp $b->key } pairs %h;
```

Differences from upstream

Fully compatible with upstream.

See also

- `unpairs` - inverse: flatten an arrayref list back to key/value.
- `pairkeys` / `pairvalues` - extract just one side.
- `pairmap` / `pairgrep` - transform or filter pairs in place.

unpairs

Flatten a list of two-element arrayrefs into a key/value list.

Synopsis

```
my @kvlist = unpairs @pairs;
my %hash   = unpairs @pairs;
```

What you get back

A flat list with exactly two elements per input arrayref: copies of `[0]` as the key and `[1]` as the value. Missing elements pad with `undef` so that the result is always even-sized; an argument that is not an ARRAY reference croaks.

Examples

```
my @kv = unpairs [a => 1], [b => 2];   # ('a', 1, 'b', 2)
my %h  = unpairs pairs %other;         # round-trip
my @kv = unpairs sort { $a->[0] cmp $b->[0] } pairs %h;
```

Differences from upstream

A plain integer argument croaks `Not a reference` where perl5 crashes, and an array reference held in a variable that once held a string is accepted where perl5 refuses it.

See also

- `pairs` - the inverse direction.

pairkeys

Extract just the keys from an even-sized key/value list.

Synopsis

```
my @keys = pairkeys @kvlist;
my @keys = pairkeys %hash;
```

What you get back

A list of every even-indexed element (indices `0`, 2, 4, ...). Equivalent to but more efficient than

```
pairmap { $a } @kvlist
```

Examples

```
my @k = pairkeys one => 1, two => 2;    # ('one', 'two')
my @k = pairkeys %ENV;                  # names of every env var
```

Differences from upstream

Fully compatible with upstream.

See also

- `pairvalues` - the value-side counterpart.
- `pairs` - keep keys and values bound together.

pairvalues

Extract just the values from an even-sized key/value list.

Synopsis

```
my @values = pairvalues @kvlist;
my @values = pairvalues %hash;
```

What you get back

A list of every odd-indexed element (indices 1, 3, 5, ...). If the input has an orphan key at the end, its corresponding value is `undef`. Equivalent to but more efficient than

```
pairmap { $b } @kvlist
```

Examples

```
my @v = pairvalues one => 1, two => 2;   # (1, 2)
my @v = pairvalues %ENV;                 # values of every env var
```

Differences from upstream

Fully compatible with upstream.

See also

- `pairkeys` - the key-side counterpart.
- `pairs` - keep keys and values bound together.

pairmap

Run a block for each pair and concatenate the results.

Synopsis

```
my @out   = pairmap { BLOCK } @kvlist;
my $count = pairmap { BLOCK } @kvlist;   # scalar context
```

For every pair the block is called in list context with `$a` set to the key and `$b` set to the value; its results are appended to the output list in order.

What you get back

In list context, the concatenation of everything every invocation of the block returned. In scalar context, the number of items that would have been returned in list context. `$a` and `$b` are aliased to the original list elements - modifications in the block are visible to the caller.

Examples

```
my @lines = pairmap { "The key $a has value $b" } %hash;
my @flat  = pairmap { ( $a, uc $b ) } %h;    # values upper-cased
my @pairs = pairmap { [ $a, $b ] } %h;       # equivalent to `pairs`
```

Differences from upstream

Fully compatible with upstream.

See also

- `pairgrep` - filter pairs by a predicate.
- `pairfirst` - stop at the first matching pair.
- `pairs` - bundle pairs into blessed arrayrefs.

pairgrep

Keep the pairs for which a block returns true.

Synopsis

```
my @kept  = pairgrep { BLOCK } @kvlist;
my $count = pairgrep { BLOCK } @kvlist;   # scalar context
```

For every pair the block is called in scalar context with `$a` set to the key and `$b` set to the value. Pairs for which the block returns true are included in the result; others are dropped.

What you get back

In list context, an even-sized list: the kept keys and values interleaved. In scalar context, the number of pairs (not elements) that matched - half of what the list-context length would be. `$a` and `$b` are aliased to the original list elements.

Examples

```
my %big     = pairgrep { $b > 100 } %totals;
my @uppers  = pairgrep { $a =~ /^[A-Z]/ } %h;
my $n_big   = pairgrep { $b > 100 } %totals;   # scalar: pair count
```

Differences from upstream

Fully compatible with upstream.

See also

- `pairmap` - transform pairs instead of filtering.
- `pairfirst` - find the first matching pair and stop.

pairfirst

Return the first pair for which a block returns true.

Synopsis

```
my ($k, $v) = pairfirst { BLOCK } @kvlist;
my $found   = pairfirst { BLOCK } @kvlist;   # scalar context
```

For each pair the block is called in scalar context with `$a` set to the key and `$b` set to the value. `pairfirst` stops at the first pair for which the block returns true.

What you get back

In list context, the matching `($key, $value)` pair, or an empty list if nothing matched. In scalar context, a boolean indicating whether any pair matched. `$a` and `$b` are aliased to the original list elements.

Examples

```
my ($k, $v) = pairfirst { $a =~ /^upper_/ } %config;
if ( pairfirst { $b > 1_000 } %totals ) {
    warn "at least one big entry";
}
```

Differences from upstream

Fully compatible with upstream.

See also

- `first` - non-pair equivalent, uses `$_`.
- `pairgrep` - collect every matching pair.

mesh

Interleave multiple arrays, padding shorter ones with `undef`.

Registered under both `mesh` and `mesh_longest`. Both forms stop at the length of the longest input and fill missing elements with `undef`.

Synopsis

```
my @out = mesh         \@a, \@b, \@c;
my @out = mesh_longest \@a, \@b, \@c;
my %h   = mesh \@keys, \@values;
```

What you get back

A flat list containing elements at position `0` from each input array, then position 1, and so on, up to the longest input. Missing positions are filled with `undef`.

Examples

```
my @r = mesh [1..3], ['a'..'c'];          # (1,'a', 2,'b', 3,'c')
my @r = mesh [1..3], ['a'..'b'];          # (1,'a', 2,'b', 3,undef)
my %h = mesh \@keys, \@values;            # build a hash
```

Edge cases

- An input that is not an arrayref croaks `Expected an ARRAY reference to mesh`.
- Empty input list returns an empty list.
- The elements are copies: modifying them leaves the inputs alone.

Differences from upstream

A hole in an input array reads as `undef`, where perl5 crashes.

See also

- `mesh_shortest` - stop at the shortest input.
- `zip` - same lockstep walk, but yields one arrayref per row.

mesh_shortest

Interleave multiple arrays, stopping at the shortest.

Synopsis

```
my @out = mesh_shortest \@a, \@b, \@c;
```

What you get back

A flat list containing elements at position `0` from each input, then 1, and so on - up to the length of the shortest input. No `undef` padding.

Examples

```
my @r = mesh_shortest [1..3], ['a','b'];   # (1,'a', 2,'b')
my @r = mesh_shortest [1..3], [];          # ()
```

Differences from upstream

Fully compatible with upstream.

See also

- `mesh` / `mesh_longest` - pad to longest instead.
- `zip_shortest` - same lockstep walk, but yields one arrayref per row.

zip

Walk multiple arrays in lockstep, returning one arrayref per row.

Registered under both `zip` and `zip_longest`. Both forms stop at the length of the longest input and fill missing positions with `undef`.

Synopsis

```
my @rows = zip         \@a, \@b, \@c;
my @rows = zip_longest \@a, \@b, \@c;
foreach ( zip \@xs, \@ys ) {
    my ($x, $y) = @$_;
}
```

What you get back

A list of arrayrefs, one per row. Row `i` contains the `i`-th element of each input array in order. Shorter inputs contribute `undef` beyond their own length.

Examples

```
my @r = zip [1..3], ['a'..'c'];   # ([1,'a'], [2,'b'], [3,'c'])
my @r = zip [1..3], ['a'..'b'];   # ([1,'a'], [2,'b'], [3,undef])
```

Differences from upstream

Fully compatible with upstream.

See also

- `zip_shortest` - stop at the shortest input.
- `mesh` - flatten the rows instead of wrapping in arrayrefs.

zip_shortest

Walk multiple arrays in lockstep, stopping at the shortest.

Synopsis

```
my @rows = zip_shortest \@a, \@b, \@c;
```

What you get back

A list of arrayrefs, one per row. Row `i` contains the `i`-th element of each input array - produced only while every input still has an `i`-th element. No `undef` padding.

Examples

```
my @r = zip_shortest [1..3], ['a','b'];   # ([1,'a'], [2,'b'])
my @r = zip_shortest [1..3], [];          # ()
```

Differences from upstream

Fully compatible with upstream.

See also

- `zip` / `zip_longest` - pad to longest instead.
- `mesh_shortest` - flatten the rows instead of wrapping in arrayrefs.

reduce

Collapse a list to a single value by repeated pairwise combination.

Synopsis

```
my $result = reduce { BLOCK } @list;
my $sum    = reduce { $a + $b } 1..10;       # 55
my $max    = reduce { $a > $b ? $a : $b } @xs;
```

The block is called in scalar context with `$a` set to the running accumulator and `$b` set to the next element. On the first call, `$a` and `$b` are the first two elements; on each subsequent call, `$a` is whatever the previous call returned.

What you get back

The value of the last block invocation. If the list is empty, `undef` is returned. If the list contains exactly one element, that element is returned and the block is not called.

Examples

```
my $concat = reduce { "$a,$b" } 'a'..'c';    # 'a,b,c'
my $best   = reduce { $a->score > $b->score ? $a : $b } @players;
```

(continues on next page)

(continued from previous page)

```
my $total  = reduce { $a + length $b } 0, @strings;   # sum of
 ↪lengths
```

```
# Guarantee a numeric identity so an empty @values doesn't yield
 ↪undef:
my $sum = reduce { $a + $b } 0, @values;
```

Differences from upstream

Fully compatible with upstream.

See also

- `reductions` - like `reduce`, but also keeps the intermediate values.
- `sum` / `product` / `min` / `max` - specialised, faster forms.
- `first` / any / `all` - element-test reductions over `$_`.

reductions

Like `reduce`, but also return every intermediate accumulator value.

Synopsis

```
my @steps = reductions { BLOCK } @list;
my @running = reductions { $a + $b } 1..4;   # (1, 3, 6, 10)
```

The block is called in the same way as for `reduce`: `$a` is the accumulator, `$b` is the next element. `reductions` differs in that it captures the accumulator at every step rather than just the final value.

What you get back

A list whose first element is the first input value and whose subsequent elements are each block invocation's return value in order. The last element equals what `reduce` would have returned for the same block and list.

Examples

```
reduce     { "$a-$b" }  'a'..'d';    # 'a-b-c-d'
reductions { "$a-$b" }  'a'..'d';    # ('a', 'a-b', 'a-b-c', 'a-b-c-
 ↪d')
```

```
my @running_sum = reductions { $a + $b } @values;
```

Differences from upstream

Fully compatible with upstream.

See also

- `reduce` - when only the final result is needed.

first

Return the first list element for which a block returns true.

Synopsis

```
my $val = first { BLOCK } @list;
my $d   = first { defined } @list;
my $big = first { $_ > 100 } @values;
```

For each element `first` sets `$_` to that element and calls the block in scalar context; the first element for which the block returns true is returned. Remaining elements are not examined.

What you get back

The matching element (the original value, not the block's return). If no element matches - or the list is empty - returns `undef`.

Examples

```
my $first_defined = first { defined $_ } @list;
my $first_big     = first { $_ > $threshold } @measurements;
my $first_admin   = first { $_->is_admin } @users;
```

Differences from upstream

Fully compatible with upstream.

See also

- any - boolean "at least one matches" test.
- `grep` - collect every matching element.
- `pairfirst` - pair-oriented counterpart.

any

True if the block returns true for at least one element.

Synopsis

```
my $bool = any { BLOCK } @list;
if ( any { length > 10 } @strings ) { ... }
```

For each element any sets `$_` to that element and calls the block in scalar context. It returns true as soon as the block returns true for any element, without examining the rest. For an empty list it returns false.

What you get back

1 if any element satisfied the block, `0` otherwise.

Examples

```
warn "has empty"   if any { $_ eq '' } @values;
die  "has admin"   if any { $_->is_admin } @users;
my $ok = any { /^--help$/ } @ARGV;
```

Differences from upstream

Fully compatible with upstream.

See also

- `all` - require every element to match.
- none - require no element to match.
- `first` - return the matching element itself.

all

True if the block returns true for every element (including empty list).

Synopsis

```
my $bool = all { BLOCK } @list;
if ( all { defined } @values ) { ... }
```

For each element `all` sets `$_` to that element and calls the block in scalar context. It returns false as soon as the block returns false for any element. An empty list yields true (the conventional “vacuous truth”).

What you get back

1 if every element satisfied the block, `0` otherwise.

Examples

```
die "found empty string" unless all { length } @strings;
my $ready = all { $_->is_ready } @workers;
my $yes   = all { $_ > 0 } ();           # 1 - vacuous truth
```

Differences from upstream

Fully compatible with upstream.

See also

- any - require at least one match.
- `notall` - inverse: at least one failure.
- none - inverse: zero matches.

none

True if the block returns true for no element (including empty list).

Synopsis

```
my $bool = none { BLOCK } @list;
if ( none { $_ < 0 } @measurements ) { ... }
```

For each element none sets `$_` to that element and calls the block in scalar context. It returns false as soon as the block returns true for any element. An empty list yields true.

What you get back

1 if every element failed the block, `0` otherwise.

Examples

```
die "negatives present" unless none { $_ < 0 } @measurements;
my $clean = none { /ERROR/ } @log_lines;
my $none  = none { 1 } ();               # 1 - nothing to fail
```

Differences from upstream

Fully compatible with upstream.

See also

- any - inverse: at least one match.
- `all` - require every element to match.
- `notall` - at least one failure.

notall

True if the block returns false for at least one element.

Synopsis

```
my $bool = notall { BLOCK } @list;
warn "some missing" if notall { defined } @values;
```

For each element `notall` sets `$_` to that element and calls the block in scalar context. It returns true as soon as the block returns false for any element. An empty list yields false.

What you get back

1 if at least one element failed the block, `0` otherwise.

Examples

```
my $incomplete = notall { defined } @row_cells;
warn "stale" if notall { $_->is_fresh } @entries;
my $false = notall { 1 } ();              # 0 - nothing to fail
```

Differences from upstream

Fully compatible with upstream.

See also

- `all` - inverse: require every element to match.
- any - at least one match.
- none - zero matches.

5.2.63 Locale

Namespace containing the following modules:

- `Locale::Maketext` - framework for localization

Locale::Maketext

framework for localization

SYNOPSIS

```
package MyProgram;
use strict;
use MyProgram::L10N;
 # ...which inherits from Locale::Maketext
my $lh = MyProgram::L10N->get_handle() || die "What language?";
...
# And then any messages your program emits, like:
warn $lh->maketext( "Can't open file [_1]: [_2]\n", $f, $! );
...
```

DESCRIPTION

It is a common feature of applications (whether run directly, or via the Web) for them to be "localized" – i.e., for them to a present an English interface to an English-speaker, a German interface to a German-speaker, and so on for all languages it's programmed with. Locale::Maketext is a framework for software localization; it provides you with the tools for organizing and accessing the bits of text and text-processing code that you need for producing localized applications.

In order to make sense of Maketext and how all its components fit together, you should probably go read Locale::Maketext::TPJ13, and *then* read the following documentation.

You may also want to read over the source for `File::Findgrep` and its constituent modules – they are a complete (if small) example application that uses Maketext.

QUICK OVERVIEW

The basic design of Locale::Maketext is object-oriented, and Locale::Maketext is an abstract base class, from which you derive a "project class". The project class (with a name like "TkBocciBall::Localize", which you then use in your module) is in turn the base class for all the "language classes" for your project (with names "TkBocciBall::Localize::it", "TkBocciBall::Localize::en", "TkBocciBall::Localize::fr", etc.).

A language class is a class containing a lexicon of phrases as class data, and possibly also some methods that are of use in interpreting phrases in the lexicon, or otherwise dealing with text in that language.

An object belonging to a language class is called a "language handle"; it's typically a flyweight object.

The normal course of action is to call:

```
use TkBocciBall::Localize;  # the localization project class
$lh = TkBocciBall::Localize->get_handle();
 # Depending on the user's locale, etc., this will
 # make a language handle from among the classes available,
 # and any defaults that you declare.
die "Couldn't make a language handle??" unless $lh;
```

From then on, you use the `maketext` function to access entries in whatever lexicon(s) belong to the language handle you got. So, this:

```
print $lh->maketext("You won!"), "\n";
```

...emits the right text for this language. If the object in `$lh` belongs to class "TkBocciBall::Localize::fr" and

=> "Tu as gagné!")`, then the above code happily tells the user "Tu as gagné!".

METHODS

Locale::Maketext offers a variety of methods, which fall into three categories:

- Methods to do with constructing language handles.
- `maketext` and other methods to do with accessing %Lexicon data for a given language handle.
- Methods that you may find it handy to use, from routines of yours that you put in %Lexicon entries.

These are covered in the following section.

Construction Methods

These are to do with constructing a language handle:

- $lh = YourProjClass->get_handle(...langtags...) || die "lg-handle?";

  This tries loading classes based on the language-tags you give (like `("en-US", "sk", "kon", "es-MX", "ja", "i-klingon")`, and for the first class that succeeds, returns YourProjClass::*language*->new().

  If it runs thru the entire given list of language-tags, and finds no classes for those exact terms, it then tries "superordinate" language classes. So if no "en-US" class (i.e., YourProjClass::en_us) was found, nor classes for anything else in that list, we then try its superordinate, "en" (i.e., YourProjClass::en), and so on thru the other language-tags in the given list: "es". (The other language-tags in our example list: happen to have no superordinates.)

  If none of those language-tags leads to loadable classes, we then try classes derived from YourProjClass->fallback_languages() and then if nothing comes of that, we use classes named by YourProjClass->fallback_language_classes(). Then in the (probably quite unlikely) event that that fails, we just return undef.

- $lh = YourProjClass->get_handle**()** || die "lg-handle?";

  When `get_handle` is called with an empty parameter list, magic happens:

  If `get_handle` senses that it's running in program that was invoked as a CGI, then it tries to get language-tags out of the environment variable "HTTP_ACCEPT_LANGUAGE", and it pretends that those were the languages passed as parameters to `get_handle`.

  Otherwise (i.e., if not a CGI), this tries various OS-specific ways to get the language-tags for the current locale/language, and then pretends that those were the value(s) passed to `get_handle`.

Currently this OS-specific stuff consists of looking in the environment variables "LANG" and "LANGUAGE"; and on MSWin machines (where those variables are typically unused), this also tries using the module Win32::Locale to get a language-tag for whatever language/locale is currently selected in the "Regional Settings" (or "International"?) Control Panel. I welcome further suggestions for making this do the Right Thing under other operating systems that support localization.

If you're using localization in an application that keeps a configuration file, you might consider something like this in your project class:

```
sub get_handle_via_config {
  my $class = $_[0];
  my $chosen_language = $Config_settings{'language'};
  my $lh;
  if($chosen_language) {
    $lh = $class->get_handle($chosen_language)
     || die "No language handle for \"$chosen_language\""
          . " or the like";
  } else {
    # Config file missing, maybe?
    $lh = $class->get_handle()
     || die "Can't get a language handle";
  }
  return $lh;
}
```

- $lh = YourProjClass::langname->new();

  This constructs a language handle. You usually **don't** call this directly, but instead let `get_handle` find a language class to `use` and to then call ->new on.

- $lh->init();

  This is called by ->new to initialize newly-constructed language handles. If you define an init method in your class, remember that it's usually considered a good idea to call $lh->SUPER::init in it (presumably at the beginning), so that all classes get a chance to initialize a new object however they see fit.

- YourProjClass->fallback_languages()

  `get_handle` appends the return value of this to the end of whatever list of languages you pass `get_handle`. Unless you override this method, your project class will inherit Locale::Maketext's `fallback_languages`, which currently returns `('i-default', 'en', 'en-US')`. ("i-default" is defined in RFC 2277).

  This method (by having it return the name of a language-tag that has an existing language class) can be used for making sure that `get_handle` will always manage to construct a language handle (assuming your language classes are in an appropriate @INC directory). Or you can use the next method:

- YourProjClass->fallback_language_classes()

  `get_handle` appends the return value of this to the end of the list of classes it will try using. Unless you override this method, your project class will inherit Locale::Maketext's `fallback_language_classes`, which currently returns

an empty list, `()`. By setting this to some value (namely, the name of a loadable language class), you can be sure that `get_handle` will always manage to construct a language handle.

The "maketext" Method

This is the most important method in Locale::Maketext:

```
$text = $lh->maketext(I<key>, ...parameters for this phrase...);
```

This looks in the %Lexicon of the language handle $lh and all its superclasses, looking for an entry whose key is the string *key*. Assuming such an entry is found, various things then happen, depending on the value found:

If the value is a scalarref, the scalar is dereferenced and returned (and any parameters are ignored).

If the value is a coderef, we return &$value($lh, ...parameters...).

If the value is a string that *doesn't* look like it's in Bracket Notation, we return it (after replacing it with a scalarref, in its %Lexicon).

If the value *does* look like it's in Bracket Notation, then we compile it into a sub, replace the string in the %Lexicon with the new coderef, and then we return &$new_sub($lh, ...parameters...).

Bracket Notation is discussed in a later section. Note that trying to compile a string into Bracket Notation can throw an exception if the string is not syntactically valid (say, by not balancing brackets right.)

Also, calling &$coderef($lh, ...parameters...) can throw any sort of exception (if, say, code in that sub tries to divide by zero). But a very common exception occurs when you have Bracket Notation text that says to call a method "foo", but there is no such method. (E.g., "You have [quat**n**,_1,ball]." will throw an exception on trying to call $lh->quat**n**($_[1],'ball') – you presumably meant "quant".) `maketext` catches these exceptions, but only to make the error message more readable, at which point it rethrows the exception.

An exception *may* be thrown if *key* is not found in any of $lh's %Lexicon hashes. What happens if a key is not found, is discussed in a later section, "Controlling Lookup Failure".

Note that you might find it useful in some cases to override the `maketext` method with an "after method", if you want to translate encodings, or even scripts:

```
package YrProj::zh_cn; # Chinese with PRC-style glyphs
use base ('YrProj::zh_tw');  # Taiwan-style
sub maketext {
  my $self = shift(@_);
  my $value = $self->maketext(@_);
  return Chineeze::taiwan2mainland($value);
}
```

Or you may want to override it with something that traps any exceptions, if that's critical to your program:

```
sub maketext {
  my($lh, @stuff) = @_;
  my $out;
  eval { $out = $lh->SUPER::maketext(@stuff) };
  return $out unless $@;
  ...otherwise deal with the exception...
}
```

Other than those two situations, I don't imagine that it's useful to override the `maketext` method. (If you run into a situation where it is useful, I'd be interested in hearing about it.)

- $lh->fail_with *or* $lh->fail_with(*PARAM*)
- $lh->failure_handler_auto

  These two methods are discussed in the section "Controlling Lookup Failure".

- $lh->denylist(@list) <or> $lh->blacklist(@list)
- $lh->allowlist(@list) <or> $lh->whitelist(@list)

  These methods are discussed in the section "Bracket Notation Security".

Utility Methods

These are methods that you may find it handy to use, generally from %Lexicon routines of yours (whether expressed as Bracket Notation or not).

- $language->quant($number, $singular)
- $language->quant($number, $singular, $plural)
- $language->quant($number, $singular, $plural, $negative)

  This is generally meant to be called from inside Bracket Notation (which is discussed later), as in

  ```
  "Your search matched [quant,_1,document]!"
  ```

  It's for *quantifying* a noun (i.e., saying how much of it there is, while giving the correct form of it). The behavior of this method is handy for English and a few other Western European languages, and you should override it for languages where it's not suitable. You can feel free to read the source, but the current implementation is basically as this pseudocode describes:

  ```
  if $number is 0 and there's a $negative,
     return $negative;
  elsif $number is 1,
     return "1 $singular";
  elsif there's a $plural,
     return "$number $plural";
  else
     return "$number " . $singular . "s";
  #
  ```

(continues on next page)

(continued from previous page)

```
# ...except that we actually call numf to
#  stringify $number before returning it.
```

So for English (with Bracket Notation) "...[quant,_1,file]..." is fine (for 0 it returns "0 files", for 1 it returns "1 file", and for more it returns "2 files", etc.)

But for "directory", you'd want "[quant,_1,directory,directories]" so that our elementary quant method doesn't think that the plural of "directory" is "directorys". And you might find that the output may sound better if you specify a negative form, as in:

```
"[quant,_1,file,files,No files] matched your query.\n"
```

Remember to keep in mind verb agreement (or adjectives too, in other languages), as in:

```
"[quant,_1,document] were matched.\n"
```

Because if _1 is one, you get "1 document **were** matched". An acceptable hack here is to do something like this:

```
"[quant,_1,document was, documents were] matched.\n"
```

- $language->numf($number)

  This returns the given number formatted nicely according to this language's conventions. Maketext's default method is mostly to just take the normal string form of the number (applying sprintf "%G" for only very large numbers), and then to add commas as necessary. (Except that we apply `tr/,././` if $language->{'numf_comma'} is true; that's a bit of a hack that's useful for languages that express two million as "2.000.000" and not as "2,000,000").

  If you want anything fancier, consider overriding this with something that uses Number::Format, or does something else entirely.

  Note that numf is called by quant for stringifying all quantifying numbers.

- $language->numerate($number, $singular, $plural, $negative)

  This returns the given noun form which is appropriate for the quantity `$number` according to this language's conventions. `numerate` is used internally by `quant` to quantify nouns. Use it directly – usually from bracket notation – to avoid `quant`'s implicit call to `numf` and output of a numeric quantity.

- $language->sprintf($format, @items)

  This is just a wrapper around Perl's normal `sprintf` function. It's provided so that you can use "sprintf" in Bracket Notation:

```
"Couldn't access datanode [sprintf,%10x=~[%s~],_1,_2]!\n"
```

  returning...

```
Couldn't access datanode      Stuff=[thangamabob]!
```

- $language->language_tag()

  Currently this just takes the last bit of `ref($language)`, turns underscores to dashes, and returns it. So if $language is an object of class Hee::HOO::Haw::en_us, $language->language_tag() returns "en-us". (Yes, the usual representation for that language tag is "en-US", but case is *never* considered meaningful in language-tag comparison.)

  You may override this as you like; Maketext doesn't use it for anything.

- $language->encoding()

  Currently this isn't used for anything, but it's provided (with default value of `(ref($language) && $language->{'encoding'}) or "iso-8859-1"`) as a sort of suggestion that it may be useful/necessary to associate encodings with your language handles (whether on a per-class or even per-handle basis.)

Language Handle Attributes and Internals

A language handle is a flyweight object – i.e., it doesn't (necessarily) carry any data of interest, other than just being a member of whatever class it belongs to.

A language handle is implemented as a blessed hash. Subclasses of yours can store whatever data you want in the hash. Currently the only hash entry used by any crucial Maketext method is "fail", so feel free to use anything else as you like.

Remember: Don't be afraid to read the Maketext source if there's any point on which this documentation is unclear. This documentation is vastly longer than the module source itself.

LANGUAGE CLASS HIERARCHIES

These are Locale::Maketext's assumptions about the class hierarchy formed by all your language classes:

- You must have a project base class, which you load, and which you then use as the first argument in the call to YourProjClass->get_handle(...). It should derive (whether directly or indirectly) from Locale::Maketext. It **doesn't matter** how you name this class, although assuming this is the localization component of your Super Mega Program, good names for your project class might be SuperMegaProgram::Localization, SuperMegaProgram::L10N, SuperMegaProgram::I18N, SuperMegaProgram::International, or even SuperMegaProgram::Languages or SuperMegaProgram::Messages.

- Language classes are what YourProjClass->get_handle will try to load. It will look for them by taking each language-tag (**skipping** it if it doesn't look like a language-tag or locale-tag!), turning it to all lowercase, turning dashes to underscores, and appending it to YourProjClass . "::". So this:

```
$lh = YourProjClass->get_handle(
  'en-US', 'fr', 'kon', 'i-klingon', 'i-klingon-romanized'
);
```

will try loading the classes YourProjClass::en_us (note lowercase!), YourProjClass::fr, YourProjClass::kon, YourProjClass::i_klingon and YourProjClass::i_klingon_romanized. (And it'll stop at the first one that actually loads.)

- I assume that each language class derives (directly or indirectly) from your project class, and also defines its @ISA, its %Lexicon, or both. But I anticipate no dire consequences if these assumptions do not hold.
- Language classes may derive from other language classes (although they should have "use *Thatclassname*" or "use base qw(...*classes*...)"). They may derive from the project class. They may derive from some other class altogether. Or via multiple inheritance, it may derive from any mixture of these.
- I foresee no problems with having multiple inheritance in your hierarchy of language classes. (As usual, however, Perl will complain bitterly if you have a cycle in the hierarchy: i.e., if any class is its own ancestor.)

ENTRIES IN EACH LEXICON

A typical %Lexicon entry is meant to signify a phrase, taking some number (0 or more) of parameters. An entry is meant to be accessed by via a string *key* in $lh->maketext(*key*, ...parameters...), which should return a string that is generally meant for be used for "output" to the user – regardless of whether this actually means printing to STDOUT, writing to a file, or putting into a GUI widget.

While the key must be a string value (since that's a basic restriction that Perl places on hash keys), the value in the lexicon can currently be of several types: a defined scalar, scalarref, or coderef. The use of these is explained above, in the section 'The "maketext" Method', and Bracket Notation for strings is discussed in the next section.

While you can use arbitrary unique IDs for lexicon keys (like "_min_larger_max_error"), it is often useful for if an entry's key is itself a valid value, like this example error message:

```
"Minimum ([_1]) is larger than maximum ([_2])!\n",
```

Compare this code that uses an arbitrary ID...

```
die $lh->maketext( "_min_larger_max_error", $min, $max )
 if $min > $max;
```

...to this code that uses a key-as-value:

```
die $lh->maketext(
 "Minimum ([_1]) is larger than maximum ([_2])!\n",
 $min, $max
) if $min > $max;
```

The second is, in short, more readable. In particular, it's obvious that the number of parameters you're feeding to that phrase (two) is the number of parameters that it *wants* to be fed. (Since you see _1 and a _2 being used in the key there.)

Also, once a project is otherwise complete and you start to localize it, you can scrape together all the various keys you use, and pass it to a translator; and then the translator's work will go faster if what he's presented is this:

```
"Minimum ([_1]) is larger than maximum ([_2])!\n",
 => "",   # fill in something here, Jacques!
```

rather than this more cryptic mess:

```
"_min_larger_max_error"
 => "",   # fill in something here, Jacques
```

I think that keys as lexicon values makes the completed lexicon entries more readable:

```
"Minimum ([_1]) is larger than maximum ([_2])!\n",
 => "Le minimum ([_1]) est plus grand que le maximum ([_2])!\n",
```

Also, having valid values as keys becomes very useful if you set up an _AUTO lexicon. _AUTO lexicons are discussed in a later section.

I almost always use keys that are themselves valid lexicon values. One notable exception is when the value is quite long. For example, to get the screenful of data that a command-line program might return when given an unknown switch, I often just use a brief, self-explanatory key such as "_USAGE_MESSAGE". At that point I then go and immediately to define that lexicon entry in the ProjectClass::L10N::en lexicon (since English is always my "project language"):

```
'_USAGE_MESSAGE' => <<'EOSTUFF',
...long long message...
EOSTUFF
```

and then I can use it as:

```
getopt('oDI', \%opts) or die $lh->maketext('_USAGE_MESSAGE');
```

Incidentally, note that each class's `%Lexicon` inherits-and-extends the lexicons in its superclasses. This is not because these are special hashes *per se*, but because you access them via the `maketext` method, which looks for entries across all the `%Lexicon` hashes in a language class *and* all its ancestor classes. (This is because the idea of "class data" isn't directly implemented in Perl, but is instead left to individual class-systems to implement as they see fit..)

Note that you may have things stored in a lexicon besides just phrases for output: for example, if your program takes input from the keyboard, asking a "(Y/N)" question, you probably need to know what the equivalent of "Y[es]/N[o]" is in whatever language. You probably also need to know what the equivalents of the answers "y" and "n" are. You can store that information in the lexicon (say, under the keys "~answer_y" and "~answer_n", and the long forms as "~answer_yes" and "~answer_no", where "~" is just an ad-hoc character meant to indicate to programmers/translators that these are not phrases for output).

Or instead of storing this in the language class's lexicon, you can (and, in some cases, really should) represent the same bit of knowledge as code in a method in the language class. (That leaves a tidy distinction between the lexicon as the things we know how to *say*, and the rest of the things in the lexicon class as things that we know how to *do*.) Consider this example of a processor for responses to French "oui/non" questions:

```
sub y_or_n {
  return undef unless defined $_[1] and length $_[1];
  my $answer = lc $_[1];  # smash case
  return 1 if $answer eq 'o' or $answer eq 'oui';
  return 0 if $answer eq 'n' or $answer eq 'non';
  return undef;
}
```

...which you'd then call in a construct like this:

```
my $response;
until(defined $response) {
  print $lh->maketext("Open the pod bay door (y/n)? ");
  $response = $lh->y_or_n( get_input_from_keyboard_somehow() );
}
if($response) { $pod_bay_door->open()         }
else          { $pod_bay_door->leave_closed() }
```

Other data worth storing in a lexicon might be things like filenames for language-targetted resources:

```
...
"_main_splash_png"
  => "/styles/en_us/main_splash.png",
"_main_splash_imagemap"
  => "/styles/en_us/main_splash.incl",
"_general_graphics_path"
  => "/styles/en_us/",
"_alert_sound"
  => "/styles/en_us/hey_there.wav",
"_forward_icon"
 => "left_arrow.png",
"_backward_icon"
 => "right_arrow.png",
# In some other languages, left equals
#  BACKwards, and right is FOREwards.
...
```

You might want to do the same thing for expressing key bindings or the like (since hard-wiring "q" as the binding for the function that quits a screen/menu/program is useful only if your language happens to associate "q" with "quit"!)

BRACKET NOTATION

Bracket Notation is a crucial feature of Locale::Maketext. I mean Bracket Notation to provide a replacement for the use of sprintf formatting. Everything you do with Bracket Notation could be done with a sub block, but bracket notation is meant to be much more concise.

Bracket Notation is a like a miniature "template" system (in the sense of Text::Template, not in the sense of C++ templates), where normal text is passed thru basically as is, but

text in special regions is specially interpreted. In Bracket Notation, you use square brackets ("[...]"), not curly braces ("{...}") to note sections that are specially interpreted.

For example, here all the areas that are taken literally are underlined with a "^", and all the in-bracket special regions are underlined with an X:

```
"Minimum ([_1]) is larger than maximum ([_2])!\n",
 ^^^^^^^^^ XX ^^^^^^^^^^^^^^^^^^^^^^^^^ XX ^^^^
```

When that string is compiled from bracket notation into a real Perl sub, it's basically turned into:

```
sub {
  my $lh = $_[0];
  my @params = @_;
  return join '',
    "Minimum (",
    ...some code here...
    ") is larger than maximum (",
    ...some code here...
    ")!\n",
}
# to be called by $lh->maketext(KEY, params...)
```

In other words, text outside bracket groups is turned into string literals. Text in brackets is rather more complex, and currently follows these rules:

- Bracket groups that are empty, or which consist only of whitespace, are ignored. (Examples: "[]", "[ ]", or a [and a] with returns and/or tabs and/or spaces between them.

  Otherwise, each group is taken to be a comma-separated group of items, and each item is interpreted as follows:

- An item that is "*_digits*" or "*_-digits*" is interpreted as $_[*value*]. I.e., "_1" becomes with $_[1], and "_-3" is interpreted as $_[-3] (in which case @_ should have at least three elements in it). Note that $_[0] is the language handle, and is typically not named directly.

- An item "_*" is interpreted to mean "all of @_ except $_[0]". I.e., `@_[1..$#_]`. Note that this is an empty list in the case of calls like $lh->maketext(*key*) where there are no parameters (except $_[0], the language handle).

- Otherwise, each item is interpreted as a string literal.

The group as a whole is interpreted as follows:

- If the first item in a bracket group looks like a method name, then that group is interpreted like this:

  ```
  $lh->that_method_name(
    ...rest of items in this group...
  ),
  ```

- If the first item in a bracket group is "*", it's taken as shorthand for the so commonly called "quant" method. Similarly, if the first item in a bracket group is "#", it's taken

to be shorthand for "numf".

- If the first item in a bracket group is the empty-string, or "_*" or "*_digits*" or "*_-digits*", then that group is interpreted as just the interpolation of all its items:

```
join('',
  ...rest of items in this group...
),
```

  Examples: "[_1]" and "[,_1]", which are synonymous; and "`[,ID-(,_4,-,_2,)]`", which compiles as `join "", "ID-(", $_[4], "-", $_[2], ")"`.

- Otherwise this bracket group is invalid. For example, in the group "[!@#,whatever]", the first item "`!@#`" is neither the empty-string, "*_number*", "*_-number*", "_*", nor a valid method name; and so Locale::Maketext will throw an exception of you try compiling an expression containing this bracket group.

Note, incidentally, that items in each group are comma-separated, not `/\s*,\s*/`-separated. That is, you might expect that this bracket group:

```
"Hoohah [foo, _1 , bar ,baz]!"
```

would compile to this:

```
sub {
  my $lh = $_[0];
  return join '',
    "Hoohah ",
    $lh->foo( $_[1], "bar", "baz"),
    "!",
}
```

But it actually compiles as this:

```
sub {
  my $lh = $_[0];
  return join '',
    "Hoohah ",
    $lh->foo(" _1 ", " bar ", "baz"),  # note the <space> in " bar "
    "!",
}
```

In the notation discussed so far, the characters "[" and "]" are given special meaning, for opening and closing bracket groups, and "," has a special meaning inside bracket groups, where it separates items in the group. This begs the question of how you'd express a literal "[" or "]" in a Bracket Notation string, and how you'd express a literal comma inside a bracket group. For this purpose I've adopted "~" (tilde) as an escape character: "~[" means a literal '[' character anywhere in Bracket Notation (i.e., regardless of whether you're in a bracket group or not), and ditto for "~]" meaning a literal ']', and "~," meaning a literal comma. (Altho "," means a literal comma outside of bracket groups – it's only inside bracket groups that commas are special.)

And on the off chance you need a literal tilde in a bracket expression, you get it with "~~".

Currently, an unescaped "~" before a character other than a bracket or a comma is taken to mean just a "~" and that character. I.e., "~X" means the same as "~~X" – i.e., one literal tilde, and then one literal "X". However, by using "~X", you are assuming that no future version of Maketext will use "~X" as a magic escape sequence. In practice this is not a great problem, since first off you can just write "~~X" and not worry about it; second off, I doubt I'll add lots of new magic characters to bracket notation; and third off, you aren't likely to want literal "~" characters in your messages anyway, since it's not a character with wide use in natural language text.

Brackets must be balanced – every openbracket must have one matching closebracket, and vice versa. So these are all **invalid**:

```
"I ate [quant,_1,rhubarb pie."
"I ate [quant,_1,rhubarb pie[."
"I ate quant,_1,rhubarb pie]."
"I ate quant,_1,rhubarb pie[."
```

Currently, bracket groups do not nest. That is, you **cannot** say:

```
"Foo [bar,baz,[quux,quuux]]\n";
```

If you need a notation that's that powerful, use normal Perl:

```
%Lexicon = (
  ...
  "some_key" => sub {
    my $lh = $_[0];
    join '',
      "Foo ",
      $lh->bar('baz', $lh->quux('quuux')),
      "\n",
  },
  ...
);
```

Or write the "bar" method so you don't need to pass it the output from calling quux.

I do not anticipate that you will need (or particularly want) to nest bracket groups, but you are welcome to email me with convincing (real-life) arguments to the contrary.

BRACKET NOTATION SECURITY

Locale::Maketext does not use any special syntax to differentiate bracket notation methods from normal class or object methods. This design makes it vulnerable to format string attacks whenever it is used to process strings provided by untrusted users.

Locale::Maketext does support denylist and allowlist functionality to limit which methods may be called as bracket notation methods.

By default, Locale::Maketext denies all methods in the Locale::Maketext namespace that begin with the '_' character, and all methods which include Perl's namespace separator characters.

The default denylist for Locale::Maketext also prevents use of the following methods in bracket notation:

```
denylist
encoding
fail_with
failure_handler_auto
fallback_language_classes
fallback_languages
get_handle
init
language_tag
maketext
new
allowlist
whitelist
blacklist
```

This list can be extended by either deny-listing additional “known bad” methods, or allow-listing only “known good” methods.

To prevent specific methods from being called in bracket notation, use the denylist() method:

```
my $lh = MyProgram::L10N->get_handle();
$lh->denylist(qw{my_internal_method my_other_method});
$lh->maketext('[my_internal_method]'); # dies
```

To limit the allowed bracked notation methods to a specific list, use the allowlist() method:

```
my $lh = MyProgram::L10N->get_handle();
$lh->allowlist('numerate', 'numf');
$lh->maketext('[_1] [numerate, _1,shoe,shoes]', 12); # works
$lh->maketext('[my_internal_method]'); # dies
```

The denylist() and allowlist() methods extend their internal lists whenever they are called. To reset the denylist or allowlist, create a new maketext object.

```
my $lh = MyProgram::L10N->get_handle();
$lh->denylist('numerate');
$lh->denylist('numf');
$lh->maketext('[_1] [numerate,_1,shoe,shoes]', 12); # dies
```

For lexicons that use an internal cache, translations which have already been cached in their compiled form are not affected by subsequent changes to the allowlist or denylist settings. Lexicons that use an external cache will have their cache cleared whenever the allowlist or denylist settings change. The difference between the two types of caching is explained in the “Readonly Lexicons” section.

Methods disallowed by the denylist cannot be permitted by the allowlist.

NOTE: denylist() is the preferred method name to use instead of the historical and non-inclusive method blacklist(). blacklist() may be removed in a future release of this

package and so it's use should be removed from usage.

NOTE: allowlist() is the preferred method name to use instead of the historical and non-inclusive method whitelist(). whitelist() may be removed in a future release of this package and so it's use should be removed from usage.

AUTO LEXICONS

If maketext goes to look in an individual %Lexicon for an entry for *key* (where *key* does not start with an underscore), and sees none, **but does see** an entry of "_AUTO" => *some_true_value*, then we actually define $Lexicon{*key*} = *key* right then and there, and then use that value as if it had been there all along. This happens before we even look in any superclass %Lexicons!

(This is meant to be somewhat like the AUTOLOAD mechanism in Perl's function call system – or, looked at another way, like the *AutoLoader* module.)

I can picture all sorts of circumstances where you just do not want lookup to be able to fail (since failing normally means that maketext throws a `die`, although see the next section for greater control over that). But here's one circumstance where _AUTO lexicons are meant to be *especially* useful:

As you're writing an application, you decide as you go what messages you need to emit. Normally you'd go to write this:

```
if(-e $filename) {
  go_process_file($filename)
} else {
  print qq{Couldn't find file "$filename"!\n};
}
```

but since you anticipate localizing this, you write:

```
use ThisProject::I18N;
my $lh = ThisProject::I18N->get_handle();
 # For the moment, assume that things are set up so
 # that we load class ThisProject::I18N::en
 # and that that's the class that $lh belongs to.
...
if(-e $filename) {
  go_process_file($filename)
} else {
  print $lh->maketext(
    qq{Couldn't find file "[_1]"!\n}, $filename
  );
}
```

Now, right after you've just written the above lines, you'd normally have to go open the file ThisProject/I18N/en.pm, and immediately add an entry:

```
"Couldn't find file \"[_1]\"!\n"
=> "Couldn't find file \"[_1]\"!\n",
```

But I consider that somewhat of a distraction from the work of getting the main code working – to say nothing of the fact that I often have to play with the program a few times before I can decide exactly what wording I want in the messages (which in this case would require me to go changing three lines of code: the call to maketext with that key, and then the two lines in ThisProject/I18N/en.pm).

However, if you set "_AUTO => 1" in the %Lexicon in, ThisProject/I18N/en.pm (assuming that English (en) is the language that all your programmers will be using for this project's internal message keys), then you don't ever have to go adding lines like this

```
"Couldn't find file \"[_1]\"!\n"
=> "Couldn't find file \"[_1]\"!\n",
```

to ThisProject/I18N/en.pm, because if _AUTO is true there, then just looking for an entry with the key "Couldn't find file "[_1]"!\n" in that lexicon will cause it to be added, with that value!

Note that the reason that keys that start with "_" are immune to _AUTO isn't anything generally magical about the underscore character – I just wanted a way to have most lexicon keys be autoable, except for possibly a few, and I arbitrarily decided to use a leading underscore as a signal to distinguish those few.

READONLY LEXICONS

If your lexicon is a tied hash the simple act of caching the compiled value can be fatal.

For example a *GDBM_File* GDBM_READER tied hash will die with something like:

```
gdbm store returned -1, errno 2, key "..." at ...
```

All you need to do is turn on caching outside of the lexicon hash itself like so:

```
sub init {
    my ($lh) = @_;
    ...
    $lh->{'use_external_lex_cache'} = 1;
    ...
}
```

And then instead of storing the compiled value in the lexicon hash it will store it in $lh->{'_external_lex_cache'}

CONTROLLING LOOKUP FAILURE

If you call $lh->maketext(*key*, ...parameters...), and there's no entry *key* in $lh's class's %Lexicon, nor in the superclass %Lexicon hash, *and* if we can't auto-make *key* (because either it starts with a "_", or because none of its lexicons have `_AUTO => 1`,), then we have failed to find a normal way to maketext *key*. What then happens in these failure conditions, depends on the $lh object's "fail" attribute.

If the language handle has no "fail" attribute, maketext will simply throw an exception (i.e., it calls `die`, mentioning the *key* whose lookup failed, and naming the line number where the calling $lh->maketext(*key*,...) was.

If the language handle has a "fail" attribute whose value is a coderef, then $lh->maketext(*key*,...params...) gives up and calls:

```
return $that_subref->($lh, $key, @params);
```

Otherwise, the "fail" attribute's value should be a string denoting a method name, so that $lh->maketext(*key*,...params...) can give up with:

```
return $lh->$that_method_name($phrase, @params);
```

The "fail" attribute can be accessed with the `fail_with` method:

```
# Set to a coderef:
$lh->fail_with( \&failure_handler );

# Set to a method name:
$lh->fail_with( 'failure_method' );

# Set to nothing (i.e., so failure throws a plain exception)
$lh->fail_with( undef );

# Get the current value
$handler = $lh->fail_with();
```

Now, as to what you may want to do with these handlers: Maybe you'd want to log what key failed for what class, and then die. Maybe you don't like `die` and instead you want to send the error message to STDOUT (or wherever) and then merely `exit()`.

Or maybe you don't want to `die` at all! Maybe you could use a handler like this:

```
# Make all lookups fall back onto an English value,
#  but only after we log it for later fingerpointing.
my $lh_backup = ThisProject->get_handle('en');
open(LEX_FAIL_LOG, ">>wherever/lex.log") || die "GNAARGH $!";
sub lex_fail {
  my($failing_lh, $key, $params) = @_;
  print LEX_FAIL_LOG scalar(localtime), "\t",
     ref($failing_lh), "\t", $key, "\n";
  return $lh_backup->maketext($key,@params);
}
```

Some users have expressed that they think this whole mechanism of having a "fail" attribute at all, seems a rather pointless complication. But I want Locale::Maketext to be usable for software projects of *any* scale and type; and different software projects have different ideas of what the right thing is to do in failure conditions. I could simply say that failure always throws an exception, and that if you want to be careful, you'll just have to wrap every call to $lh->maketext in an eval { }. However, I want programmers to reserve the right (via the "fail" attribute) to treat lookup failure as something other than an exception of the same level of severity as a config file being unreadable, or some essential resource being inaccessible.

One possibly useful value for the "fail" attribute is the method name "failure_handler_auto". This is a method defined in the class Locale::Maketext itself. You set it with:

```
$lh->fail_with('failure_handler_auto');
```

Then when you call $lh->maketext(*key*, ...parameters...) and there's no *key* in any of those lexicons, maketext gives up with

```
return $lh->failure_handler_auto($key, @params);
```

But failure_handler_auto, instead of dying or anything, compiles $key, caching it in

```
$lh->{'failure_lex'}{$key} = $compiled
```

and then calls the compiled value, and returns that. (I.e., if $key looks like bracket notation, $compiled is a sub, and we return &{$compiled}(@params); but if $key is just a plain string, we just return that.)

The effect of using "failure_auto_handler" is like an AUTO lexicon, except that it 1) compiles $key even if it starts with "_", and 2) you have a record in the new hashref $lh->{'failure_lex'} of all the keys that have failed for this object. This should avoid your program dying – as long as your keys aren't actually invalid as bracket code, and as long as they don't try calling methods that don't exist.

"failure_auto_handler" may not be exactly what you want, but I hope it at least shows you that maketext failure can be mitigated in any number of very flexible ways. If you can formalize exactly what you want, you should be able to express that as a failure handler. You can even make it default for every object of a given class, by setting it in that class's init:

```
sub init {
  my $lh = $_[0];  # a newborn handle
  $lh->SUPER::init();
  $lh->fail_with('my_clever_failure_handler');
  return;
}
sub my_clever_failure_handler {
  ...you clever things here...
}
```

HOW TO USE MAKETEXT

Here is a brief checklist on how to use Maketext to localize applications:

- Decide what system you'll use for lexicon keys. If you insist, you can use opaque IDs (if you're nostalgic for `catgets`), but I have better suggestions in the section "Entries in Each Lexicon", above. Assuming you opt for meaningful keys that double as values (like "Minimum ([_1]) is larger than maximum ([_2])!\n"), you'll have to settle on what language those should be in. For the sake of argument, I'll call this English, specifically American English, "en-US".
- Create a class for your localization project. This is the name of the class that you'll use in the idiom:

  ```
  use Projname::L10N;
  my $lh = Projname::L10N->get_handle(...) || die "Language?";
  ```

Assuming you call your class Projname::L10N, create a class consisting minimally of:

```
package Projname::L10N;
use base qw(Locale::Maketext);
...any methods you might want all your languages to share...

# And, assuming you want the base class to be an _AUTO lexicon,
# as is discussed a few sections up:

1;
```

- Create a class for the language your internal keys are in. Name the class after the language-tag for that language, in lowercase, with dashes changed to underscores. Assuming your project's first language is US English, you should call this Projname::L10N::en_us. It should consist minimally of:

```
package Projname::L10N::en_us;
use base qw(Projname::L10N);
%Lexicon = (
  '_AUTO' => 1,
);
1;
```

(For the rest of this section, I'll assume that this "first language class" of Projname::L10N::en_us has _AUTO lexicon.)

- Go and write your program. Everywhere in your program where you would say:

```
print "Foobar $thing stuff\n";
```

instead do it thru maketext, using no variable interpolation in the key:

```
print $lh->maketext("Foobar [_1] stuff\n", $thing);
```

If you get tired of constantly saying `print $lh->maketext`, consider making a functional wrapper for it, like so:

```
use Projname::L10N;
our $lh;
$lh = Projname::L10N->get_handle(...) || die "Language?";
sub pmt (@) { print( $lh->maketext(@_)) }
 # "pmt" is short for "Print MakeText"
$Carp::Verbose = 1;
 # so if maketext fails, we see made the call to pmt
```

Besides whole phrases meant for output, anything language-dependent should be put into the class Projname::L10N::en_us, whether as methods, or as lexicon entries – this is discussed in the section "Entries in Each Lexicon", above.

- Once the program is otherwise done, and once its localization for the first language works right (via the data and methods in Projname::L10N::en_us), you can get together the data for translation. If your first language lexicon isn't an _AUTO lexicon, then you already have all the messages explicitly in the lexicon (or else you'd be

getting exceptions thrown when you call $lh->maketext to get messages that aren't in there). But if you were (advisedly) lazy and are using an _AUTO lexicon, then you've got to make a list of all the phrases that you've so far been letting _AUTO generate for you. There are very many ways to assemble such a list. The most straightforward is to simply grep the source for every occurrence of "maketext" (or calls to wrappers around it, like the above `pmt` function), and to log the following phrase.

- You may at this point want to consider whether your base class (Projname::L10N), from which all lexicons inherit from (Projname::L10N::en, Projname::L10N::es, etc.), should be an _AUTO lexicon. It may be true that in theory, all needed messages will be in each language class; but in the presumably unlikely or "impossible" case of lookup failure, you should consider whether your program should throw an exception, emit text in English (or whatever your project's first language is), or some more complex solution as described in the section "Controlling Lookup Failure", above.

- Submit all messages/phrases/etc. to translators.

  (You may, in fact, want to start with localizing to *one* other language at first, if you're not sure that you've properly abstracted the language-dependent parts of your code.)

  Translators may request clarification of the situation in which a particular phrase is found. For example, in English we are entirely happy saying "*n* files found", regardless of whether we mean "I looked for files, and found *n* of them" or the rather distinct situation of "I looked for something else (like lines in files), and along the way I saw *n* files." This may involve rethinking things that you thought quite clear: should "Edit" on a toolbar be a noun ("editing") or a verb ("to edit")? Is there already a conventionalized way to express that menu option, separate from the target language's normal word for "to edit"?

  In all cases where the very common phenomenon of quantification (saying "*N* files", for **any** value of N) is involved, each translator should make clear what dependencies the number causes in the sentence. In many cases, dependency is limited to words adjacent to the number, in places where you might expect them ("I found the-?PLURAL *N* empty-?PLURAL directory-?PLURAL"), but in some cases there are unexpected dependencies ("I found-?PLURAL ..."!) as well as long-distance dependencies "The *N* directory-?PLURAL could not be deleted-?PLURAL"!).

  Remind the translators to consider the case where N is 0: "0 files found" isn't exactly natural-sounding in any language, but it may be unacceptable in many – or it may condition special kinds of agreement (similar to English "I didN'T find ANY files").

  Remember to ask your translators about numeral formatting in their language, so that you can override the `numf` method as appropriate. Typical variables in number formatting are: what to use as a decimal point (comma? period?); what to use as a thousands separator (space? nonbreaking space? comma? period? small middot? prime? apostrophe?); and even whether the so-called "thousands separator" is actually for every third digit – I've heard reports of two hundred thousand being expressible as "2,00,000" for some Indian (Subcontinental) languages, besides the less surprising "200 000", "200.000", "200,000", and "200'000". Also, using a set of numeral glyphs other than the usual ASCII "0"-"9" might be appreciated, as via `tr/0-9/\x{0966}-\x{096F}/` for getting digits in Devanagari script (for Hindi,

Konkani, others).

The basic `quant` method that Locale::Maketext provides should be good for many languages. For some languages, it might be useful to modify it (or its constituent `numerate` method) to take a plural form in the two-argument call to `quant` (as in "[quant,_1,files]") if it's all-around easier to infer the singular form from the plural, than to infer the plural form from the singular.

But for other languages (as is discussed at length in Locale::Maketext::TPJ13), simple `quant`/`numf` is not enough. For the particularly problematic Slavic languages, what you may need is a method which you provide with the number, the citation form of the noun to quantify, and the case and gender that the sentence's syntax projects onto that noun slot. The method would then be responsible for determining what grammatical number that numeral projects onto its noun phrase, and what case and gender it may override the normal case and gender with; and then it would look up the noun in a lexicon providing all needed inflected forms.

- You may also wish to discuss with the translators the question of how to relate different subforms of the same language tag, considering how this reacts with `get_handle`'s treatment of these. For example, if a user accepts interfaces in "en, fr", and you have interfaces available in "en-US" and "fr", what should they get? You may wish to resolve this by establishing that "en" and "en-US" are effectively synonymous, by having one class zero-derive from the other.

  For some languages this issue may never come up (Danish is rarely expressed as "da-DK", but instead is just "da"). And for other languages, the whole concept of a "generic" form may verge on being uselessly vague, particularly for interfaces involving voice media in forms of Arabic or Chinese.

- Once you've localized your program/site/etc. for all desired languages, be sure to show the result (whether live, or via screenshots) to the translators. Once they approve, make every effort to have it then checked by at least one other speaker of that language. This holds true even when (or especially when) the translation is done by one of your own programmers. Some kinds of systems may be harder to find testers for than others, depending on the amount of domain-specific jargon and concepts involved – it's easier to find people who can tell you whether they approve of your translation for "delete this message" in an email-via-Web interface, than to find people who can give you an informed opinion on your translation for "attribute value" in an XML query tool's interface.

SEE ALSO

I recommend reading all of these:

Locale::Maketext::TPJ13 – my *The Perl Journal* article about Maketext. It explains many important concepts underlying Locale::Maketext's design, and some insight into why Maketext is better than the plain old approach of having message catalogs that are just databases of sprintf formats.

File::Findgrep is a sample application/module that uses Locale::Maketext to localize its messages. For a larger internationalized system, see also Apache::MP3.

I18N::LangTags.

Win32::Locale.

RFC 3066, *Tags for the Identification of Languages*, as at http://sunsite.dk/RFC/rfc/rfc3066.html

RFC 2277, *IETF Policy on Character Sets and Languages* is at http://sunsite.dk/RFC/rfc/rfc2277.html – much of it is just things of interest to protocol designers, but it explains some basic concepts, like the distinction between locales and language-tags.

The manual for GNU `gettext`. The gettext dist is available in `ftp://prep.ai.mit.edu/pub/gnu/` – get a recent gettext tarball and look in its "doc/" directory, there's an easily browsable HTML version in there. The gettext documentation asks lots of questions worth thinking about, even if some of their answers are sometimes wonky, particularly where they start talking about pluralization.

The Locale/Maketext.pm source. Observe that the module is much shorter than its documentation!

COPYRIGHT AND DISCLAIMER

Copyright (c) 1999-2004 Sean M. Burke. All rights reserved.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Sean M. Burke `sburke@cpan.org`

Modules

- `Locale::Maketext::Guts` - Deprecated module to load Locale::Maketext utf8 code
- `Locale::Maketext::GutsLoader` - Deprecated module to load Locale::Maketext utf8 code
- `Locale::Maketext::Simple` - Simple interface to Locale::Maketext::Lexicon

Locale::Maketext::Guts

Deprecated module to load Locale::Maketext utf8 code

SYNOPSIS

```
# Do this instead please
use Locale::Maketext
```

DESCRIPTION

Previously Local::Maketext::GutsLoader performed some magic to load Locale::Maketext when utf8 was unavailable. The subs this module provided were merged back into Locale::Maketext

Locale::Maketext::GutsLoader

Deprecated module to load Locale::Maketext utf8 code

SYNOPSIS

```
# Do this instead please
use Locale::Maketext
```

DESCRIPTION

Previously Locale::Maketext::Guts performed some magic to load Locale::Maketext when utf8 was unavailable. The subs this module provided were merged back into Locale::Maketext.

Locale::Maketext::Simple

Simple interface to Locale::Maketext::Lexicon

VERSION

This document describes version 0.18 of Locale::Maketext::Simple, released Septermber 8, 2006.

SYNOPSIS

Minimal setup (looks for `auto/Foo/*.po` and `auto/Foo/*.mo`):

```
package Foo;
use Locale::Maketext::Simple;       # exports 'loc'
loc_lang('fr');                     # set language to French
sub hello {
    print loc("Hello, [_1]!", "World");
}
```

More sophisticated example:

```
package Foo::Bar;
use Locale::Maketext::Simple (
    Class       => 'Foo',       # search in auto/Foo/
    Style       => 'gettext',   # %1 instead of [_1]
    Export      => 'maketext',  # maketext() instead of loc()
    Subclass    => 'L10N',      # Foo::L10N instead of Foo::I18N
```

(continues on next page)

(continued from previous page)

```
    Decode      => 1,           # decode entries to unicode-strings
    Encoding    => 'locale',    # but encode lexicons in current␣
↪locale
                                # (needs Locale::Maketext::Lexicon␣
↪0.36)
);
sub japh {
    print maketext("Just another %1 hacker", "Perl");
}
```

DESCRIPTION

This module is a simple wrapper around **Locale::Maketext::Lexicon**, designed to alleviate the need of creating *Language Classes* for module authors.

The language used is chosen from the loc_lang call. If a lookup is not possible, the i-default language will be used. If the lookup is not in the i-default language, then the key will be returned.

If **Locale::Maketext::Lexicon** is not present, it implements a minimal localization function by simply interpolating `[_1]` with the first argument, `[_2]` with the second, etc. Interpolated function like `[quant,_1]` are treated as `[_1]`, with the sole exception of `[tense,_1,X]`, which will append `ing` to `_1` when X is `present`, or appending `ed` to <_1> otherwise.

OPTIONS

All options are passed either via the `use` statement, or via an explicit `import`.

Class

By default, **Locale::Maketext::Simple** draws its source from the calling package's `auto/` directory; you can override this behaviour by explicitly specifying another package as `Class`.

Path

If your PO and MO files are under a path elsewhere than `auto/`, you may specify it using the `Path` option.

Style

By default, this module uses the `maketext` style of `[_1]` and `[quant,_1]` for interpolation. Alternatively, you can specify the `gettext` style, which uses `%1` and `%quant(%1)` for interpolation.

This option is case-insensitive.

Export

By default, this module exports a single function, `loc`, into its caller's namespace. You can set it to another name, or set it to an empty string to disable exporting.

Subclass

By default, this module creates an `::I18N` subclass under the caller's package (or the package specified by `Class`), and stores lexicon data in its subclasses. You can assign a name other than `I18N` via this option.

Decode

If set to a true value, source entries will be converted into utf8-strings (available in Perl 5.6.1 or later). This feature needs the **Encode** or **Encode::compat** module.

Encoding

Specifies an encoding to store lexicon entries, instead of utf8-strings. If set to `locale`, the encoding from the current locale setting is used. Implies a true value for `Decode`.

ACKNOWLEDGMENTS

Thanks to Jos I. Boumans for suggesting this module to be written.

Thanks to Chia-Liang Kao for suggesting `Path` and `loc_lang`.

SEE ALSO

Locale::Maketext, Locale::Maketext::Lexicon

AUTHORS

Audrey Tang cpan@audreyt.org

COPYRIGHT

Copyright 2003, 2004, 2005, 2006 by Audrey Tang cpan@audreyt.org.

This software is released under the MIT license cited below. Additionally, when this software is distributed with **Perl Kit, Version 5**, you may also redistribute it and/or modify it under the same terms as Perl itself.

The "MIT" License

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

5.2.64 MIME

Namespace containing the following modules:

- `MIME::Base64` - Encode and decode base64 strings - the standard MIME flavour plus the URL-safe variant and cheap length helpers.
- `MIME::Charset` - Charset Information for MIME
- `MIME::QuotedPrint` - Quoted-printable encoding and decoding as specified by RFC 2045.

MIME::Base64

Encode and decode base64 strings - the standard MIME flavour plus the URL-safe variant and cheap length helpers.

`MIME::Base64` turns arbitrary byte strings into the 65-character ASCII subset defined by RFC 2045 (`[A-Za-z0-9+/=]`) and back. Most callers reach for it for one of a handful of jobs:

- writing MIME-style bodies where encoded output must be wrapped at 76 characters per line (the default `encode_base64` form);
- producing a single unwrapped token for an HTTP header, a credential, or a config file - `encode_base64($bytes, "");`
- building URL-safe tokens (JWT payloads, signed cookies, opaque identifiers) where `+` and `/` would need percent-escaping - `encode_base64url`;
- sizing a buffer without paying the cost of the full encode or decode - `encoded_base64_length` and `decoded_base64_length`.

Line-length wrapping

`encode_base64` breaks the output into lines of **at most 76 characters**, each terminated by the caller’s end-of-line sequence (default `"\n"`). Pass `""` as the second argument to suppress wrapping entirely; pass `"\r\n"` for CRLF output suitable for raw MIME transport. The URL-safe encoder never wraps and never pads.

Decode leniency

decode_base64 is deliberately forgiving. Any character outside the 65-character alphabet is silently skipped, so embedded whitespace, line breaks, and MIME boilerplate pass through harmlessly. Decoding stops at the first = padding group - data after padding is never consumed. The URL-safe decoder accepts input with or without padding and translates -/_ back to +// before running the same state machine.

Who calls this

Most real-world consumers are Authen::* (HTTP Basic auth), Email::MIME and friends (message bodies), JWT/JOSE libraries (via the URL-safe variant), and anything that needs to round-trip binary through a text channel. encode_base64 and decode_base64 are exported by default; the URL-safe and length helpers are exported on request.

URL-safe variant

encode_base64url($bytes) encodes with the URL- and filename-safe alphabet (- and _ in place of + and /), strips the trailing = padding and never wraps, so the result drops into URLs, cookies and JWT segments unescaped. decode_base64url($token) maps - and _ back, restores the padding and decodes; input with or without padding, or with + and /, is accepted.

This is MIME::Base64 from MIME-Base64 3.16_01; MIME/Base64.pm is upstream's. encode_base64 and decode_base64 are exported by default.

Synopsis

```
use MIME::Base64 qw(encode_base64 decode_base64 encode_base64url␣
↪decode_base64url);

my $enc = encode_base64("Aladdin:open sesame");
print $enc;                                    #␣
↪QWxhZGRpbjpvcGVuIHNlc2FtZQ==\n
print decode_base64($enc), "\n";               # Aladdin:open␣
↪sesame
print encode_base64("\xff\xfe binary", ''), "\n";  # //4gYmluYXJ5 -␣
↪one line

my $token = encode_base64url("\xfb\xff");          # -_8: no + / or␣
↪= padding
print length decode_base64url($token), "\n";       # 2
```

Strings must be bytes: a character above 0xFF dies "Wide character"; encode text first (encode_base64(Encode::encode_utf8($text))).

Differences from upstream

One, where perl 5.44 with Base64.xs fails (t/81-xs-native/MIME-Base64/130-or-better.t): `encoded_base64_length` and `decoded_base64_length` given a number or undef measure its string value, as for any other value. perl5 reads a length that number never had - `encoded_base64_length(12345)` gave 485286422 - or crashes on undef.

Functions

Standard encoding

encode_base64

Encode a byte string as standard RFC 2045 base64, optionally wrapping at 76 characters.

decode_base64

Decode a base64 string back to the original byte sequence, tolerating stray whitespace and line breaks.

Length helpers

encoded_base64_length

Return the length `encode_base64` would produce, without doing the encoding.

decoded_base64_length

Return the length `decode_base64` would produce, without doing the decoding.

Other Functions

encode_base64url

Encode bytes using the URL- and filename-safe base64 alphabet, with no padding and no line breaks.

decode_base64url

Decode a URL-safe base64 string, accepting input with or without trailing padding.

encode_base64

Encode a byte string as standard RFC 2045 base64, optionally wrapping at 76 characters.

Synopsis

```
use MIME::Base64;
my $encoded = encode_base64($bytes);          # wrapped at 76, "\n"-
 ↪terminated
```

(continues on next page)

(continued from previous page)

```
my $oneline = encode_base64($bytes, "");     # no wrapping, no␣
↪trailing EOL
my $crlf    = encode_base64($bytes, "\r\n"); # MIME-style CRLF␣
↪wrapping
```

What you get back

A plain (non-UTF8) ASCII string containing only characters from `[A-Za-z0-9+/=]` plus the caller's end-of-line sequence. With the default `"\n"` terminator, every 76-character line - including the final short one - ends with `"\n"`. Pass `""` to get a single unbroken token with no trailing newline; useful for HTTP headers, credential strings, and anywhere a single atomic value is expected. The output always pads to a multiple of four characters with `=`.

Examples

```
encode_base64("Aladdin:open sesame");
# "QWxhZGRpbjpvcGVuIHNlc2FtZQ==\n"

encode_base64("Aladdin:open sesame", "");
# "QWxhZGRpbjpvcGVuIHNlc2FtZQ=="

encode_base64("\x00\x00\x00");
# "AAAA\n"
```

Edge cases

- Empty input returns the empty string.
- A single character (or any input whose length is not a multiple of 3) is padded with `=` so the output length is a multiple of four.
- Binary bytes are encoded as-is; there is no byte-order or encoding transformation.
- Input with code points above 255 is a programming error in the caller; base64 is defined over octets only. Encode to UTF-8 first if the source is a text string.

Differences from upstream

Fully compatible with upstream `MIME::Base64` 3.16_01.

See also

- `decode_base64` - the inverse operation.
- `encode_base64url` - the URL-safe variant, no padding, no wrapping.
- `encoded_base64_length` - size the output without encoding.

decode_base64

Decode a base64 string back to the original byte sequence, tolerating stray whitespace and line breaks.

Synopsis

```
use MIME::Base64;
my $bytes = decode_base64($encoded);
```

What you get back

A byte string carrying the decoded octets. The result is binary; treat it as bytes, not text. If the input was the output of `encode_base64`, the result is byte-for-byte what was encoded.

The decoder silently skips any character outside the 65-character base64 subset, so wrapped lines, CRLFs, leading/trailing whitespace, and MIME header remnants are handled without special preprocessing. Decoding stops at the first = padding group; data appearing after padding is ignored.

Examples

```
decode_base64("QWxhZGRpbjpvcGVuIHNlc2FtZQ==");
# "Aladdin:open sesame"

decode_base64("QWxh\nZGRp\nbjpv\ncGVu\nIHNl\nc2Ft\nZQ==\n");
# "Aladdin:open sesame"   (wrapped input decodes identically)

decode_base64("AAAA");
# "\x00\x00\x00"
```

Edge cases

- Empty input returns the empty string.
- Input of all-whitespace returns the empty string.
- A truncated final group (1 valid character) produces no output for that group; 2 valid characters produce 1 byte, 3 produce 2 bytes.
- Characters after the first = padding group are discarded.
- Non-alphabet characters anywhere in the input are silently skipped, including control characters and punctuation.

Differences from upstream

Fully compatible with upstream `MIME::Base64` 3.16_01.

See also

- encode_base64 - the inverse operation.
- decode_base64url - the URL-safe variant, accepts -/_ and missing padding.
- decoded_base64_length - size the output without decoding.

encoded_base64_length

Return the length encode_base64 would produce, without doing the encoding.

Synopsis

```
use MIME::Base64 qw(encoded_base64_length);
my $n = encoded_base64_length($bytes);           # wrapping with "\n"
my $m = encoded_base64_length($bytes, "");       # no wrapping
my $k = encoded_base64_length($bytes, "\r\n");  # CRLF wrapping
```

What you get back

An integer: the exact number of bytes encode_base64($bytes, $eol) would produce, including every end-of-line sequence. Equivalent to length(encode_base64($bytes, $eol)) but avoids allocating and building the encoded string - preferred when you need to pre-size a buffer or report progress on a large payload.

Examples

```
encoded_base64_length("");                  # 0
encoded_base64_length("A");                 # 5   ("QQ==\n")
encoded_base64_length("A", "");             # 4   ("QQ==")
encoded_base64_length("A" x 57);            # 77  (one full 76-char␣
↪line + "\n")
encoded_base64_length("A" x 57, "\r\n");    # 78  (one full 76-char␣
↪line + "\r\n")
```

Edge cases

- Empty input returns 0 regardless of the $eol argument.
- The formula is ceil(n/3) * 4 + lines * length($eol) where lines = ceil(base_len / 76).
- An undefined or missing $eol counts as "\n" (length 1) to match encode_base64's default.

Differences from upstream

Fully compatible with upstream `MIME::Base64` 3.16_01.

See also

- `encode_base64` - the function whose output length is being computed.
- `decoded_base64_length` - the companion predictor for the decoder.

decoded_base64_length

Return the length `decode_base64` would produce, without doing the decoding.

Synopsis

```
use MIME::Base64 qw(decoded_base64_length);
my $n = decoded_base64_length($encoded);
```

What you get back

An integer: the exact number of bytes `decode_base64($encoded)` would produce. Equivalent to `length(decode_base64($encoded))` but avoids building the decoded string. Useful when validating that a payload fits a size limit, or pre-sizing a destination buffer.

The same leniency rules as `decode_base64` apply: stray whitespace and non-alphabet characters are ignored, and the count stops at the first = padding group.

Examples

```
decoded_base64_length("");                                  # 0
decoded_base64_length("QQ==");                              # 1
decoded_base64_length("QWxhZGRpbjpvcGVuIHNlc2FtZQ==");  # 19
decoded_base64_length("QWxh\nZGRp\nbjpv\ncGVu\n");     # 12 (newlines
↪ignored)
```

Edge cases

- Empty input, or input with no alphabet characters, returns `0`.
- A lone trailing character that would decode to no output (a group of exactly 1 valid character) contributes `0` bytes.
- Characters after the first = are not counted.

Differences from upstream

Fully compatible with upstream `MIME::Base64` 3.16_01.

See also

- `decode_base64` - the function whose output length is being computed.
- `encoded_base64_length` - the companion predictor for the encoder.

encode_base64url

Encode bytes using the URL- and filename-safe base64 alphabet, with no padding and no line breaks.

Synopsis

```
use MIME::Base64 qw(encode_base64url);
my $token = encode_base64url($bytes);
```

What you get back

A single unbroken ASCII string using the alphabet `[A-Za-z0-9-_]`. Trailing = padding is stripped. The result is safe to drop into URLs, path segments, cookie values, and JWT components without percent-escaping. The output is always one line - never wrapped.

Examples

```
encode_base64url("Aladdin:open sesame");
# "QWxhZGRpbjpvcGVuIHNlc2FtZQ"

encode_base64url("\xFB\xFF");
# "-_8"                 (note the '-' and '_' where '+' and '/' would
 ↪appear)

encode_base64url("");
# ""
```

Edge cases

- Empty input returns the empty string.
- Bytes that would encode to + under standard base64 become -; bytes that would encode to / become _.
- No = appears in the output under any input length.

Differences from upstream

Fully compatible with upstream `MIME::Base64` 3.16_01.

See also

- `decode_base64url` - the inverse operation.
- `encode_base64` - standard variant with wrapping and padding.

decode_base64url

Decode a URL-safe base64 string, accepting input with or without trailing padding.

Synopsis

```
use MIME::Base64 qw(decode_base64url);
my $bytes = decode_base64url($token);
```

What you get back

The decoded byte string. `-` and `_` in the input are mapped back to `+` and `/` before decoding, and any missing trailing padding is reconstructed. The decoder inherits `decode_base64`'s tolerance: characters outside the alphabet are silently skipped, and decoding stops at the first padding group.

Examples

```
decode_base64url("QWxhZGRpbjpvcGVuIHNlc2FtZQ");
# "Aladdin:open sesame"

decode_base64url("-_8");
# "\xFB\xFF"

decode_base64url("QWxhZGRpbjpvcGVuIHNlc2FtZQ==");   # padding also␣
↪OK
# "Aladdin:open sesame"
```

Edge cases

- Empty input returns the empty string.
- Input with stray `+` or `/` is accepted (those are valid under standard base64); the URL-safe decoder is a superset of the standard decoder for input characters.
- Truncated input (length not a multiple of four, no padding) is padded internally and decoded; the last partial group follows the same "1 char gives 0 bytes, 2 gives 1, 3 gives 2" rule as `decode_base64`.

Differences from upstream

Fully compatible with upstream `MIME::Base64` 3.16_01.

See also

- `encode_base64url` - the inverse operation.
- `decode_base64` - standard variant.

MIME::Charset

Charset Information for MIME

SYNOPSIS

```
use MIME::Charset;

$charset = MIME::Charset->new("euc-jp");
```

Getting charset information:

```
$benc = $charset->body_encoding; # e.g. "Q"
$cset = $charset->as_string; # e.g. "US-ASCII"
$henc = $charset->header_encoding; # e.g. "S"
$cset = $charset->output_charset; # e.g. "ISO-2022-JP"
```

Translating text data:

```
($text, $charset, $encoding) =
    $charset->header_encode(
       "\xc9\xc2\xc5\xaa\xc0\xde\xc3\xef\xc5\xaa".
       "\xc7\xd1\xca\xaa\xbd\xd0\xce\xcf\xb4\xef",
       Charset => 'euc-jp');
# ...returns e.g. (<converted>, "ISO-2022-JP", "B").

($text, $charset, $encoding) =
    $charset->body_encode(
        "Collectioneur path\xe9tiquement ".
        "\xe9clectique de d\xe9chets",
        Charset => 'latin1');
# ...returns e.g. (<original>, "ISO-8859-1", "QUOTED-PRINTABLE").

$len = $charset->encoded_header_len(
    "Perl\xe8\xa8\x80\xe8\xaa\x9e",
    Charset => 'utf-8',
    Encoding => "b");
# ...returns e.g. 28.
```

Manipulating module defaults:

```
MIME::Charset::alias("csEUCKR", "euc-kr");
MIME::Charset::default("iso-8859-1");
MIME::Charset::fallback("us-ascii");
```

Non-OO functions (may be deprecated in near future):

```
use MIME::Charset qw(:info);

$benc = body_encoding("iso-8859-2"); # "Q"
$cset = canonical_charset("ANSI X3.4-1968"); # "US-ASCII"
$henc = header_encoding("utf-8"); # "S"
$cset = output_charset("shift_jis"); # "ISO-2022-JP"

use MIME::Charset qw(:trans);

($text, $charset, $encoding) =
    header_encode(
       "\xc9\xc2\xc5\xaa\xc0\xde\xc3\xef\xc5\xaa".
       "\xc7\xd1\xca\xaa\xbd\xd0\xce\xcf\xb4\xef",
       "euc-jp");
# ...returns (<converted>, "ISO-2022-JP", "B");

($text, $charset, $encoding) =
    body_encode(
        "Collectioneur path\xe9tiquement ".
        "\xe9clectique de d\xe9chets",
        "latin1");
# ...returns (<original>, "ISO-8859-1", "QUOTED-PRINTABLE");

$len = encoded_header_len(
    "Perl\xe8\xa8\x80\xe8\xaa\x9e", "b", "utf-8"); # 28
```

DESCRIPTION

MIME::Charset provides information about character sets used for MIME messages on Internet.

Definitions

The **charset** is ``character set'' used in MIME to refer to a method of converting a sequence of octets into a sequence of characters. It includes both concepts of ``coded character set'' (CCS) and ``character encoding scheme'' (CES) of ISO/IEC.

The **encoding** is that used in MIME to refer to a method of representing a body part or a header body as sequence(s) of printable US-ASCII characters.

Constructor

- $charset = MIME::Charset->new([CHARSET [, OPTS]])

  Create charset object.

  OPTS may accept following key-value pair. **NOTE**: When Unicode/multibyte support is disabled (see "USE_ENCODE"), conversion will not be performed. So this option do not have any effects.

  – Mapping => MAPTYPE

    Whether to extend mappings actually used for charset names or not. `"EXTENDED"` uses extended mappings. `"STANDARD"` uses standardized strict mappings. Default is `"EXTENDED"`.

Getting Information of Charsets

- $charset->body_encoding
- body_encoding CHARSET

  Get recommended transfer-encoding of CHARSET for message body.

  Returned value will be one of `"B"` (BASE64), `"Q"` (QUOTED-PRINTABLE), `"S"` (shorter one of either) or `undef` (might not be transfer-encoded; either 7BIT or 8BIT). This may not be same as encoding for message header.

- $charset->as_string
- canonical_charset CHARSET

  Get canonical name for charset.

- $charset->decoder

  Get "Encode::Encoding" object to decode strings to Unicode by charset. If charset is not specified or not known by this module, undef will be returned.

- $charset->dup

  Get a copy of charset object.

- $charset->encoder([CHARSET])

  Get "Encode::Encoding" object to encode Unicode string using compatible charset recommended to be used for messages on Internet.

  If optional CHARSET is specified, replace encoder (and output charset name) of $charset object with those of CHARSET, therefore, $charset object will be a converter between original charset and new CHARSET.

- $charset->header_encoding
- header_encoding CHARSET

  Get recommended encoding scheme of CHARSET for message header.

  Returned value will be one of `"B"`, `"Q"`, `"S"` (shorter one of either) or `undef` (might not be encoded). This may not be same as encoding for message body.

- $charset->output_charset

- output_charset CHARSET

  Get a charset which is compatible with given CHARSET and is recommended to be used for MIME messages on Internet (if it is known by this module).

  When Unicode/multibyte support is disabled (see "USE_ENCODE"), this function will simply return the result of "canonical_charset".

Translating Text Data

- $charset->body_encode(STRING [, OPTS])
- body_encode STRING, CHARSET [, OPTS]

  Get converted (if needed) data of STRING and recommended transfer-encoding of that data for message body. CHARSET is the charset by which STRING is encoded.

  OPTS may accept following key-value pairs. **NOTE**: When Unicode/multibyte support is disabled (see "USE_ENCODE"), conversion will not be performed. So these options do not have any effects.

  – Detect7bit => YESNO

    Try auto-detecting 7-bit charset when CHARSET is not given. Default is `"YES"`.

  – Replacement => REPLACEMENT

    Specifies error handling scheme. See "Error Handling".

  3-item list of (*converted string*, *charset for output*, *transfer-encoding*) will be returned. *Transfer-encoding* will be either `"BASE64"`, `"QUOTED-PRINTABLE"`, `"7BIT"` or `"8BIT"`. If *charset for output* could not be determined and *converted string* contains non-ASCII byte(s), *charset for output* will be `undef` and *transfer-encoding* will be `"BASE64"`. *Charset for output* will be `"US-ASCII"` if and only if string does not contain any non-ASCII bytes.

- $charset->decode(STRING [,CHECK])

  Decode STRING to Unicode.

  Note: When Unicode/multibyte support is disabled (see "USE_ENCODE"), this function will die.

- detect_7bit_charset STRING

  Guess 7-bit charset that may encode a string STRING. If STRING contains any 8-bit bytes, `undef` will be returned. Otherwise, Default Charset will be returned for unknown charset.

- $charset->encode(STRING [, CHECK])

  Encode STRING (Unicode or non-Unicode) using compatible charset recommended to be used for messages on Internet (if this module knows it). Note that string will be decoded to Unicode then encoded even if compatible charset was equal to original charset.

  Note: When Unicode/multibyte support is disabled (see "USE_ENCODE"), this function will die.

- $charset->encoded_header_len(STRING [, ENCODING])

- encoded_header_len STRING, ENCODING, CHARSET

  Get length of encoded STRING for message header (without folding).

  ENCODING may be one of `"B"`, `"Q"` or `"S"` (shorter one of either `"B"` or `"Q"`).

- $charset->header_encode(STRING [, OPTS])
- header_encode STRING, CHARSET [, OPTS]

  Get converted (if needed) data of STRING and recommended encoding scheme of that data for message headers. CHARSET is the charset by which STRING is encoded.

  OPTS may accept following key-value pairs. **NOTE**: When Unicode/multibyte support is disabled (see "USE_ENCODE"), conversion will not be performed. So these options do not have any effects.

  – Detect7bit => YESNO

    Try auto-detecting 7-bit charset when CHARSET is not given. Default is `"YES"`.

  – Replacement => REPLACEMENT

    Specifies error handling scheme. See "Error Handling".

  3-item list of (*converted string*, *charset for output*, *encoding scheme*) will be returned. *Encoding scheme* will be either `"B"`, `"Q"` or `undef` (might not be encoded). If *charset for output* could not be determined and *converted string* contains non-ASCII byte(s), *charset for output* will be `"8BIT"` (this is *not* charset name but a special value to represent unencodable data) and *encoding scheme* will be `undef` (should not be encoded). *Charset for output* will be `"US-ASCII"` if and only if string does not contain any non-ASCII bytes.

- $charset->undecode(STRING [,CHECK])

  Encode Unicode string STRING to byte string by input charset of $charset. This is equivalent to `$charset->decoder->encode()`.

  Note: When Unicode/multibyte support is disabled (see "USE_ENCODE"), this function will die.

Manipulating Module Defaults

- alias ALIAS [, CHARSET]

  Get/set charset alias for canonical names determined by "canonical_charset".

  If CHARSET is given and isn't false, ALIAS will be assigned as an alias of CHARSET. Otherwise, alias won't be changed. In both cases, current charset name that ALIAS is assigned will be returned.

- default [CHARSET]

  Get/set default charset.

  Default charset is used by this module when charset context is unknown. Modules using this module are recommended to use this charset when charset context is unknown or implicit default is expected. By default, it is `"US-ASCII"`.

If CHARSET is given and isn't false, it will be set to default charset. Otherwise, default charset won't be changed. In both cases, current default charset will be returned.

NOTE: Default charset *should not* be changed.

- fallback [CHARSET]

  Get/set fallback charset.

  Fallback charset is used by this module when conversion by given charset is failed and `"FALLBACK"` error handling scheme is specified. Modules using this module may use this charset as last resort of charset for conversion. By default, it is `"UTF-8"`.

  If CHARSET is given and isn't false, it will be set to fallback charset. If CHARSET is `"NONE"`, fallback charset will be undefined. Otherwise, fallback charset won't be changed. In any cases, current fallback charset will be returned.

  NOTE: It *is* useful that `"US-ASCII"` is specified as fallback charset, since result of conversion will be readable without charset information.

- recommended CHARSET [, HEADERENC, BODYENC [, ENCCHARSET]]

  Get/set charset profiles.

  If optional arguments are given and any of them are not false, profiles for CHARSET will be set by those arguments. Otherwise, profiles won't be changed. In both cases, current profiles for CHARSET will be returned as 3-item list of (HEADERENC, BODYENC, ENCCHARSET).

  HEADERENC is recommended encoding scheme for message header. It may be one of `"B"`, `"Q"`, `"S"` (shorter one of either) or `undef` (might not be encoded).

  BODYENC is recommended transfer-encoding for message body. It may be one of `"B"`, `"Q"`, `"S"` (shorter one of either) or `undef` (might not be transfer-encoded).

  ENCCHARSET is a charset which is compatible with given CHARSET and is recommended to be used for MIME messages on Internet. If conversion is not needed (or this module doesn't know appropriate charset), ENCCHARSET is `undef`.

  NOTE: This function in the future releases can accept more optional arguments (for example, properties to handle character widths, line folding behavior, ...). So format of returned value may probably be changed. Use "header_encoding", "body_encoding" or "output_charset" to get particular profile.

Constants

- USE_ENCODE

  Unicode/multibyte support flag. Non-empty string will be set when Unicode and multibyte support is enabled. Currently, this flag will be non-empty on Perl 5.7.3 or later and empty string on earlier versions of Perl.

Error Handling

“body_encode” and “header_encode” accept following `Replacement` options:

- `"DEFAULT"`

  Put a substitution character in place of a malformed character. For UCM-based encodings, <subchar> will be used.

- `"FALLBACK"`

  Try `"DEFAULT"` scheme using *fallback charset* (see “fallback”). When fallback charset is undefined and conversion causes error, code will die on error with an error message.

- `"CROAK"`

  Code will die on error immediately with an error message. Therefore, you should trap the fatal error with eval{} unless you really want to let it die on error. Synonym is `"STRICT"`.

- `"PERLQQ"`

- `"HTMLCREF"`

- `"XMLCREF"`

  Use `FB_PERLQQ`, `FB_HTMLCREF` or `FB_XMLCREF` scheme defined by *Encode* module.

- numeric values

  Numeric values are also allowed. For more details see *“Handling Malformed Data” in Encode*.

If error handling scheme is not specified or unknown scheme is specified, `"DEFAULT"` will be assumed.

Configuration File

Built-in defaults for option parameters can be overridden by configuration file: `MIME/Charset/Defaults.pm`. For more details read `MIME/Charset/Defaults.pm.sample`.

VERSION

Consult $VERSION variable.

Development versions of this module may be found at http://hatuka.nezumi.nu/repos/MIME-Charset/.

Incompatible Changes

- Release 1.001
  - new() method returns an object when CHARSET argument is not specified.
- Release 1.005
  - Restrict characters in encoded-word according to RFC 2047 section 5 (3). This also affects return value of encoded_header_len() method.

- Release 1.008.2
  - body_encoding() method may also returns "S".
  - Return value of body_encode() method for UTF-8 may include "QUOTED-PRINTABLE" encoding item that in earlier versions was fixed to "BASE64".

SEE ALSO

Multipurpose Internet Mail Extensions (MIME).

AUTHOR

Hatuka*nezumi - IKEDA Soji <hatuka(at)nezumi.nu>

COPYRIGHT

Copyright (C) 2006-2017 Hatuka*nezumi - IKEDA Soji. This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

MIME::QuotedPrint

Quoted-printable encoding and decoding as specified by RFC 2045.

Quoted-printable is the encoding of choice for mail bodies that are *mostly* ASCII with the occasional non-ASCII byte - long passages of plain text stay readable, and only the unsafe bytes turn into =XX hex triplets. For content that is truly binary, prefer base64 via MIME::Base64.

The encoder keeps lines at most 76 characters long by inserting soft line breaks (= at end of line, then the $eol sequence). Pass a $binmode flag when you want \n itself to be encoded - useful when the decoded output must be byte-for-byte identical regardless of the receiver's line-ending conventions. An $eol of "" disables both soft breaks and \n rewriting.

The decoder always produces \n-terminated output, even when the input used \r\n line endings.

This is MIME::QuotedPrint from MIME-Base64 3.16_01; MIME/QuotedPrint.pm is upstream's. encode_qp and decode_qp are exported by default.

Synopsis

```
use MIME::QuotedPrint;

my $qp = encode_qp("caf\xe9 = coffee\n");
print $qp;                                  # caf=E9 =3D coffee
print decode_qp($qp) eq "caf\xe9 = coffee\n" ? "round trip\n" : "";

print encode_qp('x' x 80);    # 75 x's and "=", then "xxxxx=":
                              # text without a final newline ends in␣
 ↪a soft break
```

Differences from upstream

None: upstream's tests pass unchanged (t/81-xs-native/MIME-Base64/230-upstream-quoted-print.t), and t/81-xs-native/MIME-QuotedPrint checks the rest against perl 5.44.

Functions

Encoding and decoding

encode_qp

Encode a string as quoted-printable.

decode_qp

Decode a quoted-printable string back to plain bytes.

encode_qp

Encode a string as quoted-printable.

Synopsis

```
$encoded = encode_qp($str);
$encoded = encode_qp($str, $eol);
$encoded = encode_qp($str, $eol, $binmode);
```

What you get back

A scalar holding the quoted-printable encoding of `$str`. Printable ASCII bytes (plus `\t`) pass through unchanged; every other byte, and `=` itself, becomes a `=XX` triplet with uppercase hex digits. Lines are kept at most 76 characters by inserting soft breaks: a trailing `=` followed by the `$eol` sequence.

`$eol` defaults to `"\n"`. Pass `"\015\012"` to produce output suitable for direct transmission over the wire. Pass `""` to suppress soft breaks and force binary-mode behavior for `\n`.

`$binmode`, when true, encodes `\n` and `\r` as `=0A` / `=0D` rather than passing them through as line terminators. Use this when the decoded form must match the input byte for byte on every platform.

Examples

```
use MIME::QuotedPrint;
print encode_qp("Hello, world!\n");       # Hello, world!\n
print encode_qp("caf\xe9\n");             # caf=E9\n
print encode_qp("\x00\x01\x02", "", 1);   # =00=01=02
print encode_qp("x" x 80 . "\n");         # wraps at col 75 with =\n
↪soft break
```

Edge cases

- Empty `$str` returns the empty string.
- `$eol` of `""` disables soft line breaks and encodes every `\n`.
- Trailing spaces and tabs on a line are always encoded so they survive transports that strip trailing whitespace.

Differences from upstream

Fully compatible with upstream `MIME::QuotedPrint` 3.16.

See also

- `decode_qp` - reverse the encoding.
- `MIME::Base64::encode_base64` - better choice for fully binary data.

decode_qp

Decode a quoted-printable string back to plain bytes.

Synopsis

```
$decoded = decode_qp($encoded);
```

What you get back

A scalar holding the decoded byte string. Every =XX triplet is replaced by the byte whose hex value is XX; soft line breaks (a = immediately before `\r\n` or `\n`) are removed.

Line terminators are normalized: whether the input used `\n` or `\r\n`, each line in the result ends with `\n`.

Examples

```
use MIME::QuotedPrint;
print decode_qp("Hello, world!\n");     # Hello, world!\n
print decode_qp("caf=E9\n");            # "caf\xe9" (the Latin-1␣
↪byte E9)
print decode_qp("line =\r\ncontinued"); # line continued
print decode_qp("=00=01=02");           # three bytes: 00 01 02
```

Edge cases

- A stray = not followed by two hex digits or a line break is passed through literally, matching the upstream decoder.
- Trailing whitespace on a decoded line is stripped, matching the encoder's soft-break contract.

- Lowercase hex digits are accepted even though the encoder only emits uppercase.

Differences from upstream

Fully compatible with upstream `MIME::QuotedPrint` 3.16.

See also

- `encode_qp` - produce the encoded form.
- `MIME::Base64::decode_base64` - the counterpart for binary payloads.

5.2.65 Math

Namespace containing the following modules:

- `Math::BigFloat` - arbitrary size floating point math package
- `Math::BigInt` - arbitrary size integer math package
- `Math::BigRat` - arbitrary size rational number math package
- `Math::Complex` - complex numbers and associated mathematical functions
- `Math::GMP` - Arbitrary-precision integer arithmetic backed by GNU `libgmp`.
- `Math::Random`
- `Math::Trig` - trigonometric functions

Math::BigFloat

arbitrary size floating point math package

SYNOPSIS

```
use Math::BigFloat;

# Configuration methods (may be used as class methods and instance
 ↪methods)

Math::BigFloat->accuracy($n);       # set accuracy
Math::BigFloat->accuracy();         # get accuracy
Math::BigFloat->precision($n);      # set precision
Math::BigFloat->precision();        # get precision
Math::BigFloat->round_mode($m);     # set rounding mode, must be
                                    # 'even', 'odd', '+inf', '-inf',
                                    # 'zero', 'trunc', or 'common'
Math::BigFloat->round_mode();       # get class rounding mode
Math::BigFloat->div_scale($n);      # set fallback accuracy
Math::BigFloat->div_scale();        # get fallback accuracy
Math::BigFloat->trap_inf($b);       # trap infinities or not
Math::BigFloat->trap_inf();         # get trap infinities status
```

(continues on next page)

(continued from previous page)

```
Math::BigFloat->trap_nan($b);       # trap NaNs or not
Math::BigFloat->trap_nan();         # get trap NaNs status
Math::BigFloat->config($par, $val); # set configuration parameter
Math::BigFloat->config($par);       # get configuration parameter
Math::BigFloat->config();           # get hash with configuration
Math::BigFloat->config("lib");      # get name of backend library

# Generic constructor method (always returns a new object)

$x = Math::BigFloat->new($str);               # defaults to 0
$x = Math::BigFloat->new('256');              # from decimal
$x = Math::BigFloat->new('0256');             # from decimal
$x = Math::BigFloat->new('0xcafe');           # from hexadecimal
$x = Math::BigFloat->new('0x1.cafep+7');      # from hexadecimal
$x = Math::BigFloat->new('0o377');            # from octal
$x = Math::BigFloat->new('0o1.3571p+6');      # from octal
$x = Math::BigFloat->new('0b101');            # from binary
$x = Math::BigFloat->new('0b1.101p+3');       # from binary

# Specific constructor methods (no prefix needed; when used as
# instance method, the value is assigned to the invocand)

$x = Math::BigFloat->from_dec('234');         # from decimal
$x = Math::BigFloat->from_hex('c.afep+3');    # from hexadecimal
$x = Math::BigFloat->from_hex('cafe');        # from hexadecimal
$x = Math::BigFloat->from_oct('1.3267p-4');   # from octal
$x = Math::BigFloat->from_oct('377');         # from octal
$x = Math::BigFloat->from_bin('0b1.1001p-4'); # from binary
$x = Math::BigFloat->from_bin('0101');        # from binary
$x = Math::BigFloat->from_bytes($bytes);      # from byte string
$x = Math::BigFloat->from_base('why', 36);    # from any base
$x = Math::BigFloat->from_ieee754($b, $fmt);  # from IEEE-754 bytes
$x = Math::BigFloat->from_fp80($b);           # from x86 80-bit
$x = Math::BigFloat->bzero();                 # create a +0
$x = Math::BigFloat->bone();                  # create a +1
$x = Math::BigFloat->bone('-');               # create a -1
$x = Math::BigFloat->binf();                  # create a +inf
$x = Math::BigFloat->binf('-');               # create a -inf
$x = Math::BigFloat->bnan();                  # create a Not-A-
 ↪Number
$x = Math::BigFloat->bpi();                   # returns pi

$y = $x->copy();        # make a copy (unlike $y = $x)
$y = $x->as_int();      # return as BigInt
$y = $x->as_float();    # return as a Math::BigFloat
$y = $x->as_rat();      # return as a Math::BigRat

# Boolean methods (these don't modify the invocand)
```

(continues on next page)

(continued from previous page)

```
$x->is_zero();          # true if $x is 0
$x->is_one();           # true if $x is +1
$x->is_one("+");        # true if $x is +1
$x->is_one("-");        # true if $x is -1
$x->is_inf();           # true if $x is +inf or -inf
$x->is_inf("+");        # true if $x is +inf
$x->is_inf("-");        # true if $x is -inf
$x->is_nan();           # true if $x is NaN

$x->is_finite();        # true if -inf < $x < inf
$x->is_positive();      # true if $x > 0
$x->is_pos();           # true if $x > 0
$x->is_negative();      # true if $x < 0
$x->is_neg();           # true if $x < 0
$x->is_non_positive()   # true if $x <= 0
$x->is_non_negative()   # true if $x >= 0

$x->is_odd();           # true if $x is odd
$x->is_even();          # true if $x is even
$x->is_int();           # true if $x is an integer

# Comparison methods (these don't modify the invocand)

$x->bcmp($y);           # compare numbers (undef, < 0, == 0, > 0)
$x->bacmp($y);          # compare abs values (undef, < 0, == 0, > 0)
$x->beq($y);            # true if $x == $y
$x->bne($y);            # true if $x != $y
$x->blt($y);            # true if $x < $y
$x->ble($y);            # true if $x <= $y
$x->bgt($y);            # true if $x > $y
$x->bge($y);            # true if $x >= $y

# Arithmetic methods (these modify the invocand)

$x->bneg();             # negation
$x->babs();             # absolute value
$x->bsgn();             # sign function (-1, 0, 1, or NaN)
$x->binc();             # increment $x by 1
$x->bdec();             # decrement $x by 1
$x->badd($y);           # addition (add $y to $x)
$x->bsub($y);           # subtraction (subtract $y from $x)
$x->bmul($y);           # multiplication (multiply $x by $y)
$x->bmuladd($y, $z);    # $x = $x * $y + $z
$x->bdiv($y);           # division (floored), set $x to quotient
$x->bmod($y);           # modulus (x % y)
$x->bmodinv($mod);      # modular multiplicative inverse
$x->bmodpow($y, $mod);  # modular exponentiation (($x ** $y) % $mod)
$x->btdiv($y);          # division (truncated), set $x to quotient
$x->btmod($y);          # modulus (truncated)
```

(continues on next page)

(continued from previous page)

```
$x->binv()               # inverse (1/$x)
$x->bpow($y);            # power of arguments (x ** y)
$x->blog();              # logarithm of $x to base e (Euler's number)
$x->blog($base);         # logarithm of $x to base $base (e.g., base
 ↪2)
$x->bexp();              # calculate e ** $x where e is Euler's
 ↪number
$x->bilog2();            # log2($x) rounded down to nearest int
$x->bilog10();           # log10($x) rounded down to nearest int
$x->bclog2();            # log2($x) rounded up to nearest int
$x->bclog10();           # log10($x) rounded up to nearest int
$x->bnok($y);            # combinations (binomial coefficient n over
 ↪k)
$x->bperm($y);           # permutations
$x->bsin();              # sine
$x->bcos();              # cosine
$x->batan();             # inverse tangent
$x->batan2($y);          # two-argument inverse tangent
$x->bsqrt();             # calculate square root
$x->broot($y);           # $y'th root of $x (e.g. $y == 3 => cubic
 ↪root)
$x->bfac();              # factorial of $x (1*2*3*4*..$x)
$x->bdfac();             # double factorial of $x ($x*($x-2)*($x-4)*.
 ↪..)
$x->btfac();             # triple factorial of $x ($x*($x-3)*($x-6)*.
 ↪..)
$x->bmfac($k);           # $k'th multi-factorial of $x ($x*($x-$k)*..
 ↪.)
$x->bfib($k);            # $k'th Fibonacci number
$x->blucas($k);          # $k'th Lucas number

$x->blsft($n);           # left shift $n places in base 2
$x->blsft($n, $b);       # left shift $n places in base $b
$x->brsft($n);           # right shift $n places in base 2
$x->brsft($n, $b);       # right shift $n places in base $b

# Bitwise methods (these modify the invocand)

$x->bblsft($y);          # bitwise left shift
$x->bbrsft($y);          # bitwise right shift
$x->band($y);            # bitwise and
$x->bior($y);            # bitwise inclusive or
$x->bxor($y);            # bitwise exclusive or
$x->bnot();              # bitwise not (two's complement)

# Rounding methods (these modify the invocand)

$x->round($A, $P, $R);  # round to accuracy or precision using
                         #   rounding mode $R
```

(continues on next page)

(continued from previous page)

```
$x->bround($n);          # accuracy: preserve $n digits
$x->bfround($n);         # $n > 0: round to $nth digit left of dec.
 ↪point
                         # $n < 0: round to $nth digit right of dec.
 ↪point
$x->bfloor();            # round towards minus infinity
$x->bceil();             # round towards plus infinity
$x->bint();              # round towards zero

# Other mathematical methods (these don't modify the invocand)

$x->bgcd($y);            # greatest common divisor
$x->blcm($y);            # least common multiple

# Object property methods (these don't modify the invocand)

$x->sign();              # the sign, either +, - or NaN
$x->digit($n);           # the nth digit, counting from the right
$x->digit(-$n);          # the nth digit, counting from the left
$x->length();            # return number of digits in number
$x->mantissa();          # return (signed) mantissa as BigInt
$x->exponent();          # return exponent as BigInt
$x->parts();             # return (mantissa,exponent) as BigInt
$x->sparts();            # mantissa and exponent (as integers)
$x->nparts();            # mantissa and exponent (normalised)
$x->eparts();            # mantissa and exponent (engineering
 ↪notation)
$x->dparts();            # integer and fraction part
$x->fparts();            # numerator and denominator
$x->numerator();         # numerator
$x->denominator();       # denominator

# Conversion methods (these don't modify the invocand)

$x->bstr();              # decimal notation (possibly zero padded)
$x->bsstr();             # string in scientific notation with
 ↪integers
$x->bnstr();             # string in normalized notation
$x->bestr();             # string in engineering notation
$x->bdstr();             # string in decimal notation (no padding)
$x->bfstr();             # string in fractional notation

$x->to_hex();            # as signed hexadecimal string
$x->to_bin();            # as signed binary string
$x->to_oct();            # as signed octal string
$x->to_bytes();          # as byte string
$x->to_ieee754($fmt);    # to bytes encoded according to IEEE 754-
 ↪2008
$x->to_fp80();           # encode value in x86 80-bit format
```

(continues on next page)

(continued from previous page)

```
$x->as_hex();           # as signed hexadecimal string with "0x"␣
↪prefix
$x->as_bin();           # as signed binary string with "0b" prefix
$x->as_oct();           # as signed octal string with "0" prefix

# Other conversion methods (these don't modify the invocand)

$x->numify();           # return as scalar (might overflow or␣
↪underflow)
```

DESCRIPTION

Math::BigFloat provides support for arbitrary precision floating point. Overloading is also provided for Perl operators.

All operators (including basic math operations) are overloaded if you declare your big floating point numbers as

```
$x = Math::BigFloat -> new('12_3.456_789_123_456_789E-2');
```

Operations with overloaded operators preserve the arguments, which is exactly what you expect.

Input

Input values to these routines may be any scalar number or string that looks like a number. Anything that is accepted by Perl as a literal numeric constant should be accepted by this module.

- Leading and trailing whitespace is ignored.
- Leading zeros are ignored, except for floating point numbers with a binary exponent, in which case the number is interpreted as an octal floating point number. For example, “01.4p+0” gives 1.5, “00.4p+0” gives 0.5, but “0.4p+0” gives a NaN. And while “0377” gives 255, “0377p0” gives 255.
- If the string has a “0x” or “0X” prefix, it is interpreted as a hexadecimal number.
- If the string has a “0o” or “0O” prefix, it is interpreted as an octal number. A floating point literal with a “0” prefix is also interpreted as an octal number.
- If the string has a “0b” or “0B” prefix, it is interpreted as a binary number.
- Underline characters are allowed in the same way as they are allowed in literal numerical constants.
- If the string can not be interpreted, NaN is returned.
- For hexadecimal, octal, and binary floating point numbers, the exponent must be separated from the significand (mantissa) by the letter “p” or “P”, not “e” or “E” as with decimal numbers.

Some examples of valid string input

```
Input string                 Resulting value

123                          123
1.23e2                       123
12300e-2                     123

67_538_754                   67538754
-4_5_6.7_8_9e+0_1_0          -4567890000000

0x13a                        314
0x13ap0                      314
0x1.3ap+8                    314
0x0.00013ap+24               314
0x13a000p-12                 314

0o472                        314
0o1.164p+8                   314
0o0.0001164p+20              314
0o1164000p-10                314

0472                         472     Note!
01.164p+8                    314
00.0001164p+20               314
01164000p-10                 314

0b100111010                  314
0b1.0011101p+8               314
0b0.00010011101p+12          314
0b100111010000p-3            314

0x1.921fb5p+1                3.14159262180328369140625e+0
0o1.2677025p1                2.71828174591064453125
01.2677025p1                 2.71828174591064453125
0b1.1001p-4                  9.765625e-2
```

Output

Output values are usually Math::BigFloat objects.

Boolean operators *is_zero()*, *is_one()*, *is_inf()*, etc. return true or false.

Comparison operators *bcmp()* and *bacmp()*) return -1, 0, 1, or undef.

METHODS

Math::BigFloat supports all methods that Math::BigInt supports, except it calculates non-integer results when possible. Please see *Math::BigInt* for a full description of each method. Below are just the most important differences:

Configuration methods

- accuracy()

```
$x->accuracy(5);           # local for $x
CLASS->accuracy(5);        # global for all members of CLASS
                           # Note: This also applies to new()!

$A = $x->accuracy();       # read out accuracy that affects $x
$A = CLASS->accuracy();    # read out global accuracy
```

Set or get the global or local accuracy, aka how many significant digits the results have. If you set a global accuracy, then this also applies to new()!

Warning! The accuracy *sticks*, e.g. once you created a number under the influence of `CLASS->accuracy($A)`, all results from math operations with that number will also be rounded.

In most cases, you should probably round the results explicitly using one of *"round()" in Math::BigInt*, *"bround()" in Math::BigInt* or *"bfround()" in Math::BigInt* or by passing the desired accuracy to the math operation as additional parameter:

```
my $x = Math::BigInt->new(30000);
my $y = Math::BigInt->new(7);
print scalar $x->copy()->bdiv($y, 2);           # print 4300
print scalar $x->copy()->bdiv($y)->bround(2);   # print 4300
```

- precision()

```
$x->precision(-2);         # local for $x, round at the second
                           # digit right of the dot
$x->precision(2);          # ditto, round at the second digit
                           # left of the dot

CLASS->precision(5);       # Global for all members of CLASS
                           # This also applies to new()!
CLASS->precision(-5);      # ditto

$P = CLASS->precision();   # read out global precision
$P = $x->precision();      # read out precision that affects $x
```

Note: You probably want to use "accuracy()" instead. With "accuracy()" you set the number of digits each result should have, with "precision()" you set the place where to round!

Constructor methods

- from_dec()

```
$x -> from_hex("314159");
$x = Math::BigInt -> from_hex("314159");
```

Interpret input as a decimal. It is equivalent to new(), but does not accept anything but strings representing finite, decimal numbers.

- from_hex()

```
$x -> from_hex("0x1.921fb54442d18p+1");
$x = Math::BigFloat -> from_hex("0x1.921fb54442d18p+1");
```

Interpret input as a hexadecimal string.A prefix ("0x", "x", ignoring case) is optional. A single underscore character ("_") may be placed between any two digits. If the input is invalid, a NaN is returned. The exponent is in base 2 using decimal digits.

If called as an instance method, the value is assigned to the invocand.

- from_oct()

```
$x -> from_oct("1.3267p-4");
$x = Math::BigFloat -> from_oct("1.3267p-4");
```

Interpret input as an octal string. A single underscore character ("_") may be placed between any two digits. If the input is invalid, a NaN is returned. The exponent is in base 2 using decimal digits.

If called as an instance method, the value is assigned to the invocand.

- from_bin()

```
$x -> from_bin("0b1.1001p-4");
$x = Math::BigFloat -> from_bin("0b1.1001p-4");
```

Interpret input as a hexadecimal string. A prefix ("0b" or "b", ignoring case) is optional. A single underscore character ("_") may be placed between any two digits. If the input is invalid, a NaN is returned. The exponent is in base 2 using decimal digits.

If called as an instance method, the value is assigned to the invocand.

- from_bytes()

```
$x = Math::BigFloat->from_bytes("\xf3\x6b");  # $x = 62315
```

Interpret the input as a byte string, assuming big endian byte order. The output is always a non-negative, finite integer.

See *"from_bytes()" in Math::BigInt*.

- from_ieee754()

Interpret the input as a value encoded as described in IEEE754-2008. The input can be given as a byte string, hex string, or binary string. The input is assumed to be in big-endian byte-order.

```
# Both $dbl, $xr, $xh, and $xb below are 3.141592...

$dbl = unpack "d>", "\x40\x09\x21\xfb\x54\x44\x2d\x18";
```

(continues on next page)

(continued from previous page)

```
$raw = "\x40\x09\x21\xfb\x54\x44\x2d\x18";           # raw bytes
$xr  = Math::BigFloat -> from_ieee754($raw, "binary64");

$hex = "400921fb54442d18";
$xh  = Math::BigFloat -> from_ieee754($hex, "binary64");

$bin =
 ↪"0100000000001001001000011111101101010100010001000010110100011000
 ↪";
$xb  = Math::BigFloat -> from_ieee754($bin, "binary64");
```

Supported formats are all IEEE 754 binary formats: “binary16”, “binary32”, “binary64”, “binary128”, “binary160”, “binary192”, “binary224”, “binary256”, etc. where the number of bits is a multiple of 32 for all formats larger than “binary128”. Aliases are “half” (“binary16”), “single” (“binary32”), “double” (“binary64”), “quadruple” (“binary128”), “octuple” (“binary256”), and “sexdecuple” (“binary512”).

See also “to_ieee754()”.

- from_fp80()

  Interpret the input as a value encoded as an x86 80-bit floating point number. The input can be given as a 10 character byte string, 20 character hex string, or 80 character binary string. The input is assumed to be in big-endian byte-order.

```
    # Both $xr, $xh, and $xb below are 3.141592...

    $dbl = unpack "d>", "\x40\x09\x21\xfb\x54\x44\x2d\x18";

    $raw = "\x40\x00\xc9\x0f\xda\xa2\x21\x68\xc2\x35";  # raw␣
 ↪bytes
    $xr  = Math::BigFloat -> from_fp80($raw);

    $hex = "4000c90fdaa22168c235";
    $xh  = Math::BigFloat -> from_fp80($hex);

    $bin = "0100000000000000110010010000111111011010"
         . "1010001000100001011010001100001000110101";
    $xb  = Math::BigFloat -> from_fp80($bin);
See also L</to_ieee754()>.
```

- from_base()

  See *“from_base()” in Math::BigInt*.

- bpi()

```
print Math::BigFloat->bpi(100), "\n";
```

Calculate PI to N digits (including the 3 before the dot). The result is rounded according to the current rounding mode, which defaults to “even”.

This method was added in v1.87 of Math::BigInt (June 2007).

- as_int()

```
$y = $x -> as_int();        # $y is a Math::BigInt
```

Returns $x as a Math::BigInt object regardless of upgrading and downgrading. If $x is finite, but not an integer, $x is truncated.

- as_rat()

```
$y = $x -> as_rat();        # $y is a Math::BigRat
```

Returns $x a Math::BigRat object regardless of upgrading and downgrading. The invocand is not modified.

- as_float()

```
$y = $x -> as_float();      # $y is a Math::BigFloat
```

Returns $x a Math::BigFloat object regardless of upgrading and downgrading. The invocand is not modified.

Arithmetic methods

- bdiv()

```
$x->bdiv($y);               # set $x to quotient
($q, $r) = $x->bdiv($y);    # also remainder
```

This is an alias for "bfdiv()".

- bmod()

```
$x->bmod($y);
```

Returns $x modulo $y. When $x is finite, and $y is finite and non-zero, the result is identical to the remainder after floored division (F-division). If, in addition, both $x and $y are integers, the result is identical to the result from Perl's % operator.

- bfdiv()

```
$q = $x->bfdiv($y);
($q, $r) = $x->bfdiv($y);
```

In scalar context, divides $x by $y and returns the result to the given accuracy or precision or the default accuracy. In list context, does floored division (F-division), returning an integer $q and a remainder $r

```
$q = floor($x / $y)
$r = $x - $q * $y
```

so that the following relationship always holds

```
$x = $q * $y + $r
```

The remainer (modulo) is equal to what is returned by `$x->bmod($y)`.

- binv()

```
$x->binv();
```

Invert the value of $x, i.e., compute 1/$x.

- bmuladd()

```
$x->bmuladd($y,$z);
```

Multiply $x by $y, and then add $z to the result.

This method was added in v1.87 of Math::BigInt (June 2007).

- bexp()

```
$x->bexp($accuracy);            # calculate e ** X
```

Calculates the expression `e ** $x` where e is Euler's number.

This method was added in v1.82 of Math::BigInt (April 2007).

- bnok()

See *"bnok()" in Math::BigInt*.

- bperm()

See *"bperm()" in Math::BigInt*.

- bsin()

```
my $x = Math::BigFloat->new(1);
print $x->bsin(100), "\n";
```

Calculate the sinus of $x, modifying $x in place.

This method was added in v1.87 of Math::BigInt (June 2007).

- bcos()

```
my $x = Math::BigFloat->new(1);
print $x->bcos(100), "\n";
```

Calculate the cosinus of $x, modifying $x in place.

This method was added in v1.87 of Math::BigInt (June 2007).

- batan()

```
my $x = Math::BigFloat->new(1);
print $x->batan(100), "\n";
```

Calculate the arcus tanges of $x, modifying $x in place. See also "batan2()".

This method was added in v1.87 of Math::BigInt (June 2007).

- batan2()

```
my $y = Math::BigFloat->new(2);
my $x = Math::BigFloat->new(3);
print $y->batan2($x), "\n";
```

Calculate the arcus tanges of $y divided by $x, modifying $y in place. See also "batan()".

This method was added in v1.87 of Math::BigInt (June 2007).

- bgcd()

```
$x -> bgcd($y);               # GCD of $x and $y
$x -> bgcd($y, $z, ...);      # GCD of $x, $y, $z, ...
```

Returns the greatest common divisor (GCD), which is the number with the largest absolute value such that $x/$gcd, $y/$gcd, ... is an integer. For example, when the operands are 0.8 and 1.2, the GCD is 0.4. This is a generalisation of the ordinary GCD for integers. See *"gcd()" in Math::BigInt*.

String conversion methods

- bstr()

```
my $x = Math::BigRat->new('8/4');
print $x->bstr(), "\n";             # prints 1/2
```

Returns a string representing the number.

- bsstr()

  See *"bsstr()" in Math::BigInt*.

- bnstr()

  See *"bnstr()" in Math::BigInt*.

- bestr()

  See *"bestr()" in Math::BigInt*.

- bdstr()

  See *"bdstr()" in Math::BigInt*.

- to_bytes()

  See *"to_bytes()" in Math::BigInt*.

- to_ieee754()

  Encodes the invocand as a byte string in the given format as specified in IEEE 754-2008. Note that the encoded value is the nearest possible representation of the value. This value might not be exactly the same as the value in the invocand.

```
# $x = 3.14159265358979323
$x = Math::BigFloat -> bpi(30);

```

(continues on next page)

(continued from previous page)

```
$b = $x -> to_ieee754("binary64");  # encode as 8 bytes
$h = unpack "H*", $b;               # "400921fb54442d18"

# 3.14159265358979311599796 3...
$y = Math::BigFloat -> from_ieee754($h, "binary64");
```

All binary formats in IEEE 754-2008 are accepted. For convenience, som aliases are recognized: “half” for “binary16”, “single” for “binary32”, “double” for “binary64”, “quadruple” for “binary128”, “octuple” for “binary256”, and “sexdecuple” for “binary512”.

See also “from_ieee754()”, https://en.wikipedia.org/wiki/IEEE_754.

ACCURACY AND PRECISION

See also: Rounding.

Math::BigFloat supports both precision (rounding to a certain place before or after the dot) and accuracy (rounding to a certain number of digits). For a full documentation, examples and tips on these topics please see the large section about rounding in *Math::BigInt*.

Since things like `sqrt(2)` or `1 / 3` must presented with a limited accuracy lest a operation consumes all resources, each operation produces no more than the requested number of digits.

If there is no global precision or accuracy set, **and** the operation in question was not called with a requested precision or accuracy, **and** the input $x has no accuracy or precision set, then a fallback parameter will be used. For historical reasons, it is called `div_scale` and can be accessed via:

```
$d = Math::BigFloat->div_scale();       # query
Math::BigFloat->div_scale($n);          # set to $n digits
```

The default value for `div_scale` is 40.

In case the result of one operation has more digits than specified, it is rounded. The rounding mode taken is either the default mode, or the one supplied to the operation after the *scale*:

```
$x = Math::BigFloat->new(2);
Math::BigFloat->accuracy(5);              # 5 digits max
$y = $x->copy()->bdiv(3);                 # gives 0.66667
$y = $x->copy()->bdiv(3,6);               # gives 0.666667
$y = $x->copy()->bdiv(3,6,undef,'odd');   # gives 0.666667
Math::BigFloat->round_mode('zero');
$y = $x->copy()->bdiv(3,6);               # will also give 0.666667
```

Note that `Math::BigFloat->accuracy()` and `Math::BigFloat->precision()` set the global variables, and thus **any** newly created number will be subject to the global rounding **immediately**. This means that in the examples above, the 3 as argument to “bdiv()” will also get an accuracy of **5**.

It is less confusing to either calculate the result fully, and afterwards round it explicitly, or use the additional parameters to the math functions like so:

```
use Math::BigFloat;
$x = Math::BigFloat->new(2);
$y = $x->copy()->bdiv(3);
print $y->bround(5),"\n";                # gives 0.66667

or

use Math::BigFloat;
$x = Math::BigFloat->new(2);
$y = $x->copy()->bdiv(3,5);              # gives 0.66667
print "$y\n";
```

Rounding

- bfround (+$scale)

  Rounds to the $scale'th place left from the '.', counting from the dot. The first digit is numbered 1.

- bfround (-$scale)

  Rounds to the $scale'th place right from the '.', counting from the dot.

- bfround (0)

  Rounds to an integer.

- bround (+$scale)

  Preserves accuracy to $scale digits from the left (aka significant digits) and pads the rest with zeros. If the number is between 1 and -1, the significant digits count from the first non-zero after the '.'

- bround (-$scale) and bround (0)

  These are effectively no-ops.

All rounding functions take as a second parameter a rounding mode from one of the following: 'even', 'odd', '+inf', '-inf', 'zero', 'trunc' or 'common'.

The default rounding mode is 'even'. By using `Math::BigFloat->round_mode($round_mode);` you can get and set the default mode for subsequent rounding. The usage of `$Math::BigFloat::$round_mode` is no longer supported. The second parameter to the round functions then overrides the default temporarily.

The "as_int()" method returns a BigInt from a Math::BigFloat. It uses 'trunc' as rounding mode to make it equivalent to:

```
$x = 2.5;
$y = int($x) + 2;
```

You can override this by passing the desired rounding mode as parameter to "as_int()":

```
$x = Math::BigFloat->new(2.5);
$y = $x->as_number('odd');      # $y = 3
```

NUMERIC LITERALS

After `use Math::BigFloat ':constant'` all numeric literals in the given scope are converted to `Math::BigFloat` objects. This conversion happens at compile time.

For example,

```
perl -MMath::BigFloat=:constant -le 'print 2e-150'
```

prints the exact value of `2e-150`. Note that without conversion of constants the expression `2e-150` is calculated using Perl scalars, which leads to an inaccuracte result.

Note that strings are not affected, so that

```
use Math::BigFloat qw/:constant/;

$y = "1234567890123456789012345678901234567890"
        + "123456789123456789";
```

does not give you what you expect. You need an explicit Math::BigFloat->new() around at least one of the operands. You should also quote large constants to prevent loss of precision:

```
use Math::BigFloat;

$x = Math::BigFloat->new("123456788912345678912345678912345678 9");
```

Without the quotes Perl converts the large number to a floating point constant at compile time, and then converts the result to a Math::BigFloat object at runtime, which results in an inaccurate result.

Hexadecimal, octal, and binary floating point literals

Perl (and this module) accepts hexadecimal, octal, and binary floating point literals, but use them with care with Perl versions before v5.32.0, because some versions of Perl silently give the wrong result. Below are some examples of different ways to write the number decimal 314.

Hexadecimal floating point literals:

```
0x1.3ap+8          0X1.3AP+8
0x1.3ap8           0X1.3AP8
0x13a0p-4          0X13A0P-4
```

Octal floating point literals (with “0” prefix):

```
01.164p+8          01.164P+8
01.164p8           01.164P8
011640p-4          011640P-4
```

Octal floating point literals (with "0o" prefix) (requires v5.34.0):

```
0o1.164p+8          0O1.164P+8
0o1.164p8           0O1.164P8
0o11640p-4          0O11640P-4
```

Binary floating point literals:

```
0b1.0011101p+8      0B1.0011101P+8
0b1.0011101p8       0B1.0011101P8
0b10011101000p-2    0B10011101000P-2
```

Math library

Math with the numbers is done (by default) by a module called Math::BigInt::Calc. This is equivalent to saying:

```
use Math::BigFloat lib => "Calc";
```

You can change this by using:

```
use Math::BigFloat lib => "GMP";
```

Note: General purpose packages should not be explicit about the library to use; let the script author decide which is best.

Note: The keyword 'lib' will warn when the requested library could not be loaded. To suppress the warning use 'try' instead:

```
use Math::BigFloat try => "GMP";
```

If your script works with huge numbers and Calc is too slow for them, you can also for the loading of one of these libraries and if none of them can be used, the code will die:

```
use Math::BigFloat only => "GMP,Pari";
```

The following would first try to find Math::BigInt::Foo, then Math::BigInt::Bar, and when this also fails, revert to Math::BigInt::Calc:

```
use Math::BigFloat lib => "Foo,Math::BigInt::Bar";
```

See the respective low-level library documentation for further details.

See *Math::BigInt* for more details about using a different low-level library.

EXPORTS

`Math::BigFloat` exports nothing by default, but can export the "bpi()" method:

```
use Math::BigFloat qw/bpi/;

print bpi(10), "\n";
```

- Modifying and =

  Beware of:

```
$x = Math::BigFloat->new(5);
$y = $x;
```

  It will not do what you think, e.g. making a copy of $x. Instead it just makes a second reference to the **same** object and stores it in $y. Thus anything that modifies $x will modify $y (except overloaded math operators), and vice versa. See *Math::BigInt* for details and how to avoid that.

- precision() vs. accuracy()

  A common pitfall is to use "precision()" when you want to round a result to a certain number of digits:

```
use Math::BigFloat;

Math::BigFloat->precision(4);           # does not do what you
                                        # think it does
my $x = Math::BigFloat->new(12345);     # rounds $x to "12000"!
print "$x\n";                           # print "12000"
my $y = Math::BigFloat->new(3);         # rounds $y to "0"!
print "$y\n";                           # print "0"
$z = $x / $y;                           # 12000 / 0 => NaN!
print "$z\n";
print $z->precision(),"\n";             # 4
```

  Replacing "precision()" with "accuracy()" is probably not what you want, either:

```
use Math::BigFloat;

Math::BigFloat->accuracy(4);          # enables global rounding:
my $x = Math::BigFloat->new(123456);  # rounded immediately
                                      #   to "12350"
print "$x\n";                         # print "123500"
my $y = Math::BigFloat->new(3);       # rounded to "3
print "$y\n";                         # print "3"
print $z = $x->copy()->bdiv($y),"\n"; # 41170
print $z->accuracy(),"\n";            # 4
```

  What you want to use instead is:

```
use Math::BigFloat;

my $x = Math::BigFloat->new(123456);    # no rounding
print "$x\n";                           # print "123456"
my $y = Math::BigFloat->new(3);         # no rounding
print "$y\n";                           # print "3"
print $z = $x->copy()->bdiv($y,4),"\n"; # 41150
print $z->accuracy(),"\n";              # undef
```

  In addition to computing what you expected, the last example also does **not** "taint"

the result with an accuracy or precision setting, which would influence any further operation.

BUGS

Please report any bugs or feature requests to `bug-math-bigint at rt.cpan.org`, or through the web interface at https://rt.cpan.org/Ticket/Create.html?Queue=Math-BigInt (requires login). We will be notified, and then you'll automatically be notified of progress on your bug as I make changes.

SUPPORT

You can find documentation for this module with the perldoc command.

```
perldoc Math::BigFloat
```

You can also look for information at:

- GitHub

  https://github.com/pjacklam/p5-Math-BigInt

- RT: CPAN's request tracker

  https://rt.cpan.org/Dist/Display.html?Name=Math-BigInt

- MetaCPAN

  https://metacpan.org/release/Math-BigInt

- CPAN Testers Matrix

  http://matrix.cpantesters.org/?dist=Math-BigInt

LICENSE

This program is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

SEE ALSO

Math::BigInt and *Math::BigRat* as well as the backend libraries *Math::BigInt::FastCalc*, Math::BigInt::GMP, and Math::BigInt::Pari, Math::BigInt::GMPz, and Math::BigInt::BitVect.

The pragmas *bigint*, *bigfloat*, and *bigrat* might also be of interest. In addition there is the *bignum* pragma which does upgrading and downgrading.

AUTHORS

- Mark Biggar, overloaded interface by Ilya Zakharevich, 1996-2001.
- Completely rewritten by Tels http://bloodgate.com in 2001-2008.
- Florian Ragwitz flora@cpan.org, 2010.
- Peter John Acklam pjacklam@gmail.com, 2011-.

Math::BigInt

arbitrary size integer math package

SYNOPSIS

```
use Math::BigInt;

# or make it faster with huge numbers: install (optional)
# Math::BigInt::GMP and always use (it falls back to
# pure Perl if the GMP library is not installed):
# (See also the L</Math Library> section!)

# to warn if Math::BigInt::GMP cannot be found, use
use Math::BigInt lib => 'GMP';

# to suppress the warning if Math::BigInt::GMP cannot be found, use
# use Math::BigInt try => 'GMP';

# to die if Math::BigInt::GMP cannot be found, use
# use Math::BigInt only => 'GMP';

# Configuration methods (may be used as class methods and instance
 ↪methods)

Math::BigInt->accuracy($n);       # set accuracy
Math::BigInt->accuracy();         # get accuracy
Math::BigInt->precision($n);      # set precision
Math::BigInt->precision();        # get precision
Math::BigInt->round_mode($m);     # set rounding mode, must be
                                  # 'even', 'odd', '+inf', '-inf',
                                  # 'zero', 'trunc', or 'common'
Math::BigInt->round_mode();       # get class rounding mode
Math::BigInt->div_scale($n);      # set fallback accuracy
Math::BigInt->div_scale();        # get fallback accuracy
Math::BigInt->trap_inf($b);       # trap infinities or not
Math::BigInt->trap_inf();         # get trap infinities status
Math::BigInt->trap_nan($b);       # trap NaNs or not
Math::BigInt->trap_nan();         # get trap NaNs status
Math::BigInt->config($par, $val); # set configuration parameter
Math::BigInt->config($par);       # get configuration parameter
Math::BigInt->config();           # get hash with configuration
Math::BigFloat->config("lib");    # get name of backend library

# Generic constructor method (always returns a new object)

$x = Math::BigInt->new($str);             # defaults to 0
$x = Math::BigInt->new('256');            # from decimal
$x = Math::BigInt->new('0256');           # from decimal
$x = Math::BigInt->new('0xcafe');         # from hexadecimal
```

(continues on next page)

(continued from previous page)

```
$x = Math::BigInt->new('0x1.fap+7');      # from hexadecimal
$x = Math::BigInt->new('0o377');          # from octal
$x = Math::BigInt->new('0o1.35p+6');      # from octal
$x = Math::BigInt->new('0b101');          # from binary
$x = Math::BigInt->new('0b1.101p+3');     # from binary

# Specific constructor methods (no prefix needed; when used as
# instance method, the value is assigned to the invocand)

$x = Math::BigInt->from_dec('234');       # from decimal
$x = Math::BigInt->from_hex('cafe');      # from hexadecimal
$x = Math::BigInt->from_hex('1.fap+7');   # from hexadecimal
$x = Math::BigInt->from_oct('377');       # from octal
$x = Math::BigInt->from_oct('1.35p+6');   # from octal
$x = Math::BigInt->from_bin('1101');      # from binary
$x = Math::BigInt->from_bin('1.101p+3');  # from binary
$x = Math::BigInt->from_bytes($bytes);    # from byte string
$x = Math::BigInt->from_base('why', 36);  # from any base
$x = Math::BigInt->from_base_num([1, 0], 2);  # from any base
$x = Math::BigInt->from_ieee754($b, $fmt);    # from IEEE-754 bytes
$x = Math::BigInt->from_fp80($b);         # from x86 80-bit
$x = Math::BigInt->bzero();               # create a +0
$x = Math::BigInt->bone();                # create a +1
$x = Math::BigInt->bone('-');             # create a -1
$x = Math::BigInt->binf();                # create a +inf
$x = Math::BigInt->binf('-');             # create a -inf
$x = Math::BigInt->bnan();                # create a Not-A-Number
$x = Math::BigInt->bpi();                 # returns pi

$y = $x->copy();        # make a copy (unlike $y = $x)
$y = $x->as_int();      # return as a Math::BigInt
$y = $x->as_float();    # return as a Math::BigFloat
$y = $x->as_rat();      # return as a Math::BigRat

# Boolean methods (these don't modify the invocand)

$x->is_zero();          # true if $x is 0
$x->is_one();           # true if $x is +1
$x->is_one("+");        # true if $x is +1
$x->is_one("-");        # true if $x is -1
$x->is_inf();           # true if $x is +inf or -inf
$x->is_inf("+");        # true if $x is +inf
$x->is_inf("-");        # true if $x is -inf
$x->is_nan();           # true if $x is NaN

$x->is_finite();        # true if -inf < $x < inf
$x->is_positive();      # true if $x > 0
$x->is_pos();           # true if $x > 0
$x->is_negative();      # true if $x < 0
```

(continues on next page)

(continued from previous page)

```
$x->is_neg();           # true if $x < 0
$x->is_non_positive()   # true if $x <= 0
$x->is_non_negative()   # true if $x >= 0

$x->is_odd();           # true if $x is odd
$x->is_even();          # true if $x is even
$x->is_int();           # true if $x is an integer

# Comparison methods (these don't modify the invocand)

$x->bcmp($y);           # compare numbers (undef, < 0, == 0, > 0)
$x->bacmp($y);          # compare abs values (undef, < 0, == 0, > 0)
$x->beq($y);            # true if $x == $y
$x->bne($y);            # true if $x != $y
$x->blt($y);            # true if $x < $y
$x->ble($y);            # true if $x <= $y
$x->bgt($y);            # true if $x > $y
$x->bge($y);            # true if $x >= $y

# Arithmetic methods (these modify the invocand)

$x->bneg();             # negation
$x->babs();             # absolute value
$x->bsgn();             # sign function (-1, 0, 1, or NaN)
$x->bdigitsum();        # sum of decimal digits
$x->binc();             # increment $x by 1
$x->bdec();             # decrement $x by 1
$x->badd($y);           # addition (add $y to $x)
$x->bsub($y);           # subtraction (subtract $y from $x)
$x->bmul($y);           # multiplication (multiply $x by $y)
$x->bmuladd($y, $z);    # $x = $x * $y + $z
$x->bdiv($y);           # division (floored)
$x->bmod($y);           # modulus (x % y)
$x->bmodinv($mod);      # modular multiplicative inverse
$x->bmodpow($y, $mod);  # modular exponentiation (($x ** $y) % $mod)
$x->btdiv($y);          # division (truncated), set $x to quotient
$x->btmod($y);          # modulus (truncated)
$x->binv()              # inverse (1/$x)
$x->bpow($y);           # power of arguments (x ** y)
$x->blog();             # logarithm of $x to base e (Euler's number)
$x->blog($base);        # logarithm of $x to base $base (e.g., base␣
↪2)
$x->bexp();             # calculate e ** $x where e is Euler's␣
↪number
$x->bilog2();           # log2($x) rounded down to nearest int
$x->bilog10();          # log10($x) rounded down to nearest int
$x->bclog2();           # log2($x) rounded up to nearest int
$x->bclog10();          # log10($x) rounded up to nearest int
$x->bnok($y);           # combinations (binomial coefficient n over␣
```

(continues on next page)

(continued from previous page)

```
↪k)
$x->bperm($y);          # permutations
$x->buparrow($n, $y);   # Knuth's up-arrow notation
$x->bhyperop($n, $y);   # n'th hyperoprator
$x->backermann($y);     # the Ackermann function
$x->bsin();             # sine
$x->bcos();             # cosine
$x->batan();            # inverse tangent
$x->batan2($y);         # two-argument inverse tangent
$x->bsqrt();            # calculate square root
$x->broot($y);          # $y'th root of $x (e.g. $y == 3 => cubic␣
↪root)
$x->bfac();             # factorial of $x (1*2*3*4*..$x)
$x->bdfac();            # double factorial of $x ($x*($x-2)*($x-4)*.
↪..)
$x->btfac();            # triple factorial of $x ($x*($x-3)*($x-6)*.
↪..)
$x->bmfac($k);          # $k'th multi-factorial of $x ($x*($x-$k)*..
↪.)
$x->bfib($k);           # $k'th Fibonacci number
$x->blucas($k);         # $k'th Lucas number

$x->blsft($n);          # left shift $n places in base 2
$x->blsft($n, $b);      # left shift $n places in base $b
$x->brsft($n);          # right shift $n places in base 2
$x->brsft($n, $b);      # right shift $n places in base $b

# Bitwise methods (these modify the invocand)

$x->bblsft($y);         # bitwise left shift
$x->bbrsft($y);         # bitwise right shift
$x->band($y);           # bitwise and
$x->bior($y);           # bitwise inclusive or
$x->bxor($y);           # bitwise exclusive or
$x->bnot();             # bitwise not (two's complement)

# Rounding methods (these modify the invocand)

$x->round($A, $P, $R);  # round to accuracy or precision using
                        #   rounding mode $R
$x->bround($n);         # accuracy: preserve $n digits
$x->bfround($n);        # $n > 0: round to $nth digit left of dec.␣
↪point
                        # $n < 0: round to $nth digit right of dec.␣
↪point
$x->bfloor();           # round towards minus infinity
$x->bceil();            # round towards plus infinity
$x->bint();             # round towards zero
```

(continues on next page)

(continued from previous page)

```
# Other mathematical methods (these don't modify the invocand)

$x->bgcd($y);           # greatest common divisor
$x->blcm($y);           # least common multiple

# Object property methods (these don't modify the invocand)

$x->sign();             # the sign, either +, - or NaN
$x->digit($n);          # the nth digit, counting from the right
$x->digit(-$n);         # the nth digit, counting from the left
$x->digitsum();         # sum of decimal digits
$x->length();           # return number of digits in number
$x->mantissa();         # return (signed) mantissa as a Math::BigInt
$x->exponent();         # return exponent as a Math::BigInt
$x->parts();            # return (mantissa,exponent) as a␣
 ↪Math::BigInt
$x->sparts();           # mantissa and exponent (as integers)
$x->nparts();           # mantissa and exponent (normalised)
$x->eparts();           # mantissa and exponent (engineering␣
 ↪notation)
$x->dparts();           # integer and fraction part
$x->fparts();           # numerator and denominator
$x->numerator();        # numerator
$x->denominator();      # denominator

# Conversion methods (these don't modify the invocand)

$x->bstr();             # decimal notation (possibly zero padded)
$x->bsstr();            # string in scientific notation with␣
 ↪integers
$x->bnstr();            # string in normalized notation
$x->bestr();            # string in engineering notation
$x->bdstr();            # string in decimal notation (no padding)
$x->bfstr();            # string in fractional notation

$x->to_hex();           # as signed hexadecimal string
$x->to_bin();           # as signed binary string
$x->to_oct();           # as signed octal string
$x->to_bytes();         # as byte string
$x->to_base($b);        # as string in any base
$x->to_base_num($b);    # as array of integers in any base
$x->to_ieee754($fmt);   # to bytes encoded according to IEEE 754-
 ↪2008
$x->to_fp80();          # encode value in x86 80-bit format

$x->as_hex();           # as signed hexadecimal string with "0x"␣
 ↪prefix
$x->as_bin();           # as signed binary string with "0b" prefix
$x->as_oct();           # as signed octal string with "0" prefix
```

(continues on next page)

(continued from previous page)

```
# Other conversion methods (these don't modify the invocand)

$x->numify();           # return as scalar (might overflow or␣
↪underflow)
```

DESCRIPTION

Math::BigInt provides support for arbitrary precision integers. Overloading is also provided for Perl operators.

Input

Input values to these routines may be any scalar number or string that looks like a number and represents an integer. Anything that is accepted by Perl as a literal numeric constant should be accepted by this module, except that finite non-integers return NaN.

- Leading and trailing whitespace is ignored.
- Leading zeros are ignored, except for floating point numbers with a binary exponent, in which case the number is interpreted as an octal floating point number. For example, “01.4p+0” gives 1.5, “00.4p+0” gives 0.5, but “0.4p+0” gives a NaN. And while “0377” gives 255, “0377p0” gives 255.
- If the string has a “0x” or “0X” prefix, it is interpreted as a hexadecimal number.
- If the string has a “0o” or “0O” prefix, it is interpreted as an octal number. A floating point literal with a “0” prefix is also interpreted as an octal number.
- If the string has a “0b” or “0B” prefix, it is interpreted as a binary number.
- Underline characters are allowed in the same way as they are allowed in literal numerical constants.
- If the string can not be interpreted, or does not represent a finite integer, NaN is returned.
- For hexadecimal, octal, and binary floating point numbers, the exponent must be separated from the significand (mantissa) by the letter “p” or “P”, not “e” or “E” as with decimal numbers.

Some examples of valid string input

```
Input string                Resulting value

123                         123
1.23e2                      123
12300e-2                    123

67_538_754                  67538754
-4_5_6.7_8_9e+0_1_0         -4567890000000

0x13a                       314
```

(continues on next page)

(continued from previous page)

```
0x13ap0                         314
0x1.3ap+8                       314
0x0.00013ap+24                  314
0x13a000p-12                    314

0o472                           314
0o1.164p+8                      314
0o0.0001164p+20                 314
0o1164000p-10                   314

0472                            472     Note!
01.164p+8                       314
00.0001164p+20                  314
01164000p-10                    314

0b100111010                     314
0b1.0011101p+8                  314
0b0.00010011101p+12             314
0b100111010000p-3               314
```

Input given as scalar numbers might lose precision. Quote your input to ensure that no digits are lost:

```
$x = Math::BigInt->new( 56789012345678901234 );   # bad
$x = Math::BigInt->new('56789012345678901234');   # good
```

Currently, `Math::BigInt-new()>` (no input argument) and `Math::BigInt-new("")>` return 0. This might change in the future, so always use the following explicit forms to get a zero:

```
$zero = Math::BigInt->bzero();
```

Output

Output values are usually Math::BigInt objects.

Boolean operators "is_zero()", "is_one()", "is_inf()", etc. return true or false.

Comparison operators "bcmp()" and "bacmp()") return -1, 0, 1, or undef.

METHODS

Configuration methods

Each of the methods below (except "config()", "accuracy()" and "precision()") accepts three additional parameters. These arguments `$A`, `$P` and `$R` are `accuracy`, `precision` and `round_mode`. Please see the section about "ACCURACY AND PRECISION" for more information.

Setting a class variable effects all object instance that are created afterwards.

- accuracy()

```
Math::BigInt->accuracy(5);      # set class accuracy
$x->accuracy(5);                # set instance accuracy

$A = Math::BigInt->accuracy();  # get class accuracy
$A = $x->accuracy();            # get instance accuracy
```

Set or get the accuracy, i.e., the number of significant digits. The accuracy must be an integer. If the accuracy is set to `undef`, no rounding is done.

Alternatively, one can round the results explicitly using one of "round()", "bround()" or "bfround()" or by passing the desired accuracy to the method as an additional parameter:

```
my $x = Math::BigInt->new(30000);
my $y = Math::BigInt->new(7);
print scalar $x->copy()->bdiv($y, 2);               # prints␣
↪4300
print scalar $x->copy()->bdiv($y)->bround(2);       # prints␣
↪4300
```

Please see the section about "ACCURACY AND PRECISION" for further details.

```
$y = Math::BigInt->new(1234567);    # $y is not rounded
Math::BigInt->accuracy(4);          # set class accuracy to 4
$x = Math::BigInt->new(1234567);    # $x is rounded␣
↪automatically
print "$x $y";                      # prints "1235000 1234567"

print $x->accuracy();       # prints "4"
print $y->accuracy();       # also prints "4", since
                            #   class accuracy is 4

Math::BigInt->accuracy(5);  # set class accuracy to 5
print $x->accuracy();       # prints "4", since instance
                            #   accuracy is 4
print $y->accuracy();       # prints "5", since no instance
                            #   accuracy, and class accuracy is␣
↪5
```

Note: Each class has it's own globals separated from Math::BigInt, but it is possible to subclass Math::BigInt and make the globals of the subclass aliases to the ones from Math::BigInt.

- precision()

```
Math::BigInt->precision(-2);     # set class precision
$x->precision(-2);               # set instance precision

$P = Math::BigInt->precision();  # get class precision
$P = $x->precision();            # get instance precision
```

Set or get the precision, i.e., the place to round relative to the decimal point. The precision must be a integer. Setting the precision to $P means that each number is

rounded up or down, depending on the rounding mode, to the nearest multiple of 10**$P. If the precision is set to `undef`, no rounding is done.

You might want to use "accuracy()" instead. With "accuracy()" you set the number of digits each result should have, with "precision()" you set the place where to round.

Please see the section about "ACCURACY AND PRECISION" for further details.

```
$y = Math::BigInt->new(1234567);    # $y is not rounded
Math::BigInt->precision(4);         # set class precision to 4
$x = Math::BigInt->new(1234567);    # $x is rounded
 ↪automatically
print $x;                           # prints "1230000"
```

Note: Each class has its own globals separated from Math::BigInt, but it is possible to subclass Math::BigInt and make the globals of the subclass aliases to the ones from Math::BigInt.

- round_mode()

  Set/get the rounding mode.

- div_scale()

  Set/get the fallback accuracy. This is the accuracy used when neither accuracy nor precision is set explicitly. It is used when a computation might otherwise attempt to return an infinite number of digits.

- trap_inf()

  Set/get the value determining whether infinities should cause a fatal error or not.

- trap_nan()

  Set/get the value determining whether NaNs should cause a fatal error or not.

- upgrade()

  Set/get the class for upgrading. When a computation might result in a non-integer, the operands are upgraded to this class. This is used for instance by *bignum*. The default is `undef`, i.e., no upgrading.

```
# with no upgrading
$x = Math::BigInt->new(12);
$y = Math::BigInt->new(5);
print $x / $y, "\n";                # 2 as a Math::BigInt

# with upgrading to Math::BigFloat
Math::BigInt -> upgrade("Math::BigFloat");
print $x / $y, "\n";                # 2.4 as a Math::BigFloat

# with upgrading to Math::BigRat (after loading Math::BigRat)
Math::BigInt -> upgrade("Math::BigRat");
print $x / $y, "\n";                # 12/5 as a Math::BigRat
```

- downgrade()

Set/get the class for downgrading. The default is undef, i.e., no downgrading. Downgrading is not done by Math::BigInt.

- modify()

```
$x->modify('bpowd');
```

This method returns 0 if the object can be modified with the given operation, or 1 if not.

This is used for instance by Math::BigInt::Constant.

- config()

```
Math::BigInt->config("trap_nan" => 1);      # set
$accu = Math::BigInt->config("accuracy");   # get
```

Set or get class variables. Read-only parameters are marked as RO. Read-write parameters are marked as RW. The following parameters are supported.

```
Parameter       RO/RW   Description
                        Example
============================================================
lib             RO      Name of the math backend library
                        Math::BigInt::Calc
lib_version     RO      Version of the math backend library
                        0.30
class           RO      The class of config you just called
                        Math::BigRat
version         RO      version number of the class you used
                        0.10
upgrade         RW      To which class numbers are upgraded
                        undef
downgrade       RW      To which class numbers are downgraded
                        undef
precision       RW      Global precision
                        undef
accuracy        RW      Global accuracy
                        undef
round_mode      RW      Global round mode
                        even
div_scale       RW      Fallback accuracy for division etc.
                        40
trap_nan        RW      Trap NaNs
                        undef
trap_inf        RW      Trap +inf/-inf
                        undef
```

Constructor methods

- new()

```
$x = Math::BigInt->new($str,$A,$P,$R);
```

Creates a new Math::BigInt object from a scalar or another Math::BigInt object. The input is accepted as decimal, hexadecimal (with leading '0x'), octal (with leading ('0o') or binary (with leading '0b').

See "Input" for more info on accepted input formats.

- from_dec()

```
$x = Math::BigInt->from_dec("314159");    # input is decimal
```

Interpret input as a decimal. It is equivalent to "new()", but does not accept anything but strings representing finite, decimal numbers.

- from_hex()

```
$x = Math::BigInt->from_hex("0xcafe");    # input is hexadecimal
```

Interpret input as a hexadecimal string. A "0x" or "x" prefix is optional. A single underscore character may be placed right after the prefix, if present, or between any two digits. If the input is invalid, a NaN is returned.

- from_oct()

```
$x = Math::BigInt->from_oct("0775");      # input is octal
```

Interpret the input as an octal string and return the corresponding value. A "0" (zero) prefix is optional. A single underscore character may be placed right after the prefix, if present, or between any two digits. If the input is invalid, a NaN is returned.

- from_bin()

```
$x = Math::BigInt->from_bin("0b10011");   # input is binary
```

Interpret the input as a binary string. A "0b" or "b" prefix is optional. A single underscore character may be placed right after the prefix, if present, or between any two digits. If the input is invalid, a NaN is returned.

- from_bytes()

```
$x = Math::BigInt->from_bytes("\xf3\x6b");  # $x = 62315
```

Interpret the input as a byte string, assuming big endian byte order. The output is always a non-negative, finite integer.

In some special cases, "from_bytes()" matches the conversion done by unpack():

```
$b = "\x4e";                              # one char byte string
$x = Math::BigInt->from_bytes($b);        # = 78
$y = unpack "C", $b;                      # ditto, but scalar
```

(continues on next page)

(continued from previous page)

```
$b = "\xf3\x6b";                          # two char byte string
$x = Math::BigInt->from_bytes($b);        # = 62315
$y = unpack "S>", $b;                     # ditto, but scalar

$b = "\x2d\xe0\x49\xad";                  # four char byte string
$x = Math::BigInt->from_bytes($b);        # = 769673645
$y = unpack "L>", $b;                     # ditto, but scalar

$b = "\x2d\xe0\x49\xad\x2d\xe0\x49\xad"; # eight char byte␣
↪string
$x = Math::BigInt->from_bytes($b);        # = 3305723134637787565
$y = unpack "Q>", $b;                     # ditto, but scalar
```

- from_ieee754()

```
# set $x to 314159
$x = Math::BigInt -> from_ieee754("40490fdb", "binary32");
```

Interpret the input as a value encoded as described in IEEE754-2008. NaN is returned if the value is neither +/-infinity nor an integer.

See *“from_ieee754()” in Math::BigFloat*.

- from_fp80()

```
# set $x to 314159
$x = Math::BigInt -> from_fp80("40119965e00000000000");
```

Interpret the input as a value encoded in the x86 extended-precision 80-bit format.

See *“from_fp80()” in Math::BigFloat*.

- from_base()

Given a string, a base, and an optional collation sequence, interpret the string as a number in the given base. The collation sequence describes the value of each character in the string.

If a collation sequence is not given, a default collation sequence is used. If the base is less than or equal to 36, the collation sequence is the string consisting of the 36 characters “0” to “9” and “A” to “Z”. In this case, the letter case in the input is ignored. If the base is greater than 36, and smaller than or equal to 62, the collation sequence is the string consisting of the 62 characters “0” to “9”, “A” to “Z”, and “a” to “z”. A base larger than 62 requires the collation sequence to be specified explicitly.

These examples show standard binary, octal, and hexadecimal conversion. All cases return 250.

```
$x = Math::BigInt->from_base("11111010", 2);
$x = Math::BigInt->from_base("372", 8);
$x = Math::BigInt->from_base("fa", 16);
```

When the base is less than or equal to 36, and no collation sequence is given, the letter case is ignored, so both of these also return 250:

```
$x = Math::BigInt->from_base("6Y", 16);
$x = Math::BigInt->from_base("6y", 16);
```

When the base greater than 36, and no collation sequence is given, the default collation sequence contains both uppercase and lowercase letters, so the letter case in the input is not ignored:

```
$x = Math::BigInt->from_base("6S", 37);         # $x is 250
$x = Math::BigInt->from_base("6s", 37);         # $x is 276
$x = Math::BigInt->from_base("121", 3);         # $x is 16
$x = Math::BigInt->from_base("XYZ", 36);        # $x is 44027
$x = Math::BigInt->from_base("Why", 42);        # $x is 58314
```

The collation sequence can be any set of unique characters. These two cases are equivalent

```
$x = Math::BigInt->from_base("100", 2, "01");   # $x is 4
$x = Math::BigInt->from_base("|--", 2, "-|");   # $x is 4
```

- from_base_num()

  Returns a new Math::BigInt object given an array of values and a base. This method is equivalent to "from_base()", but works on numbers in an array rather than characters in a string. Unlike "from_base()", all input values may be arbitrarily large.

```
$x = Math::BigInt->from_base_num([1, 1, 0, 1], 2)     # $x is 13
$x = Math::BigInt->from_base_num([3, 125, 39], 128)   # $x is␣
↪65191
```

- bzero()

```
$x = Math::BigInt->bzero();
$x->bzero();
```

  Returns a new Math::BigInt object representing zero. If used as an instance method, assigns the value to the invocand.

- bone()

```
$x = Math::BigInt->bone();            # +1
$x = Math::BigInt->bone("+");         # +1
$x = Math::BigInt->bone("-");         # -1
$x->bone();                           # +1
$x->bone("+");                        # +1
$x->bone('-');                        # -1
```

  Creates a new Math::BigInt object representing one. The optional argument is either '-' or '+', indicating whether you want plus one or minus one. If used as an instance method, assigns the value to the invocand.

- binf()

```
$x = Math::BigInt->binf($sign);
```

Creates a new Math::BigInt object representing infinity. The optional argument is either '-' or '+', indicating whether you want infinity or minus infinity. If used as an instance method, assigns the value to the invocand.

```
$x->binf();
$x->binf('-');
```

- bnan()

```
$x = Math::BigInt->bnan();
```

Creates a new Math::BigInt object representing NaN (Not A Number). If used as an instance method, assigns the value to the invocand.

```
$x->bnan();
```

- bpi()

```
$x = Math::BigInt->bpi(100);        # 3
$x->bpi(100);                       # 3
```

Creates a new Math::BigInt object representing PI. If used as an instance method, assigns the value to the invocand. With Math::BigInt this always returns 3.

If upgrading is in effect, returns PI, rounded to N digits with the current rounding mode:

```
use Math::BigFloat;
use Math::BigInt upgrade => "Math::BigFloat";
print Math::BigInt->bpi(3), "\n";           # 3.14
print Math::BigInt->bpi(100), "\n";         # 3.1415....
```

- copy()

```
$x->copy();         # make a true copy of $x (unlike $y = $x)
```

- as_int()

```
$y = $x -> as_int();        # $y is a Math::BigInt
```

Returns $x as a Math::BigInt object regardless of upgrading and downgrading. If $x is finite, but not an integer, $x is truncated.

- as_rat()

```
$y = $x -> as_rat();        # $y is a Math::BigRat
```

Returns $x a Math::BigRat object regardless of upgrading and downgrading. The invocand is not modified.

- as_float()

```
$y = $x -> as_float();      # $y is a Math::BigFloat
```

Returns $x a Math::BigFloat object regardless of upgrading and downgrading. The invocand is not modified.

Boolean methods

None of these methods modify the invocand object.

- is_zero()

```
$x->is_zero();              # true if $x is 0
```

Returns true if the invocand is zero and false otherwise.

- is_one()

```
$x->is_one();               # true if $x is +1
$x->is_one("+");            # ditto
$x->is_one("-");            # true if $x is -1
```

Returns true if the invocand is one and false otherwise.

- is_finite()

```
$x->is_finite();    # true if $x is not +inf, -inf or NaN
```

Returns true if the invocand is a finite number, i.e., it is neither +inf, -inf, nor NaN.

- is_inf()

```
$x->is_inf();               # true if $x is +inf or -inf
$x->is_inf("+");            # true if $x is +inf
$x->is_inf("-");            # true if $x is -inf
```

Returns true if the invocand is infinite and false otherwise.

- is_nan()

```
$x->is_nan();               # true if $x is NaN
```

- is_positive()
- is_pos()

```
$x->is_positive();          # true if > 0
$x->is_pos();               # ditto
```

Returns true if the invocand is positive and false otherwise. A NaN is neither positive nor negative.

- is_negative()
- is_neg()

```
$x->is_negative();          # true if < 0
$x->is_neg();               # ditto
```

Returns true if the invocand is negative and false otherwise. A NaN is neither positive nor negative.

- is_non_positive()

```
$x->is_non_positive();      # true if <= 0
```

Returns true if the invocand is negative or zero.

- is_non_negative()

```
$x->is_non_negative();      # true if >= 0
```

Returns true if the invocand is positive or zero.

- is_odd()

```
$x->is_odd();               # true if odd, false for even
```

Returns true if the invocand is odd and false otherwise. NaN, +inf, and -inf are neither odd nor even.

- is_even()

```
$x->is_even();              # true if $x is even
```

Returns true if the invocand is even and false otherwise. NaN, +inf, -inf are not integers and are neither odd nor even.

- is_int()

```
$x->is_int();               # true if $x is an integer
```

Returns true if the invocand is an integer and false otherwise. NaN, +inf, -inf are not integers.

Comparison methods

None of these methods modify the invocand object. Note that a NaN is neither less than, greater than, or equal to anything else, even a NaN.

- bcmp()

```
$x->bcmp($y);
```

Returns -1, 0, 1 depending on whether $x is less than, equal to, or grater than $y. Returns undef if any operand is a NaN.

- bacmp()

```
$x->bacmp($y);
```

Returns -1, 0, 1 depending on whether the absolute value of $x is less than, equal to, or grater than the absolute value of $y. Returns undef if any operand is a NaN.

- beq()

```
$x -> beq($y);
```

Returns true if and only if $x is equal to $y, and false otherwise.

- bne()

```
$x -> bne($y);
```

Returns true if and only if $x is not equal to $y, and false otherwise.

- blt()

```
$x -> blt($y);
```

Returns true if and only if $x is equal to $y, and false otherwise.

- ble()

```
$x -> ble($y);
```

Returns true if and only if $x is less than or equal to $y, and false otherwise.

- bgt()

```
$x -> bgt($y);
```

Returns true if and only if $x is greater than $y, and false otherwise.

- bge()

```
$x -> bge($y);
```

Returns true if and only if $x is greater than or equal to $y, and false otherwise.

Arithmetic methods

These methods modify the invocand object and returns it.

- bneg()

```
$x->bneg();
```

Negate the number, e.g. change the sign between '+' and '-', or between '+inf' and '-inf', respectively. Does nothing for NaN or zero.

- babs()

```
$x->babs();
```

Set the number to its absolute value, e.g. change the sign from '-' to '+' and from '-inf' to '+inf', respectively. Does nothing for NaN or positive numbers.

- bsgn()

```
$x->bsgn();
```

Signum function. Set the number to -1, 0, or 1, depending on whether the number is negative, zero, or positive, respectively. Does not modify NaNs.

- bnorm()

```
$x->bnorm();                    # normalize (no-op)
```

Normalize the number. This is a no-op and is provided only for backwards compatibility.

- binc()

```
$x->binc();                     # increment x by 1
```

- bdec()

```
$x->bdec();                     # decrement x by 1
```

- badd()

```
$x->badd($y);                   # addition (add $y to $x)
```

- bsub()

```
$x->bsub($y);                   # subtraction (subtract $y from $x)
```

- bmul()

```
$x->bmul($y);                   # multiplication (multiply $x by $y)
```

- bdiv()

```
$x->bdiv($y);                   # set $x to quotient
($q, $r) = $x->bdiv($y);        # also return remainder
```

The behaviour of “bdiv()” and “bmod()” is based on Perl’s % operator, which is the remainder after performing floored division.

Because of this, “bdiv()” and “bmod()” are aliases for “bfdiv()” and “bfmod()”, respectively.

- bmod()

```
$x->bmod($y);                   # modulus (x % y)
```

This is an alias for “bfmod()”.

- bfdiv()

```
$x->bfdiv($y);                  # return quotient
($q, $r) = $x->bfdiv($y);       # return quotient and remainder
```

Divides $x by $y by doing floored division (F-division), where the quotient is the floored (rounded towards negative infinity) quotient of the two operands. In list context, returns the quotient and the remainder. In scalar context, only the quotient is returned.

```
$q = floor($x / $y)          # quotient
$r = $x - $q * $y            # remainder
```

With F-division, the remainder is either zero or has the same sign as the divisor.

```
 7 /  4 => ( 1,  3)
-7 /  4 => (-2,  1)
-7 / -4 => ( 1, -3)
 7 / -4 => (-2, -1)
```

The behavior of the overloaded operator % agrees with the behavior of Perl's built-in % operator (as documented in the perlop manpage), and the equation

```
$x == ($x / $y) * $y + ($x % $y)
```

holds true for any finite $x and finite, non-zero $y.

Perl's "use integer" might change the behaviour of % and / for scalars. This is because under 'use integer' Perl does what the underlying C library thinks is right, and this varies. However, "use integer" does not change the way things are done with Math::BigInt objects.

- bfmod()

```
$x->bfmod($y);               # floored modulus (x % y)
```

Returns $x modulo $y, i.e., the remainder after floored division (F-division). This method is like Perl's % operator. See "bfdiv()".

- btdiv()

```
$x->btdiv($y);               # divide, set $x to quotient
```

Divides $x by $y by doing truncated division (T-division), where quotient is the truncated (rouneded towards zero) quotient of the two operands. In list context, returns the quotient and the remainder. The remainder is either zero or has the same sign as the first operand. In scalar context, only the quotient is returned.

- btmod()

```
$x->btmod($y);               # modulus
```

Returns the remainer after truncated division (T-division). See "btdiv()".

- binv()

```
$x->binv();
```

Invert the value of $x, i.e., compute 1/$x.

- bsqrt()

```
$x->bsqrt();                    # calculate square root
```

Returns the square root truncated to an integer.

If you want a better approximation of the square root, then use:

```
$x = Math::BigFloat->new(12);
Math::BigFloat->precision(0);
Math::BigFloat->round_mode("even");
print $x->copy->bsqrt(),"\n";           # 4

Math::BigFloat->precision(2);
print $x->bsqrt(),"\n";                 # 3.46
print $x->bsqrt(3),"\n";                # 3.464
```

- bpow()

```
$x->bpow($y);                   # power of arguments (x ** y)
```

Returns $x raised to the power of $y. The first two modifies $x, the last one doesn't:

```
print $x->bpow($i),"\n";        # modifies $x
print $x **= $i,"\n";           # ditto
print $x ** $i,"\n";            # leaves $x alone
```

The form `$x **= $y` is faster than `$x = $x ** $y;`, though.

- broot()

```
$x->broot($N);
```

Calculates the $N'th root of `$x`.

- bmuladd()

```
$x->bmuladd($y,$z);
```

Multiply $x by $y, and then add $z to the result,

This method was added in v1.88 of Math::BigInt.

- bmodpow()

```
$num->bmodpow($exp,$mod);           # modular exponentiation
                                    # ($num**$exp % $mod)
```

Returns the value of `$num` taken to the power `$exp` in the modulus `$mod` using binary exponentiation. `bmodpow` is far superior to writing

```
$num ** $exp % $mod
```

because it is much faster - it reduces internal variables into the modulus whenever possible, so it operates on smaller numbers.

`bmodpow` also supports negative exponents.

```
bmodpow($num, -1, $mod)
```

is exactly equivalent to

```
bmodinv($num, $mod)
```

- bmodinv()

```
$x->bmodinv($mod);          # modular multiplicative inverse
```

Returns the multiplicative inverse of $x modulo $mod. If

```
$y = $x -> copy() -> bmodinv($mod)
```

then $y is the number closest to zero, and with the same sign as $mod, satisfying

```
($x * $y) % $mod = 1 % $mod
```

If $x and $y are non-zero, they must be relative primes, i.e., bgcd($y, $mod)==1. 'NaN' is returned when no modular multiplicative inverse exists.

- blog()

```
$x->blog($base, $accuracy);         # logarithm of x to the
↪base $base
```

If $base is not defined, Euler's number (e) is used:

```
print $x->blog(undef, 100);         # log(x) to 100 digits
```

- bexp()

```
$x->bexp($accuracy);                # calculate e ** X
```

Calculates the expression e ** $x where e is Euler's number.

This method was added in v1.82 of Math::BigInt (April 2007).

See also "blog()".

- bilog2()

Base 2 logarithm rounded down towards the nearest integer.

```
$x->bilog2();               # int(log2(x)) = int(log(x)/log(2))
```

In list context a second argument is returned. This is 1 if the result is exact, i.e., the input is an exact power of 2, and 0 otherwise.

- bilog10()

Base 10 logarithm rounded down towards the nearest integer.

```
$x->bilog10();              # int(log10(x)) = int(log(x)/
↪log(10))
```

In list context a second argument is returned. This is 1 if the result is exact, i.e., the input is an exact power of 10, and 0 otherwise.

- bclog2()

  Base 2 logarithm rounded up towards the nearest integer.

```
$x->bclog2();               # ceil(log2(x)) = ceil(log(x)/
 ↪log(2))
```

  In list context a second argument is returned. This is 1 if the result is exact, i.e., the input is an exact power of 2, and 0 otherwise.

- bclog10()

  Base 10 logarithm rounded up towards the nearest integer.

```
$x->bclog10();              # ceil(log10(x)) = ceil(log(x)/
 ↪log(10))
```

  In list context a second argument is returned. This is 1 if the result is exact, i.e., the input is an exact power of 10, and 0 otherwise.

- bnok()

  Combinations.

```
$n->bnok($k);               # binomial coefficient n over k
```

  Calculates the binomial coefficient n over k, also called the "choose" function, which is the number of ways to choose a sample of k elements from a set of n distinct objects where order does not matter and replacements are not allowed. The result is equivalent to

```
          / n \      n!
C(n, k) = |   | = --------  where 0 <= k <= n
          \ k /   k!(n-k)!
```

  when n and k are non-negative. This method implements the full Kronenburg extension (Kronenburg, M.J. "The Binomial Coefficient for Negative Arguments." 18 May 2011. http://arxiv.org/abs/1105.3689/) illustrated by the following pseudo-code:

```
if n >= 0 and k >= 0:
    return binomial(n, k)
if k >= 0:
    return (-1)^k*binomial(-n+k-1, k)
if k <= n:
    return (-1)^(n-k)*binomial(-k-1, n-k)
else
    return 0
```

  The behaviour is identical to the behaviour of the Maple and Mathematica function for negative integers n, k.

- bperm()

  Permutations

```
$n->bperm($k);
```

Calculates the number of ways to choose a sample of k elements from a set of n distinct objects where order does matter and replacements are not allowed.

```
            n!
P(n, k) = ------  where 0 <= k <= n
          (n-k)!
```

- bhyperop()
- hyperop()

```
$a -> bhyperop($n, $b);          # modifies $a
$x = $a -> hyperop($n, $b);      # does not modify $a
```

H_n(a, b) = a[n]b is the _n_th hyperoperator,

```
n = 0 : succession (b + 1)
n = 1 : addition (a + b)
n = 2 : multiplication (a * b)
n = 3 : exponentiation (a ** b)
n = 4 : tetration (a ** a ** ... ** a) (b occurrences of a)
...

                    / b+1                       if n = 0
                    | a                         if n = 1 and b = 0
H_n(a, b) = a[n]b = | 0                         if n = 2 and b = 0
                    | 1                         if n >= 3 and b =␣
 ↪0
                    \ H_(n-1)(a, H_n(a, b-1)) otherwise
```

Note that the result can be a very large number, even for small operands. Also note that the backend library `Math::BigInt::GMP` silently returns the incorrect result when the numbers are larger than it can handle. It is better to use `Math::BigInt::GMPz` or `Math::BigInt::Pari`; they throw an error if they can't handle the number.

See also "buparrow()", https://en.wikipedia.org/wiki/Hyperoperation.

- buparrow()
- uparrow()

```
$a -> buparrow($n, $b);          # modifies $a
$x = $a -> uparrow($n, $b);      # does not modify $a
```

This method implements Knuth's up-arrow notation, where $n is a non-negative integer representing the number of up-arrows. $n = 0 gives multiplication, $n = 1 gives exponentiation, $n = 2 gives tetration, $n = 3 gives hexation etc. The following illustrates the relation between the first values of $n.

The "buparrow()" method is equivalent to the "bhyperop()" method with an offset of two. The following two give the same result:

```
$x -> buparrow($n, $b);
$x -> bhyperop($n + 2, $b);
```

See also "bhyperop()", https://en.wikipedia.org/wiki/Knuth%27s_up-arrow_notation.

- backermann()
- ackermann()

```
$m -> backermann($n);           # modifies $a
$x = $m -> ackermann($n);       # does not modify $a
```

This method implements the Ackermann function:

```
          / n + 1               if m = 0
A(m, n) = | A(m-1, 1)           if m > 0 and n = 0
          \ A(m-1, A(m, n-1))   if m > 0 and n > 0
```

Its value grows rapidly, even for small inputs. For example, A(4, 2) is an integer of 19729 decimal digits.

See https://en.wikipedia.org/wiki/Ackermann_function

- bsin()

```
my $x = Math::BigInt->new(1);
print $x->bsin(100), "\n";
```

Calculate the sine of $x, modifying $x in place.

In Math::BigInt, unless upgrading is in effect, the result is truncated to an integer.

This method was added in v1.87 of Math::BigInt (June 2007).

- bcos()

```
my $x = Math::BigInt->new(1);
print $x->bcos(100), "\n";
```

Calculate the cosine of $x, modifying $x in place.

In Math::BigInt, unless upgrading is in effect, the result is truncated to an integer.

This method was added in v1.87 of Math::BigInt (June 2007).

- batan()

```
my $x = Math::BigFloat->new(0.5);
print $x->batan(100), "\n";
```

Calculate the arcus tangens of $x, modifying $x in place.

In Math::BigInt, unless upgrading is in effect, the result is truncated to an integer.

This method was added in v1.87 of Math::BigInt (June 2007).

- batan2()

```
my $x = Math::BigInt->new(1);
my $y = Math::BigInt->new(1);
print $y->batan2($x), "\n";
```

Calculate the arcus tangens of $y divided by $x, modifying $y in place.

In Math::BigInt, unless upgrading is in effect, the result is truncated to an integer.

This method was added in v1.87 of Math::BigInt (June 2007).

- bfac()

```
$x->bfac();               # factorial of $x
```

Returns the factorial of $x, i.e., $x*($x-1)*($x-2)*...*2*1, the product of all positive integers up to and including $x. $x must be > -1. The factorial of N is commonly written as N!, or N!1, when using the multifactorial notation.

- bdfac()

```
$x->bdfac();                # double factorial of $x
```

Returns the double factorial of $x, i.e., $x*($x-2)*($x-4)*... $x must be > -2. The double factorial of N is commonly written as N!!, or N!2, when using the multifactorial notation.

- btfac()

```
$x->btfac();               # triple factorial of $x
```

Returns the triple factorial of $x, i.e., $x*($x-3)*($x-6)*... $x must be > -3. The triple factorial of N is commonly written as N!!!, or N!3, when using the multifactorial notation.

- bmfac()

```
$x->bmfac($k);           # $k'th multifactorial of $x
```

Returns the multi-factorial of $x, i.e., $x*($x-$k)*($x-2*$k)*... $x must be > -$k. The multi-factorial of N is commonly written as N!K.

- bfib()

```
$F = $n->bfib();            # a single Fibonacci number
@F = $n->bfib();            # a list of Fibonacci numbers
```

In scalar context, returns a single Fibonacci number. In list context, returns a list of Fibonacci numbers. The invocand is the last element in the output.

The Fibonacci sequence is defined by

```
F(0) = 0
F(1) = 1
F(n) = F(n-1) + F(n-2)
```

In list context, F(0) and F(n) is the first and last number in the output, respectively. For example, if $n is 12, then `@F = $n->bfib()` returns the following values, F(0) to F(12):

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144
```

The sequence can also be extended to negative index n using the re-arranged recurrence relation

```
F(n-2) = F(n) - F(n-1)
```

giving the bidirectional sequence

```
  n  -7  -6  -5  -4  -3  -2  -1   0   1   2   3   4   5   6   7
F(n)  13  -8   5  -3   2  -1   1   0   1   1   2   3   5   8  13
```

If $n is -12, the following values, F(0) to F(12), are returned:

```
0, 1, -1, 2, -3, 5, -8, 13, -21, 34, -55, 89, -144
```

- blucas()

```
$F = $n->blucas();          # a single Lucas number
@F = $n->blucas();          # a list of Lucas numbers
```

In scalar context, returns a single Lucas number. In list context, returns a list of Lucas numbers. The invocand is the last element in the output.

The Lucas sequence is defined by

```
L(0) = 2
L(1) = 1
L(n) = L(n-1) + L(n-2)
```

In list context, L(0) and L(n) is the first and last number in the output, respectively. For example, if $n is 12, then `@L = $n->blucas()` returns the following values, L(0) to L(12):

```
2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123, 199, 322
```

The sequence can also be extended to negative index n using the re-arranged recurrence relation

```
L(n-2) = L(n) - L(n-1)
```

giving the bidirectional sequence

```
  n  -7  -6  -5  -4  -3  -2  -1   0   1   2   3   4   5   6   7
L(n)  29 -18  11  -7   4  -3   1   2   1   3   4   7  11  18  29
```

If $n is -12, the following values, L(0) to L(-12), are returned:

```
2, 1, -3, 4, -7, 11, -18, 29, -47, 76, -123, 199, -322
```

- blsft()

  Left shift.

```
$x->blsft($n);               # left shift $n places in base 2
$x->blsft($n, $b);           # left shift $n places in base $b
```

  The latter is equivalent to

```
$x -> bmul($b -> copy() -> bpow($n));
```

- brsft()

  Right shift.

```
$x->brsft($n);               # right shift $n places in base 2
$x->brsft($n, $b);           # right shift $n places in base $b
```

  The latter is equivalent to

```
$x -> bdiv($b -> copy() -> bpow($n));
```

Bitwise methods

For all bitwise methods, the operands are truncated to integers, i.e., rounded towards zero, if necessary, before the method is applied. The bitwise methods never upgrade, and they always return an integer.

- bblsft()

  Bitwise left shift. This is equivalent to Perl's << operator.

```
$x -> bblsft($n);            # left shift $n places in base 2
```

  If $n is negative, the shifting is done in the opposite direction, so these two are equivalent for all $x and $n

```
$y = $x -> bblsft($n);
$y = $x -> bbrsft(-$n);
```

  and also equivalent to

```
$y = $x -> bmul(ref($x) -> new(2) -> bpow($n));   # if $n > 0
$y = $x -> bdiv(ref($x) -> new(2) -> bpow($n));   # if $n < 0
```

- bbrsft()

  Bitwise right shift. This is equivalent to Perl's >> operator.

```
$x -> bbrsft($n);            # right shift $n places in base 2
```

  If $n is negative, the shifting is done in the opposite direction, so these two are equivalent for all $x and $n

```
$y = $x -> bbrsft($n);
$y = $x -> bblsft(-$n);
```

and also equivalent to

```
$y = $x -> bdiv(ref($x) -> new(2) -> bpow($n));   # if $n > 0
$y = $x -> bmul(ref($x) -> new(2) -> bpow(-$n));  # if $n < 0
```

- band()

```
$x->band($y);                 # bitwise and
```

- bior()

```
$x->bior($y);                 # bitwise inclusive or
```

- bxor()

```
$x->bxor($y);                 # bitwise exclusive or
```

- bnot()

```
$x->bnot();                   # bitwise not (two's complement)
```

Two's complement (bitwise not). This is equivalent to, but faster than,

```
$x->binc()->bneg();
```

Rounding methods

- round()

```
$x->round($A,$P,$round_mode);
```

Round $x to accuracy $A or precision $P using the round mode $round_mode.

- bround()

```
$x->bround($N);               # accuracy: preserve $N digits
```

Rounds $x to an accuracy of $N digits.

- bfround()

```
$x->bfround($N);
```

Rounds to a multiple of 10**$N. Examples:

```
Input           N          Result

123456.123456   3          123500
123456.123456   2          123450
```

(continues on next page)

(continued from previous page)

```
123456.123456   -2          123456.12
123456.123456   -3          123456.123
```

- bfloor()

```
$x->bfloor();
```

Round $x towards minus infinity, i.e., set $x to the largest integer less than or equal to $x.

- bceil()

```
$x->bceil();
```

Round $x towards plus infinity, i.e., set $x to the smallest integer greater than or equal to $x.

- bint()

```
$x->bint();
```

Round $x towards zero.

Other mathematical methods

- bgcd()

```
$x -> bgcd($y);             # GCD of $x and $y
$x -> bgcd($y, $z, ...);    # GCD of $x, $y, $z, ...
```

Returns the greatest common divisor (GCD), which is the largest positive integer that divides each of the operands.

- blcm()

```
$x -> blcm($y);             # LCM of $x and $y
$x -> blcm($y, $z, ...);    # LCM of $x, $y, $z, ...
```

Returns the least common multiple (LCM).

Object property methods

- sign()

```
$x->sign();
```

Return the sign, of $x, meaning either +, -, -inf, +inf or NaN.

If you want $x to have a certain sign, use one of the following methods:

```
$x->babs();                 # '+'
$x->babs()->bneg();         # '-'
```

(continues on next page)

(continued from previous page)

```
$x->bnan();                    # 'NaN'
$x->binf();                    # '+inf'
$x->binf('-');                 # '-inf'
```

- digit()

```
$x->digit($n);       # return the nth digit, counting from right
```

If $n is negative, returns the digit counting from left.

- bdigitsum()

```
$x->bdigitsum();
```

Computes the sum of the base 10 digits and assigns the result to the invocand.

- digitsum()

```
$x->digitsum();
```

Computes the sum of the base 10 digits and returns it.

- length()

```
$x->length();
($xl, $fl) = $x->length();
```

Returns the number of digits in the decimal representation of the number. In list context, returns the length of the integer and fraction part. For Math::BigInt objects, the length of the fraction part is always 0.

The following probably doesn't do what you expect:

```
$c = Math::BigInt->new(123);
print $c->length(),"\n";                # prints 30
```

It prints both the number of digits in the number and in the fraction part since print calls "length()" in list context. Use something like:

```
print scalar $c->length(),"\n";         # prints 3
```

- mantissa()

```
$x->mantissa();
```

Return the signed mantissa of $x as a Math::BigInt.

- exponent()

```
$x->exponent();
```

Return the exponent of $x as a Math::BigInt.

- parts()

```
$x->parts();
```

Returns the significand (mantissa) and the exponent as integers. In Math::BigFloat, both are returned as Math::BigInt objects.

- sparts()

  Returns the significand (mantissa) and the exponent as integers. In scalar context, only the significand is returned. The significand is the integer with the smallest absolute value. The output of "sparts()" corresponds to the output from "bsstr()".

  In Math::BigInt, this method is identical to "parts()".

- nparts()

  Returns the significand (mantissa) and exponent corresponding to normalized notation. In scalar context, only the significand is returned. For finite non-zero numbers, the significand's absolute value is greater than or equal to 1 and less than 10. The output of "nparts()" corresponds to the output from "bnstr()". In Math::BigInt, if the significand can not be represented as an integer, upgrading is performed or NaN is returned.

- eparts()

  Returns the significand (mantissa) and exponent corresponding to engineering notation. In scalar context, only the significand is returned. For finite non-zero numbers, the significand's absolute value is greater than or equal to 1 and less than 1000, and the exponent is a multiple of 3. The output of "eparts()" corresponds to the output from "bestr()". In Math::BigInt, if the significand can not be represented as an integer, upgrading is performed or NaN is returned.

- dparts()

  Returns the integer part and the fraction part. If the fraction part can not be represented as an integer, upgrading is performed or NaN is returned. The output of "dparts()" corresponds to the output from "bdstr()".

- fparts()

  Returns the smallest possible numerator and denominator so that the numerator divided by the denominator gives back the original value. For finite numbers, both values are integers. Mnemonic: fraction.

- numerator()

  Together with "denominator()", returns the smallest integers so that the numerator divided by the denominator reproduces the original value. With Math::BigInt, "numerator()" simply returns a copy of the invocand.

- denominator()

  Together with "numerator()", returns the smallest integers so that the numerator divided by the denominator reproduces the original value. With Math::BigInt, "denominator()" always returns either a 1 or a NaN.

String conversion methods

- bstr()

  Returns a string representing the number using decimal notation. In Math::BigFloat, the output is zero padded according to the current accuracy or precision, if any of those are defined.

- bsstr()

  Returns a string representing the number using scientific notation where both the significand (mantissa) and the exponent are integers. The output corresponds to the output from "sparts()".

```
  123 is returned as "123e+0"
 1230 is returned as "123e+1"
12300 is returned as "123e+2"
12000 is returned as "12e+3"
10000 is returned as "1e+4"
```

- bnstr()

  Returns a string representing the number using normalized notation, the most common variant of scientific notation. For finite non-zero numbers, the absolute value of the significand is greater than or equal to 1 and less than 10. The output corresponds to the output from "nparts()".

```
  123 is returned as "1.23e+2"
 1230 is returned as "1.23e+3"
12300 is returned as "1.23e+4"
12000 is returned as "1.2e+4"
10000 is returned as "1e+4"
```

- bestr()

  Returns a string representing the number using engineering notation. For finite non-zero numbers, the absolute value of the significand is greater than or equal to 1 and less than 1000, and the exponent is a multiple of 3. The output corresponds to the output from "eparts()".

```
  123 is returned as "123e+0"
 1230 is returned as "1.23e+3"
12300 is returned as "12.3e+3"
12000 is returned as "12e+3"
10000 is returned as "10e+3"
```

- bdstr()

  Returns a string representing the number using decimal notation. The output corresponds to the output from "dparts()".

```
  123 is returned as "123"
 1230 is returned as "1230"
12300 is returned as "12300"
```

(continues on next page)

(continued from previous page)

```
12000 is returned as "12000"
10000 is returned as "10000"
```

- bfstr()

  Returns a string representing the number using fractional notation. The output corresponds to the output from "fparts()".

```
    12.345 is returned as "2469/200"
   123.45 is returned as "2469/20"
  1234.5 is returned as "2469/2"
 12345 is returned as "12345"
123450 is returned as "123450"
```

- to_hex()

```
$x->to_hex();
```

  Returns a hexadecimal string representation of the number. See also "from_hex()".

- to_oct()

```
$x->to_oct();
```

  Returns an octal string representation of the number. See also "from_oct()".

- to_bin()

```
$x->to_bin();
```

  Returns a binary string representation of the number. See also "from_bin()".

- to_bytes()

```
$x = Math::BigInt->new("1667327589");
$s = $x->to_bytes();                    # $s = "cafe"
```

  Returns a byte string representation of the number using big endian byte order. The invocand must be a non-negative, finite integer. See also "from_bytes()".

- to_ieee754()

  See *"to_ieee754()" in Math::BigFloat*.

- to_fp80()

  See *"to_fp80()" in Math::BigFloat*.

- to_base()

```
$x = Math::BigInt->new("250");
$x->to_base(2);     # returns "11111010"
$x->to_base(8);     # returns "372"
$x->to_base(16);    # returns "fa"
```

Returns a string representation of the number in the given base. If a collation sequence is given, the collation sequence determines which characters are used in the output.

Here are some more examples

```
$x = Math::BigInt->new("16")->to_base(3);       # returns "121"
$x = Math::BigInt->new("44027")->to_base(36);   # returns "XYZ"
$x = Math::BigInt->new("58314")->to_base(42);   # returns "Why"
$x = Math::BigInt->new("4")->to_base(2, "-|");  # returns "|--"
```

If the collation sequence are the bytes from "\x00" to "\xff", and the base is 256, then "to_base()" returns the same output as "to_bytes()". In the following example, $x and $y are identical:

```
$cs = join "", map chr, 0 .. 255;  # collation sequence
$x = Math::BigInt -> to_base("1230129310", 256, $cs)
$y = Math::BigInt -> to_bytes("1230129310");
```

See "from_base()" for information and examples.

- to_base_num()

  Converts the given number to the given base. This method is equivalent to "to_base()", but returns numbers in an array rather than characters in a string. In the output, the first element is the most significant.

```
$x = Math::BigInt->new(13);          # decimal 13 is binary 1101
$x->to_base_num(2);                  # returns [1, 1, 0, 1]

$x = Math::BigInt->new(65191);
$x->to_base_num(128);                # returns [3, 125, 39]
```

- as_hex()

```
$x->as_hex();
```

  As, "to_hex()", but with a "0x" prefix.

- as_oct()

```
$x->as_oct();
```

  As, "to_oct()", but with a "0" prefix.

- as_bin()

```
$x->as_bin();
```

  As, "to_bin()", but with a "0b" prefix.

- as_bytes()

  This is an alias for "to_bytes()".

Other conversion methods

- numify()

```
print $x->numify();
```

  Returns a Perl scalar from $x. It is used automatically whenever a scalar is needed, for instance in array index operations.

Utility methods

These utility methods are made public

- dec_str_to_dec_flt_str()

  Takes a string representing any valid number using decimal notation and converts it to a string representing the same number using decimal floating point notation. The output consists of five parts joined together: the sign of the significand, the absolute value of the significand as the smallest possible integer, the letter “e”, the sign of the exponent, and the absolute value of the exponent. If the input is invalid, nothing is returned.

```
$str2 = $class -> dec_str_to_dec_flt_str($str1);
```

  Some examples

```
Input           Output
31400.00e-4     +314e-2
-0.00012300e8   -123e+2
0               +0e+0
```

- hex_str_to_dec_flt_str()

  Takes a string representing any valid number using hexadecimal notation and converts it to a string representing the same number using decimal floating point notation. The output has the same format as that of “dec_str_to_dec_flt_str()”.

```
$str2 = $class -> hex_str_to_dec_flt_str($str1);
```

  Some examples

```
Input           Output
0xff            +255e+0
```

  Some examples

- oct_str_to_dec_flt_str()

  Takes a string representing any valid number using octal notation and converts it to a string representing the same number using decimal floating point notation. The output has the same format as that of “dec_str_to_dec_flt_str()”.

```
$str2 = $class -> oct_str_to_dec_flt_str($str1);
```

- bin_str_to_dec_flt_str()

Takes a string representing any valid number using binary notation and converts it to a string representing the same number using decimal floating point notation. The output has the same format as that of "dec_str_to_dec_flt_str()".

```
$str2 = $class -> bin_str_to_dec_flt_str($str1);
```

- dec_str_to_dec_str()

  Takes a string representing any valid number using decimal notation and converts it to a string representing the same number using decimal notation. If the number represents an integer, the output consists of a sign and the absolute value. If the number represents a non-integer, the output consists of a sign, the integer part of the number, the decimal point ".", and the fraction part of the number without any trailing zeros. If the input is invalid, nothing is returned.

- hex_str_to_dec_str()

  Takes a string representing any valid number using hexadecimal notation and converts it to a string representing the same number using decimal notation. The output has the same format as that of "dec_str_to_dec_str()".

- oct_str_to_dec_str()

  Takes a string representing any valid number using octal notation and converts it to a string representing the same number using decimal notation. The output has the same format as that of "dec_str_to_dec_str()".

- bin_str_to_dec_str()

  Takes a string representing any valid number using binary notation and converts it to a string representing the same number using decimal notation. The output has the same format as that of "dec_str_to_dec_str()".

ACCURACY AND PRECISION

Math::BigInt and Math::BigFloat have full support for accuracy and precision based rounding, both automatically after every operation, as well as manually.

This section describes the accuracy/precision handling in Math::BigInt and Math::BigFloat as it used to be and as it is now, complete with an explanation of all terms and abbreviations.

Not yet implemented things (but with correct description) are marked with '!', things that need to be answered are marked with '?'.

In the next paragraph follows a short description of terms used here (because these may differ from terms used by others people or documentation).

During the rest of this document, the shortcuts A (for accuracy), P (for precision), R (rounding mode), and F (fallback) are be used.

Accuracy A

Number of significant digits. Leading zeros are not counted. A number may have an accuracy greater than the non-zero digits when there are zeros in it or trailing zeros. For example, 123.456 has A of 6, 10203 has 5, 123.0506 has 7, 123.45000 has 8 and 0.000123 has 3.

The string output (of floating point numbers) is padded with zeros:

```
Initial value    P      A       Result          String
------------------------------------------------------------
1234.01                 3       1230            1230
1234.01                 6       1234.01         1234.01
1234.1                  8       1234.1          1234.1000
```

For Math::BigInt objects, no padding occurs.

Precision P

Precision is a fixed number of digits before (positive) or after (negative) the decimal point. For example, 123.45 has a precision of -2. 0 means an integer like 123 (or 120). A precision of 2 means at least two digits to the left of the decimal point are zero, so 123 with P = 1 becomes 120. Note that numbers with zeros before the decimal point may have different precisions, because 1200 can have P = 0, 1 or 2 (depending on what the initial value was). It could also have p < 0, when the digits after the decimal point are zero.

The string output (of floating point numbers) is padded with zeros:

```
Initial value    P      A       Result          String
------------------------------------------------------------
1234.01         -3              1000            1000
1234            -2              1200            1200
1234.5          -1              1230            1230
1234.001         1              1234            1234.0
1234.01          0              1234            1234
1234.01          2              1234.01         1234.01
1234.01          5              1234.01         1234.01000
```

For Math::BigInt objects, no padding occurs.

Rounding mode R

When rounding a number, different 'styles' or 'kinds' of rounding are possible. (Note that random rounding, as in Math::Round, is not implemented.)

Directed rounding

These round modes always round in the same direction.

- 'trunc'

  Round towards zero. Remove all digits following the rounding place, i.e., replace them with zeros. Thus, 987.65 rounded to tens (P=1) becomes 980, and rounded to the fourth significant digit becomes 987.6 (A=4). 123.456 rounded to the second place after the decimal point (P=-2) becomes 123.46. This corresponds to the IEEE 754 rounding mode 'roundTowardZero'.

Rounding to nearest

These rounding modes round to the nearest digit. They differ in how they determine which way to round in the ambiguous case when there is a tie.

- 'even'

  Round towards the nearest even digit, e.g., when rounding to nearest integer, -5.5 becomes -6, 4.5 becomes 4, but 4.501 becomes 5. This corresponds to the IEEE 754 rounding mode 'roundTiesToEven'.

- 'odd'

  Round towards the nearest odd digit, e.g., when rounding to nearest integer, 4.5 becomes 5, -5.5 becomes -5, but 5.501 becomes 6. This corresponds to the IEEE 754 rounding mode 'roundTiesToOdd'.

- '+inf'

  Round towards plus infinity, i.e., always round up. E.g., when rounding to the nearest integer, 4.5 becomes 5, -5.5 becomes -5, and 4.501 also becomes 5. This corresponds to the IEEE 754 rounding mode 'roundTiesToPositive'.

- '-inf'

  Round towards minus infinity, i.e., always round down. E.g., when rounding to the nearest integer, 4.5 becomes 4, -5.5 becomes -6, but 4.501 becomes 5. This corresponds to the IEEE 754 rounding mode 'roundTiesToNegative'.

- 'zero'

  Round towards zero, i.e., round positive numbers down and negative numbers up. E.g., when rounding to the nearest integer, 4.5 becomes 4, -5.5 becomes -5, but 4.501 becomes 5. This corresponds to the IEEE 754 rounding mode 'roundTiesToZero'.

- 'common'

  Round away from zero, i.e., round to the number with the largest absolute value. E.g., when rounding to the nearest integer, -1.5 becomes -2, 1.5 becomes 2 and 1.49 becomes 1. This corresponds to the IEEE 754 rounding mode 'roundTiesToAway'.

Fallback F

When neither A nor P are defined, the fallback accuracy is used when computing values that would potentially give an infinite number of digits, e.g., division, roots, logarithms, trigonometric functions etc.

More details on rounding

The handling of A & P in MBI/MBF (the old core code shipped with Perl versions <= 5.7.2) is like this:

- Precision

```
* bfround($p) is able to round to $p number of digits after the␣
↪decimal
```

(continues on next page)

(continued from previous page)

```
  point
* otherwise P is unused
```

- Accuracy (significant digits)

```
 * bround($a) rounds to $a significant digits
 * only bdiv() and bsqrt() take A as (optional) parameter
   + other operations simply create the same number (bneg etc), or
     more (bmul) of digits
   + rounding/truncating is only done when explicitly calling one
     of bround or bfround, and never for Math::BigInt (not implemented)
 * bsqrt() simply hands its accuracy argument over to bdiv.
 * the documentation and the comment in the code indicate two
   different ways on how bdiv() determines the maximum number
   of digits it should calculate, and the actual code does yet
   another thing
   POD:
     max($Math::BigFloat::div_scale, length(dividend)+length(divisor))
   Comment:
     result has at most max(scale, length(dividend), length(divisor)) digits
   Actual code:
     scale = max(scale, length(dividend)-1,length(divisor)-1);
     scale += length(divisor) - length(dividend);
   So for lx = 3, ly = 9, scale = 10, scale will actually be 16 (10
   So for lx = 3, ly = 9, scale = 10, scale will actually be 16
   (10+9-3). Actually, the 'difference' added to the scale is cal-
   culated from the number of "significant digits" in dividend and
   divisor, which is derived by looking at the length of the man-
   tissa. Which is wrong, since it includes the + sign (oops) and
   actually gets 2 for '+100' and 4 for '+101'. Oops again. Thus
   124/3 with div_scale=1 will get you '41.3' based on the strange
   assumption that 124 has 3 significant digits, while 120/7 will
   get you '17', not '17.1' since 120 is thought to have 2 signif-
   icant digits. The rounding after the division then uses the
   remainder and $y to determine whether it must round up or down.
```

(continues on next page)

(continued from previous page)

```
?  I have no idea which is the right way. That's why I used a␣
↪slightly more
?  simple scheme and tweaked the few failing testcases to match␣
↪it.
```

This is how it works now:

- Setting/Accessing

```
* You can set the A global via Math::BigInt->accuracy() or
  Math::BigFloat->accuracy() or whatever class you are using.
* You can also set P globally by using Math::SomeClass->
↪precision()
  likewise.
* Globals are classwide, and not inherited by subclasses.
* to undefine A, use Math::SomeClass->accuracy(undef);
* to undefine P, use Math::SomeClass->precision(undef);
* Setting Math::SomeClass->accuracy() clears automatically
  Math::SomeClass->precision(), and vice versa.
* To be valid, A must be > 0, P can have any value.
* If P is negative, this means round to the P'th place to the␣
↪right of the
  decimal point; positive values mean to the left of the␣
↪decimal point.
  P of 0 means round to integer.
* to find out the current global A, use Math::SomeClass->
↪accuracy()
* to find out the current global P, use Math::SomeClass->
↪precision()
* use $x->accuracy() respective $x->precision() for the local
  setting of $x.
* Please note that $x->accuracy() respective $x->precision()
  return eventually defined global A or P, when $x's A or P is␣
↪not
  set.
```

- Creating numbers

```
* When you create a number, you can give the desired A or P via:
  $x = Math::BigInt->new($number,$A,$P);
* Only one of A or P can be defined, otherwise the result is NaN
* If no A or P is give ($x = Math::BigInt->new($number) form),␣
↪then the
  globals (if set) will be used. Thus changing the global␣
↪defaults later on
  will not change the A or P of previously created numbers (i.e.
↪, A and P of
  $x will be what was in effect when $x was created)
* If given undef for A and P, NO rounding will occur, and the␣
↪globals will
```

(continues on next page)

(continued from previous page)

```
  NOT be used. This is used by subclasses to create numbers␣
↪without
  suffering rounding in the parent. Thus a subclass is able to␣
↪have its own
  globals enforced upon creation of a number by using
  $x = Math::BigInt->new($number,undef,undef):

      use Math::BigInt::SomeSubclass;
      use Math::BigInt;

      Math::BigInt->accuracy(2);
      Math::BigInt::SomeSubclass->accuracy(3);
      $x = Math::BigInt::SomeSubclass->new(1234);

  $x is now 1230, and not 1200. A subclass might choose to␣
↪implement
  this otherwise, e.g. falling back to the parent's A and P.
```

- Usage

```
* If A or P are enabled/defined, they are used to round the␣
↪result of each
  operation according to the rules below
* Negative P is ignored in Math::BigInt, since Math::BigInt␣
↪objects never
  have digits after the decimal point
* Math::BigFloat uses Math::BigInt internally, but setting A or␣
↪P inside
  Math::BigInt as globals does not tamper with the parts of a␣
↪Math::BigFloat.
  A flag is used to mark all Math::BigFloat numbers as 'never␣
↪round'.
```

- Precedence

```
 * It only makes sense that a number has only one of A or P at␣
↪a time.
   If you set either A or P on one object, or globally, the␣
↪other one will
   be automatically cleared.
 * If two objects are involved in an operation, and one of them␣
↪has A in
   effect, and the other P, this results in an error (NaN).
 * A takes precedence over P (Hint: A comes before P).
   If neither of them is defined, nothing is used, i.e. the␣
↪result will have
   as many digits as it can (with an exception for bdiv/bsqrt)␣
↪and will not
   be rounded.
 * There is another setting for bdiv() (and thus for bsqrt()).␣
```

(continues on next page)

(continued from previous page)

```
↪If neither of
  A or P is defined, bdiv() will use a fallback (F) of $div_
↪scale digits.
  If either the dividend's or the divisor's mantissa has more␣
↪digits than
  the value of F, the higher value will be used instead of F.
  This is to limit the digits (A) of the result (just consider␣
↪what would
  happen with unlimited A and P in the case of 1/3 :-)
* bdiv will calculate (at least) 4 more digits than required␣
↪(determined by
  A, P or F), and, if F is not used, round the result
  (this will still fail in the case of a result like 0.
↪12345000000001 with A
  or P of 5, but this can not be helped - or can it?)
* Thus you can have the math done by on Math::Big* class in␣
↪two modi:
  + never round (this is the default):
    This is done by setting A and P to undef. No math operation
    will round the result, with bdiv() and bsqrt() as␣
↪exceptions to guard
    against overflows. You must explicitly call bround(),␣
↪bfround() or
    round() (the latter with parameters).
    Note: Once you have rounded a number, the settings will
↪'stick' on it
    and 'infect' all other numbers engaged in math operations␣
↪with it, since
    local settings have the highest precedence. So, to get␣
↪SaferRound[tm],
    use a copy() before rounding like this:

      $x = Math::BigFloat->new(12.34);
      $y = Math::BigFloat->new(98.76);
      $z = $x * $y;                           # 1218.6984
      print $x->copy()->bround(3);            # 12.3 (but A is␣
↪now 3!)
      $z = $x * $y;                           # still 1218.
↪6984, without
                                              # copy would␣
↪have been 1210!

  + round after each op:
    After each single operation (except for testing like is_
↪zero()), the
    method round() is called and the result is rounded␣
↪appropriately. By
    setting proper values for A and P, you can have all-the-
↪same-A or
```

(continues on next page)

(continued from previous page)

```
    all-the-same-P modes. For example, Math::Currency might␣
↪set A to undef,
    and P to -2, globally.

?Maybe an extra option that forbids local A & P settings would␣
↪be in order,
?so that intermediate rounding does not 'poison' further math?
```

- Overriding globals

```
* you will be able to give A, P and R as an argument to all the␣
↪calculation
  routines; the second parameter is A, the third one is P, and␣
↪the fourth is
  R (shift right by one for binary operations like badd). P is␣
↪used only if
  the first parameter (A) is undefined. These three parameters␣
↪override the
  globals in the order detailed as follows, i.e. the first␣
↪defined value
  wins:
  (local: per object, global: global default, parameter:␣
↪argument to sub)
    + parameter A
    + parameter P
    + local A (if defined on both of the operands: smaller one␣
↪is taken)
    + local P (if defined on both of the operands: bigger one␣
↪is taken)
    + global A
    + global P
    + global F
* bsqrt() will hand its arguments to bdiv(), as it used to,␣
↪only now for two
  arguments (A and P) instead of one
```

- Local settings

```
* You can set A or P locally by using $x->accuracy() or
  $x->precision()
  and thus force different A and P for different objects/
↪numbers.
* Setting A or P this way immediately rounds $x to the new␣
↪value.
* $x->accuracy() clears $x->precision(), and vice versa.
```

- Rounding

```
* the rounding routines will use the respective global or local␣
↪settings.
```

(continues on next page)

(continued from previous page)

```
  bround() is for accuracy rounding, while bfround() is for
↪precision
* the two rounding functions take as the second parameter one
↪of the
  following rounding modes (R):
  'even', 'odd', '+inf', '-inf', 'zero', 'trunc', 'common'
* you can set/get the global R by using Math::SomeClass->round_
↪mode()
  or by setting $Math::SomeClass::round_mode
* after each operation, $result->round() is called, and the
↪result may
  eventually be rounded (that is, if A or P were set either
↪locally,
  globally or as parameter to the operation)
* to manually round a number, call $x->round($A,$P,$round_mode);
  this will round the number by using the appropriate rounding
↪function
  and then normalize it.
* rounding modifies the local settings of the number:

      $x = Math::BigFloat->new(123.456);
      $x->accuracy(5);
      $x->bround(4);

  Here 4 takes precedence over 5, so 123.5 is the result and $x-
↪>accuracy()
  will be 4 from now on.
```

- Default values

```
* A: undef
* P: undef
* R: 'even'
* F: 40
```

- Remarks

```
* The defaults are set up so that the new code gives the same
↪results as
  the old code (except in a few cases on bdiv):
  + Both A and P are undefined and thus will not be used for
↪rounding
    after each operation.
  + round() is thus a no-op, unless given extra parameters A
↪and P
```

INTERNALS

You should neither care about nor depend on the internal representation; it might change without notice. Use **ONLY** method calls like `$x->sign();` instead relying on the internal representation.

Math Library

The mathematical computations are performed by a backend library. It is not required to specify which backend library to use, but some backend libraries are much faster than the default library.

The default library

The default library is *Math::BigInt::Calc*, which is implemented in pure Perl and hence does not require a compiler.

Specifying a library

The simple case

```
use Math::BigInt;
```

is equivalent to saying

```
use Math::BigInt try => 'Calc';
```

You can use a different backend library with, e.g.,

```
use Math::BigInt try => 'GMP';
```

which attempts to load the Math::BigInt::GMP library, and falls back to the default library if the specified library can't be loaded.

Multiple libraries can be specified by separating them by a comma, e.g.,

```
use Math::BigInt try => 'GMP,Pari';
```

If you request a specific set of libraries and do not allow fallback to the default library, specify them using "only",

```
use Math::BigInt only => 'GMP,Pari';
```

If you prefer a specific set of libraries, but want to see a warning if the fallback library is used, specify them using "lib",

```
use Math::BigInt lib => 'GMP,Pari';
```

The following first tries to find Math::BigInt::Foo, then Math::BigInt::Bar, and if this also fails, reverts to Math::BigInt::Calc:

```
use Math::BigInt try => 'Foo,Math::BigInt::Bar';
```

Which library to use?

Note: General purpose packages should not be explicit about the library to use; let the script author decide which is best.

Math::BigInt::GMP, Math::BigInt::Pari, and Math::BigInt::GMPz are in cases involving big numbers much faster than *Math::BigInt::Calc*. However these libraries are slower when dealing with very small numbers (less than about 20 digits) and when converting very large numbers to decimal (for instance for printing, rounding, calculating their length in decimal etc.).

So please select carefully what library you want to use.

Different low-level libraries use different formats to store the numbers, so mixing them won't work. You should not depend on the number having a specific internal format.

See the respective math library module documentation for further details.

Loading multiple libraries

The first library that is successfully loaded is the one that will be used. Any further attempts at loading a different module will be ignored. This is to avoid the situation where module A requires math library X, and module B requires math library Y, causing modules A and B to be incompatible. For example,

```
use Math::BigInt;                    # loads default "Calc"
use Math::BigFloat only => "GMP";    # ignores "GMP"
```

Sign

The sign is either '+', '-', 'NaN', '+inf' or '-inf'.

A sign of 'NaN' is used to represent values that are not numbers, e.g., the result of 0/0. '+inf' and '-inf' represen positive and negative infinity, respectively. For example you get '+inf' when dividing a positive number by 0, and '-inf' when dividing any negative number by 0.

EXAMPLES

```
use Math::BigInt;

sub bigint { Math::BigInt->new(shift); }

$x = Math::BigInt->bstr("1234")       # string "1234"
$x = "$x";                            # same as bstr()
$x = Math::BigInt->bneg("1234");      # Math::BigInt "-1234"
$x = Math::BigInt->babs("-12345");    # Math::BigInt "12345"
$x = Math::BigInt->bnorm("-0.00");    # Math::BigInt "0"
$x = bigint(1) + bigint(2);           # Math::BigInt "3"
$x = bigint(1) + "2";                 # ditto ("2" becomes a␣
↪Math::BigInt)
$x = bigint(1);                       # Math::BigInt "1"
```

(continues on next page)

(continued from previous page)

```
$x = $x + 5 / 2;                        # Math::BigInt "3"
$x = $x ** 3;                           # Math::BigInt "27"
$x *= 2;                                # Math::BigInt "54"
$x = Math::BigInt->new(0);              # Math::BigInt "0"
$x--;                                   # Math::BigInt "-1"
$x = Math::BigInt->badd(4,5)            # Math::BigInt "9"
print $x->bsstr();                      # 9e+0
```

Examples for rounding:

```
use Math::BigFloat;
use Test::More;

$x = Math::BigFloat->new(123.4567);
$y = Math::BigFloat->new(123.456789);
Math::BigFloat->accuracy(4);            # no more A than 4

is ($x->copy()->bround(),123.4);        # even rounding
print $x->copy()->bround(),"\n";        # 123.4
Math::BigFloat->round_mode('odd');      # round to odd
print $x->copy()->bround(),"\n";        # 123.5
Math::BigFloat->accuracy(5);            # no more A than 5
Math::BigFloat->round_mode('odd');      # round to odd
print $x->copy()->bround(),"\n";        # 123.46
$y = $x->copy()->bround(4),"\n";        # A = 4: 123.4
print "$y, ",$y->accuracy(),"\n";       # 123.4, 4

Math::BigFloat->accuracy(undef);        # A not important now
Math::BigFloat->precision(2);           # P important
print $x->copy()->bnorm(),"\n";         # 123.46
print $x->copy()->bround(),"\n";        # 123.46
```

Examples for converting:

```
my $x = Math::BigInt->new('0b1'.'01' x 123);
print "bin: ",$x->as_bin()," hex:",$x->as_hex()," dec: ",$x,"\n";
```

NUMERIC LITERALS

After `use Math::BigInt ':constant'` all numeric literals in the given scope are converted to `Math::BigInt` objects. This conversion happens at compile time. Every non-integer is convert to a NaN.

For example,

```
perl -MMath::BigInt=:constant -le 'print 2**150'
```

prints the exact value of `2**150`. Note that without conversion of constants to objects the expression `2**150` is calculated using Perl scalars, which leads to an inaccurate result.

Please note that strings are not affected, so that

```
use Math::BigInt qw/:constant/;

$x = "12345678901234567890123456789012345678890"
        + "123456789123456789";
```

does give you what you expect. You need an explicit Math::BigInt->new() around at least one of the operands. You should also quote large constants to prevent loss of precision:

```
use Math::BigInt;

$x = Math::BigInt->new("123456788912345678912345678912345678889");
```

Without the quotes Perl first converts the large number to a floating point constant at compile time, and then converts the result to a Math::BigInt object at run time, which results in an inaccurate result.

Hexadecimal, octal, and binary floating point literals

Perl (and this module) accepts hexadecimal, octal, and binary floating point literals, but use them with care with Perl versions before v5.32.0, because some versions of Perl silently give the wrong result. Below are some examples of different ways to write the number decimal 314.

Hexadecimal floating point literals:

```
0x1.3ap+8          0X1.3AP+8
0x1.3ap8           0X1.3AP8
0x13a0p-4          0X13A0P-4
```

Octal floating point literals (with “0” prefix):

```
01.164p+8          01.164P+8
01.164p8           01.164P8
011640p-4          011640P-4
```

Octal floating point literals (with “0o” prefix) (requires v5.34.0):

```
0o1.164p+8         0O1.164P+8
0o1.164p8          0O1.164P8
0o11640p-4         0O11640P-4
```

Binary floating point literals:

```
0b1.0011101p+8     0B1.0011101P+8
0b1.0011101p8      0B1.0011101P8
0b10011101000p-2   0B10011101000P-2
```

PERFORMANCE

Using the form $x += $y; etc over $x = $x + $y is faster, since a copy of $x must be made in the second case. For long numbers, the copy can eat up to 20% of the work (in the case of addition/subtraction, less for multiplication/division). If $y is very small compared to $x, the form $x += $y is MUCH faster than $x = $x + $y since making the copy of $x takes more time then the actual addition.

With a technique called copy-on-write, the cost of copying with overload could be minimized or even completely avoided. A test implementation of COW did show performance gains for overloaded math, but introduced a performance loss due to a constant overhead for all other operations. So Math::BigInt does currently not COW.

The rewritten version of this module (vs. v0.01) is slower on certain operations, like "new()", "bstr()" and "numify()". The reason are that it does now more work and handles much more cases. The time spent in these operations is usually gained in the other math operations so that code on the average should get (much) faster. If they don't, please contact the author.

Some operations may be slower for small numbers, but are significantly faster for big numbers. Other operations are now constant (O(1), like "bneg()", "babs()" etc), instead of O(N) and thus nearly always take much less time. These optimizations were done on purpose.

If you find the Calc module to slow, try to install any of the replacement modules and see if they help you.

Alternative math libraries

You can use an alternative library to drive Math::BigInt. See the section "Math Library" for more information.

For more benchmark results see http://bloodgate.com/perl/benchmarks.html.

SUBCLASSING

Subclassing Math::BigInt

The basic design of Math::BigInt allows simple subclasses with very little work, as long as a few simple rules are followed:

- The public API must remain consistent, i.e. if a sub-class is overloading addition, the sub-class must use the same name, in this case badd(). The reason for this is that Math::BigInt is optimized to call the object methods directly.
- The private object hash keys like `$x->{sign}` may not be changed, but additional keys can be added, like `$x->{_custom}`.
- Accessor functions are available for all existing object hash keys and should be used instead of directly accessing the internal hash keys. The reason for this is that Math::BigInt itself has a pluggable interface which permits it to support different storage methods.

More complex sub-classes may have to replicate more of the logic internal of Math::BigInt if they need to change more basic behaviors. A subclass that needs to merely change the output only needs to overload "bstr()".

All other object methods and overloaded functions can be directly inherited from the parent class.

At the very minimum, any subclass needs to provide its own "new()" and can store additional hash keys in the object. There are also some package globals that must be defined, e.g.:

```
# Globals
our $accuracy = 2;          # round to 2 decimal places
our $precision = undef;
our $round_mode = 'even';
our $div_scale = 40;
```

Additionally, you might want to provide the following two globals to allow auto-upgrading and auto-downgrading:

```
our $upgrade = undef;
our $downgrade = undef;
```

This allows Math::BigInt to correctly retrieve package globals from the subclass, like `$SubClass::precision`. See `t/Math/BigInt/Subclass.pm`, `t/Math/BigFloat/SubClass.pm`, or `t/Math/BigRat/SubClass.pm` for subclass examples.

Don't forget to

```
use overload;
```

in your subclass to automatically inherit the overloading from the parent. If you like, you can change part of the overloading, look at Math::String for an example.

UPGRADING

When used like this:

```
use Math::BigInt upgrade => 'Foo::Bar';

use Math::BigInt;
Math::BigInt -> upgrade('Foo::Bar');
```

any operation whose result cannot be represented as an Math::BigInt object is upgraded to the class Foo::Bar. Usually this is used in conjunction with Math::BigRat or Math::BigFloat:

```
use Math::BigInt upgrade => 'Math::BigFloat';
```

For example, the following returns 3 as a Math::BigInt when no upgrading is defined, and 3.125 as a Math::BigFloat if Math::BigInt is set to upgrade to Math::BigFloat:

```
$x = Math::BigInt -> new(25) -> bdiv(8);
```

As a shortcut, you can use the module *bignum*:

```
use bignum;
```

which is also good for one-liners:

```
perl -Mbignum -le 'print 2 ** 255'
```

This makes it possible to mix arguments of different classes (as in 2.5 + 2) as well as preserve accuracy (as in sqrt(3)).

Auto-upgrade

The following methods upgrade themselves unconditionally; that is if upgrade is in effect, they always hands up their work:

```
bdiv bfdiv btdiv bsqrt blog bexp bpi bsin bcos batan batan2
```

All other methods upgrade themselves only when one (or all) of their arguments are of the class mentioned in $upgrade.

EXPORTS

`Math::BigInt` exports nothing by default, but can export the following methods:

```
bgcd
blcm
```

CAVEATS

Some things might not work as you expect them. Below is documented what is known to be troublesome:

- Comparing numbers as strings

  Both "bstr()" and "bsstr()" as well as stringify via overload drop the leading '+'. This is to be consistent with Perl and to make `cmp` (especially with overloading) to work as you expect. It also solves problems with `Test.pm` and *Test::More*, which stringify arguments before comparing them.

  Mark Biggar said, when asked about to drop the '+' altogether, or make only `cmp` work:

  ```
  I agree (with the first alternative), don't add the '+' on␣
  ↪positive
  numbers.  It's not as important anymore with the new internal␣
  ↪form
  for numbers.  It made doing things like abs and neg easier, but
  those have to be done differently now anyway.
  ```

  So, the following examples now works as expected:

  ```
  use Test::More tests => 1;
  use Math::BigInt;

  my $x = Math::BigInt -> new(3*3);
  ```

(continues on next page)

(continued from previous page)

```
my $y = Math::BigInt -> new(3*3);

is($x,3*3, 'multiplication');
print "$x eq 9" if $x eq $y;
print "$x eq 9" if $x eq '9';
print "$x eq 9" if $x eq 3*3;
```

Additionally, the following still works:

```
print "$x == 9" if $x == $y;
print "$x == 9" if $x == 9;
print "$x == 9" if $x == 3*3;
```

There is now a "bsstr()" method to get the string in scientific notation aka 1e+2 instead of 100. Be advised that overloaded 'eq' always uses bstr() for comparison, but Perl represents some numbers as 100 and others as 1e+308. If in doubt, convert both arguments to Math::BigInt before comparing them as strings:

```
use Test::More tests => 3;
use Math::BigInt;

$x = Math::BigInt->new('1e56');
$y = 1e56;
is($x,$y);                      # fails
is($x->bsstr(), $y);            # okay
$y = Math::BigInt->new($y);
is($x, $y);                     # okay
```

Alternatively, simply use <=> for comparisons, this always gets it right. There is not yet a way to get a number automatically represented as a string that matches exactly the way Perl represents it.

- oct()/hex()

  These perl routines currently (as of Perl v.5.8.6) cannot handle passed inf.

```
te@linux:~> perl -wle 'print 2 ** 3333'
Inf
te@linux:~> perl -wle 'print 2 ** 3333 == 2 ** 3333'
1
te@linux:~> perl -wle 'print oct(2 ** 3333)'
0
te@linux:~> perl -wle 'print hex(2 ** 3333)'
Illegal hexadecimal digit 'I' ignored at -e line 1.
0
```

  The same problems occur if you pass them Math::BigInt->binf() objects. Since overloading these routines is not possible, this cannot be fixed from Math::BigInt.

- int()

  "int()" returns (at least for Perl v5.7.1 and up) another Math::BigInt, not a Perl scalar:

```
$x = Math::BigInt->new(123);
$y = int($x);                           # 123 as a Math::BigInt
$x = Math::BigFloat->new(123.45);
$y = int($x);                           # 123 as a␣
↪Math::BigFloat
```

If you want a real Perl scalar, use "numify()":

```
$y = $x->numify();                      # 123 as a scalar
```

This is seldom necessary, though, because this is done automatically, like when you access an array:

```
$z = $array[$x];                        # does work␣
↪automatically
```

- Modifying and =

  Beware of:

```
$x = Math::BigFloat->new(5);
$y = $x;
```

  This makes a second reference to the **same** object and stores it in $y. Thus anything that modifies $x (except overloaded operators) also modifies $y, and vice versa. Or in other words, = is only safe if you modify your Math::BigInt objects only via overloaded math. As soon as you use a method call it breaks:

```
$x->bmul(2);
print "$x, $y\n";       # prints '10, 10'
```

  If you want a true copy of $x, use:

```
$y = $x->copy();
```

  You can also chain the calls like this, this first makes a copy and then multiply it by 2:

```
$y = $x->copy()->bmul(2);
```

  See also the documentation for overload.pm regarding =.

- Overloading -$x

  The following:

```
$x = -$x;
```

  is slower than

```
$x->bneg();
```

  since overload calls `sub($x,0,1);` instead of `neg($x)`. The first variant needs to preserve $x since it does not know that it later gets overwritten. This makes a copy

of $x and takes O(N), but $x->bneg() is O(1).

- Mixing different object types

  With overloaded operators, it is the first (dominating) operand that determines which method is called. Here are some examples showing what actually gets called in various cases.

```
use Math::BigInt;
use Math::BigFloat;

$mbf  = Math::BigFloat->new(5);
$mbi2 = Math::BigInt->new(5);
$mbi  = Math::BigInt->new(2);
                                  # what actually gets called:
$float = $mbf + $mbi;             # $mbf->badd($mbi)
$float = $mbf / $mbi;             # $mbf->bdiv($mbi)
$integer = $mbi + $mbf;           # $mbi->badd($mbf)
$integer = $mbi2 / $mbi;          # $mbi2->bdiv($mbi)
$integer = $mbi2 / $mbf;          # $mbi2->bdiv($mbf)
```

  For instance, Math::BigInt->bdiv() always returns a Math::BigInt, regardless of whether the second operant is a Math::BigFloat. To get a Math::BigFloat you either need to call the operation manually, make sure each operand already is a Math::BigFloat, or cast to that type via Math::BigFloat->new():

```
$float = Math::BigFloat->new($mbi2) / $mbi;      # = 2.5
```

  Beware of casting the entire expression, as this would cast the result, at which point it is too late:

```
$float = Math::BigFloat->new($mbi2 / $mbi);      # = 2
```

  Beware also of the order of more complicated expressions like:

```
$integer = ($mbi2 + $mbi) / $mbf;                # int / float =>
 ↪ int
$integer = $mbi2 / Math::BigFloat->new($mbi);    # ditto
```

  If in doubt, break the expression into simpler terms, or cast all operands to the desired resulting type.

  Scalar values are a bit different, since:

```
$float = 2 + $mbf;
$float = $mbf + 2;
```

  both result in the proper type due to the way the overloaded math works.

  This section also applies to other overloaded math packages, like Math::String.

  One solution to you problem might be autoupgrading|upgrading. See the pragmas *bignum*, *bigint* and *bigrat* for an easy way to do this.

BUGS

Please report any bugs or feature requests to `bug-math-bigint at rt.cpan.org`, or through the web interface at https://rt.cpan.org/Ticket/Create.html?Queue=Math-BigInt (requires login). We will be notified, and then you'll automatically be notified of progress on your bug as I make changes.

SUPPORT

You can find documentation for this module with the perldoc command.

```
perldoc Math::BigInt
```

You can also look for information at:

- GitHub

  https://github.com/pjacklam/p5-Math-BigInt

- RT: CPAN's request tracker

  https://rt.cpan.org/Dist/Display.html?Name=Math-BigInt

- MetaCPAN

  https://metacpan.org/release/Math-BigInt

- CPAN Testers Matrix

  http://matrix.cpantesters.org/?dist=Math-BigInt

LICENSE

This program is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

SEE ALSO

Math::BigFloat and *Math::BigRat* as well as the backend libraries *Math::BigInt::FastCalc*, Math::BigInt::GMP, and Math::BigInt::Pari, Math::BigInt::GMPz, and Math::BigInt::BitVect.

The pragmas *bigint*, *bigfloat*, and *bigrat* might also be of interest. In addition there is the *bignum* pragma which does upgrading and downgrading.

AUTHORS

- Mark Biggar, overloaded interface by Ilya Zakharevich, 1996-2001.
- Completely rewritten by Tels http://bloodgate.com, 2001-2008.
- Florian Ragwitz flora@cpan.org, 2010.
- Peter John Acklam pjacklam@gmail.com, 2011-.

Many people contributed in one or more ways to the final beast, see the file CREDITS for an (incomplete) list. If you miss your name, please drop me a mail. Thank you!

Modules

- `Math::BigInt::Calc` - pure Perl module to support Math::BigInt
- `Math::BigInt::FastCalc` - The XS half of Math::BigInt::FastCalc 0.5020: the hot primitives of Math::BigInt's default library, in C.
- `Math::BigInt::Lib` - virtual parent class for Math::BigInt libraries

Math::BigInt::Calc

pure Perl module to support Math::BigInt

SYNOPSIS

```
# to use it with Math::BigInt
use Math::BigInt lib => 'Calc';

# to use it with Math::BigFloat
use Math::BigFloat lib => 'Calc';

# to use it with Math::BigRat
use Math::BigRat lib => 'Calc';

# explicitly set base length and whether to "use integer"
use Math::BigInt::Calc base_len => 4, use_int => 1;
use Math::BigInt lib => 'Calc';
```

DESCRIPTION

Math::BigInt::Calc inherits from Math::BigInt::Lib.

In this library, the numbers are represented interenally in base B = 10**N, where N is the largest possible integer that does not cause overflow in the intermediate computations. The base B elements are stored in an array, with the least significant element stored in array element zero. There are no leading zero elements, except a single zero element when the number is zero. For instance, if B = 10000, the number 1234567890 is represented internally as [7890, 3456, 12].

OPTIONS

When the module is loaded, it computes the maximum exponent, i.e., power of 10, that can be used with and without "use integer" in the computations. The default is to use this maximum exponent. If the combination of the 'base_len' value and the 'use_int' value exceeds the maximum value, an error is thrown.

- base_len

  The base length can be specified explicitly with the 'base_len' option. The value must be a positive integer.

```
use Math::BigInt::Calc base_len => 4;  # use 10000 as internal␣
↪base
```

- use_int

  This option is used to specify whether “use integer” should be used in the internal computations. The value is interpreted as a boolean value, so use 0 or “” for false and anything else for true. If the ‘base_len’ is not specified together with ‘use_int’, the current value for the base length is used.

```
use Math::BigInt::Calc use_int => 1;   # use "use integer"␣
↪internally
```

METHODS

This overview constains only the methods that are specific to `Math::BigInt::Calc`. For the other methods, see *Math::BigInt::Lib*.

- _base_len()

  Specify the desired base length and whether to enable “use integer” in the computations.

```
Math::BigInt::Calc -> _base_len($base_len, $use_int);
```

  Note that it is better to specify the base length and whether to use integers as options when the module is loaded, for example like this

```
use Math::BigInt::Calc base_len => 6, use_int => 1;
```

SEE ALSO

Math::BigInt::Lib for a description of the API.

Alternative libraries *Math::BigInt::FastCalc*, Math::BigInt::GMP, Math::BigInt::Pari, Math::BigInt::GMPz, and Math::BigInt::BitVect.

Some of the modules that use these libraries *Math::BigInt*, *Math::BigFloat*, and *Math::BigRat*.

Math::BigInt::FastCalc

The XS half of Math::BigInt::FastCalc 0.5020: the hot primitives of Math::BigInt’s default library, in C.

`Math/BigInt/FastCalc.pm` is upstream’s, a subclass of Math::BigInt::Calc, which stores a number as an array of base-10**N “parts”, least significant first. These XSUBs replace Calc’s Perl for creating, copying, incrementing, decrementing, comparing and testing such arrays; everything else is Calc’s.

```
use Math::BigInt lib => 'FastCalc';
print Math::BigInt->new("123456789012345678901234567890") + 1;
# 123456789012345678901234567891
```

Nothing calls these XSUBs but the library itself. `Math::BigInt` uses FastCalc when asked to, with `lib => 'FastCalc'` (or `try` / `only`); its default library is plain `Math::BigInt::Calc`.

Differences from upstream

None.

Math::BigInt::Lib

virtual parent class for Math::BigInt libraries

SYNOPSIS

```
# In the backend library for Math::BigInt et al.

package Math::BigInt::MyBackend;

use Math::BigInt::Lib;
our @ISA = qw< Math::BigInt::Lib >;

sub _new { ... }
sub _str { ... }
sub _add { ... }
str _sub { ... }
...

# In your main program.

use Math::BigInt lib => 'MyBackend';
```

DESCRIPTION

This module provides support for big integer calculations. It is not intended to be used directly, but rather as a parent class for backend libraries used by Math::BigInt, Math::BigFloat, Math::BigRat, and related modules.

Other backend libraries include Math::BigInt::Calc, Math::BigInt::FastCalc, Math::BigInt::GMP, and Math::BigInt::Pari.

In order to allow for multiple big integer libraries, Math::BigInt was rewritten to use a plug-in library for core math routines. Any module which conforms to the API can be used by Math::BigInt by using this in your program:

```
use Math::BigInt lib => 'libname';
```

'libname' is either the long name, like 'Math::BigInt::Pari', or only the short version, like 'Pari'.

General Notes

A library only needs to deal with unsigned big integers. Testing of input parameter validity is done by the caller, so there is no need to worry about underflow (e.g., in `_sub()` and `_dec()`) or about division by zero (e.g., in `_div()` and `_mod()`)) or similar cases.

Some libraries use methods that don't modify their argument, and some libraries don't even use objects, but rather unblessed references. Because of this, liberary methods are always called as class methods, not instance methods:

```
$x = Class -> method($x, $y);     # like this
$x = $x -> method($y);            # not like this ...
$x -> method($y);                 # ... or like this
```

And with boolean methods

```
$bool = Class -> method($x, $y);  # like this
$bool = $x -> method($y);         # not like this
```

Return values are always objects, strings, Perl scalars, or true/false for comparison routines.

API version

- CLASS->api_version()

  This method is no longer used and can be omitted. Methods that are not implemented by a subclass will be inherited from this class.

Constructors

The following methods are mandatory: _new(), _str(), _add(), and _sub(). However, computations will be very slow without _mul() and _div().

- CLASS->_new(STR)

  Convert a string representing an unsigned decimal number to an object representing the same number. The input is normalized, i.e., it matches `^(0|[1-9]\d*)$`.

- CLASS->_zero()

  Return an object representing the number zero.

- CLASS->_one()

  Return an object representing the number one.

- CLASS->_two()

  Return an object representing the number two.

- CLASS->_ten()

  Return an object representing the number ten.

- CLASS->_from_bin(STR)

  Return an object given a string representing a binary number. The input has a '0b' prefix and matches the regular expression `^0[bB](0|1[01]*)$`.

- CLASS->_from_oct(STR)

  Return an object given a string representing an octal number. The input has a '0' prefix and matches the regular expression `^0[1-7]*$`.

- CLASS->_from_hex(STR)

  Return an object given a string representing a hexadecimal number. The input has a '0x' prefix and matches the regular expression `^0x(0|[1-9a-fA-F][\da-fA-F]*)$`.

- CLASS->_from_bytes(STR)

  Returns an object given a byte string representing the number. The byte string is in big endian byte order, so the two-byte input string "\x01\x00" should give an output value representing the number 256.

- CLASS->_from_base(STR, BASE, COLLSEQ)

  Returns an object given a string STR, a base BASE, and a collation sequence COLLSEQ. Each character in STR represents a numerical value identical to the character's position in COLLSEQ. All characters in STR must be present in COLLSEQ.

  If BASE is less than or equal to 94, and a collation sequence is not specified, the following default collation sequence is used. It contains of all the 94 printable ASCII characters except space/blank:

```
0123456789                  # ASCII  48 to  57
ABCDEFGHIJKLMNOPQRSTUVWXYZ  # ASCII  65 to  90
abcdefghijklmnopqrstuvwxyz  # ASCII  97 to 122
!"#$%&'()*+,-./             # ASCII  33 to  47
:;<=>?@                     # ASCII  58 to  64
[\]^_`                      # ASCII  91 to  96
{|}~                        # ASCII 123 to 126
```

  If the default collation sequence is used, and the BASE is less than or equal to 36, the letter case in STR is ignored.

  For instance, with base 3 and collation sequence "-/|", the character "-" represents 0, "/" represents 1, and "|" represents 2. So if STR is "/|-", the output is 1 * 3**2 + 2 * 3**1 + 0 * 3**0 = 15.

  The following examples show standard binary, octal, decimal, and hexadecimal conversion. All examples return 250.

```
$x = $class -> _from_base("11111010", 2)
$x = $class -> _from_base("372", 8)
$x = $class -> _from_base("250", 10)
$x = $class -> _from_base("FA", 16)
```

  Some more examples, all returning 250:

```
$x = $class -> _from_base("100021", 3)
$x = $class -> _from_base("3322", 4)
$x = $class -> _from_base("2000", 5)
$x = $class -> _from_base("caaa", 5, "abcde")
```

(continues on next page)

(continued from previous page)

```
$x = $class -> _from_base("42", 62)
$x = $class -> _from_base("2!", 94)
```

- CLASS->_from_base_num(ARRAY, BASE)

  Returns an object given an array of values and a base. This method is equivalent to `_from_base()`, but works on numbers in an array rather than characters in a string. Unlike `_from_base()`, all input values may be arbitrarily large.

```
$x = $class -> _from_base_num([1, 1, 0, 1], 2)    # $x is 13
$x = $class -> _from_base_num([3, 125, 39], 128)  # $x is 65191
```

Mathematical functions

- CLASS->_add(OBJ1, OBJ2)

  Addition. Returns the result of adding OBJ2 to OBJ1.

- CLASS->_mul(OBJ1, OBJ2)

  Multiplication. Returns the result of multiplying OBJ2 and OBJ1.

- CLASS->_div(OBJ1, OBJ2)

  Division. In scalar context, returns the quotient after dividing OBJ1 by OBJ2 and truncating the result to an integer. In list context, return the quotient and the remainder.

- CLASS->_sub(OBJ1, OBJ2, FLAG)

- CLASS->_sub(OBJ1, OBJ2)

  Subtraction. Returns the result of subtracting OBJ2 by OBJ1. If `flag` is false or omitted, OBJ1 might be modified. If `flag` is true, OBJ2 might be modified.

- CLASS->_sadd(OBJ1, SIGN1, OBJ2, SIGN2)

  Signed addition. Returns the result of adding OBJ2 with sign SIGN2 to OBJ1 with sign SIGN1.

```
($obj3, $sign3) = $class -> _sadd($obj1, $sign1, $obj2, $sign2);
```

- CLASS->_ssub(OBJ1, SIGN1, OBJ2, SIGN2)

  Signed subtraction. Returns the result of subtracting OBJ2 with sign SIGN2 to OBJ1 with sign SIGN1.

```
($obj3, $sign3) = $class -> _sadd($obj1, $sign1, $obj2, $sign2);
```

- CLASS->_dec(OBJ)

  Returns the result after decrementing OBJ by one.

- CLASS->_inc(OBJ)

  Returns the result after incrementing OBJ by one.

- CLASS->_mod(OBJ1, OBJ2)

  Returns OBJ1 modulo OBJ2, i.e., the remainder after dividing OBJ1 by OBJ2.

- CLASS->_sqrt(OBJ)

  Returns the square root of OBJ, truncated to an integer.

- CLASS->_root(OBJ, N)

  Returns the Nth root of OBJ, truncated to an integer.

- CLASS->_fac(OBJ)

  Returns the factorial of OBJ, i.e., the product of all positive integers up to and including OBJ.

- CLASS->_dfac(OBJ)

  Returns the double factorial of OBJ. If OBJ is an even integer, returns the product of all positive, even integers up to and including OBJ, i.e., 2*4*6*...*OBJ. If OBJ is an odd integer, returns the product of all positive, odd integers, i.e., 1*3*5*...*OBJ.

- CLASS->_pow(OBJ1, OBJ2)

  Returns OBJ1 raised to the power of OBJ2. By convention, 0**0 = 1.

- CLASS->_modinv(OBJ1, OBJ2)

  Returns the modular multiplicative inverse, i.e., return OBJ3 so that

  ```
  (OBJ3 * OBJ1) % OBJ2 = 1 % OBJ2
  ```

  The result is returned as two arguments. If the modular multiplicative inverse does not exist, both arguments are undefined. Otherwise, the arguments are a number (object) and its sign ("+" or "-").

  The output value, with its sign, must either be a positive value in the range 1,2,...,OBJ2-1 or the same value subtracted OBJ2. For instance, if the input arguments are objects representing the numbers 7 and 5, the method must either return an object representing the number 3 and a "+" sign, since (3*7) % 5 = 1

- CLASS->_modpow(OBJ1, OBJ2, OBJ3)

  Returns the modular exponentiation, i.e., (OBJ1 ** OBJ2) % OBJ3.

- CLASS->_rsft(OBJ, N, B)

  Returns the result after shifting OBJ N digits to thee right in base B. This is equivalent to performing integer division by B**N and discarding the remainder, except that it might be much faster.

  For instance, if the object $obj represents the hexadecimal number 0xabcde, then `_rsft($obj, 2, 16)` returns an object representing the number 0xabc. The "remainer", 0xde, is discarded and not returned.

- CLASS->_lsft(OBJ, N, B)

  Returns the result after shifting OBJ N digits to the left in base B. This is equivalent to multiplying by B**N, except that it might be much faster.

- CLASS->_log_int(OBJ, B)

  Returns the logarithm of OBJ to base BASE truncted to an integer. This method has two output arguments, the OBJECT and a STATUS. The STATUS is Perl scalar; it is 1 if OBJ is the exact result, 0 if the result was truncted to give OBJ, and undef if it is unknown whether OBJ is the exact result.

- CLASS->_ilog2(OBJ)

  Returns the base 2 logarithm of OBJ rounded downwards to the nearest integer, i.e., `int(log2(OBJ))`. In list context, this method returns two output arguments, the OBJECT and a STATUS. The STATUS is Perl scalar; it is 1 if OBJ is the exact result, 0 if the result was truncted to give OBJ, and undef if it is unknown whether OBJ is the exact result.

  This method is equivalent to the more general method _log_int() when it is used with base 2 argument, but _ilog2() method might be faster.

- CLASS->_ilog10(OBJ)

  Returns the base 10 logarithm of OBJ rounded downwards to the nearest integer, i.e., `int(log2(OBJ))`. In list context, this method returns two output arguments, the OBJECT and a STATUS. The STATUS is Perl scalar; it is 1 if OBJ is the exact result, 0 if the result was truncted to give OBJ, and undef if it is unknown whether OBJ is the exact result.

  This method is equivalent to the more general method _log_int() when it is used with base 10 argument, but _ilog10() method might be faster.

  Also, the output from _ilog10() is always 1 smaller than the output from _len().

- CLASS->_clog2(OBJ)

  Returns the base 2 logarithm of OBJ rounded upwards to the nearest integer, i.e., `ceil(log2(OBJ))`. In list context, this method returns two output arguments, the OBJECT and a STATUS. The STATUS is Perl scalar; it is 1 if OBJ is the exact result, 0 if the result was truncted to give OBJ, and undef if it is unknown whether OBJ is the exact result.

- CLASS->_clog10(OBJ)

  Returns the base 10 logarithm of OBJ rounded upnwards to the nearest integer, i.e., `ceil(log2(OBJ))`. In list context, this method returns two output arguments, the OBJECT and a STATUS. The STATUS is Perl scalar; it is 1 if OBJ is the exact result, 0 if the result was truncted to give OBJ, and undef if it is unknown whether OBJ is the exact result.

- CLASS->_gcd(OBJ1, OBJ2)

  Returns the greatest common divisor of OBJ1 and OBJ2.

- CLASS->_lcm(OBJ1, OBJ2)

  Return the least common multiple of OBJ1 and OBJ2.

- CLASS->_fib(OBJ)

  In scalar context, returns the nth Fibonacci number: _fib(0) returns 0, _fib(1) returns 1, _fib(2) returns 1, _fib(3) returns 2 etc. In list context, returns the Fibonacci numbers from F(0) to F(n): 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

- CLASS->_lucas(OBJ)

  In scalar context, returns the nth Lucas number: _lucas(0) returns 2, _lucas(1) returns 1, _lucas(2) returns 3, etc. In list context, returns the Lucas numbers from L(0) to L(n): 2, 1, 3, 4, 7, 11, 18, 29,47, 76, ...

Bitwise operators

- CLASS->_and(OBJ1, OBJ2)

  Returns bitwise and.

- CLASS->_or(OBJ1, OBJ2)

  Returns bitwise or.

- CLASS->_xor(OBJ1, OBJ2)

  Returns bitwise exclusive or.

- CLASS->_sand(OBJ1, OBJ2, SIGN1, SIGN2)

  Returns bitwise signed and.

- CLASS->_sor(OBJ1, OBJ2, SIGN1, SIGN2)

  Returns bitwise signed or.

- CLASS->_sxor(OBJ1, OBJ2, SIGN1, SIGN2)

  Returns bitwise signed exclusive or.

Boolean operators

- CLASS->_is_zero(OBJ)

  Returns a true value if OBJ is zero, and false value otherwise.

- CLASS->_is_one(OBJ)

  Returns a true value if OBJ is one, and false value otherwise.

- CLASS->_is_two(OBJ)

  Returns a true value if OBJ is two, and false value otherwise.

- CLASS->_is_ten(OBJ)

  Returns a true value if OBJ is ten, and false value otherwise.

- CLASS->_is_even(OBJ)

  Return a true value if OBJ is an even integer, and a false value otherwise.

- CLASS->_is_odd(OBJ)

  Return a true value if OBJ is an even integer, and a false value otherwise.

- CLASS->_acmp(OBJ1, OBJ2)

  Compare OBJ1 and OBJ2 and return -1, 0, or 1, if OBJ1 is numerically less than, equal to, or larger than OBJ2, respectively.

- CLASS->_scmp(OBJ1, SIGN1, OBJ2, SIGN2)

  Compare OBJ1 and OBJ2 and return -1, 0, or 1, if OBJ1 is numerically less than, equal to, or larger than OBJ2, respectively.

String conversion

- CLASS->_str(OBJ)

  Returns a string representing OBJ in decimal notation. The returned string should have no leading zeros, i.e., it should match `^(0|[1-9]\d*)$`.

- CLASS->_to_bin(OBJ)

  Returns the binary string representation of OBJ.

- CLASS->_to_oct(OBJ)

  Returns the octal string representation of the number.

- CLASS->_to_hex(OBJ)

  Returns the hexadecimal string representation of the number.

- CLASS->_to_bytes(OBJ)

  Returns a byte string representation of OBJ. The byte string is in big endian byte order, so if OBJ represents the number 256, the output should be the two-byte string "\x01\x00".

- CLASS->_to_base(OBJ, BASE, COLLSEQ)

  Returns a string representation of OBJ in base BASE with collation sequence COLLSEQ.

```
$val = $class -> _new("210");
$str = $class -> _to_base($val, 10, "xyz")  # $str is "zyx"

$val = $class -> _new("32");
$str = $class -> _to_base($val, 2, "-|")  # $str is "|-----"
```

  See _from_base() for more information.

- CLASS->_to_base_num(OBJ, BASE)

  Converts the given number to the given base. This method is equivalent to `_to_base()`, but returns numbers in an array rather than characters in a string. In the output, the first element is the most significant. Unlike `_to_base()`, all input values may be arbitrarily large.

```
$x = $class -> _to_base_num(13, 2)        # $x is [1, 1, 0, 1]
$x = $class -> _to_base_num(65191, 128)   # $x is [3, 125, 39]
```

- CLASS->_as_bin(OBJ)

  Like `_to_bin()` but with a '0b' prefix.

- CLASS->_as_oct(OBJ)

  Like `_to_oct()` but with a '0' prefix.

- CLASS->_as_hex(OBJ)

  Like `_to_hex()` but with a '0x' prefix.

- CLASS->_as_bytes(OBJ)

  This is an alias to `_to_bytes()`.

Numeric conversion

- CLASS->_num(OBJ)

  Returns a Perl scalar number representing the number OBJ as close as possible. Since Perl scalars have limited precision, the returned value might not be exactly the same as OBJ.

Miscellaneous

- CLASS->_copy(OBJ)

  Returns a true copy OBJ.

- CLASS->_len(OBJ)

  Returns the number of the decimal digits in OBJ. The output is a Perl scalar.

- CLASS->_zeros(OBJ)

  Returns the number of trailing decimal zeros. The output is a Perl scalar. The number zero has no trailing decimal zeros.

- CLASS->_digit(OBJ, N)

  Returns the Nth digit in OBJ as a Perl scalar. N is a Perl scalar, where zero refers to the rightmost (least significant) digit, and negative values count from the left (most significant digit). If $obj represents the number 123, then

```
CLASS->_digit($obj,  0)     # returns 3
CLASS->_digit($obj,  1)     # returns 2
CLASS->_digit($obj,  2)     # returns 1
CLASS->_digit($obj, -1)     # returns 1
```

- CLASS->_digitsum(OBJ)

  Returns the sum of the base 10 digits.

- CLASS->_check(OBJ)

  Returns true if the object is invalid and false otherwise. Preferably, the true value is a string describing the problem with the object. This is a check routine to test the internal state of the object for corruption.

- CLASS->_set(OBJ)

  xxx

API version 2

The following methods are required for an API version of 2 or greater.

Constructors

- CLASS->_1ex(N)

  Return an object representing the number 10**N where N >= 0 is a Perl scalar.

Mathematical functions

- CLASS->_nok(OBJ1, OBJ2)

  Return the binomial coefficient OBJ1 over OBJ1.

Miscellaneous

- CLASS->_alen(OBJ)

  Return the approximate number of decimal digits of the object. The output is a Perl scalar.

WRAP YOUR OWN

If you want to port your own favourite C library for big numbers to the Math::BigInt interface, you can take any of the already existing modules as a rough guideline. You should really wrap up the latest Math::BigInt and Math::BigFloat testsuites with your module, and replace in them any of the following:

```
use Math::BigInt;
```

by this:

```
use Math::BigInt lib => 'yourlib';
```

This way you ensure that your library really works 100% within Math::BigInt.

BUGS

Please report any bugs or feature requests to `bug-math-bigint at rt.cpan.org`, or through the web interface at https://rt.cpan.org/Ticket/Create.html?Queue=Math-BigInt (requires login). We will be notified, and then you'll automatically be notified of progress on your bug as I make changes.

SUPPORT

You can find documentation for this module with the perldoc command.

```
perldoc Math::BigInt::Calc
```

You can also look for information at:

- GitHub Source Repository

  https://github.com/pjacklam/p5-Math-BigInt

- RT: CPAN's request tracker

  https://rt.cpan.org/Public/Dist/Display.html?Name=Math-BigInt

- MetaCPAN

  https://metacpan.org/release/Math-BigInt

- CPAN Testers Matrix

  http://matrix.cpantesters.org/?dist=Math-BigInt

LICENSE

This program is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

AUTHOR

Peter John Acklam, pjacklam@gmail.com

Code and documentation based on the Math::BigInt::Calc module by Tels nospam-abuse@bloodgate.com

SEE ALSO

Math::BigInt, *Math::BigInt::Calc*, Math::BigInt::GMP, *Math::BigInt::FastCalc* and Math::BigInt::Pari.

Math::BigRat

arbitrary size rational number math package

SYNOPSIS

```
use Math::BigRat;

# Generic constructor method (always returns a new object)

$x = Math::BigRat->new($str);              # defaults to 0
$x = Math::BigRat->new('256');             # from decimal
$x = Math::BigRat->new('0256');            # from decimal
$x = Math::BigRat->new('0xcafe');          # from hexadecimal
$x = Math::BigRat->new('0x1.fap+7');       # from hexadecimal
$x = Math::BigRat->new('0o377');           # from octal
$x = Math::BigRat->new('0o1.35p+6');       # from octal
$x = Math::BigRat->new('0b101');           # from binary
$x = Math::BigRat->new('0b1.101p+3');      # from binary
```

(continues on next page)

(continued from previous page)

```
# Specific constructor methods (no prefix needed; when used as
# instance method, the value is assigned to the invocand)

$x = Math::BigRat->from_dec('234');       # from decimal
$x = Math::BigRat->from_hex('cafe');      # from hexadecimal
$x = Math::BigRat->from_hex('1.fap+7');   # from hexadecimal
$x = Math::BigRat->from_oct('377');       # from octal
$x = Math::BigRat->from_oct('1.35p+6');   # from octal
$x = Math::BigRat->from_bin('1101');      # from binary
$x = Math::BigRat->from_bin('1.101p+3');  # from binary
$x = Math::BigRat->from_bytes($bytes);    # from byte string
$x = Math::BigRat->from_base('why', 36);  # from any base
$x = Math::BigRat->from_base_num([1, 0], 2);  # from any base
$x = Math::BigRat->from_ieee754($b, $fmt);    # from IEEE-754 bytes
$x = Math::BigRat->from_fp80($b);         # from x86 80-bit
$x = Math::BigRat->bzero();               # create a +0
$x = Math::BigRat->bone();                # create a +1
$x = Math::BigRat->bone('-');             # create a -1
$x = Math::BigRat->binf();                # create a +inf
$x = Math::BigRat->binf('-');             # create a -inf
$x = Math::BigRat->bnan();                # create a Not-A-Number
$x = Math::BigRat->bpi();                 # returns pi

$y = $x->copy();         # make a copy (unlike $y = $x)
$y = $x->as_int();       # return as a Math::BigInt
$y = $x->as_float();     # return as a Math::BigFloat
$y = $x->as_rat();       # return as a Math::BigRat

# Boolean methods (these don't modify the invocand)

$x->is_zero();           # true if $x is 0
$x->is_one();            # true if $x is +1
$x->is_one("+");         # true if $x is +1
$x->is_one("-");         # true if $x is -1
$x->is_inf();            # true if $x is +inf or -inf
$x->is_inf("+");         # true if $x is +inf
$x->is_inf("-");         # true if $x is -inf
$x->is_nan();            # true if $x is NaN

$x->is_finite();         # true if -inf < $x < inf
$x->is_positive();       # true if $x > 0
$x->is_pos();            # true if $x > 0
$x->is_negative();       # true if $x < 0
$x->is_neg();            # true if $x < 0
$x->is_non_positive()    # true if $x <= 0
$x->is_non_negative()    # true if $x >= 0

$x->is_odd();            # true if $x is odd
$x->is_even();           # true if $x is even
```

(continues on next page)

(continued from previous page)

```
$x->is_int();           # true if $x is an integer

# Comparison methods (these don't modify the invocand)

$x->bcmp($y);           # compare numbers (undef, < 0, == 0, > 0)
$x->bacmp($y);          # compare abs values (undef, < 0, == 0, > 0)
$x->beq($y);            # true if $x == $y
$x->bne($y);            # true if $x != $y
$x->blt($y);            # true if $x < $y
$x->ble($y);            # true if $x <= $y
$x->bgt($y);            # true if $x > $y
$x->bge($y);            # true if $x >= $y

# Arithmetic methods (these modify the invocand)

$x->bneg();             # negation
$x->babs();             # absolute value
$x->bsgn();             # sign function (-1, 0, 1, or NaN)
$x->bdigitsum();        # sum of decimal digits
$x->binc();             # increment $x by 1
$x->bdec();             # decrement $x by 1
$x->badd($y);           # addition (add $y to $x)
$x->bsub($y);           # subtraction (subtract $y from $x)
$x->bmul($y);           # multiplication (multiply $x by $y)
$x->bmuladd($y, $z);    # $x = $x * $y + $z
$x->bdiv($y);           # division (floored)
$x->bmod($y);           # modulus (x % y)
$x->bmodinv($mod);      # modular multiplicative inverse
$x->bmodpow($y, $mod);  # modular exponentiation (($x ** $y) % $mod)
$x->btdiv($y);          # division (truncated), set $x to quotient
$x->btmod($y);          # modulus (truncated)
$x->binv()              # inverse (1/$x)
$x->bpow($y);           # power of arguments (x ** y)
$x->blog();             # logarithm of $x to base e (Euler's number)
$x->blog($base);        # logarithm of $x to base $base (e.g., base
 ↪2)
$x->bexp();             # calculate e ** $x where e is Euler's
 ↪number
$x->bilog2();           # log2($x) rounded down to nearest int
$x->bilog10();          # log10($x) rounded down to nearest int
$x->bclog2();           # log2($x) rounded up to nearest int
$x->bclog10();          # log10($x) rounded up to nearest int
$x->bnok($y);           # combinations (binomial coefficient n over
 ↪k)
$x->bperm($y);          # permutations
$x->buparrow($n, $y);   # Knuth's up-arrow notation
$x->bhyperop($n, $y);   # n'th hyperoprator
$x->backermann($y);     # the Ackermann function
$x->bsin();             # sine
```

(continues on next page)

(continued from previous page)

```
$x->bcos();             # cosine
$x->batan();            # inverse tangent
$x->batan2($y);         # two-argument inverse tangent
$x->bsqrt();            # calculate square root
$x->broot($y);          # $y'th root of $x (e.g. $y == 3 => cubic root)
$x->bfac();             # factorial of $x (1*2*3*4*..$x)
$x->bdfac();            # double factorial of $x ($x*($x-2)*($x-4)*..)
$x->btfac();            # triple factorial of $x ($x*($x-3)*($x-6)*..)
$x->bmfac($k);          # $k'th multi-factorial of $x ($x*($x-$k)*..)
$x->bfib($k);           # $k'th Fibonacci number
$x->blucas($k);         # $k'th Lucas number

$x->blsft($n);          # left shift $n places in base 2
$x->blsft($n, $b);      # left shift $n places in base $b
$x->brsft($n);          # right shift $n places in base 2
$x->brsft($n, $b);      # right shift $n places in base $b

# Bitwise methods (these modify the invocand)

$x->bblsft($y);         # bitwise left shift
$x->bbrsft($y);         # bitwise right shift
$x->band($y);           # bitwise and
$x->bior($y);           # bitwise inclusive or
$x->bxor($y);           # bitwise exclusive or
$x->bnot();             # bitwise not (two's complement)

# Rounding methods (these modify the invocand)

$x->round($A, $P, $R);  # round to accuracy or precision using
                        #   rounding mode $R
$x->bround($n);         # accuracy: preserve $n digits
$x->bfround($n);        # $n > 0: round to $nth digit left of dec. point
                        # $n < 0: round to $nth digit right of dec. point
$x->bfloor();           # round towards minus infinity
$x->bceil();            # round towards plus infinity
$x->bint();             # round towards zero

# Other mathematical methods (these don't modify the invocand)

$x->bgcd($y);           # greatest common divisor
$x->blcm($y);           # least common multiple

# Object property methods (these don't modify the invocand)
```

(continues on next page)

(continued from previous page)

```
$x->sign();             # the sign, either +, - or NaN
$x->digit($n);          # the nth digit, counting from the right
$x->digit(-$n);         # the nth digit, counting from the left
$x->digitsum();         # sum of decimal digits
$x->length();           # return number of digits in number
$x->mantissa();         # return (signed) mantissa as a Math::BigInt
$x->exponent();         # return exponent as a Math::BigInt
$x->parts();            # return (mantissa,exponent) as a␣
↪Math::BigInt
$x->sparts();           # mantissa and exponent (as integers)
$x->nparts();           # mantissa and exponent (normalised)
$x->eparts();           # mantissa and exponent (engineering␣
↪notation)
$x->dparts();           # integer and fraction part
$x->fparts();           # numerator and denominator
$x->numerator();        # numerator
$x->denominator();      # denominator

# Conversion methods (these don't modify the invocand)

$x->bstr();             # decimal notation (possibly zero padded)
$x->bnstr();            # string in normalized notation
$x->bestr();            # string in engineering notation
$x->bdstr();            # string in decimal notation (no padding)
$x->bfstr();            # string in fractional notation

$x->to_hex();           # as signed hexadecimal string
$x->to_bin();           # as signed binary string
$x->to_oct();           # as signed octal string
$x->to_bytes();         # as byte string
$x->to_base($b);        # as string in any base
$x->to_base_num($b);    # as array of integers in any base
$x->to_ieee754($fmt);   # to bytes encoded according to IEEE 754-
↪2008
$x->to_fp80();          # encode value in x86 80-bit format

$x->as_hex();           # as signed hexadecimal string with "0x"␣
↪prefix
$x->as_bin();           # as signed binary string with "0b" prefix
$x->as_oct();           # as signed octal string with "0" prefix

# Other conversion methods (these don't modify the invocand)

$x->numify();           # return as scalar (might overflow or␣
↪underflow)
```

DESCRIPTION

Math::BigRat complements *Math::BigInt* and *Math::BigFloat* by providing support for arbitrary big rational numbers.

Math Library

You can change the underlying module that does the low-level math operations by using:

```
use Math::BigRat try => 'GMP';
```

Note: This needs Math::BigInt::GMP installed.

The following would first try to find Math::BigInt::Foo, then Math::BigInt::Bar, and when this also fails, revert to Math::BigInt::Calc:

```
use Math::BigRat try => 'Foo,Math::BigInt::Bar';
```

If you want to get warned when the fallback occurs, replace “try” with “lib”:

```
use Math::BigRat lib => 'Foo,Math::BigInt::Bar';
```

If you want the code to die instead, replace “try” with “only”:

```
use Math::BigRat only => 'Foo,Math::BigInt::Bar';
```

METHODS

Any methods not listed here are derived from Math::BigFloat (or Math::BigInt), so make sure you check these two modules for further information.

- new()

  ```
  $x = Math::BigRat->new('1/3');
  ```

  Create a new Math::BigRat object. Input can come in various forms:

  ```
  $x = Math::BigRat->new(123);                                #␣
  ↪scalars
  $x = Math::BigRat->new('inf');                              #␣
  ↪infinity
  $x = Math::BigRat->new('123.3');                            # float
  $x = Math::BigRat->new('1/3');                              #␣
  ↪simple string
  $x = Math::BigRat->new('1 / 3');                            # spaced
  $x = Math::BigRat->new('1 / 0.1');                          # w/␣
  ↪floats
  $x = Math::BigRat->new(Math::BigInt->new(3));               # BigInt
  $x = Math::BigRat->new(Math::BigFloat->new('3.1'));         #␣
  ↪BigFloat
  $x = Math::BigRat->new(Math::BigInt::Lite->new('2'));       #␣
  ↪BigLite
  ```

(continues on next page)

(continued from previous page)

```
# You can also give D and N as different objects:
$x = Math::BigRat->new(
        Math::BigInt->new(-123),
        Math::BigInt->new(7),
     );                         # => -123/7
```

- from_dec()

```
my $h = Math::BigRat->from_dec("1.2");
```

Create a BigRat from a decimal number in string form. It is equivalent to "new()", but does not accept anything but strings representing finite, decimal numbers.

- from_hex()

```
my $h = Math::BigRat->from_hex("0x10");
```

Create a BigRat from a hexadecimal number in string form.

- from_oct()

```
my $o = Math::BigRat->from_oct("020");
```

Create a BigRat from an octal number in string form.

- from_bin()

```
my $b = Math::BigRat->from_bin("0b10000000");
```

Create a BigRat from an binary number in string form.

- from_bytes()

```
$x = Math::BigRat->from_bytes("\xf3\x6b");  # $x = 62315
```

Interpret the input as a byte string, assuming big endian byte order. The output is always a non-negative, finite integer.

See *"from_bytes()" in Math::BigInt*.

- from_ieee754()

```
# set $x to 13176795/4194304, the closest value to pi that can␣
↪be
# represented in the binary32 (single) format
$x = Math::BigRat -> from_ieee754("40490fdb", "binary32");
```

Interpret the input as a value encoded as described in IEEE754-2008.

See *"from_ieee754()" in Math::BigFloat*.

- from_fp80()

```
# set $x to 14488038916154245685/4611686018427387904, the
 ↪closest value
# to pi that can be represented in the x86 80-bit format
$x = Math::BigRat -> from_fp80("4000c90fdaa22168c235");
```

Interpret the input as a value encoded in the x86 extended-precision 80-bit format.

See *"from_fp80()" in Math::BigFloat*.

- from_base()

  See *"from_base()" in Math::BigInt*.

- bzero()

```
$x = Math::BigRat->bzero();
```

Creates a new BigRat object representing zero. If used on an object, it will set it to zero:

```
$x->bzero();
```

- bone()

```
$x = Math::BigRat->bone($sign);
```

Creates a new BigRat object representing one. The optional argument is either '-' or '+', indicating whether you want one or minus one. If used on an object, it will set it to one:

```
$x->bone();                 # +1
$x->bone('-');              # -1
```

- binf()

```
$x = Math::BigRat->binf($sign);
```

Creates a new BigRat object representing infinity. The optional argument is either '-' or '+', indicating whether you want infinity or minus infinity. If used on an object, it will set it to infinity:

```
$x->binf();
$x->binf('-');
```

- bnan()

```
$x = Math::BigRat->bnan();
```

Creates a new BigRat object representing NaN (Not A Number). If used on an object, it will set it to NaN:

```
$x->bnan();
```

- bpi()

```
$x = Math::BigRat -> bpi();         # default accuracy
$x = Math::BigRat -> bpi(7);        # specified accuracy
```

Returns a rational approximation of PI accurate to the specified accuracy or the default accuracy if no accuracy is specified. If called as an instance method, the value is assigned to the invocand.

```
$x = Math::BigRat -> bpi(1);        # returns "3"
$x = Math::BigRat -> bpi(3);        # returns "22/7"
$x = Math::BigRat -> bpi(7);        # returns "355/113"
```

- copy()

```
my $z = $x->copy();
```

Makes a deep copy of the object.

Please see the documentation in *Math::BigInt* for further details.

- as_int()

```
$y = $x -> as_int();        # $y is a Math::BigInt
```

Returns $x as a Math::BigInt object regardless of upgrading and downgrading. If $x is finite, but not an integer, $x is truncated.

- as_rat()

```
$y = $x -> as_rat();        # $y is a Math::BigRat
```

Returns $x a Math::BigRat object regardless of upgrading and downgrading. The invocand is not modified.

- as_float()

```
$x = Math::BigRat->new('13/7');
print $x->as_float(), "\n";             # '1'

$x = Math::BigRat->new('2/3');
print $x->as_float(5), "\n";            # '0.66667'
```

Returns a copy of the object as Math::BigFloat object regardless of upgrading and downgrading, preserving the accuracy as wanted, or the default of 40 digits.

- bround()/round()/bfround()

Are not yet implemented.

- is_zero()

```
print "$x is 0\n" if $x->is_zero();
```

Return true if $x is exactly zero, otherwise false.

- is_one()

```
print "$x is 1\n" if $x->is_one();
```

Return true if $x is exactly one, otherwise false.

- is_finite()

```
$x->is_finite();    # true if $x is not +inf, -inf or NaN
```

Returns true if the invocand is a finite number, i.e., it is neither +inf, -inf, nor NaN.

- is_positive()
- is_pos()

```
print "$x is >= 0\n" if $x->is_positive();
```

Return true if $x is positive (greater than or equal to zero), otherwise false. Please note that '+inf' is also positive, while 'NaN' and '-inf' aren't.

"is_positive()" is an alias for "is_pos()".

- is_negative()
- is_neg()

```
print "$x is < 0\n" if $x->is_negative();
```

Return true if $x is negative (smaller than zero), otherwise false. Please note that '-inf' is also negative, while 'NaN' and '+inf' aren't.

"is_negative()" is an alias for "is_neg()".

- is_odd()

```
print "$x is odd\n" if $x->is_odd();
```

Return true if $x is odd, otherwise false.

- is_even()

```
print "$x is even\n" if $x->is_even();
```

Return true if $x is even, otherwise false.

- is_int()

```
print "$x is an integer\n" if $x->is_int();
```

Return true if $x has a denominator of 1 (e.g. no fraction parts), otherwise false. Please note that '-inf', 'inf' and 'NaN' aren't integer.

Comparison methods

None of these methods modify the invocand object. Note that a `NaN` is neither less than, greater than, or equal to anything else, even a `NaN`.

- bcmp()

```
$x->bcmp($y);
```

Compares $x with $y and takes the sign into account. Returns -1, 0, 1 or undef.

- bacmp()

```
$x->bacmp($y);
```

Compares $x with $y while ignoring their sign. Returns -1, 0, 1 or undef.

- beq()

```
$x -> beq($y);
```

Returns true if and only if $x is equal to $y, and false otherwise.

- bne()

```
$x -> bne($y);
```

Returns true if and only if $x is not equal to $y, and false otherwise.

- blt()

```
$x -> blt($y);
```

Returns true if and only if $x is equal to $y, and false otherwise.

- ble()

```
$x -> ble($y);
```

Returns true if and only if $x is less than or equal to $y, and false otherwise.

- bgt()

```
$x -> bgt($y);
```

Returns true if and only if $x is greater than $y, and false otherwise.

- bge()

```
$x -> bge($y);
```

Returns true if and only if $x is greater than or equal to $y, and false otherwise.

- blsft()/brsft()

Used to shift numbers left/right.

Please see the documentation in *Math::BigInt* for further details.

- bneg()

```
$x->bneg();
```

Used to negate the object in-place.

- bnorm()

```
$x->bnorm();
```

Reduce the number to the shortest form. This routine is called automatically whenever it is needed.

- binc()

```
$x->binc();
```

Increments $x by 1 and returns the result.

- bdec()

```
$x->bdec();
```

Decrements $x by 1 and returns the result.

- badd()

```
$x->badd($y);
```

Adds $y to $x and returns the result.

- bsub()

```
$x->bsub($y);
```

Subtracts $y from $x and returns the result.

- bmul()

```
$x->bmul($y);
```

Multiplies $y to $x and returns the result.

- bdiv()

```
$q = $x->bdiv($y);
($q, $r) = $x->bdiv($y);
```

In scalar context, divides $x by $y and returns the result. In list context, does floored division (F-division), returning an integer $q and a remainder $r so that `$x = $q * $y + $r`. The remainer (modulo) is equal to what is returned by `$x->bmod($y)`.

- bmod()

```
$x->bmod($y);
```

Returns $x modulo $y. When $x is finite, and $y is finite and non-zero, the result is identical to the remainder after floored division (F-division). If, in addition, both $x and $y are integers, the result is identical to the result from Perl's % operator.

- binv()

```
$x->binv();
```

Inverse of $x.

- bsqrt()

```
$x->bsqrt();
```

Calculate the square root of $x.

- bpow()

```
$x->bpow($y);
```

Compute $x ** $y.

Please see the documentation in *Math::BigInt* for further details.

- broot()

```
$x->broot($n);
```

Calculate the N'th root of $x.

- bmodpow()

```
$num->bmodpow($exp,$mod);           # modular exponentiation
                                    # ($num**$exp % $mod)
```

Returns the value of `$num` taken to the power `$exp` in the modulus `$mod` using binary exponentiation. `bmodpow` is far superior to writing

```
$num ** $exp % $mod
```

because it is much faster - it reduces internal variables into the modulus whenever possible, so it operates on smaller numbers.

`bmodpow` also supports negative exponents.

```
bmodpow($num, -1, $mod)
```

is exactly equivalent to

```
bmodinv($num, $mod)
```

- bmodinv()

```
$x->bmodinv($mod);          # modular multiplicative inverse
```

Returns the multiplicative inverse of `$x` modulo `$mod`. If

```
$y = $x -> copy() -> bmodinv($mod)
```

then `$y` is the number closest to zero, and with the same sign as `$mod`, satisfying

```
($x * $y) % $mod = 1 % $mod
```

If $x and $y are non-zero, they must be relative primes, i.e., bgcd($y, $mod)==1. 'NaN' is returned when no modular multiplicative inverse exists.

- blog()

```
$x->blog($base, $accuracy);         # logarithm of x to the
 ↪base $base
```

If $base is not defined, Euler's number (e) is used:

```
print $x->blog(undef, 100);         # log(x) to 100 digits
```

- bexp()

```
$x->bexp($accuracy);        # calculate e ** X
```

Calculates two integers A and B so that A/B is equal to e ** $x, where e is Euler's number.

This method was added in v0.20 of Math::BigRat (May 2007).

See also "blog()".

- bnok()

See *"bnok()" in Math::BigInt*.

- bperm()

See *"bperm()" in Math::BigInt*.

- bfac()

```
$x->bfac();
```

Calculates the factorial of $x. For instance:

```
print Math::BigRat->new('3/1')->bfac(), "\n";   # 1*2*3
print Math::BigRat->new('5/1')->bfac(), "\n";   # 1*2*3*4*5
```

Works currently only for integers.

- band()

```
$x->band($y);               # bitwise and
```

- bior()

```
$x->bior($y);               # bitwise inclusive or
```

- bxor()

```
$x->bxor($y);               # bitwise exclusive or
```

- bnot()

```
$x->bnot();                    # bitwise not (two's complement)
```

- bfloor()

```
$x->bfloor();
```

Round $x towards minus infinity, i.e., set $x to the largest integer less than or equal to $x.

- bceil()

```
$x->bceil();
```

Round $x towards plus infinity, i.e., set $x to the smallest integer greater than or equal to $x.

- bint()

```
$x->bint();
```

Round $x towards zero.

- bgcd()

```
$x -> bgcd($y);               # GCD of $x and $y
$x -> bgcd($y, $z, ...);      # GCD of $x, $y, $z, ...
```

Returns the greatest common divisor (GCD), which is the number with the largest absolute value such that $x/$gcd, $y/$gcd, ... is an integer. For example, when the operands are 4/5 and 6/5, the GCD is 2/5. This is a generalisation of the ordinary GCD for integers. See *"gcd()" in Math::BigInt*.

- digit()

```
print Math::BigRat->new('123/1')->digit(1);     # 1
print Math::BigRat->new('123/1')->digit(-1);    # 3
```

Return the N'ths digit from X when X is an integer value.

- length()

```
$len = $x->length();
```

Return the length of $x in digits for integer values.

- parts()

```
($n, $d) = $x->parts();
```

Return a list consisting of (signed) numerator and (unsigned) denominator as BigInts.

- dparts()

Returns the integer part and the fraction part.

- fparts()

  Returns the smallest possible numerator and denominator so that the numerator divided by the denominator gives back the original value. For finite numbers, both values are integers. Mnemonic: fraction.

- numerator()

```
$n = $x->numerator();
```

  Returns a copy of the numerator (the part above the line) as signed BigInt.

- denominator()

```
$d = $x->denominator();
```

  Returns a copy of the denominator (the part under the line) as positive BigInt.

String conversion methods

- bstr()

```
my $x = Math::BigRat->new('8/4');
print $x->bstr(), "\n";             # prints 1/2
```

  Returns a string representing the number.

- bnstr()

  See *“bnstr()” in Math::BigInt*.

- bestr()

  See *“bestr()” in Math::BigInt*.

- bdstr()

  See *“bdstr()” in Math::BigInt*.

- to_bytes()

  See *“to_bytes()” in Math::BigInt*.

- to_ieee754()

  See *“to_ieee754()” in Math::BigFloat*.

- to_fp80()

  See *“to_fp80()” in Math::BigFloat*.

- as_hex()

```
$x = Math::BigRat->new('13');
print $x->as_hex(), "\n";               # '0xd'
```

  Returns the BigRat as hexadecimal string. Works only for integers.

- as_oct()

```
$x = Math::BigRat->new('13');
print $x->as_oct(), "\n";               # '015'
```

Returns the BigRat as octal string. Works only for integers.

- as_bin()

```
$x = Math::BigRat->new('13');
print $x->as_bin(), "\n";               # '0x1101'
```

Returns the BigRat as binary string. Works only for integers.

- numify()

```
my $y = $x->numify();
```

Returns the object as a scalar. This will lose some data if the object cannot be represented by a normal Perl scalar (integer or float), so use "as_int()" or "as_float()" instead.

This routine is automatically used whenever a scalar is required:

```
my $x = Math::BigRat->new('3/1');
@array = (0, 1, 2, 3);
$y = $array[$x];                # set $y to 3
```

- config()

```
Math::BigRat->config("trap_nan" => 1);      # set
$accu = Math::BigRat->config("accuracy");   # get
```

Set or get configuration parameter values. Read-only parameters are marked as RO. Read-write parameters are marked as RW. The following parameters are supported.

```
Parameter       RO/RW   Description
                        Example
============================================================
lib             RO      Name of the math backend library
                        Math::BigInt::Calc
lib_version     RO      Version of the math backend library
                        0.30
class           RO      The class of config you just called
                        Math::BigRat
version         RO      version number of the class you used
                        0.10
upgrade         RW      To which class numbers are upgraded
                        undef
downgrade       RW      To which class numbers are downgraded
                        undef
precision       RW      Global precision
                        undef
accuracy        RW      Global accuracy
```

(continues on next page)

(continued from previous page)

```
                        undef
round_mode      RW      Global round mode
                        even
div_scale       RW      Fallback accuracy for div, sqrt etc.
                        40
trap_nan        RW      Trap NaNs
                        undef
trap_inf        RW      Trap +inf/-inf
                        undef
```

NUMERIC LITERALS

After `use Math::BigRat ':constant'` all numeric literals in the given scope are converted to `Math::BigRat` objects. This conversion happens at compile time. Every non-integer is convert to a NaN.

For example,

```
perl -MMath::BigRat=:constant -le 'print 2**150'
```

prints the exact value of `2**150`. Note that without conversion of constants to objects the expression `2**150` is calculated using Perl scalars, which leads to an inaccurate result.

Please note that strings are not affected, so that

```
use Math::BigRat qw/:constant/;

$x = "12345678901234567890123456789012345678901234567890"
        + "123456789123456789";
```

does give you what you expect. You need an explicit Math::BigRat->new() around at least one of the operands. You should also quote large constants to prevent loss of precision:

```
use Math::BigRat;

$x = Math::BigRat->new("12345678891234567891234567891234567891234567890");
```

Without the quotes Perl first converts the large number to a floating point constant at compile time, and then converts the result to a Math::BigRat object at run time, which results in an inaccurate result.

Hexadecimal, octal, and binary floating point literals

Perl (and this module) accepts hexadecimal, octal, and binary floating point literals, but use them with care with Perl versions before v5.32.0, because some versions of Perl silently give the wrong result. Below are some examples of different ways to write the number decimal 314.

Hexadecimal floating point literals:

```
0x1.3ap+8          0X1.3AP+8
0x1.3ap8           0X1.3AP8
0x13a0p-4          0X13A0P-4
```

Octal floating point literals (with "0" prefix):

```
01.164p+8          01.164P+8
01.164p8           01.164P8
011640p-4          011640P-4
```

Octal floating point literals (with "0o" prefix) (requires v5.34.0):

```
0o1.164p+8         0O1.164P+8
0o1.164p8          0O1.164P8
0o11640p-4         0O11640P-4
```

Binary floating point literals:

```
0b1.0011101p+8     0B1.0011101P+8
0b1.0011101p8      0B1.0011101P8
0b10011101000p-2   0B10011101000P-2
```

BUGS

Please report any bugs or feature requests to `bug-math-bigint at rt.cpan.org`, or through the web interface at https://rt.cpan.org/Ticket/Create.html?Queue=Math-BigInt (requires login). We will be notified, and then you'll automatically be notified of progress on your bug as I make changes.

SUPPORT

You can find documentation for this module with the perldoc command.

```
perldoc Math::BigInt
```

You can also look for information at:

- GitHub

  https://github.com/pjacklam/p5-Math-BigInt

- RT: CPAN's request tracker

  https://rt.cpan.org/Dist/Display.html?Name=Math-BigInt

- MetaCPAN

  https://metacpan.org/release/Math-BigInt

- CPAN Testers Matrix

  http://matrix.cpantesters.org/?dist=Math-BigInt

LICENSE

This program is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

SEE ALSO

Math::BigInt and *Math::BigFloat* as well as the backend libraries *Math::BigInt::FastCalc*, Math::BigInt::GMP, and Math::BigInt::Pari, Math::BigInt::GMPz, and Math::BigInt::BitVect.

The pragmas *bigint*, *bigfloat*, and *bigrat* might also be of interest. In addition there is the *bignum* pragma which does upgrading and downgrading.

AUTHORS

- Tels http://bloodgate.com/ 2001-2009.
- Maintained by Peter John Acklam pjacklam@gmail.com 2011-

Math::Complex

complex numbers and associated mathematical functions

SYNOPSIS

```
use Math::Complex;

$z = Math::Complex->make(5, 6);
$t = 4 - 3*i + $z;
$j = cplxe(1, 2*pi/3);
```

DESCRIPTION

This package lets you create and manipulate complex numbers. By default, *Perl* limits itself to real numbers, but an extra `use` statement brings full complex support, along with a full set of mathematical functions typically associated with and/or extended to complex numbers.

If you wonder what complex numbers are, they were invented to be able to solve the following equation:

```
x*x = -1
```

and by definition, the solution is noted *i* (engineers use *j* instead since *i* usually denotes an intensity, but the name does not matter). The number *i* is a pure *imaginary* number.

The arithmetics with pure imaginary numbers works just like you would expect it with real numbers... you just have to remember that

```
i*i = -1
```

so you have:

```
5i + 7i = i * (5 + 7) = 12i
4i - 3i = i * (4 - 3) = i
4i * 2i = -8
6i / 2i = 3
1 / i = -i
```

Complex numbers are numbers that have both a real part and an imaginary part, and are usually noted:

```
a + bi
```

where a is the *real* part and b is the *imaginary* part. The arithmetic with complex numbers is straightforward. You have to keep track of the real and the imaginary parts, but otherwise the rules used for real numbers just apply:

```
(4 + 3i) + (5 - 2i) = (4 + 5) + i(3 - 2) = 9 + i
(2 + i) * (4 - i) = 2*4 + 4i -2i -i*i = 8 + 2i + 1 = 9 + 2i
```

A graphical representation of complex numbers is possible in a plane (also called the *complex plane*, but it's really a 2D plane). The number

```
z = a + bi
```

is the point whose coordinates are (a, b). Actually, it would be the vector originating from (0, 0) to (a, b). It follows that the addition of two complex numbers is a vectorial addition.

Since there is a bijection between a point in the 2D plane and a complex number (i.e. the mapping is unique and reciprocal), a complex number can also be uniquely identified with polar coordinates:

```
[rho, theta]
```

where rho is the distance to the origin, and theta the angle between the vector and the *x* axis. There is a notation for this using the exponential form, which is:

```
rho * exp(i * theta)
```

where *i* is the famous imaginary number introduced above. Conversion between this form and the cartesian form a + bi is immediate:

```
a = rho * cos(theta)
b = rho * sin(theta)
```

which is also expressed by this formula:

```
z = rho * exp(i * theta) = rho * (cos theta + i * sin theta)
```

In other words, it's the projection of the vector onto the *x* and *y* axes. Mathematicians call *rho* the *norm* or *modulus* and *theta* the *argument* of the complex number. The *norm* of z is marked here as abs(z).

The polar notation (also known as the trigonometric representation) is much more handy for performing multiplications and divisions of complex numbers, whilst the cartesian notation is better suited for additions and subtractions. Real numbers are on the *x* axis, and therefore *y* or *theta* is zero or *pi*.

All the common operations that can be performed on a real number have been defined to work on complex numbers as well, and are merely *extensions* of the operations defined on real numbers. This means they keep their natural meaning when there is no imaginary part, provided the number is within their definition set.

For instance, the `sqrt` routine which computes the square root of its argument is only defined for non-negative real numbers and yields a non-negative real number (it is an application from **R**+ to **R**+). If we allow it to return a complex number, then it can be extended to negative real numbers to become an application from **R** to **C** (the set of complex numbers):

```
sqrt(x) = x >= 0 ? sqrt(x) : sqrt(-x)*i
```

It can also be extended to be an application from **C** to **C**, whilst its restriction to **R** behaves as defined above by using the following definition:

```
sqrt(z = [r,t]) = sqrt(r) * exp(i * t/2)
```

Indeed, a negative real number can be noted `[x,pi]` (the modulus *x* is always non-negative, so `[x,pi]` is really `-x`, a negative number) and the above definition states that

```
sqrt([x,pi]) = sqrt(x) * exp(i*pi/2) = [sqrt(x),pi/2] = sqrt(x)*i
```

which is exactly what we had defined for negative real numbers above. The `sqrt` returns only one of the solutions: if you want the both, use the `root` function.

All the common mathematical functions defined on real numbers that are extended to complex numbers share that same property of working *as usual* when the imaginary part is zero (otherwise, it would not be called an extension, would it?).

A *new* operation possible on a complex number that is the identity for real numbers is called the *conjugate*, and is noted with a horizontal bar above the number, or `~z` here.

```
 z = a + bi
~z = a - bi
```

Simple... Now look:

```
z * ~z = (a + bi) * (a - bi) = a*a + b*b
```

We saw that the norm of `z` was noted `abs(z)` and was defined as the distance to the origin, also known as:

```
rho = abs(z) = sqrt(a*a + b*b)
```

so

```
z * ~z = abs(z) ** 2
```

If z is a pure real number (i.e. `b == 0`), then the above yields:

```
a * a = abs(a) ** 2
```

which is true (`abs` has the regular meaning for real number, i.e. stands for the absolute value). This example explains why the norm of `z` is noted `abs(z)`: it extends the `abs` function to complex numbers, yet is the regular `abs` we know when the complex number actually has no imaginary part... This justifies *a posteriori* our use of the `abs` notation for the norm.

OPERATIONS

Given the following notations:

```
z1 = a + bi = r1 * exp(i * t1)
z2 = c + di = r2 * exp(i * t2)
z = <any complex or real number>
```

the following (overloaded) operations are supported on complex numbers:

```
z1 + z2 = (a + c) + i(b + d)
z1 - z2 = (a - c) + i(b - d)
z1 * z2 = (r1 * r2) * exp(i * (t1 + t2))
z1 / z2 = (r1 / r2) * exp(i * (t1 - t2))
z1 ** z2 = exp(z2 * log z1)
~z = a - bi
abs(z) = r1 = sqrt(a*a + b*b)
sqrt(z) = sqrt(r1) * exp(i * t/2)
exp(z) = exp(a) * exp(i * b)
log(z) = log(r1) + i*t
sin(z) = 1/2i (exp(i * z1) - exp(-i * z))
cos(z) = 1/2 (exp(i * z1) + exp(-i * z))
atan2(y, x) = atan(y / x) # Minding the right quadrant, note the␣
↪order.
```

The definition used for complex arguments of atan2() is

```
-i log((x + iy)/sqrt(x*x+y*y))
```

Note that atan2(0, 0) is not well-defined.

The following extra operations are supported on both real and complex numbers:

```
Re(z) = a
Im(z) = b
arg(z) = t
abs(z) = r

cbrt(z) = z ** (1/3)
log10(z) = log(z) / log(10)
logn(z, n) = log(z) / log(n)
```

(continues on next page)

(continued from previous page)

```
tan(z) = sin(z) / cos(z)

csc(z) = 1 / sin(z)
sec(z) = 1 / cos(z)
cot(z) = 1 / tan(z)

asin(z) = -i * log(i*z + sqrt(1-z*z))
acos(z) = -i * log(z + i*sqrt(1-z*z))
atan(z) = i/2 * log((i+z) / (i-z))

acsc(z) = asin(1 / z)
asec(z) = acos(1 / z)
acot(z) = atan(1 / z) = -i/2 * log((i+z) / (z-i))

sinh(z) = 1/2 (exp(z) - exp(-z))
cosh(z) = 1/2 (exp(z) + exp(-z))
tanh(z) = sinh(z) / cosh(z) = (exp(z) - exp(-z)) / (exp(z) + exp(-
 ↪z))

csch(z) = 1 / sinh(z)
sech(z) = 1 / cosh(z)
coth(z) = 1 / tanh(z)

asinh(z) = log(z + sqrt(z*z+1))
acosh(z) = log(z + sqrt(z*z-1))
atanh(z) = 1/2 * log((1+z) / (1-z))

acsch(z) = asinh(1 / z)
asech(z) = acosh(1 / z)
acoth(z) = atanh(1 / z) = 1/2 * log((1+z) / (z-1))
```

arg, *abs*, *log*, *csc*, *cot*, *acsc*, *acot*, *csch*, *coth*, *acosech*, *acotanh*, have aliases *rho*, *theta*, *ln*, *cosec*, *cotan*, *acosec*, *acotan*, *cosech*, *cotanh*, *acosech*, *acotanh*, respectively. `Re`, `Im`, `arg`, `abs`, `rho`, and `theta` can be used also as mutators. The `cbrt` returns only one of the solutions: if you want all three, use the `root` function.

The *root* function is available to compute all the *n* roots of some complex, where *n* is a strictly positive integer. There are exactly *n* such roots, returned as a list. Getting the number mathematicians call `j` such that:

```
1 + j + j*j = 0;
```

is a simple matter of writing:

```
$j = (root(1, 3))[1];
```

The _k_th root for `z = [r,t]` is given by:

```
(root(z, n))[k] = r**(1/n) * exp(i * (t + 2*k*pi)/n)
```

You can return the _k_th root directly by `root(z, n, k)`, indexing starting from *zero*

and ending at *n - 1*.

The *spaceship* numeric comparison operator, <=>, is also defined. In order to ensure its restriction to real numbers is conform to what you would expect, the comparison is run on the real part of the complex number first, and imaginary parts are compared only when the real parts match.

CREATION

To create a complex number, use either:

```
$z = Math::Complex->make(3, 4);
$z = cplx(3, 4);
```

if you know the cartesian form of the number, or

```
$z = 3 + 4*i;
```

if you like. To create a number using the polar form, use either:

```
$z = Math::Complex->emake(5, pi/3);
$x = cplxe(5, pi/3);
```

instead. The first argument is the modulus, the second is the angle (in radians, the full circle is 2*pi). (Mnemonic: `e` is used as a notation for complex numbers in the polar form).

It is possible to write:

```
$x = cplxe(-3, pi/4);
```

but that will be silently converted into `[3,-3pi/4]`, since the modulus must be non-negative (it represents the distance to the origin in the complex plane).

It is also possible to have a complex number as either argument of the `make`, `emake`, `cplx`, and `cplxe`: the appropriate component of the argument will be used.

```
$z1 = cplx(-2,  1);
$z2 = cplx($z1, 4);
```

The `new`, `make`, `emake`, `cplx`, and `cplxe` will also understand a single (string) argument of the forms

```
2-3i
-3i
[2,3]
[2,-3pi/4]
[2]
```

in which case the appropriate cartesian and exponential components will be parsed from the string and used to create new complex numbers. The imaginary component and the theta, respectively, will default to zero.

The `new`, `make`, `emake`, `cplx`, and `cplxe` will also understand the case of no arguments: this means plain zero or (0, 0).

DISPLAYING

When printed, a complex number is usually shown under its cartesian style *a+bi*, but there are legitimate cases where the polar style *[r,t]* is more appropriate. The process of converting the complex number into a string that can be displayed is known as *stringification*.

By calling the class method `Math::Complex::display_format` and supplying either `"polar"` or `"cartesian"` as an argument, you override the default display style, which is `"cartesian"`. Not supplying any argument returns the current settings.

This default can be overridden on a per-number basis by calling the `display_format` method instead. As before, not supplying any argument returns the current display style for this number. Otherwise whatever you specify will be the new display style for *this* particular number.

For instance:

```
use Math::Complex;

Math::Complex::display_format('polar');
$j = (root(1, 3))[1];
print "j = $j\n";               # Prints "j = [1,2pi/3]"
$j->display_format('cartesian');
print "j = $j\n";               # Prints "j = -0.5+0.
 ↪866025403784439i"
```

The polar style attempts to emphasize arguments like *k*pi/n* (where *n* is a positive integer and *k* an integer within [-9, +9]), this is called *polar pretty-printing*.

For the reverse of stringifying, see the `make` and `emake`.

CHANGED IN PERL 5.6

The `display_format` class method and the corresponding `display_format` object method can now be called using a parameter hash instead of just a one parameter.

The old display format style, which can have values `"cartesian"` or `"polar"`, can be changed using the `"style"` parameter.

```
$j->display_format(style => "polar");
```

The one parameter calling convention also still works.

```
$j->display_format("polar");
```

There are two new display parameters.

The first one is `"format"`, which is a sprintf()-style format string to be used for both numeric parts of the complex number(s). The is somewhat system-dependent but most often it corresponds to `"%.15g"`. You can revert to the default by setting the `format` to `undef`.

```
# the $j from the above example

$j->display_format('format' => '%.5f');
print "j = $j\n";               # Prints "j = -0.50000+0.86603i"
$j->display_format('format' => undef);
print "j = $j\n";               # Prints "j = -0.5+0.86603i"
```

Notice that this affects also the return values of the `display_format` methods: in list context the whole parameter hash will be returned, as opposed to only the style parameter value. This is a potential incompatibility with earlier versions if you have been calling the `display_format` method in list context.

The second new display parameter is `"polar_pretty_print"`, which can be set to true or false, the default being true. See the previous section for what this means.

USAGE

Thanks to overloading, the handling of arithmetics with complex numbers is simple and almost transparent.

Here are some examples:

```
use Math::Complex;

$j = cplxe(1, 2*pi/3);  # $j ** 3 == 1
print "j = $j, j**3 = ", $j ** 3, "\n";
print "1 + j + j**2 = ", 1 + $j + $j**2, "\n";

$z = -16 + 0*i;                 # Force it to be a complex
print "sqrt($z) = ", sqrt($z), "\n";

$k = exp(i * 2*pi/3);
print "$j - $k = ", $j - $k, "\n";

$z->Re(3);                      # Re, Im, arg, abs,
$j->arg(2);                     # (the last two aka rho, theta)
                                # can be used also as mutators.
```

CONSTANTS

PI

The constant `pi` and some handy multiples of it (pi2, pi4, and pip2 (pi/2) and pip4 (pi/4)) are also available if separately exported:

```
use Math::Complex ':pi';
$third_of_circle = pi2 / 3;
```

Inf

The floating point infinity can be exported as a subroutine Inf():

```
use Math::Complex qw(Inf sinh);
my $AlsoInf = Inf() + 42;
my $AnotherInf = sinh(1e42);
print "$AlsoInf is $AnotherInf\n" if $AlsoInf == $AnotherInf;
```

Note that the stringified form of infinity varies between platforms: it can be for example any of

```
inf
infinity
INF
1.#INF
```

or it can be something else.

Also note that in some platforms trying to use the infinity in arithmetic operations may result in Perl crashing because using an infinity causes SIGFPE or its moral equivalent to be sent. The way to ignore this is

```
local $SIG{FPE} = sub { };
```

ERRORS DUE TO DIVISION BY ZERO OR LOGARITHM OF ZERO

The division (/) and the following functions

```
log     ln      log10   logn
tan     sec     csc     cot
atan    asec    acsc    acot
tanh    sech    csch    coth
atanh   asech   acsch   acoth
```

cannot be computed for all arguments because that would mean dividing by zero or taking logarithm of zero. These situations cause fatal runtime errors looking like this

```
cot(0): Division by zero.
(Because in the definition of cot(0), the divisor sin(0) is 0)
Died at ...
```

or

```
atanh(-1): Logarithm of zero.
Died at...
```

For the `csc`, `cot`, `asec`, `acsc`, `acot`, `csch`, `coth`, `asech`, `acsch`, the argument cannot be `0` (zero). For the logarithmic functions and the `atanh`, `acoth`, the argument cannot be `1` (one). For the `atanh`, `acoth`, the argument cannot be `-1` (minus one). For the `atan`, `acot`, the argument cannot be `i` (the imaginary unit). For the `atan`, `acoth`, the argument cannot be `-i` (the negative imaginary unit). For the `tan`, `sec`, `tanh`, the argument cannot

be *pi/2 + k * pi*, where *k* is any integer. atan2(0, 0) is undefined, and if the complex arguments are used for atan2(), a division by zero will happen if z1**2+z2**2 == 0.

Note that because we are operating on approximations of real numbers, these errors can happen when merely `too close' to the singularities listed above.

ERRORS DUE TO INDIGESTIBLE ARGUMENTS

The `make` and `emake` accept both real and complex arguments. When they cannot recognize the arguments they will die with error messages like the following

```
Math::Complex::make: Cannot take real part of ...
Math::Complex::make: Cannot take real part of ...
Math::Complex::emake: Cannot take rho of ...
Math::Complex::emake: Cannot take theta of ...
```

BUGS

Saying `use Math::Complex;` exports many mathematical routines in the caller environment and even overrides some (`sqrt`, `log`, `atan2`). This is construed as a feature by the Authors, actually... ;-)

All routines expect to be given real or complex numbers. Don't attempt to use BigFloat, since Perl has currently no rule to disambiguate a '+' operation (for instance) between two overloaded entities.

In Cray UNICOS there is some strange numerical instability that results in root(), cos(), sin(), cosh(), sinh(), losing accuracy fast. Beware. The bug may be in UNICOS math libs, in UNICOS C compiler, in Math::Complex. Whatever it is, it does not manifest itself anywhere else where Perl runs.

SEE ALSO

Math::Trig

AUTHORS

Daniel S. Lewart <lewart!at!uiuc.edu>, Jarkko Hietaniemi <jhi!at!iki.fi>, Raphael Manfredi <Raphael_Manfredi!at!pobox.com>, Zefram zefram@fysh.org

LICENSE

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

Math::GMP

Arbitrary-precision integer arithmetic backed by GNU `libgmp`.

`Math::GMP` lets you work with integers of unlimited size. The interpreter's native integer tops out somewhere around `2**63`; pass anything bigger through a `Math::GMP` object and every arithmetic operator keeps working, up to the memory you have.

Build a value with `Math::GMP->new`, then use it like a normal number - the module overloads `+ - * / % ** <=> == & | ^ << >>` so `$x * $y` and `$x ** 1000` Do The Right Thing. Only the final stringification crosses back into ordinary Perl:

```
use Math::GMP;
my $x = Math::GMP->new('2');
my $huge = $x ** 1024;                    # a 309-digit integer
print "$huge\n";
```

A `Math::GMP` value is a blessed reference whose underlying scalar holds a pointer to a GMP `mpz_t`. Passing one to `ref` returns `"Math::GMP"`; passing it to a stringifying or numifying operator goes through the overloaded `op_stringify` / `op_numify`.

The `:constant` pragma extends the overloading to bare integer literals in your source:

```
use Math::GMP qw(:constant);
my $n = 2 ** (256 * 1024);                # evaluated in GMP, not␣
↪doubles
```

Without `:constant` the exponent `256 * 1024` would be computed as a C `double` and overflow to infinity before `Math::GMP` ever sees it.

Division by zero

`libgmp` raises `SIGFPE` on division by zero instead of returning a Perl-level error. On Perl 5.36 and later you can trap it with a `$SIG{FPE}` handler; on older Perls use `POSIX::sigaction`. Returning from the handler re-raises the signal, so use it for diagnostics (`Carp::confess`) rather than recovery.

libgmp itself is the system's `libgmp.so.10`, loaded with the module. This is a native implementation of the XS half of Math::GMP 2.25; its `Math/GMP.pm` is used unchanged, and nothing is exported by default.

Differences from upstream

`_gmp_build_version` reports the libgmp actually loaded, which is the library this module is built on; upstream reports the GMP headers it was compiled against, which there are none of here.

Functions

Construction

new_from_scalar

Build a `Math::GMP` object from a decimal/auto-detected string.

new_from_scalar_with_base

Build a `Math::GMP` object from a string in the given base.

destroy

Release the `mpz_t` backing a `Math::GMP` object.

gmp_copy

Return an independent copy of `$x`.

Conversion

stringify

Return the decimal string representation of a `Math::GMP` value.

get_str_gmp

Return the string representation of `$x` in the given base.

sizeinbase_gmp

Return the number of digits `$x` would take in the given base.

uintify

Return the low bits of `$x` as a plain unsigned Perl integer.

intify

Return `$x` as a plain signed Perl integer.

Arithmetic

add_ui_gmp

Add an unsigned integer to `$x` in place.

op_add

Overloaded `+` - return `$m + $n` as a new `Math::GMP`.

op_sub

Overloaded `-` - return `$m - $n` as a new `Math::GMP`.

op_mul

Overloaded `*` - return `$m * $n` as a new `Math::GMP`.

bmulf

Return `$x` multiplied by a floating-point value, truncated to an integer.

op_div

Overloaded `/` - integer division `int($m / $n)`, truncated toward zero.

bdiv

Return the quotient and remainder of `$m / $n` as a two-element list.

op_mod

Overloaded `%` - remainder of `$m / $n`, sign follows the dividend.

op_pow

Overloaded `**` - return `$m` raised to the integer power `$n`.

Comparison

op_spaceship

Overloaded `<=>` - signed comparison returning `-1`, `0`, or `1`.

op_eq

Overloaded `==` - return `1` when `$m` equals `$n`, else `0`.

Bitwise

mul_2exp_gmp

Return `$x` shifted left by `$e` bits (`$x * 2**$e`).

Synopsis

```
Math::GMP->new(0b1011)->mul_2exp_gmp(4);   # 0b10110000
```

Equivalent to `$x << $e` on a `Math::GMP` value; the operator form goes through `blshift`. `$e` is a plain non-negative integer.

div_2exp_gmp

Return `$x` shifted right by `$e` bits (`int($x / 2**$e)`).

Synopsis

```
Math::GMP->new(200)->div_2exp_gmp(2);      # 50
```

Truncates toward zero, matching `libgmp`'s `mpz_tdiv_q_2exp`. Use `brshift` for the overloaded-operator spelling.

mod_2exp_gmp

Return the low `$cnt` bits of `$x` (`$x mod 2**$cnt`).

Synopsis

```
Math::GMP->new(0b10001011)->mod_2exp_gmp(4);    # 0b1011
```

Does not modify `$x`. The sign of the result matches `$x` - this is truncated (`tdiv`) rather than Euclidean (`mmod`) semantics.

band

Bit-wise AND of `$m` and `$n`.

Synopsis

```
Math::GMP->new(6)->band(3);      # 0b110 & 0b011 = 0b010 = 2
```

Overload hook for `&`. The optional third argument (`swap`) is accepted and ignored.

bxor

Bit-wise XOR of `$m` and `$n`.

bior

Bit-wise inclusive OR of `$m` and `$n`.

blshift

Return `$m << $n` - left shift by `$n` bits.

brshift

Return `$m >> $n` - right shift by `$n` bits, truncating toward zero.

gmp_tstbit

Return bit `$n` of `$x` (0-indexed, from the least significant end).

Number theory

powm_gmp

Compute `$base ** $exp mod $mod` without building the full power.

`mmod_gmp`

Return the non-negative remainder of `$a / $b`.

`bmodinv`

Return the modular inverse of `$x` mod `$y`.

`legendre`

Return the Legendre symbol `($m / $n)` as `-1`, `0`, or `1`.

`jacobi`

Return the Jacobi symbol `($m / $n)` as `-1`, `0`, or `1`.

`probab_prime`

Probabilistic primality test: return `0`, `1`, or `2`.

`bgcd`

Return the greatest common divisor of `$m` and `$n`.

`blcm`

Return the least common multiple of `$m` and `$n`.

`fibonacci`

Return the `$n`-th Fibonacci number.

`bfac`

Return `$n!` - the factorial of a non-negative integer.

`bnok`

Return the binomial coefficient `$n choose $k`.

Roots and powers

`broot`

Return the integer `$n`-th root of `$x`, truncated toward zero.

brootrem

Return the integer $n-th root of $x and the shortfall.

bsqrt

Return the integer square root of $x.

bsqrtrem

Return the integer square root of $x and the shortfall.

Synopsis

```
my ($r, $q) = Math::GMP->new(7)->bsqrtrem;   # 2, 3
# because 2**2 + 3 == 7
```

The returned list satisfies `$r ** 2 + $q == $x`.

is_perfect_power

Return 1 when $x is a perfect power `$a ** $b` with `$b > 1`.

is_perfect_square

Return 1 when $x is the square of an integer, else `0`.

Synopsis

```
Math::GMP->new(49)->is_perfect_square;   # 1
Math::GMP->new(50)->is_perfect_square;   # 0
```

Overload glue

op_stringify

Overloaded `""` - stringify a `Math::GMP` as its decimal digits.

op_numify

Overloaded `0+` - convert a `Math::GMP` to a plain Perl number.

op_bool

Overloaded `bool` - true when $x is non-zero.

Library info

_gmp_build_version

Return the version of `libgmp` this module is built on, as a v-string.

_gmp_lib_version

Return the version of `libgmp` currently loaded into the process.

Other Functions

new

Build a `Math::GMP` object from a string or integer scalar.

_gmp_build_version

Return the version of `libgmp` this module is built on, as a v-string.

```
printf "built on libgmp %vd\n", Math::GMP::_gmp_build_version();   #␣
 ↪6.3.0
```

Upstream reports the GMP headers it was compiled against. libgmp is bound here when the module loads, not at compile time, so the library it is built on is the one loaded: this is `_gmp_lib_version`'s value.

_gmp_lib_version

Return the version of `libgmp` currently loaded into the process.

Synopsis

```
my $v = Math::GMP::_gmp_lib_version();
printf "running libgmp %vd\n", $v;   # e.g. "running libgmp 6.3.0"
```

What you get back

A v-string, as `v6.3.0` would be: its characters are the version numbers, so it is formatted with `sprintf "%vd"`, compared with `lt`/`gt` against another v-string, and accepted by `version->parse`. Printed as it is, it shows those characters, not the dotted digits.

Differences from upstream

Fully compatible with upstream.

get_str_gmp

Return the string representation of `$x` in the given base.

Synopsis

```
my $x = Math::GMP->new(3 * 7 + 2 * 7**2 + 6 * 7**3);
print $x->get_str_gmp(7);      # "6230"
print $x->get_str_gmp(16);     # hex
```

Base is 2 through 36 for lowercase digits or `-2` through `-36` for uppercase, matching `libgmp`'s `mpz_get_str` convention.

sizeinbase_gmp

Return the number of digits `$x` would take in the given base.

Synopsis

```
Math::GMP->new(10**100)->sizeinbase_gmp(10);    # 101
Math::GMP->new(255)->sizeinbase_gmp(2);         # 8
```

The count may be off by one on the high side - this is `libgmp`'s documented behaviour for `mpz_sizeinbase`, which trades accuracy for speed. Use it for buffer sizing, not for exact digit counts.

intify

Return `$x` as a plain signed Perl integer.

Synopsis

```
my $n = Math::GMP->new(-42)->intify;    # -42
```

Truncates when `$x` does not fit in a platform `signed long`. For values outside that range use `get_str_gmp` or stringify to preserve precision.

add_ui_gmp

Add an unsigned integer to `$x` in place.

Synopsis

```
my $x = Math::GMP->new(1_000_000);
$x->add_ui_gmp(7);
print $x;                   # 1000007
```

Mutates `$x` and returns nothing. The second argument must be a non-negative plain Perl integer. Useful in tight counting loops where allocating a new `Math::GMP` per `+=` would dominate the cost.

powm_gmp

Compute `$base ** $exp mod $mod` without building the full power.

Synopsis

```
my $base = Math::GMP->new(157);
my $exp  = Math::GMP->new(100);
my $mod  = Math::GMP->new(5013);
my $r    = $base->powm_gmp($exp, $mod);   # 157**100 mod 5013
```

The workhorse of RSA-style modular exponentiation: fast enough for thousand-bit exponents, and never materialises the intermediate `$base ** $exp`. Behaviour is undefined when `$mod` is `0`.

mmod_gmp

Return the non-negative remainder of `$a / $b`.

Synopsis

```
Math::GMP->new(-7)->mmod_gmp(3);        # 2, not -1
```

Unlike `%` on `Math::GMP` (which uses truncated division and follows the sign of the dividend), `mmod_gmp` always returns a result in `[0, abs($b))`. This matches the mathematical "mod" operation used in number-theory work.

op_numify

Overloaded `0+` - convert a `Math::GMP` to a plain Perl number.

Invoked when a `Math::GMP` is used in numeric context outside of the overloaded arithmetic operators (for example, passed to a function that does `$arg + 0`). Precision is limited to what an `IV` or `UV` can hold; for large values the top bits are lost. Call `intify` or `uintify` if you want to be explicit about the cast, or stringify if you want to keep all the digits.

bmulf

Return `$x` multiplied by a floating-point value, truncated to an integer.

Synopsis

```
my $x = Math::GMP->new(3)->bpow(100);
my $y = $x->bmulf(1.5);                 # floor(3**100 * 1.5)
```

The double `$float` is converted to a GMP float, multiplied, then truncated toward zero. Internal precision is raised just enough to hold both operands exactly; the default `mpf` precision is restored before returning.

Edge cases

- `$float == 0.0` gives a `Math::GMP` of `0`.
- Negative `$float` gives a result with the expected sign.
- `NaN` / infinity gives an implementation-defined result; do not rely on it.

bdiv

Return the quotient and remainder of `$m / $n` as a two-element list.

Synopsis

```
my $x = Math::GMP->new(7);
my ($q, $r) = $x->bdiv(3);      # $q = 2, $r = 1
```

Division truncates toward zero, and the remainder has the same sign as the dividend - the `tdiv_qr` convention. For non-negative remainders, combine with `mmod_gmp`.

bmodinv

Return the modular inverse of `$x` mod `$y`.

Synopsis

```
my $x = Math::GMP->new(5);
my $inv = $x->bmodinv(7);      # 3, because 5 * 3 == 1 (mod 7)
```

When `$x` has no inverse modulo `$y` (i.e. `gcd($x, $y) != 1`), returns `0`. Behaviour is undefined when `$y` is `0`.

Edge cases

- `bmodinv` returning `0` is ambiguous: `0` is never a valid inverse, so treat it as "no inverse exists".

legendre

Return the Legendre symbol `($m / $n)` as `-1`, `0`, or `1`.

Synopsis

```
Math::GMP->new(6)->legendre(Math::GMP->new(11));   # -1, 0, or 1
```

Defined only when `$n` is an odd prime. When `$n` is not an odd prime the result is currently `0`, but do not rely on that - upstream documents this as "may change in the future".

probab_prime

Probabilistic primality test: return `0`, 1, or 2.

Synopsis

```
my $x = Math::GMP->new(7);
my $v = $x->probab_prime(10);   # 2 - definitely prime
```

What you get back

- `0` - definitely composite.
- 1 - probably prime (no witness found in `$reps` rounds).
- 2 - definitely prime (small numbers, or `libgmp` proved it).

`$reps` is the Miller-Rabin round count; higher values cost more but tighten the confidence on a 1 answer. A round count of 25 is a common default for cryptographic work.

bgcd

Return the greatest common divisor of `$m` and `$n`.

Synopsis

```
my $x = Math::GMP->new(6);
print $x->bgcd(4);              # 2
```

Also registered as `gcd`. Result is always non-negative; `bgcd(0, 0)` is `0`.

blcm

Return the least common multiple of `$m` and `$n`.

Synopsis

```
my $x = Math::GMP->new(6);
print $x->blcm(4);              # 12
```

Also registered as `lcm`. Result is always non-negative; `blcm($x, 0)` is `0`.

fibonacci

Return the $n-th Fibonacci number.

Synopsis

```
my $f = Math::GMP::fibonacci(100);    # 354224848179261915075
```

Called as a function, not a method. `$n` is a plain Perl non-negative integer. `libgmp`'s recursive doubling algorithm handles indices in the millions in fractions of a second.

blshift

Return `$m << $n` - left shift by `$n` bits.

Synopsis

```
Math::GMP->new(0b11) << 4;                # 0b110000 = 48
Math::GMP->new(0b11)->blshift(4, 0);      # same; the third argument
↪is
                                          # the overload swap flag
```

Overload hook for <<. Only the low bits of `$n` that fit in an `unsigned long` count, because it is passed through `mpz_get_ui`: the shift is `abs($n)` modulo 2**64, so a negative `$n` shifts left by its magnitude and `2**64 + 1` shifts by 1.

brshift

Return `$m >> $n` - right shift by `$n` bits, truncating toward zero.

Synopsis

```
Math::GMP->new(0b11001) >> 3;               # 0b11 = 3
Math::GMP->new(0b11001)->brshift(3, 0);     # same; the third
↪argument
                                            # is the overload swap
↪flag
```

Overload hook for >>. Uses truncated division, so negative values shift toward zero rather than toward minus infinity.

bfac

Return `$n!` - the factorial of a non-negative integer.

Synopsis

```
my $x = Math::GMP->new(5);
print $x->bfac;                 # 1*2*3*4*5 = 120
```

The receiver is read as an `unsigned long` - factorials larger than that range are out of reach of a single call. In practice this is rarely limiting: `libgmp` struggles with memory long before `LONG_MAX` enters the picture.

bnok

Return the binomial coefficient `$n choose $k`.

Synopsis

```
my $x = Math::GMP->new(5);
print $x->bnok(2);              # 10
```

Equivalent to `$n! / ($k! * ($n - $k)!)`, but computed without building the factorials - suitable for counting problems where `$n` is large.

gmp_copy

Return an independent copy of `$x`.

Because `Math::GMP` values are references to heap-allocated `mpz_t` structures, two Perl variables can share the same underlying integer. For the overloaded arithmetic operators that does not matter - they always allocate a fresh result. If you are about to call an in-place mutator (currently just `add_ui_gmp`) and want to keep the original intact, copy first.

gmp_tstbit

Return bit `$n` of `$x` (0-indexed, from the least significant end).

Synopsis

```
my $x = Math::GMP->new(0b1010);
print $x->gmp_tstbit(1);        # 1
print $x->gmp_tstbit(2);        # 0
```

For negative values, `libgmp` uses a two's-complement view. Bits above the most significant set bit of a positive value read as `0`.

broot

Return the integer `$n`-th root of `$x`, truncated toward zero.

Synopsis

```
my $x = Math::GMP->new(100);
print $x->broot(3);             # 4, since 4**3 = 64 and 5**3 = 125
```

`$n` must be a positive integer. For odd `$n` negative values of `$x` are fine; for even `$n` they croak inside `libgmp`.

brootrem

Return the integer $n-th root of $x and the shortfall.

Synopsis

```
my $x = Math::GMP->new(100);
my ($root, $rem) = $x->brootrem(3);    # 4, 36
# because 4**3 + 36 == 100
```

The returned list satisfies `$root ** $n + $rem == $x` exactly. Use when you need both pieces in one pass - calling `broot` and computing the remainder yourself doubles the work.

Differences from upstream

Replicates upstream's explicit workaround for `libgmp` versions earlier than `5.1.0`, where `mpz_rootrem` returned wrong results for negative arguments with odd $n. Modern `libgmp` builds skip the workaround automatically.

bsqrt

Return the integer square root of $x.

Synopsis

```
Math::GMP->new(6)->bsqrt;      # 2 - int(sqrt(6))
Math::GMP->new(49)->bsqrt;     # 7
```

Equivalent to `broot(2)` but noticeably faster. Croaks on negative arguments inside `libgmp`.

is_perfect_power

Return 1 when $x is a perfect power `$a ** $b` with `$b > 1`.

Synopsis

```
Math::GMP->new(27)->is_perfect_power;    # 1 (3**3)
Math::GMP->new(28)->is_perfect_power;    # 0
```

Returns `0` otherwise. Detects any exponent - squares, cubes, fifth powers, and so on.

new

Build a `Math::GMP` object from a string or integer scalar.

Synopsis

```
my $x = Math::GMP->new(123);
my $y = Math::GMP->new('12345678901234567890123456789O');
my $z = Math::GMP->new('abcd', 36);          # base 36
```

What you get back

A blessed reference of class `Math::GMP`. You can feed it to any overloaded operator, or to the named methods on this page.

The first argument is read as a string. A leading `+` is stripped, and embedded spaces and underscores are removed - so `'1_000_000'` and `' 1 000 000 '` both parse. When no base is given every character must be a digit, `-`, `x` or a hex letter, which admits the usual `0x`-prefixed hex form; any other character dies with `Argument to Math::GMP->new is not a string representing an integer`. A string that passes this check but is not a valid number, such as `'abc'` or `'1-2'`, gives `0`. With an explicit base (2 to 62) the usual GMP digit-set applies.

Edge cases

- Missing or false first argument gives `Math::GMP` of `0`.
- Floating-point input is not special-cased here; a stringified value with a decimal point or an exponent sign fails the character check and dies.
- Base `0` (the default) means "guess from the prefix": `0x` for hex, `0` for octal, otherwise decimal.

Math::Random

Namespace containing the following modules:

- `Math::Random::MT` - The Mersenne Twister pseudo-random number generator, MT19937: a long period (2**19937 - 1), fast, and the same sequence on every platform for the same seed.

Math::Random::MT

The Mersenne Twister pseudo-random number generator, MT19937: a long period (2**19937 - 1), fast, and the same sequence on every platform for the same seed. It is not for cryptography.

`Math::Random::MT->new(@seeds)` makes a generator; one seed is a 32-bit integer, several are an array of them, and none takes a seed from perl's own `rand`. `$gen->rand($n)` returns a number in `[0, $n)` (`$n` defaults to 1), `$gen->irand` an integer in `[0, 2**32)`, and `$gen->get_seed` the seed. Imported, `srand`, `rand` and `irand` work on one generator of the module's own, as perl's built-ins do.

This is a native implementation of Math::Random::MT 1.17's `MT.xs` and `_mt.c` (Matsumoto and Nishimura's mt19937ar.c); its `Math/Random/MT.pm` is used unchanged.

Synopsis

```
use Math::Random::MT;

my $gen = Math::Random::MT->new(5489);
print $gen->irand, "\n";        # 3499211612, mt19937ar's first␣
↪output
my $die = 1 + int $gen->rand(6);

use Math::Random::MT qw(srand rand);
srand(42);
print rand(10), "\n";
```

Functions

Generators

init

`Math::Random::MT::init()`: a new generator object, not yet seeded; new seeds it.

genrand

`$gen->genrand`: the next number in `[0, 1)`.

genirand

`$gen->genirand`: the next integer in `[0, 2**32)`.

Seeding

init_seed

`$gen->init_seed($seed)`: seed with one 32-bit integer.

setup_array

`$gen->setup_array(@seeds)`: seed with an array of 32-bit integers.

get_seed

`$gen->get_seed`: the seed it was given with `init_seed`, or 19650218 after `setup_array`.

Math::Trig

trigonometric functions

SYNOPSIS

```
use Math::Trig;

$x = tan(0.9);
$y = acos(3.7);
$z = asin(2.4);

$halfpi = pi/2;

$rad = deg2rad(120);

# Import constants pi2, pi4, pip2, pip4 (2*pi, 4*pi, pi/2, pi/4).
use Math::Trig ':pi';

# Import the conversions between cartesian/spherical/cylindrical.
use Math::Trig ':radial';

    # Import the great circle formulas.
use Math::Trig ':great_circle';
```

DESCRIPTION

`Math::Trig` defines many trigonometric functions not defined by the core Perl which defines only the `sin()` and `cos()`. The constant **pi** is also defined as are a few convenience functions for angle conversions, and *great circle formulas* for spherical movement.

TRIGONOMETRIC FUNCTIONS

The tangent

- **tan**

The cofunctions of the sine, cosine, and tangent (cosec/csc and cotan/cot are aliases)

csc, **cosec**, **sec**, **sec**, **cot**, **cotan**

The arcus (also known as the inverse) functions of the sine, cosine, and tangent

asin, **acos**, **atan**

The principal value of the arc tangent of y/x

atan2(y, x)

The arcus cofunctions of the sine, cosine, and tangent (acosec/acsc and acotan/acot are aliases). Note that atan2(0, 0) is not well-defined.

acsc, **acosec**, **asec**, **acot**, **acotan**

The hyperbolic sine, cosine, and tangent

sinh, **cosh**, **tanh**

The cofunctions of the hyperbolic sine, cosine, and tangent (cosech/csch and cotanh/coth are aliases)

csch, **cosech**, **sech**, **coth**, **cotanh**

The area (also known as the inverse) functions of the hyperbolic sine, cosine, and tangent

asinh, **acosh**, **atanh**

The area cofunctions of the hyperbolic sine, cosine, and tangent (acsch/acosech and acoth/acotanh are aliases)

acsch, **acosech**, **asech**, **acoth**, **acotanh**

The trigonometric constant **pi** and some of handy multiples of it are also defined.

pi, pi2, pi4, pip2, pip4

ERRORS DUE TO DIVISION BY ZERO

The following functions

```
acoth
acsc
acsch
asec
asech
atanh
cot
coth
csc
csch
sec
sech
tan
tanh
```

cannot be computed for all arguments because that would mean dividing by zero or taking logarithm of zero. These situations cause fatal runtime errors looking like this

```
cot(0): Division by zero.
(Because in the definition of cot(0), the divisor sin(0) is 0)
Died at ...
```

or

```
atanh(-1): Logarithm of zero.
Died at...
```

For the `csc`, `cot`, `asec`, `acsc`, `acot`, `csch`, `coth`, `asech`, `acsch`, the argument cannot be `0` (zero). For the `atanh`, `acoth`, the argument cannot be 1 (one). For the `atanh`, `acoth`, the argument cannot be `-1` (minus one). For the `tan`, `sec`, `tanh`, `sech`, the argument cannot be *pi/2 + k * pi*, where *k* is any integer.

Note that atan2(0, 0) is not well-defined.

SIMPLE (REAL) ARGUMENTS, COMPLEX RESULTS

Please note that some of the trigonometric functions can break out from the **real axis** into the **complex plane**. For example `asin(2)` has no definition for plain real numbers but it has definition for complex numbers.

In Perl terms this means that supplying the usual Perl numbers (also known as scalars, please see *perldata*) as input for the trigonometric functions might produce as output results that no more are simple real numbers: instead they are complex numbers.

The `Math::Trig` handles this by using the `Math::Complex` package which knows how to handle complex numbers, please see *Math::Complex* for more information. In practice you need not to worry about getting complex numbers as results because the `Math::Complex` takes care of details like for example how to display complex numbers. For example:

```
print asin(2), "\n";
```

should produce something like this (take or leave few last decimals):

```
1.5707963267949-1.31695789692482i
```

That is, a complex number with the real part of approximately `1.571` and the imaginary part of approximately `-1.317`.

PLANE ANGLE CONVERSIONS

(Plane, 2-dimensional) angles may be converted with the following functions.

- deg2rad

```
$radians  = deg2rad($degrees);
```

- grad2rad

```
$radians  = grad2rad($gradians);
```

- rad2deg

```
$degrees  = rad2deg($radians);
```

- grad2deg

```
$degrees  = grad2deg($gradians);
```

- deg2grad

```
$gradians = deg2grad($degrees);
```

- rad2grad

```
$gradians = rad2grad($radians);
```

The full circle is 2 *pi* radians or *360* degrees or *400* gradians. The result is by default wrapped to be inside the [0, {2pi,360,400}) circle. If you don't want this, supply a true second argument:

```
$zillions_of_radians  = deg2rad($zillions_of_degrees, 1);
$negative_degrees     = rad2deg($negative_radians, 1);
```

You can also do the wrapping explicitly by rad2rad(), deg2deg(), and grad2grad().

- rad2rad

  ```
  $radians_wrapped_by_2pi = rad2rad($radians);
  ```

- deg2deg

  ```
  $degrees_wrapped_by_360 = deg2deg($degrees);
  ```

- grad2grad

  ```
  $gradians_wrapped_by_400 = grad2grad($gradians);
  ```

RADIAL COORDINATE CONVERSIONS

Radial coordinate systems are the **spherical** and the **cylindrical** systems, explained shortly in more detail.

You can import radial coordinate conversion functions by using the `:radial` tag:

```
use Math::Trig ':radial';

($rho, $theta, $z)     = cartesian_to_cylindrical($x, $y, $z);
($rho, $theta, $phi)   = cartesian_to_spherical($x, $y, $z);
($x, $y, $z)           = cylindrical_to_cartesian($rho, $theta, $z);
($rho_s, $theta, $phi) = cylindrical_to_spherical($rho_c, $theta,
 ↪$z);
($x, $y, $z)           = spherical_to_cartesian($rho, $theta, $phi);
($rho_c, $theta, $z)   = spherical_to_cylindrical($rho_s, $theta,
 ↪$phi);
```

All angles are in radians.

COORDINATE SYSTEMS

Cartesian coordinates are the usual rectangular *(x, y, z)*-coordinates.

Spherical coordinates, *(rho, theta, phi)*, are three-dimensional coordinates which define a point in three-dimensional space. They are based on a sphere surface. The radius of the sphere is **rho**, also known as the *radial* coordinate. The angle in the *xy*-plane (around the *z*-axis) is **theta**, also known as the *azimuthal* coordinate. The angle from the *z*-axis is **phi**, also known as the *polar* coordinate. The North Pole is therefore *rho, 0, 0*, and the Gulf of Guinea (think of the missing big chunk of Africa) *rho, 0, pi/2*. In geographical terms *phi* is latitude (northward positive, southward negative) and *theta* is longitude (eastward positive, westward negative).

BEWARE: some texts define *theta* and *phi* the other way round, some texts define the *phi* to start from the horizontal plane, some texts use *r* in place of *rho*.

Cylindrical coordinates, *(rho, theta, z)*, are three-dimensional coordinates which define a point in three-dimensional space. They are based on a cylinder surface. The radius of the cylinder is **rho**, also known as the *radial* coordinate. The angle in the *xy*-plane (around the *z*-axis) is **theta**, also known as the *azimuthal* coordinate. The third coordinate is the *z*, pointing up from the **theta**-plane.

3-D ANGLE CONVERSIONS

Conversions to and from spherical and cylindrical coordinates are available. Please notice that the conversions are not necessarily reversible because of the equalities like *pi* angles being equal to *-pi* angles.

- cartesian_to_cylindrical

```
($rho, $theta, $z) = cartesian_to_cylindrical($x, $y, $z);
```

- cartesian_to_spherical

```
($rho, $theta, $phi) = cartesian_to_spherical($x, $y, $z);
```

- cylindrical_to_cartesian

```
($x, $y, $z) = cylindrical_to_cartesian($rho, $theta, $z);
```

- cylindrical_to_spherical

```
($rho_s, $theta, $phi) = cylindrical_to_spherical($rho_c,
↪$theta, $z);
```

  Notice that when `$z` is not 0 `$rho_s` is not equal to `$rho_c`.

- spherical_to_cartesian

```
($x, $y, $z) = spherical_to_cartesian($rho, $theta, $phi);
```

- spherical_to_cylindrical

```
($rho_c, $theta, $z) = spherical_to_cylindrical($rho_s, $theta,
↪$phi);
```

  Notice that when `$z` is not 0 `$rho_c` is not equal to `$rho_s`.

GREAT CIRCLE DISTANCES AND DIRECTIONS

A great circle is section of a circle that contains the circle diameter: the shortest distance between two (non-antipodal) points on the spherical surface goes along the great circle connecting those two points.

great_circle_distance

Returns the great circle distance between two points on a sphere.

```
$distance = great_circle_distance($theta0, $phi0, $theta1, $phi1, [,
↪$rho]);
```

Where ($theta0, $phi0) and ($theta1, $phi1) are the spherical coordinates of the two points, respectively. The distance is in $rho units. The $rho is optional. It defaults to 1 (the unit sphere).

If you are using geographic coordinates, latitude and longitude, you need to adjust for the fact that latitude is zero at the equator increasing towards the north and decreasing towards the south. Assuming ($lat0, $lon0) and ($lat1, $lon1) are the geographic coordinates in radians of the two points, the distance can be computed with

```
$distance = great_circle_distance($lon0, pi/2 - $lat0,
                                  $lon1, pi/2 - $lat1, $rho);
```

great_circle_direction

The direction you must follow the great circle (also known as *bearing*) can be computed by the great_circle_direction() function:

```
use Math::Trig 'great_circle_direction';

$direction = great_circle_direction($theta0, $phi0, $theta1, $phi1);
```

great_circle_bearing

Alias 'great_circle_bearing' for 'great_circle_direction' is also available.

```
use Math::Trig 'great_circle_bearing';

$direction = great_circle_bearing($theta0, $phi0, $theta1, $phi1);
```

The result of great_circle_direction is in radians, zero indicating straight north, pi or -pi straight south, pi/2 straight west, and -pi/2 straight east.

great_circle_destination

You can inversely compute the destination if you know the starting point, direction, and distance:

```
use Math::Trig 'great_circle_destination';

# $diro is the original direction,
# for example from great_circle_bearing().
# $distance is the angular distance in radians,
# for example from great_circle_distance().
# $thetad and $phid are the destination coordinates,
```

(continues on next page)

(continued from previous page)

```
# $dird is the final direction at the destination.

($thetad, $phid, $dird) =
  great_circle_destination($theta, $phi, $diro, $distance);
```

or the midpoint if you know the end points:

great_circle_midpoint

```
use Math::Trig 'great_circle_midpoint';

($thetam, $phim) =
  great_circle_midpoint($theta0, $phi0, $theta1, $phi1);
```

The great_circle_midpoint() is just a special case of

great_circle_waypoint

```
use Math::Trig 'great_circle_waypoint';

($thetai, $phii) =
  great_circle_waypoint($theta0, $phi0, $theta1, $phi1, $way);
```

Where $way indicates the position of the waypoint along the great circle arc through the starting point ($theta0, $phi0) and the end point ($theta1, $phi1) relative to the distance from the starting point to the end point. So $way = 0 gives the starting point, $way = 1 gives the end point, $way < 0 gives a point "behind" the starting point, and $way > 1 gives a point beyond the end point. $way defaults to 0.5 if not given.

Note that antipodal points (where their distance is *pi* radians) do not have unique waypoints between them, and therefore `undef` is returned in such cases. If the points are the same, so the distance between them is zero, all waypoints are identical to the starting/end point.

The thetas, phis, direction, and distance in the above are all in radians.

You can import all the great circle formulas by

```
use Math::Trig ':great_circle';
```

Notice that the resulting directions might be somewhat surprising if you are looking at a flat worldmap: in such map projections the great circles quite often do not look like the shortest routes – but for example the shortest possible routes from Europe or North America to Asia do often cross the polar regions. (The common Mercator projection does **not** show great circles as straight lines: straight lines in the Mercator projection are lines of constant bearing.)

EXAMPLES

To calculate the distance between London (51.3N 0.5W) and Tokyo (35.7N 139.8E) in kilometers:

```
use Math::Trig qw(great_circle_distance deg2rad);

# Notice the 90 - latitude: phi zero is at the North Pole.
sub NESW { deg2rad($_[0]), deg2rad(90 - $_[1]) }
my @L = NESW( -0.5, 51.3);
my @T = NESW(139.8, 35.7);
my $km = great_circle_distance(@L, @T, 6378); # About 9600 km.
```

The direction you would have to go from London to Tokyo (in radians, straight north being zero, straight east being pi/2).

```
use Math::Trig qw(great_circle_direction);

my $rad = great_circle_direction(@L, @T); # About 0.547 or 0.174 pi.
```

The midpoint between London and Tokyo being

```
use Math::Trig qw(great_circle_midpoint rad2deg);

my @M = great_circle_midpoint(@L, @T);
sub SWNE { rad2deg( $_[0] ), 90 - rad2deg( $_[1] ) }
my @lonlat = SWNE(@M);
```

or about 69 N 89 E, on the Putorana Plateau of Siberia.

NOTE: you **cannot** get from A to B like this:

```
Dist = great_circle_distance(A, B)
Dir  = great_circle_direction(A, B)
C    = great_circle_destination(A, Dist, Dir)
```

and expect C to be B, because the bearing constantly changes when going from A to B (except in some special case like the meridians or the circles of latitudes) and in great_circle_destination() one gives a **constant** bearing to follow.

CAVEAT FOR GREAT CIRCLE FORMULAS

The answers may be off by few percentages because of the irregular (slightly aspherical) form of the Earth. The errors are at worst about 0.55%, but generally below 0.3%.

Real-valued asin and acos

For small inputs asin() and acos() may return complex numbers even when real numbers would be enough and correct, this happens because of floating-point inaccuracies. You can see these inaccuracies for example by trying theses:

```
print cos(1e-6)**2+sin(1e-6)**2 - 1,"\n";
printf "%.20f", cos(1e-6)**2+sin(1e-6)**2,"\n";
```

which will print something like this

```
-1.11022302462516e-16
0.99999999999999988898
```

even though the expected results are of course exactly zero and one. The formulas used to compute asin() and acos() are quite sensitive to this, and therefore they might accidentally slip into the complex plane even when they should not. To counter this there are two interfaces that are guaranteed to return a real-valued output.

- asin_real

  ```
  use Math::Trig qw(asin_real);

  $real_angle = asin_real($input_sin);
  ```

  Return a real-valued arcus sine if the input is between [-1, 1], **inclusive** the endpoints. For inputs greater than one, pi/2 is returned. For inputs less than minus one, -pi/2 is returned.

- acos_real

  ```
  use Math::Trig qw(acos_real);

  $real_angle = acos_real($input_cos);
  ```

  Return a real-valued arcus cosine if the input is between [-1, 1], **inclusive** the endpoints. For inputs greater than one, zero is returned. For inputs less than minus one, pi is returned.

BUGS

Saying `use Math::Trig;` exports many mathematical routines in the caller environment and even overrides some (`sin`, `cos`). This is construed as a feature by the Authors, actually... ;-)

The code is not optimized for speed, especially because we use `Math::Complex` and thus go quite near complex numbers while doing the computations even when the arguments are not. This, however, cannot be completely avoided if we want things like `asin(2)` to give an answer instead of giving a fatal runtime error.

Do not attempt navigation using these formulas.

SEE ALSO

Math::Complex

AUTHORS

Jarkko Hietaniemi <jhi!at!iki.fi>, Raphael Manfredi <Raphael_Manfredi!at!pobox.com>, Zefram zefram@fysh.org

LICENSE

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

5.2.66 Memoize

Make functions faster by trading space for time - cache a function's results by its arguments and return the cached result on a repeated call.

`memoize('f')` replaces `f` with a wrapper that remembers every result it has computed. The next call with the same arguments returns the remembered result without running `f` again. This is only sound for a function whose result depends on nothing but its arguments and which has no side effects; the call to `memoize` is your assertion that it is such a function.

The wrapper is installed under the function's own name, so existing callers hit the cache transparently, and it is returned as a code reference. `INSTALL => 'other'` installs it under another name, `INSTALL => undef` installs nothing and only returns it. Scalar- and list-context results are cached separately, because a function may return different things in each; `SCALAR_CACHE` and `LIST_CACHE` choose the storage per context: `MEMORY` (in-process, the default), `[HASH => \%h]` (a hash you supply, possibly tied to a DBM file so the cache persists across runs), `FAULT` (croak if called in that context) and `MERGE` (one cache for both contexts). `NORMALIZER` names a function that turns the argument list into the cache key when the default key, the arguments joined with `chr(28)`, is not the right notion of "same arguments".

`flush_cache('f')` empties a function's cache; `unmemoize('f')` restores the original function. Both are exported on request only.

The cache is native: nothing here is a Perl closure, the key is built and probed in the runtime, and a hit costs one hash probe and a copy of the stored result.

A pure numeric function is memoized further down still. It is compiled, its results are kept in a native table, and `memoize` puts the compiled function itself under the name rather than anything wrapped around it - so a loop calling it can be compiled too, and a repeated call is then a table probe in registers with no call at all. This covers one to four numeric arguments and recursion (the compiled body's recursive calls probe the same table), so `memoize('fib')`, and equally a two- or three-argument recurrence, run compiled, memoized and - for a repeated call - allocation- and call-free. Deep recursion behaves as in perl: it is limited by memory, not by a fixed depth, and the `Deep recursion` warning appears as usual under `use warnings`. Anything outside that shape - five or more arguments, a non-numeric argument, a `NORMALIZER`, list context - uses the general cache above, which is correct for every function `memoize` accepts.

Synopsis

```
use Memoize;
memoize('slow_function');
slow_function(@args);          # computed
slow_function(@args);          # returned from the cache

use Memoize qw(memoize flush_cache unmemoize);
```

(continues on next page)

(continued from previous page)

```
memoize('fib', NORMALIZER => sub { $_[0] });
flush_cache('fib');
unmemoize('fib');
```

Differences from upstream

Upstream Memoize is pure Perl and wraps each function in a Perl closure; here the wrapper is native, and that shows in four places. From inside the memoized function `caller` sees its own caller directly, with no `Memoize::__ANON__` frame in between - there is no Perl closure to make one. For the same reason the `Deep recursion` warning names the memoized function (`subroutine "main::fib"`) where the pure-Perl module's recursion is first counted on its wrapper (`anonymous subroutine`). A `die` from `flush_cache` or `unmemoize` reports the caller's location, as XS errors do, rather than a line inside `Memoize.pm`. The module does not define `CLONE`, because this runtime has no interpreter threads.

Modules

- `Memoize::AnyDBM_File` - glue to provide EXISTS for AnyDBM_File for Storable use
- `Memoize::Expire` - Plug-in module for automatic expiration of memoized values
- `Memoize::NDBM_File` - glue to provide EXISTS for NDBM_File for Storable use
- `Memoize::SDBM_File` - DEPRECATED compatibility shim
- `Memoize::Storable` - store Memoized data in Storable database

Functions

Memoization

memoize

Memoize a function: `memoize('name', %options)` or `memoize(\&code, %options)`.

flush_cache

Empty a memoized function's caches: `flush_cache('name')` or `flush_cache(\&wrapper)`.

unmemoize

Undo `memoize`: `unmemoize('name')` or `unmemoize(\&wrapper)`.

Memoize::AnyDBM_File

glue to provide EXISTS for AnyDBM_File for Storable use

DESCRIPTION

This class does the same thing as *AnyDBM_File*, except that instead of *NDBM_File* itself it loads *Memoize::NDBM_File* if *NDBM_File* lacks EXISTS support.

Code which requires perl 5.37.3 or newer should simply use AnyBDM_File directly.

Memoize::Expire

Plug-in module for automatic expiration of memoized values

SYNOPSIS

```
use Memoize;
use Memoize::Expire;
tie my %cache => 'Memoize::Expire',
                   LIFETIME => $lifetime,    # In seconds
                   NUM_USES => $n_uses;

memoize 'function', SCALAR_CACHE => [HASH => \%cache ];
```

DESCRIPTION

Memoize::Expire is a plug-in module for Memoize. It allows the cached values for memoized functions to expire automatically. This manual assumes you are already familiar with the Memoize module. If not, you should study that manual carefully first, paying particular attention to the HASH feature.

Memoize::Expire is a layer of software that you can insert in between Memoize itself and whatever underlying package implements the cache. The layer presents a hash variable whose values expire whenever they get too old, have been used too often, or both. You tell `Memoize` to use this forgetful hash as its cache instead of the default, which is an ordinary hash.

To specify a real-time timeout, supply the `LIFETIME` option with a numeric value. Cached data will expire after this many seconds, and will be looked up afresh when it expires. When a data item is looked up afresh, its lifetime is reset.

If you specify `NUM_USES` with an argument of *n*, then each cached data item will be discarded and looked up afresh after the _n_th time you access it. When a data item is looked up afresh, its number of uses is reset.

If you specify both arguments, data will be discarded from the cache when either expiration condition holds.

Memoize::Expire uses a real hash internally to store the cached data. You can use the `HASH` option to Memoize::Expire to supply a tied hash in place of the ordinary hash that Memoize::Expire will normally use. You can use this feature to add Memoize::Expire as a

layer in between a persistent disk hash and Memoize. If you do this, you get a persistent disk cache whose entries expire automatically. For example:

```
#   Memoize
#      |
#   Memoize::Expire  enforces data expiration policy
#      |
#   DB_File  implements persistence of data in a disk file
#      |
#   Disk file

use Memoize;
use Memoize::Expire;
use DB_File;

# Set up persistence
tie my %disk_cache => 'DB_File', $filename, O_CREAT|O_RDWR, 0666];

# Set up expiration policy, supplying persistent hash as a target
tie my %cache => 'Memoize::Expire',
                   LIFETIME => $lifetime,    # In seconds
                   NUM_USES => $n_uses,
                   HASH => \%disk_cache;

# Set up memoization, supplying expiring persistent hash for cache
memoize 'function', SCALAR_CACHE => [ HASH => \%cache ];
```

INTERFACE

There is nothing special about Memoize::Expire. It is just an example. If you don't like the policy that it implements, you are free to write your own expiration policy module that implements whatever policy you desire. Here is how to do that. Let us suppose that your module will be named MyExpirePolicy.

Short summary: You need to create a package that defines four methods:

-

TIEHASH

```
Construct and return cache object.
```

-

EXISTS

```
Given a function argument, is the corresponding function value in␣
↪the
cache, and if so, is it fresh enough to use?
```

-

FETCH

```
Given a function argument, look up the corresponding function value␣
 ↪in
the cache and return it.
```

-

STORE

```
Given a function argument and the corresponding function value,␣
 ↪store
them into the cache.
```

-

CLEAR

```
(Optional.)  Flush the cache completely.
```

The user who wants the memoization cache to be expired according to your policy will say so by writing

```
tie my %cache => 'MyExpirePolicy', args...;
memoize 'function', SCALAR_CACHE => [HASH => \%cache];
```

This will invoke `MyExpirePolicy->TIEHASH(args)`. MyExpirePolicy::TIEHASH should do whatever is appropriate to set up the cache, and it should return the cache object to the caller.

For example, MyExpirePolicy::TIEHASH might create an object that contains a regular Perl hash (which it will to store the cached values) and some extra information about the arguments and how old the data is and things like that. Let us call this object `C`.

When Memoize needs to check to see if an entry is in the cache already, it will invoke `C->EXISTS(key)`. `key` is the normalized function argument. MyExpirePolicy::EXISTS should return 0 if the key is not in the cache, or if it has expired, and 1 if an unexpired value is in the cache. It should *not* return `undef`, because there is a bug in some versions of Perl that will cause a spurious FETCH if the EXISTS method returns `undef`.

If your EXISTS function returns true, Memoize will try to fetch the cached value by invoking `C->FETCH(key)`. MyExpirePolicy::FETCH should return the cached value. Otherwise, Memoize will call the memoized function to compute the appropriate value, and will store it into the cache by calling `C->STORE(key, value)`.

Here is a very brief example of a policy module that expires each cache item after ten seconds.

```
package Memoize::TenSecondExpire;

sub TIEHASH {
  my ($package, %args) = @_;
  my $cache = $args{HASH} || {};
  bless $cache => $package;
}
```

(continues on next page)

(continued from previous page)

```
sub EXISTS {
  my ($cache, $key) = @_;
  if (exists $cache->{$key} &&
      $cache->{$key}{EXPIRE_TIME} > time) {
    return 1
  } else {
    return 0;  # Do NOT return undef here
  }
}

sub FETCH {
  my ($cache, $key) = @_;
  return $cache->{$key}{VALUE};
}

sub STORE {
  my ($cache, $key, $newvalue) = @_;
  $cache->{$key}{VALUE} = $newvalue;
  $cache->{$key}{EXPIRE_TIME} = time + 10;
}
```

To use this expiration policy, the user would say

```
use Memoize;
tie my %cache10sec => 'Memoize::TenSecondExpire';
memoize 'function', SCALAR_CACHE => [HASH => \%cache10sec];
```

Memoize would then call `function` whenever a cached value was entirely absent or was older than ten seconds.

You should always support a `HASH` argument to `TIEHASH` that ties the underlying cache so that the user can specify that the cache is also persistent or that it has some other interesting semantics. The example above demonstrates how to do this, as does `Memoize::Expire`.

Another sample module, Memoize::Saves, is available in a separate distribution on CPAN. It implements a policy that allows you to specify that certain function values would always be looked up afresh. See the documentation for details.

ALTERNATIVES

Brent Powers has a Memoize::ExpireLRU module that was designed to work with Memoize and provides expiration of least-recently-used data. The cache is held at a fixed number of entries, and when new data comes in, the least-recently used data is expired.

Joshua Chamas's Tie::Cache module may be useful as an expiration manager. (If you try this, let me know how it works out.)

If you develop any useful expiration managers that you think should be distributed with Memoize, please let me know.

CAVEATS

This module is experimental, and may contain bugs. Please report bugs to the address below.

Number-of-uses is stored as a 16-bit unsigned integer, so can't exceed 65535.

Because of clock granularity, expiration times may occur up to one second sooner than you expect. For example, suppose you store a value with a lifetime of ten seconds, and you store it at 12:00:00.998 on a certain day. Memoize will look at the clock and see 12:00:00. Then 9.01 seconds later, at 12:00:10.008 you try to read it back. Memoize will look at the clock and see 12:00:10 and conclude that the value has expired. This will probably not occur if you have `Time::HiRes` installed.

AUTHOR

Mark-Jason Dominus

Mike Cariaso provided valuable insight into the best way to solve this problem.

SEE ALSO

perl(1)

The Memoize man page.

Memoize::NDBM_File

glue to provide EXISTS for NDBM_File for Storable use

DESCRIPTION

This class provides EXISTS support for *NDBM_File*.

In Perl 5.37.3, support for `EXISTS` was added to *NDBM_File* itself. Any code which already requires perl >= 5.37.3 should be rewritten to use *NDBM_File* directly.

Memoize::SDBM_File

DEPRECATED compatibility shim

DESCRIPTION

This class used to provide EXISTS support for *SDBM_File* before support for `EXISTS` was added to *SDBM_File* itself in Perl 5.6.0.

Any code still using this class should be rewritten to use *SDBM_File* directly.

Memoize::Storable

store Memoized data in Storable database

DESCRIPTION

See *Memoize*.

memoize

Memoize a function: `memoize('name', %options)` or `memoize(\&code, %options)`.

Returns a code reference to the memoized wrapper. Given a name, the wrapper is also installed under that name (qualified with the caller's package unless it contains `::`), so every existing caller now goes through the cache; `\&name` changes accordingly. Given a code reference, nothing is installed unless `INSTALL` says so.

Options:

- `INSTALL => 'name'` installs the wrapper under `name` instead; `INSTALL => undef` installs it nowhere.
- `NORMALIZER => 'name'` or `=> \&code` computes the cache key from the argument list (called in the same context as the wrapper; in list context its first return value is the key). Without it the key is the arguments joined with `chr(28)`, so `f(1, 2)` and `f("1", "2")` share an entry and references are keyed by address.
- `SCALAR_CACHE` and `LIST_CACHE` choose the storage for results computed in scalar and list context: `MEMORY` (the default), `[HASH => \%hash]` to use your hash (tie it to a DBM file for a persistent cache), `FAULT` to croak when called in that context, and `MERGE` to share one cache between the two contexts. `TIE` is accepted for compatibility and warns that it is deprecated.

Croaks `Cannot operate on nonexistent function` for a name that does not name a defined sub, and `Usage: memoize 'functionname'|coderef {OPTIONS}` for anything that is neither a name nor a plain code reference.

A `die` inside the function propagates and caches nothing. A recursive function benefits fully: its recursive calls reach the cache too, so `fib(50)` becomes linear. Only memoize functions whose result depends solely on their arguments and that have no side effects - that is your assertion, not something the module checks.

flush_cache

Empty a memoized function's caches: `flush_cache('name')` or `flush_cache(\&wrapper)`.

Both the scalar- and the list-context cache are cleared; the function stays memoized and starts refilling on the next call. Croaks `... not memoized` for a function that is not currently memoized, and `Tied cache hash for ... does not support flushing` when a `HASH` cache is tied to a class without a `CLEAR` method. Not exported by default: `use Memoize qw(flush_cache)`.

unmemoize

Undo `memoize`: `unmemoize('name')` or `unmemoize(\&wrapper)`.

Reinstalls the original function under the name the wrapper was installed at and returns a code reference to the original. The cache is discarded with the wrapper; a wrapper reference you still hold keeps working but can no longer be flushed or unmemoized. Croaks `Could not unmemoize function ..., because it was not memoized to begin with` otherwise. Not exported by default: `use Memoize qw(unmemoize)`.

5.2.67 Module

Namespace containing the following modules:

- `Module::CoreList` - what modules shipped with versions of perl
- `Module::Load` - runtime require of both modules and files
- `Module::Loaded` - mark modules as loaded or unloaded
- `Module::Metadata` - Gather package and POD information from perl module files

Module::CoreList

what modules shipped with versions of perl

SYNOPSIS

```
use Module::CoreList;

print $Module::CoreList::version{5.00503}{CPAN}; # prints 1.48

print Module::CoreList->first_release('File::Spec');
# prints 5.00405

print Module::CoreList->first_release_by_date('File::Spec');
# prints 5.005

print Module::CoreList->first_release('File::Spec', 0.82);
# prints 5.006001

if (Module::CoreList::is_core('File::Spec')) {
  print "File::Spec is a core module\n";
}

print join ', ', Module::CoreList->find_modules(qr/Data/);
   # prints 'Data::Dumper'
print join ', ',
         Module::CoreList->find_modules(qr/test::h.*::.*s/i, 5.
↪008008);
   # prints 'Test::Harness::Assert, Test::Harness::Straps'
```

(continues on next page)

(continued from previous page)

```
print join ", ", @{ $Module::CoreList::families{5.005} };
   # prints "5.005, 5.00503, 5.00504"
```

DESCRIPTION

Module::CoreList provides information on which core and dual-life modules shipped with each version of perl.

It provides a number of mechanisms for querying this information.

There is a utility called corelist provided with this module which is a convenient way of querying from the command-line.

There is a functional programming API available for programmers to query information.

Programmers may also query the contained hash structures to find relevant information.

FUNCTIONS API

These are the functions that are available, they may either be called as functions or class methods:

```
Module::CoreList::first_release('File::Spec'); # as a function

Module::CoreList->first_release('File::Spec'); # class method
```

- `first_release( MODULE )`

  Behaviour since version 2.11

  Requires a MODULE name as an argument, returns the perl version when that module first appeared in core as ordered by perl version number or undef (in scalar context) or an empty list (in list context) if that module is not in core.

- `first_release_by_date( MODULE )`

  Requires a MODULE name as an argument, returns the perl version when that module first appeared in core as ordered by release date or undef (in scalar context) or an empty list (in list context) if that module is not in core.

- `find_modules( REGEX, [ LIST OF PERLS ] )`

  Takes a regex as an argument, returns a list of modules that match the regex given. If only a regex is provided applies to all modules in all perl versions. Optionally you may provide a list of perl versions to limit the regex search.

- `find_version( PERL_VERSION )`

  Takes a perl version as an argument. Upon successful completion, returns a reference to a hash. Each element of that hash has a key which is the name of a module (*e.g.,* 'File::Path') shipped with that version of perl and a value which is the version number (*e.g.,* '2.09') of that module which shipped with that version of perl . Returns `undef` otherwise.

- `is_core( MODULE, [ MODULE_VERSION, [ PERL_VERSION ] ] )`

  Available in version 2.99 and above.

  Returns true if MODULE was bundled with the specified version of Perl. You can optionally specify a minimum version of the module, and can also specify a version of Perl. If a version of Perl isn't specified, `is_core()` will use the numeric version of Perl that is running (ie `$]`).

  If you want to specify the version of Perl, but don't care about the version of the module, pass `undef` for the module version:

- `is_deprecated( MODULE, PERL_VERSION )`

  Available in version 2.22 and above.

  Returns true if MODULE is marked as deprecated in PERL_VERSION. If PERL_VERSION is omitted, it defaults to the current version of Perl.

- `deprecated_in( MODULE )`

  Available in version 2.77 and above.

  Returns the first perl version where the MODULE was marked as deprecated. Returns `undef` if the MODULE has not been marked as deprecated.

- `removed_from( MODULE )`

  Available in version 2.32 and above

  Takes a module name as an argument, returns the first perl version where that module was removed from core. Returns undef if the given module was never in core or remains in core.

- `removed_from_by_date( MODULE )`

  Available in version 2.32 and above

  Takes a module name as an argument, returns the first perl version by release date where that module was removed from core. Returns undef if the given module was never in core or remains in core.

- `changes_between( PERL_VERSION, PERL_VERSION )`

  Available in version 2.66 and above.

  Given two perl versions, this returns a list of pairs describing the changes in core module content between them. The list is suitable for storing in a hash. The keys are library names and the values are hashrefs. Each hashref has an entry for one or both of `left` and `right`, giving the versions of the library in each of the left and right perl distributions.

  For example, it might return these data (among others) for the difference between 5.008000 and 5.008001:

```
'Pod::ParseLink'  => { left => '1.05', right => '1.06' },
'Pod::ParseUtils' => { left => '0.22', right => '0.3'  },
'Pod::Perldoc'    => {                 right => '3.10' },
'Pod::Perldoc::BaseTo' => {            right => undef  },
```

This shows us two libraries being updated and two being added, one of which has an undefined version in the right-hand side version.

DATA STRUCTURES

These are the hash data structures that are available:

- `%Module::CoreList::version`

  A hash of hashes that is keyed on perl version as indicated in $]. The second level hash is module => version pairs.

  Note, it is possible for the version of a module to be unspecified, whereby the value is `undef`, so use `exists $version{$foo}{$bar}` if that's what you're testing for.

  Starting with 2.10, the special module name `Unicode` refers to the version of the Unicode Character Database bundled with Perl.

- `%Module::CoreList::delta`

  Available in version 3.00 and above.

  It is a hash of hashes that is keyed on perl version. Each keyed hash will have the following keys:

```
delta_from - a previous perl version that the changes are based
 ↪on
changed    - a hash of module/versions that have changed
removed    - a hash of modules that have been removed
```

- `%Module::CoreList::released`

  Keyed on perl version this contains ISO formatted versions of the release dates, as gleaned from perlhist.

- `%Module::CoreList::families`

  New, in 1.96, a hash that clusters known perl releases by their major versions.

- `%Module::CoreList::deprecated`

  A hash of hashes keyed on perl version and on module name. If a module is defined it indicates that that module is deprecated in that perl version and is scheduled for removal from core at some future point.

- `%Module::CoreList::upstream`

  A hash that contains information on where patches should be directed for each core module.

  UPSTREAM indicates where patches should go. `undef` implies that this hasn't been discussed for the module at hand. `blead` indicates that the copy of the module in the blead sources is to be considered canonical, `cpan` means that the module on CPAN is to be patched first. `first-come` means that blead can be patched freely if it is in sync with the latest release on CPAN.

- `%Module::CoreList::bug_tracker`

  A hash that contains information on the appropriate bug tracker for each core module.

BUGS is an email or url to post bug reports. For modules with UPSTREAM => 'blead', use mailto:perl5-porters@perl.org. rt.cpan.org appears to automatically provide a URL for CPAN modules; any value given here overrides the default: http://rt.cpan.org/Public/Dist/Display.html?Name=\protect\TU\textdollarModuleName

CAVEATS

Module::CoreList currently covers the 5.000, 5.001, 5.002, 5.003_07, 5.004, 5.004_05, 5.005, 5.005_03, 5.005_04 and 5.7.3 releases of perl.

All stable releases of perl since 5.6.0 are covered.

All development releases of perl since 5.9.0 are covered.

HISTORY

Moved to Changes file.

AUTHOR

Richard Clamp richardc@unixbeard.net

Currently maintained by the perl 5 porters perl5-porters@perl.org.

LICENSE

Copyright (C) 2002-2009 Richard Clamp. All Rights Reserved.

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

SEE ALSO

corelist, Module::Info, perl, http://perlpunks.de/corelist

Modules

- `Module::CoreList::Utils` - what utilities shipped with versions of perl

Module::CoreList::Utils

what utilities shipped with versions of perl

SYNOPSIS

```
use Module::CoreList::Utils;

print $Module::CoreList::Utils::utilities{5.009003}{ptar}; # prints␣
 ↪1

print Module::CoreList::Utils->first_release('corelist');
```

(continues on next page)

(continued from previous page)

```
# prints 5.008009

print Module::CoreList::Utils->first_release_by_date('corelist');
# prints 5.009002
```

DESCRIPTION

Module::CoreList::Utils provides information on which core and dual-life utilities shipped with each version of perl.

It provides a number of mechanisms for querying this information.

There is a functional programming API available for programmers to query information.

Programmers may also query the contained hash structure to find relevant information.

FUNCTIONS API

These are the functions that are available, they may either be called as functions or class methods:

```
Module::CoreList::Utils::first_release('corelist'); # as a function

Module::CoreList::Utils->first_release('corelist'); # class method
```

- `utilities`

  Requires a perl version as an argument, returns a list of utilities that shipped with that version of perl, or undef/empty list if that perl doesn't exist.

- `first_release( UTILITY )`

  Requires a UTILITY name as an argument, returns the perl version when that utility first appeared in core as ordered by perl version number or undef (in scalar context) or an empty list (in list context) if that utility is not in core.

- `first_release_by_date( UTILITY )`

  Requires a UTILITY name as an argument, returns the perl version when that utility first appeared in core as ordered by release date or undef (in scalar context) or an empty list (in list context) if that utility is not in core.

- `removed_from( UTILITY )`

  Takes a UTILITY name as an argument, returns the first perl version where that utility was removed from core. Returns undef if the given utility was never in core or remains in core.

- `removed_from_by_date( UTILITY )`

  Takes a UTILITY name as an argument, returns the first perl version by release date where that utility was removed from core. Returns undef if the given utility was never in core or remains in core.

DATA STRUCTURES

These are the hash data structures that are available:

- `%Module::CoreList::Utils::utilities`

  A hash of hashes that is keyed on perl version as indicated in $]. The second level hash is utility / defined pairs.

AUTHOR

Chris `BinGOs` Williams chris@bingosnet.co.uk

Currently maintained by the perl 5 porters perl5-porters@perl.org.

This module is the result of archaeology undertaken during QA Hackathon in Lancaster, April 2013.

LICENSE

Copyright (C) 2013 Chris Williams. All Rights Reserved.

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

SEE ALSO

corelist, *Module::CoreList*, perl, http://perlpunks.de/corelist

Module::Load

runtime require of both modules and files

SYNOPSIS

```
use Module::Load;

my $module = 'Data::Dumper';

load Data::Dumper;     # loads that module, but not import any␣
↪functions
                       # -> cannot use 'Dumper' function

load 'Data::Dumper';   # ditto
load $module           # tritto

autoload Data::Dumper; # loads that module and imports the default␣
↪functions
                       # -> can use 'Dumper' function

my $script = 'some/script.pl'
load $script;
```

(continues on next page)

(continued from previous page)

```
load 'some/script.pl';  # use quotes because of punctuations

load thing;             # try 'thing' first, then 'thing.pm'

load CGI, ':all';       # like 'use CGI qw[:standard]'
```

DESCRIPTION

`Module::Load` eliminates the need to know whether you are trying to require either a file or a module.

If you consult `perldoc -f require` you will see that `require` will behave differently when given a bareword or a string.

In the case of a string, `require` assumes you are wanting to load a file. But in the case of a bareword, it assumes you mean a module.

This gives nasty overhead when you are trying to dynamically require modules at runtime, since you will need to change the module notation (`Acme::Comment`) to a file notation fitting the particular platform you are on.

`Module::Load` eliminates the need for this overhead and will just DWYM.

Difference between `load` and `autoload`

`Module::Load` imports the two functions - `load` and `autoload`

`autoload` imports the default functions automatically, but `load` do not import any functions.

`autoload` is usable under `BEGIN{};`.

Both the functions can import the functions that are specified.

Following codes are same.

```
load File::Spec::Functions, qw/splitpath/;

autoload File::Spec::Functions, qw/splitpath/;
```

FUNCTIONS

- load

  Loads a specified module.

  See “Rules” for detailed loading rule.

- autoload

  Loads a specified module and imports the default functions.

  Except importing the functions, ‘autoload’ is same as ‘load’.

- load_remote

  Loads a specified module to the specified package.

```
use Module::Load 'load_remote';

my $pkg = 'Other::Package';

load_remote $pkg, 'Data::Dumper'; # load a module to
 ↪'Other::Package'
                                  # but do not import 'Dumper'␣
 ↪function
```

  A module for loading must be quoted.

  Except specifing the package and quoting module name, 'load_remote' is same as 'load'.

- autoload_remote

  Loads a specified module and imports the default functions to the specified package.

```
use Module::Load 'autoload_remote';

my $pkg = 'Other::Package';

autoload_remote $pkg, 'Data::Dumper'; # load a module to
 ↪'Other::Package'
                                      # and imports 'Dumper'␣
 ↪function
```

  A module for loading must be quoted.

  Except specifing the package and quoting module name, 'autoload_remote' is same as 'load_remote'.

Rules

All functions have the following rules to decide what it thinks you want:

- If the argument has any characters in it other than those matching `\w`, `:` or `'`, it must be a file
- If the argument matches only `[\w:']`, it must be a module
- If the argument matches only `\w`, it could either be a module or a file. We will try to find `file.pm` first in `@INC` and if that fails, we will try to find `file` in @INC. If both fail, we die with the respective error messages.

IMPORTS THE FUNCTIONS

'load' and 'autoload' are imported by default, but 'load_remote' and 'autoload_remote' are not imported.

To use 'load_remote' or 'autoload_remote', specify at 'use'.

- “load”,”autoload”,”load_remote”,”autoload_remote”

  Imports the selected functions.

```
# imports 'load' and 'autoload' (default)
use Module::Load;

# imports 'autoload' only
use Module::Load 'autoload';

# imports 'autoload' and 'autoload_remote', but don't import
 ↪'load';
use Module::Load qw/autoload autoload_remote/;
```

- ‘all’

  Imports all the functions.

```
use Module::Load 'all'; # imports load, autoload, load_remote,␣
 ↪autoload_remote
```

- ‘’,’none’,undef

  Not import any functions (`load` and `autoload` are not imported).

```
use Module::Load '';

use Module::Load 'none';

use Module::Load undef;
```

Caveats

Because of a bug in perl (#19213), at least in version 5.6.1, we have to hardcode the path separator for a require on Win32 to be `/`, like on Unix rather than the Win32 `\`. Otherwise perl will not read its own %INC accurately double load files if they are required again, or in the worst case, core dump.

`Module::Load` cannot do implicit imports, only explicit imports. (in other words, you always have to specify explicitly what you wish to import from a module, even if the functions are in that modules’ `@EXPORT`)

SEE ALSO

Module::Runtime provides functions for loading modules, checking the validity of a module name, converting a module name to partial `.pm` path, and related utility functions.

“require” in perlfunc and “use” in perlfunc.

Mojo::Loader is a “class loader and plugin framework”, and is included in the Mojolicious distribution.

Module::Loader is a module for finding and loading modules in a given namespace, inspired by `Mojo::Loader`.

ACKNOWLEDGEMENTS

Thanks to Jonas B. Nielsen for making explicit imports work.

BUG REPORTS

Please report bugs or other issues to bug-module-load@rt.cpan.org.

AUTHOR

This module by Jos Boumans kane@cpan.org.

COPYRIGHT

This library is free software; you may redistribute and/or modify it under the same terms as Perl itself.

Modules

- `Module::Load::Conditional` - Looking up module information / loading at runtime

Module::Load::Conditional

Looking up module information / loading at runtime

SYNOPSIS

```
use Module::Load::Conditional qw[can_load check_install requires];


my $use_list = {
        CPANPLUS        => 0.05,
        LWP             => 5.60,
        'Test::More'    => undef,
};

print can_load( modules => $use_list )
        ? 'all modules loaded successfully'
        : 'failed to load required modules';


my $rv = check_install( module => 'LWP', version => 5.60 )
            or print 'LWP is not installed!';

print 'LWP up to date' if $rv->{uptodate};
print "LWP version is $rv->{version}\n";
print "LWP is installed as file $rv->{file}\n";
```

(continues on next page)

(continued from previous page)

```
print "LWP requires the following modules to be installed:\n";
print join "\n", requires('LWP');

### allow M::L::C to peek in your %INC rather than just
### scanning @INC
$Module::Load::Conditional::CHECK_INC_HASH = 1;

### reset the 'can_load' cache
undef $Module::Load::Conditional::CACHE;

### don't have Module::Load::Conditional issue warnings --
### default is '1'
$Module::Load::Conditional::VERBOSE = 0;

### The last error that happened during a call to 'can_load'
my $err = $Module::Load::Conditional::ERROR;
```

DESCRIPTION

Module::Load::Conditional provides simple ways to query and possibly load any of the modules you have installed on your system during runtime.

It is able to load multiple modules at once or none at all if one of them was not able to load. It also takes care of any error checking and so forth.

Methods

$href = check_install(module => NAME [, version => VERSION, verbose => BOOL]);

`check_install` allows you to verify if a certain module is installed or not. You may call it with the following arguments:

- module

  The name of the module you wish to verify – this is a required key

- version

  The version this module needs to be – this is optional

- verbose

  Whether or not to be verbose about what it is doing – it will default to $Module::Load::Conditional::VERBOSE

It will return undef if it was not able to find where the module was installed, or a hash reference with the following keys if it was able to find the file:

- file

  Full path to the file that contains the module

- dir

Directory, or more exact the `@INC` entry, where the module was loaded from.

- version

  The version number of the installed module - this will be `undef` if the module had no (or unparsable) version number, or if the variable `$Module::Load::Conditional::FIND_VERSION` was set to true. (See the `GLOBAL VARIABLES` section below for details)

- uptodate

  A boolean value indicating whether or not the module was found to be at least the version you specified. If you did not specify a version, uptodate will always be true if the module was found. If no parsable version was found in the module, uptodate will also be true, since `check_install` had no way to verify clearly.

  See also `$Module::Load::Conditional::DEPRECATED`, which affects the outcome of this value.

$bool = can_load(modules => { NAME => VERSION [,NAME => VERSION] }, [verbose => BOOL, nocache => BOOL, autoload => BOOL])

`can_load` will take a list of modules, optionally with version numbers and determine if it is able to load them. If it can load *ALL* of them, it will. If one or more are unloadable, none will be loaded.

This is particularly useful if you have More Than One Way (tm) to solve a problem in a program, and only wish to continue down a path if all modules could be loaded, and not load them if they couldn't.

This function uses the `load` function or the `autoload_remote` function from Module::Load under the hood.

`can_load` takes the following arguments:

- modules

  This is a hashref of module/version pairs. The version indicates the minimum version to load. If no version is provided, any version is assumed to be good enough.

- verbose

  This controls whether warnings should be printed if a module failed to load. The default is to use the value of $Module::Load::Conditional::VERBOSE.

- nocache

  `can_load` keeps its results in a cache, so it will not load the same module twice, nor will it attempt to load a module that has already failed to load before. By default, `can_load` will check its cache, but you can override that by setting `nocache` to true.

- autoload

  This controls whether imports the functions of a loaded modules to the caller package. The default is no importing any functions.

  See the `autoload` function and the `autoload_remote` function from *Module::Load* for details.

@list = requires(MODULE);

`requires` can tell you what other modules a particular module requires. This is particularly useful when you're intending to write a module for public release and are listing its prerequisites.

`requires` takes but one argument: the name of a module. It will then first check if it can actually load this module, and return undef if it can't. Otherwise, it will return a list of modules and pragmas that would have been loaded on the module's behalf.

Note: The list `require` returns has originated from your current perl and your current install.

Global Variables

The behaviour of Module::Load::Conditional can be altered by changing the following global variables:

$Module::Load::Conditional::VERBOSE

This controls whether Module::Load::Conditional will issue warnings and explanations as to why certain things may have failed. If you set it to 0, Module::Load::Conditional will not output any warnings. The default is 0;

$Module::Load::Conditional::FIND_VERSION

This controls whether Module::Load::Conditional will try to parse (and eval) the version from the module you're trying to load.

If you don't wish to do this, set this variable to `false`. Understand then that version comparisons are not possible, and Module::Load::Conditional can not tell you what module version you have installed. This may be desirable from a security or performance point of view. Note that `$FIND_VERSION` code runs safely under `taint mode`.

The default is 1;

$Module::Load::Conditional::CHECK_INC_HASH

This controls whether `Module::Load::Conditional` checks your `%INC` hash to see if a module is available. By default, only `@INC` is scanned to see if a module is physically on your filesystem, or available via an `@INC-hook`. Setting this variable to `true` will trust any entries in `%INC` and return them for you.

The default is 0;

$Module::Load::Conditional::FORCE_SAFE_INC

This controls whether `Module::Load::Conditional` sanitises `@INC` by removing ".". The current default setting is `0`, but this may change in a future release.

$Module::Load::Conditional::CACHE

This holds the cache of the `can_load` function. If you explicitly want to remove the current cache, you can set this variable to `undef`

$Module::Load::Conditional::ERROR

This holds a string of the last error that happened during a call to `can_load`. It is useful to inspect this when `can_load` returns `undef`.

$Module::Load::Conditional::DEPRECATED

This controls whether `Module::Load::Conditional` checks if a dual-life core module has been deprecated. If this is set to true `check_install` will return false to `uptodate`, if a dual-life module is found to be loaded from `$Config{privlibexp}`

The default is 0;

See Also

`Module::Load`

BUG REPORTS

Please report bugs or other issues to bug-module-load-conditional@rt.cpan.org.

AUTHOR

This module by Jos Boumans kane@cpan.org.

COPYRIGHT

This library is free software; you may redistribute and/or modify it under the same terms as Perl itself.

Module::Loaded

mark modules as loaded or unloaded

SYNOPSIS

```
use Module::Loaded;

$bool = mark_as_loaded('Foo');   # Foo.pm is now marked as loaded
$loc  = is_loaded('Foo');        # location of Foo.pm set to the
                                 # loaders location
eval "require 'Foo'";            # is now a no-op

$bool = mark_as_unloaded('Foo'); # Foo.pm no longer marked as loaded
eval "require 'Foo'";            # Will try to find Foo.pm in @INC
```

DESCRIPTION

When testing applications, often you find yourself needing to provide functionality in your test environment that would usually be provided by external modules. Rather than munging the `%INC` by hand to mark these external modules as loaded, so they are not attempted to be loaded by perl, this module offers you a very simple way to mark modules as loaded and/or unloaded.

FUNCTIONS

$bool = mark_as_loaded(PACKAGE);

Marks the package as loaded to perl. `PACKAGE` can be a bareword or string.

If the module is already loaded, `mark_as_loaded` will carp about this and tell you from where the `PACKAGE` has been loaded already.

$bool = mark_as_unloaded(PACKAGE);

Marks the package as unloaded to perl, which is the exact opposite of `mark_as_loaded`. `PACKAGE` can be a bareword or string.

If the module is already unloaded, `mark_as_unloaded` will carp about this and tell you the `PACKAGE` has been unloaded already.

$loc = is_loaded(PACKAGE);

`is_loaded` tells you if `PACKAGE` has been marked as loaded yet. `PACKAGE` can be a bareword or string.

It returns falls if `PACKAGE` has not been loaded yet and the location from where it is said to be loaded on success.

BUG REPORTS

Please report bugs or other issues to <bug-module-loaded@rt.cpan.org<gt>.

AUTHOR

This module by Jos Boumans kane@cpan.org.

COPYRIGHT

This library is free software; you may redistribute and/or modify it under the same terms as Perl itself.

Module::Metadata

Gather package and POD information from perl module files

VERSION

version 1.000039

SYNOPSIS

```
use Module::Metadata;

# information about a .pm file
my $info = Module::Metadata->new_from_file( $file );
my $version = $info->version;

# CPAN META 'provides' field for .pm files in a directory
my $provides = Module::Metadata->provides(
  dir => 'lib', version => 2
);
```

DESCRIPTION

This module provides a standard way to gather metadata about a .pm file through (mostly) static analysis and (some) code execution. When determining the version of a module, the `$VERSION` assignment is `eval`ed, as is traditional in the CPAN toolchain.

CLASS METHODS

new_from_file($filename, collect_pod => 1, decode_pod => 1)

Constructs a `Module::Metadata` object given the path to a file. Returns undef if the filename does not exist.

`collect_pod` is a optional boolean argument that determines whether POD data is collected and stored for reference. POD data is not collected by default. POD headings are always collected.

If the file begins by an UTF-8, UTF-16BE or UTF-16LE byte-order mark, then it is skipped before processing, and the content of the file is also decoded appropriately starting from perl 5.8.

Alternatively, if `decode_pod` is set, it will decode the collected pod sections according to the `=encoding` declaration.

new_from_handle($handle, $filename, collect_pod => 1, decode_pod => 1)

This works just like `new_from_file`, except that a handle can be provided as the first argument.

Note that there is no validation to confirm that the handle is a handle or something that can act like one. Passing something that isn't a handle will cause a exception when trying to read from it. The `filename` argument is mandatory or undef will be returned.

You are responsible for setting the decoding layers on `$handle` if required.

new_from_module($module, collect_pod => 1, inc => \@dirs, decode_pod => 1)

Constructs a `Module::Metadata` object given a module or package name. Returns undef if the module cannot be found.

In addition to accepting the `collect_pod` and `decode_pod` arguments as described above, this method accepts a `inc` argument which is a reference to an array of directories to search for the module. If none are given, the default is @INC.

If the file that contains the module begins by an UTF-8, UTF-16BE or UTF-16LE byte-order mark, then it is skipped before processing, and the content of the file is also decoded appropriately starting from perl 5.8.

find_module_by_name($module, \@dirs)

Returns the path to a module given the module or package name. A list of directories can be passed in as an optional parameter, otherwise @INC is searched.

Can be called as either an object or a class method.

find_module_dir_by_name($module, \@dirs)

Returns the entry in `@dirs` (or `@INC` by default) that contains the module `$module`. A list of directories can be passed in as an optional parameter, otherwise @INC is searched.

Can be called as either an object or a class method.

provides(%options)

This is a convenience wrapper around `package_versions_from_directory` to generate a CPAN META `provides` data structure. It takes key/value pairs. Valid option keys include:

- version **(required)**

  Specifies which version of the *CPAN::Meta::Spec* should be used as the format of the `provides` output. Currently only '1.4' and '2' are supported (and their format is identical). This may change in the future as the definition of `provides` changes.

  The `version` option is required. If it is omitted or if an unsupported version is given, then `provides` will throw an error.

- dir

  Directory to search recursively for `.pm` files. May not be specified with `files`.

- files

  Array reference of files to examine. May not be specified with `dir`.

- prefix

  String to prepend to the `file` field of the resulting output. This defaults to `lib`, which is the common case for most CPAN distributions with their `.pm` files in `lib`. This option ensures the META information has the correct relative path even when

the `dir` or `files` arguments are absolute or have relative paths from a location other than the distribution root.

For example, given `dir` of 'lib' and `prefix` of 'lib', the return value is a hashref of the form:

```
{
  'Package::Name' => {
    version => '0.123',
    file => 'lib/Package/Name.pm'
  },
  'OtherPackage::Name' => ...
}
```

package_versions_from_directory($dir, \@files?)

Scans `$dir` for .pm files (unless `@files` is given, in which case looks for those files in `$dir` - and reads each file for packages and versions, returning a hashref of the form:

```
{
  'Package::Name' => {
    version => '0.123',
    file => 'Package/Name.pm'
  },
  'OtherPackage::Name' => ...
}
```

The `DB` and `main` packages are always omitted, as are any "private" packages that have leading underscores in the namespace (e.g. `Foo::_private`)

Note that the file path is relative to `$dir` if that is specified. This **must not** be used directly for CPAN META `provides`. See the `provides` method instead.

log_info (internal)

Used internally to perform logging; imported from Log::Contextual if Log::Contextual has already been loaded, otherwise simply calls warn.

OBJECT METHODS

name()

Returns the name of the package represented by this module. If there is more than one package, it makes a best guess based on the filename. If it's a script (i.e. not a *.pm) the package name is 'main'.

version($package)

Returns the version as defined by the $VERSION variable for the package as returned by the `name` method if no arguments are given. If given the name of a package it will attempt to return the version of that package if it is specified in the file.

`filename()`

Returns the absolute path to the file. Note that this file may not actually exist on disk yet, e.g. if the module was read from an in-memory filehandle.

`packages_inside()`

Returns a list of packages. Note: this is a raw list of packages discovered (or assumed, in the case of `main`). It is not filtered for `DB`, `main` or private packages the way the `provides` method does. Invalid package names are not returned, for example "Foo:Bar". Strange but valid package names are returned, for example "Foo::Bar::", and are left up to the caller on how to handle.

`pod_inside()`

Returns a list of POD sections.

`contains_pod()`

Returns true if there is any POD in the file.

`pod($section)`

Returns the POD data in the given section.

`is_indexable($package)` or `is_indexable()`

Available since version 1.000020.

Returns a boolean indicating whether the package (if provided) or any package (otherwise) is eligible for indexing by PAUSE, the Perl Authors Upload Server. Note This only checks for valid `package` declarations, and does not take any ownership information into account.

GIVING THANKS

If you found this module to be useful, please show your appreciation by adding a +1 in MetaCPAN and a star in GitHub.

SUPPORT

Bugs may be submitted through the RT bug tracker (or bug-Module-Metadata@rt.cpan.org).

There is also a mailing list available for users of this distribution, at http://lists.perl.org/list/cpan-workers.html.

There is also an irc channel available for users of this distribution, at `#toolchain` on `irc.perl.org`.

AUTHOR

Original code from Module::Build::ModuleInfo by Ken Williams kwilliams@cpan.org, Randy W. Sims RandyS@ThePierianSpring.org

Released as Module::Metadata by Matt S Trout (mst) mst@shadowcat.co.uk with assistance from David Golden (xdg) dagolden@cpan.org.

CONTRIBUTORS

- Karen Etheridge ether@cpan.org
- David Golden dagolden@cpan.org
- Vincent Pit perl@profvince.com
- Matt S Trout mst@shadowcat.co.uk
- Chris Nehren apeiron@cpan.org
- Graham Knop haarg@haarg.org
- Olivier Mengué dolmen@cpan.org
- Tomas Doran bobtfish@bobtfish.net
- Christian Walde walde.christian@googlemail.com
- Craig A. Berry cberry@cpan.org
- Tatsuhiko Miyagawa miyagawa@bulknews.net
- tokuhirom tokuhirom@gmail.com
- Chris 'BinGOs' Williams chris@bingosnet.co.uk
- David Mitchell davem@iabyn.com
- David Steinbrunner dsteinbrunner@pobox.com
- Edward Zborowski ed@rubensteintech.com
- Erik Huelsmann ehuels@gmail.com
- Gareth Harper gareth@broadbean.com
- James Raspass jraspass@gmail.com
- Jerry D. Hedden jdhedden@cpan.org
- Josh Jore jjore@cpan.org
- Kent Fredric kentnl@cpan.org
- Leon Timmermans fawaka@gmail.com
- Peter Rabbitson ribasushi@cpan.org
- Steve Hay steve.m.hay@googlemail.com

COPYRIGHT & LICENSE

Original code Copyright (c) 2001-2011 Ken Williams. Additional code Copyright (c) 2010-2011 Matt Trout and David Golden. All rights reserved.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

5.2.68 Moose

The compiled part of Moose and Class::MOP: the attribute readers of the metaclass objects, the method map cache, and a few helpers of Moose::Exporter, the type constraints and role application.

Moose's own `.pm` files load this when `Class::MOP` is loaded; it is not used directly. `Class::MOP::get_code_info` is the one function meant to be called by other code.

This is a native implementation of Moose 2.4001's `mop.c` and `xs/*.xs`; its `.pm` files are used unchanged.

Synopsis

```
use Class::MOP;

my ($package, $name) = Class::MOP::get_code_info(\&Foo::bar);  # (
↪'Foo', 'bar')
my $meta = Class::MOP::Class->initialize('Foo');
print $meta->name, "\n";                          # Foo
print $meta->get_method('bar')->is_stub ? "stub\n" : "defined\n";
```

Differences from upstream

- `_method_map` and `is_stub` called on something that is not a reference to a hash die with the readers' messages, "can't call _method_map as a class method" and "object is not a hashref"; perl crashes. `_method_map` on an object with no `package` entry treats it as undefined, as perl does for an undefined one; perl crashes.
- `is_stub` on a method whose body is not a code reference dies "method body is not a CODE reference"; perl crashes, or reads the value as code and answers at random.
- Moose::Exporter's `_flag_as_reexport` does nothing to what is not a reference, and `_export_is_flagged` answers false for it; perl crashes on both.
- Applying a role to an instance that is not a blessed reference (`_reset_amagic`) does nothing; perl crashes.
- `_reset_amagic` no longer walks every scalar of the program looking for other references to the object: in perl 5.44 the overloading flag it sets on each of them lives on the class, which the first one has already set. The result is the same, without a cost that grows with the size of the program.

5.2.69 NDBM_File

Tie a Perl hash to an ndbm database file, so keys and values persist between runs.

NDBM_File is a tied-hash interface: bind a hash to a database with `tie`, and from then on `$h{key}`, `delete`, `exists`, `keys` and `each` read and write the file. Most programs reach it through `AnyDBM_File`, which picks NDBM_File first.

Synopsis

```
use Fcntl;
use NDBM_File;

tie my %h, 'NDBM_File', 'mydb', O_RDWR|O_CREAT, 0640
    or die "tie failed: $!";
$h{greeting} = 'hello';
untie %h;
```

Opening a database

```
tie %h, 'NDBM_File', $basename, $flags, $mode;
```

`$flags` are `Fcntl`'s `O_RDONLY`, `O_RDWR`, `O_CREAT`, `O_TRUNC`; `$mode` is the permission of the files it creates. `tie` returns the database object, or undef with `$!` set when the file cannot be opened.

DBM filters

`(tied %h)->filter_store_key(sub {...})`, `filter_store_value`, `filter_fetch_key` and `filter_fetch_value` install a sub that rewrites every key or value on its way into, or out of, the database. The sub changes `$_` in place; a store filter works on a copy, so your own variable keeps its value. Each method returns the filter it replaces (or undef), and `undef` as the argument removes the filter. The `DBM_Filter` module offers ready-made ones (`utf8`, `null`, `int32` and others). A filter that uses the same database croaks `recursion detected in filter_store_key` (or the filter's own name).

Files

pperl keeps the data in an LMDB file, `NAME.pag`, with an empty `NAME.dir` beside it, as ndbm has the two, and LMDB's lock file `NAME.pag-lock`. The ndbm library is not used, and its files cannot be read: tying a real gdbm or Berkeley DB file croaks, naming what the file is, and the file is left as it was. A key may be at most 511 bytes, 510 if it starts with a NUL byte; a longer one croaks.

Writes reach the file when the hash is untied: until then the tie holds the file's one writer slot, and another process that writes the file waits for it (readers never wait). Set `$Peta::FFI::LMDB::DBM_WRITE = 'through'` before `tie` to commit every write as it happens, for files that several processes write; see `Peta::FFI::LMDB`.

Differences from upstream

- The files are LMDB files: real ndbm (gdbm) and Berkeley DB files are refused, and pperl's files cannot be read by perl 5's NDBM_File. Covered by `t/82-native/Peta-FFI-LMDB/060-store.t`.
- Keys are limited to 511 bytes. Covered by `t/82-native/Peta-FFI-LMDB/060-store.t`.
- Writes reach the file at `untie` (or each one, with `$Peta::FFI::LMDB::DBM_WRITE = 'through'`), and a writer holds the file until then. Covered by `t/82-native/Peta-FFI-LMDB/060-store.t`.
- Deleting the key `each` just returned is safe: the walk goes on with the next key. Covered by `t/82-native/Peta-FFI-LMDB/060-store.t`.

Everything else behaves as with perl 5's NDBM_File. Covered by its own test, `t/81-xs-native/NDBM_File/100-upstream-ndbm.t`.

Functions

Errors

error

`(tied %h)->error`: true when an operation on the database has failed since it was tied or since the last `clearerr` - a `delete` on a database tied `O_RDONLY`, for one.

clearerr

`(tied %h)->clearerr`: reset the failure flag `error` reports.

5.2.70 NEXT

Provide a pseudo-class NEXT (et al) that allows method redispatch

SYNOPSIS

```
use NEXT;

package P;
sub P::method   { print "$_[0]: P method\n";   $_[0]->NEXT::method()
 ↪}
sub P::DESTROY  { print "$_[0]: P dtor\n";     $_[0]->
 ↪NEXT::DESTROY() }

package Q;
use base qw( P );
sub Q::AUTOLOAD { print "$_[0]: Q AUTOLOAD\n"; $_[0]->
 ↪NEXT::AUTOLOAD() }
sub Q::DESTROY  { print "$_[0]: Q dtor\n";     $_[0]->
 ↪NEXT::DESTROY() }
```

(continues on next page)

(continued from previous page)

```
package R;
sub R::method   { print "$_[0]: R method\n";   $_[0]->NEXT::method()␣
↪}
sub R::AUTOLOAD { print "$_[0]: R AUTOLOAD\n"; $_[0]->
↪NEXT::AUTOLOAD() }
sub R::DESTROY  { print "$_[0]: R dtor\n";     $_[0]->
↪NEXT::DESTROY() }

package S;
use base qw( Q R );
sub S::method   { print "$_[0]: S method\n";   $_[0]->NEXT::method()␣
↪}
sub S::AUTOLOAD { print "$_[0]: S AUTOLOAD\n"; $_[0]->
↪NEXT::AUTOLOAD() }
sub S::DESTROY  { print "$_[0]: S dtor\n";     $_[0]->
↪NEXT::DESTROY() }

package main;

my $obj = bless {}, "S";

$obj->method();             # Calls S::method, P::method, R::method
$obj->missing_method(); # Calls S::AUTOLOAD, Q::AUTOLOAD,␣
↪R::AUTOLOAD

# Clean-up calls S::DESTROY, Q::DESTROY, P::DESTROY, R::DESTROY
```

DESCRIPTION

The NEXT module adds a pseudoclass named NEXT to any program that uses it. If a method m calls `$self->NEXT::m()`, the call to m is redispatched as if the calling method had not originally been found.

Note: before using this module, you should look at next::method in the core *mro* module. mro has been a core module since Perl 5.9.5.

In other words, a call to `$self->NEXT::m()` resumes the depth-first, left-to-right search of `$self`'s class hierarchy that resulted in the original call to m.

Note that this is not the same thing as `$self->SUPER::m()`, which begins a new dispatch that is restricted to searching the ancestors of the current class. `$self->NEXT::m()` can backtrack past the current class – to look for a suitable method in other ancestors of `$self` – whereas `$self->SUPER::m()` cannot.

A typical use would be in the destructors of a class hierarchy, as illustrated in the SYNOPSIS above. Each class in the hierarchy has a DESTROY method that performs some class-specific action and then redispatches the call up the hierarchy. As a result, when an object of class S is destroyed, the destructors of *all* its parent classes are called (in depth-first, left-to-right order).

Another typical use of redispatch would be in AUTOLOAD'ed methods. If such a method determined that it was not able to handle a particular call, it might choose to redispatch

that call, in the hope that some other AUTOLOAD (above it, or to its left) might do better.

By default, if a redispatch attempt fails to find another method elsewhere in the objects class hierarchy, it quietly gives up and does nothing (but see "Enforcing redispatch"). This gracious acquiescence is also unlike the (generally annoying) behaviour of SUPER, which throws an exception if it cannot redispatch.

Note that it is a fatal error for any method (including AUTOLOAD) to attempt to redispatch any method that does not have the same name. For example:

```
sub S::oops { print "oops!\n"; $_[0]->NEXT::other_method() }
```

Enforcing redispatch

It is possible to make NEXT redispatch more demandingly (i.e. like SUPER does), so that the redispatch throws an exception if it cannot find a "next" method to call.

To do this, simple invoke the redispatch as:

```
$self->NEXT::ACTUAL::method();
```

rather than:

```
$self->NEXT::method();
```

The ACTUAL tells NEXT that there must actually be a next method to call, or it should throw an exception.

NEXT::ACTUAL is most commonly used in AUTOLOAD methods, as a means to decline an AUTOLOAD request, but preserve the normal exception-on-failure semantics:

```
sub AUTOLOAD {
        if ($AUTOLOAD =~ /foo|bar/) {
                # handle here
        }
        else {  # try elsewhere
                shift()->NEXT::ACTUAL::AUTOLOAD(@_);
        }
}
```

By using NEXT::ACTUAL, if there is no other AUTOLOAD to handle the method call, an exception will be thrown (as usually happens in the absence of a suitable AUTOLOAD).

Avoiding repetitions

If NEXT redispatching is used in the methods of a "diamond" class hierarchy:

```
#     A   B
#    / \ /
#   C   D
#    \ /
#     E
```

(continues on next page)

(continued from previous page)

```
use NEXT;

package A;
sub foo { print "called A::foo\n"; shift->NEXT::foo() }

package B;
sub foo { print "called B::foo\n"; shift->NEXT::foo() }

package C; @ISA = qw( A );
sub foo { print "called C::foo\n"; shift->NEXT::foo() }

package D; @ISA = qw(A B);
sub foo { print "called D::foo\n"; shift->NEXT::foo() }

package E; @ISA = qw(C D);
sub foo { print "called E::foo\n"; shift->NEXT::foo() }

E->foo();
```

then derived classes may (re-)inherit base-class methods through two or more distinct paths (e.g. in the way `E` inherits `A::foo` twice – through `C` and `D`). In such cases, a sequence of `NEXT` redispatches will invoke the multiply inherited method as many times as it is inherited. For example, the above code prints:

```
called E::foo
called C::foo
called A::foo
called D::foo
called A::foo
called B::foo
```

(i.e. `A::foo` is called twice).

In some cases this *may* be the desired effect within a diamond hierarchy, but in others (e.g. for destructors) it may be more appropriate to call each method only once during a sequence of redispatches.

To cover such cases, you can redispatch methods via:

```
$self->NEXT::DISTINCT::method();
```

rather than:

```
$self->NEXT::method();
```

This causes the redispatcher to only visit each distinct `method` method once. That is, to skip any classes in the hierarchy that it has already visited during redispatch. So, for example, if the previous example were rewritten:

```
package A;
sub foo { print "called A::foo\n"; shift->NEXT::DISTINCT::foo() }
```

(continues on next page)

(continued from previous page)

```
package B;
sub foo { print "called B::foo\n"; shift->NEXT::DISTINCT::foo() }

package C; @ISA = qw( A );
sub foo { print "called C::foo\n"; shift->NEXT::DISTINCT::foo() }

package D; @ISA = qw(A B);
sub foo { print "called D::foo\n"; shift->NEXT::DISTINCT::foo() }

package E; @ISA = qw(C D);
sub foo { print "called E::foo\n"; shift->NEXT::DISTINCT::foo() }

E->foo();
```

then it would print:

```
called E::foo
called C::foo
called A::foo
called D::foo
called B::foo
```

and omit the second call to `A::foo` (since it would not be distinct from the first call to `A::foo`).

Note that you can also use:

```
$self->NEXT::DISTINCT::ACTUAL::method();
```

or:

```
$self->NEXT::ACTUAL::DISTINCT::method();
```

to get both unique invocation *and* exception-on-failure.

Note that, for historical compatibility, you can also use `NEXT::UNSEEN` instead of `NEXT::DISTINCT`.

Invoking all versions of a method with a single call

Yet another pseudo-class that `NEXT` provides is `EVERY`. Its behaviour is considerably simpler than that of the `NEXT` family. A call to:

```
$obj->EVERY::foo();
```

calls *every* method named `foo` that the object in `$obj` has inherited. That is:

```
use NEXT;

package A; @ISA = qw(B D X);
```

(continues on next page)

(continued from previous page)

```
sub foo { print "A::foo " }

package B; @ISA = qw(D X);
sub foo { print "B::foo " }

package X; @ISA = qw(D);
sub foo { print "X::foo " }

package D;
sub foo { print "D::foo " }

package main;

my $obj = bless {}, 'A';
$obj->EVERY::foo();        # prints" A::foo B::foo X::foo D::foo
```

Prefixing a method call with `EVERY::` causes every method in the object's hierarchy with that name to be invoked. As the above example illustrates, they are not called in Perl's usual "left-most-depth-first" order. Instead, they are called "breadth-first-dependency-wise".

That means that the inheritance tree of the object is traversed breadth-first and the resulting order of classes is used as the sequence in which methods are called. However, that sequence is modified by imposing a rule that the appropriate method of a derived class must be called before the same method of any ancestral class. That's why, in the above example, `X::foo` is called before `D::foo`, even though `D` comes before `X` in `@B::ISA`.

In general, there's no need to worry about the order of calls. They will be left-to-right, breadth-first, most-derived-first. This works perfectly for most inherited methods (including destructors), but is inappropriate for some kinds of methods (such as constructors, cloners, debuggers, and initializers) where it's more appropriate that the least-derived methods be called first (as more-derived methods may rely on the behaviour of their "ancestors"). In that case, instead of using the `EVERY` pseudo-class:

```
$obj->EVERY::foo();        # prints" A::foo B::foo X::foo D::foo
```

you can use the `EVERY::LAST` pseudo-class:

```
$obj->EVERY::LAST::foo();  # prints" D::foo X::foo B::foo A::foo
```

which reverses the order of method call.

Whichever version is used, the actual methods are called in the same context (list, scalar, or void) as the original call via `EVERY`, and return:

- A hash of array references in list context. Each entry of the hash has the fully qualified method name as its key and a reference to an array containing the method's list-context return values as its value.
- A reference to a hash of scalar values in scalar context. Each entry of the hash has the fully qualified method name as its key and the method's scalar-context return values as its value.

- Nothing in void context (obviously).

Using EVERY methods

The typical way to use an EVERY call is to wrap it in another base method, that all classes inherit. For example, to ensure that every destructor an object inherits is actually called (as opposed to just the left-most-depth-first-est one):

```
package Base;
sub DESTROY { $_[0]->EVERY::Destroy }

package Derived1;
use base 'Base';
sub Destroy {...}

package Derived2;
use base 'Base', 'Derived1';
sub Destroy {...}
```

et cetera. Every derived class than needs its own clean-up behaviour simply adds its own Destroy method (*not* a DESTROY method), which the call to EVERY::LAST::Destroy in the inherited destructor then correctly picks up.

Likewise, to create a class hierarchy in which every initializer inherited by a new object is invoked:

```
package Base;
sub new {
        my ($class, %args) = @_;
        my $obj = bless {}, $class;
        $obj->EVERY::LAST::Init(\%args);
}

package Derived1;
use base 'Base';
sub Init {
        my ($argsref) = @_;
        ...
}

package Derived2;
use base 'Base', 'Derived1';
sub Init {
        my ($argsref) = @_;
        ...
}
```

et cetera. Every derived class than needs some additional initialization behaviour simply adds its own Init method (*not* a new method), which the call to EVERY::LAST::Init in the inherited constructor then correctly picks up.

SEE ALSO

mro (in particular next::method), which has been a core module since Perl 5.9.5.

AUTHOR

Damian Conway (damian@conway.org)

BUGS AND IRRITATIONS

Because it's a module, not an integral part of the interpreter, NEXT has to guess where the surrounding call was found in the method look-up sequence. In the presence of diamond inheritance patterns it occasionally guesses wrong.

It's also too slow (despite caching).

Comment, suggestions, and patches welcome.

COPYRIGHT


```
Copyright (c) 2000-2001, Damian Conway. All Rights Reserved.
This module is free software. It may be used, redistributed
   and/or modified under the same terms as Perl itself.
```


5.2.71 Net

Namespace containing the following modules:

- Net::Cmd - Network Command class (as used by FTP, SMTP etc)
- Net::Config - Local configuration data for libnet
- Net::Domain - Attempt to evaluate the current host's internet name and domain
- Net::FTP - FTP Client class
- Net::NNTP - NNTP Client class
- Net::Netrc - OO interface to users netrc file
- Net::POP3 - Post Office Protocol 3 Client class (RFC1939)
- Net::Ping - check a remote host for reachability
- Net::SMTP - Simple Mail Transfer Protocol Client
- Net::SSLeay - Perl bindings for OpenSSL's libssl and libcrypto
- Net::Time - time and daytime network client interface
- Net::hostent - by-name interface to Perl's built-in gethost*() functions
- Net::netent - by-name interface to Perl's built-in getnet*() functions
- Net::protoent - by-name interface to Perl's built-in getproto*() functions
- Net::servent - by-name interface to Perl's built-in getserv*() functions

Net::Cmd

Network Command class (as used by FTP, SMTP etc)

SYNOPSIS

```
use Net::Cmd;

@ISA = qw(Net::Cmd);
```

DESCRIPTION

`Net::Cmd` is a collection of methods that can be inherited by a sub-class of `IO::Socket::INET`. These methods implement the functionality required for a command based protocol, for example FTP and SMTP.

If your sub-class does not also derive from `IO::Socket::INET` or similar (e.g. `IO::Socket::IP`, `IO::Socket::INET6` or `IO::Socket::SSL`) then you must provide the following methods by other means yourself: `close()` and `timeout()`.

Public Methods

These methods provide a user interface to the `Net::Cmd` object.

- `debug($level)`

  Set the level of debug information for this object. If `$level` is not given then the current state is returned. Otherwise the state is changed to `$level` and the previous state returned.

  Different packages may implement different levels of debug but a non-zero value results in copies of all commands and responses also being sent to STDERR.

  If `$level` is `undef` then the debug level will be set to the default debug level for the class.

  This method can also be called as a *static* method to set/get the default debug level for a given class.

- `message()`

  Returns the text message returned from the last command. In a scalar context it returns a single string, in a list context it will return each line as a separate element. (See "PSEUDO RESPONSES" below.)

- `code()`

  Returns the 3-digit code from the last command. If a command is pending then the value 0 is returned. (See "PSEUDO RESPONSES" below.)

- `ok()`

  Returns non-zero if the last code value was greater than zero and less than 400. This holds true for most command servers. Servers where this does not hold may override this method.

- `status()`

  Returns the most significant digit of the current status code. If a command is pending then `CMD_PENDING` is returned.

- `datasend($data)`

  Send data to the remote server, converting LF to CRLF. Any line starting with a '.' will be prefixed with another '.'. `$data` may be an array or a reference to an array. The `$data` passed in must be encoded by the caller to octets of whatever encoding is required, e.g. by using the Encode module's `encode()` function.

- `dataend()`

  End the sending of data to the remote server. This is done by ensuring that the data already sent ends with CRLF then sending '.CRLF' to end the transmission. Once this data has been sent `dataend` calls `response` and returns true if `response` returns CMD_OK.

Protected Methods

These methods are not intended to be called by the user, but used or over-ridden by a sub-class of `Net::Cmd`

- `debug_print($dir, $text)`

  Print debugging information. `$dir` denotes the direction *true* being data being sent to the server. Calls `debug_text` before printing to STDERR.

- `debug_text($dir, $text)`

  This method is called to print debugging information. `$text` is the text being sent. The method should return the text to be printed.

  This is primarily meant for the use of modules such as FTP where passwords are sent, but we do not want to display them in the debugging information.

- `command($cmd[, $args, ... ])`

  Send a command to the command server. All arguments are first joined with a space character and CRLF is appended, this string is then sent to the command server.

  Returns undef upon failure.

- `unsupported()`

  Sets the status code to 580 and the response text to 'Unsupported command'. Returns zero.

- `response()`

  Obtain a response from the server. Upon success the most significant digit of the status code is returned. Upon failure, timeout etc., *CMD_ERROR* is returned.

- `parse_response($text)`

  This method is called by `response` as a method with one argument. It should return an array of 2 values, the 3-digit status code and a flag which is true when this is part of a multi-line response and this line is not the last.

- `getline()`

  Retrieve one line, delimited by CRLF, from the remote server. Returns *undef* upon failure.

  NOTE: If you do use this method for any reason, please remember to add some `debug_print` calls into your method.

- `ungetline($text)`

  Unget a line of text from the server.

- `rawdatasend($data)`

  Send data to the remote server without performing any conversions. `$data` is a scalar. As with `datasend()`, the `$data` passed in must be encoded by the caller to octets of whatever encoding is required, e.g. by using the Encode module's `encode()` function.

- `read_until_dot()`

  Read data from the remote server until a line consisting of a single '.'. Any lines starting with '..' will have one of the '.'s removed.

  Returns a reference to a list containing the lines, or *undef* upon failure.

- `tied_fh()`

  Returns a filehandle tied to the Net::Cmd object. After issuing a command, you may read from this filehandle using read() or <>. The filehandle will return EOF when the final dot is encountered. Similarly, you may write to the filehandle in order to send data to the server after issuing a command that expects data to be written.

  See the Net::POP3 and Net::SMTP modules for examples of this.

Pseudo Responses

Normally the values returned by `message()` and `code()` are obtained from the remote server, but in a few circumstances, as detailed below, `Net::Cmd` will return values that it sets. You can alter this behavior by overriding DEF_REPLY_CODE() to specify a different default reply code, or overriding one of the specific error handling methods below.

- Initial value

  Before any command has executed or if an unexpected error occurs `code()` will return "421" (temporary connection failure) and `message()` will return undef.

- Connection closed

  If the underlying `IO::Handle` is closed, or if there are any read or write failures, the file handle will be forced closed, and `code()` will return "421" (temporary connection failure) and `message()` will return "[$pkg] Connection closed" (where $pkg is the name of the class that subclassed `Net::Cmd`). The _set_status_closed() method can be overridden to set a different message (by calling set_status()) or otherwise trap this error.

- Timeout

  If there is a read or write timeout `code()` will return "421" (temporary connection failure) and `message()` will return "[$pkg] Timeout" (where $pkg is the name

of the class that subclassed `Net::Cmd`). The _set_status_timeout() method can be overridden to set a different message (by calling set_status()) or otherwise trap this error.

EXPORTS

The following symbols are, or can be, exported by this module:

- Default Exports

  `CMD_INFO`, `CMD_OK`, `CMD_MORE`, `CMD_REJECT`, `CMD_ERROR`, `CMD_PENDING`.

  (These correspond to possible results of `response()` and `status()`.)

- Optional Exports

  None.

- Export Tags

  None.

KNOWN BUGS

See https://rt.cpan.org/Dist/Display.html?Status=Active&Queue=libnet.

AUTHOR

Graham Barr <gbarr@pobox.com>.

Steve Hay <shay@cpan.org> is now maintaining libnet as of version 1.22_02.

COPYRIGHT

Copyright (C) 1995-2006 Graham Barr. All rights reserved.

Copyright (C) 2013-2016, 2020, 2022 Steve Hay. All rights reserved.

LICENCE

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself, i.e. under the terms of either the GNU General Public License or the Artistic License, as specified in the `LICENCE` file.

VERSION

Version 3.15

DATE

20 March 2023

HISTORY

See the Changes file.

Net::Config

Local configuration data for libnet

SYNOPSIS

```
use Net::Config qw(%NetConfig);
```

DESCRIPTION

`Net::Config` holds configuration data for the modules in the libnet distribution. During installation you will be asked for these values.

The configuration data is held globally in a file in the perl installation tree, but a user may override any of these values by providing their own. This can be done by having a `.libnetrc` file in their home directory. This file should return a reference to a HASH containing the keys described below. For example

```
# .libnetrc
{
    nntp_hosts => [ "my_preferred_host" ],
    ph_hosts   => [ "my_ph_server" ],
}
__END__
```

Class Methods

`Net::Config` defines the following methods. They are methods as they are invoked as class methods. This is because `Net::Config` inherits from `Net::LocalCfg` so you can override these methods if you want.

- `requires_firewall($host)`

  Attempts to determine if a given host is outside your firewall. Possible return values are.

  ```
  -1  Cannot lookup hostname
   0  Host is inside firewall (or there is no ftp_firewall entry)
   1  Host is outside the firewall
  ```

  This is done by using hostname lookup and the `local_netmask` entry in the configuration data.

NetConfig Values

- nntp_hosts
- snpp_hosts
- pop3_hosts
- smtp_hosts
- ph_hosts
- daytime_hosts
- time_hosts

  Each is a reference to an array of hostnames (in order of preference), which should be used for the given protocol

- inet_domain

  Your internet domain name

- ftp_firewall

  If you have an FTP proxy firewall (**NOT** an HTTP or SOCKS firewall) then this value should be set to the firewall hostname. If your firewall does not listen to port 21, then this value should be set to `"hostname:port"` (eg `"hostname:99"`)

- ftp_firewall_type

  There are many different ftp firewall products available. But unfortunately there is no standard for how to traverse a firewall. The list below shows the sequence of commands that Net::FTP will use

```
user        Username for remote host
pass        Password for remote host
fwuser      Username for firewall
fwpass      Password for firewall
remote.host The hostname of the remote ftp server
```

  – 0

    There is no firewall

  – 1

```
USER user@remote.host
PASS pass
```

  – 2

```
USER fwuser
PASS fwpass
USER user@remote.host
PASS pass
```

  – 3

```
USER fwuser
PASS fwpass
SITE remote.site
USER user
PASS pass
```

– 4

```
USER fwuser
PASS fwpass
OPEN remote.site
USER user
PASS pass
```

– 5

```
USER user@fwuser@remote.site
PASS pass@fwpass
```

– 6

```
USER fwuser@remote.site
PASS fwpass
USER user
PASS pass
```

– 7

```
USER user@remote.host
PASS pass
AUTH fwuser
RESP fwpass
```

- ftp_ext_passive
- ftp_int_passive

  FTP servers can work in passive or active mode. Active mode is when you want to transfer data you have to tell the server the address and port to connect to. Passive mode is when the server provide the address and port and you establish the connection.

  With some firewalls active mode does not work as the server cannot connect to your machine (because you are behind a firewall) and the firewall does not re-write the command. In this case you should set `ftp_ext_passive` to a *true* value.

  Some servers are configured to only work in passive mode. If you have one of these you can force `Net::FTP` to always transfer in passive mode; when not going via a firewall, by setting `ftp_int_passive` to a *true* value.

- local_netmask

  A reference to a list of netmask strings in the form `"134.99.4.0/24"`. These are used by the `requires_firewall` function to determine if a given host is inside or outside your firewall.

The following entries are used during installation & testing on the libnet package

- test_hosts

  If true then `make test` may attempt to connect to hosts given in the configuration.

- test_exists

  If true then `Configure` will check each hostname given that it exists

EXPORTS

The following symbols are, or can be, exported by this module:

- Default Exports

  `%NetConfig`.

- Optional Exports

  None.

- Export Tags

  None.

KNOWN BUGS

None.

AUTHOR

Graham Barr <gbarr@pobox.com>.

Steve Hay <shay@cpan.org> is now maintaining libnet as of version 1.22_02.

COPYRIGHT

Copyright (C) 2000 Graham Barr. All rights reserved.

Copyright (C) 2013-2014, 2016, 2020 Steve Hay. All rights reserved.

LICENCE

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself, i.e. under the terms of either the GNU General Public License or the Artistic License, as specified in the `LICENCE` file.

VERSION

Version 3.15

DATE

20 March 2023

HISTORY

See the Changes file.

Net::Domain

Attempt to evaluate the current host's internet name and domain

SYNOPSIS

```
use Net::Domain qw(hostname hostfqdn hostdomain domainname);
```

DESCRIPTION

Using various methods **attempt** to find the Fully Qualified Domain Name (FQDN) of the current host. From this determine the host-name and the host-domain.

Each of the functions will return *undef* if the FQDN cannot be determined.

Functions

- `hostfqdn()`

  Identify and return the FQDN of the current host.

- `domainname()`

  An alias for hostfqdn().

- `hostname()`

  Returns the smallest part of the FQDN which can be used to identify the host.

- `hostdomain()`

  Returns the remainder of the FQDN after the *hostname* has been removed.

EXPORTS

The following symbols are, or can be, exported by this module:

- Default Exports

  None.

- Optional Exports

  `hostname`, `hostdomain`, `hostfqdn`, `domainname`.

- Export Tags

  None.

KNOWN BUGS

See https://rt.cpan.org/Dist/Display.html?Status=Active&Queue=libnet.

AUTHOR

Graham Barr <gbarr@pobox.com>.

Adapted from Sys::Hostname by David Sundstrom <sunds@asictest.sc.ti.com>.

Steve Hay <shay@cpan.org> is now maintaining libnet as of version 1.22_02.

COPYRIGHT

Copyright (C) 1995-1998 Graham Barr. All rights reserved.

Copyright (C) 2013-2014, 2020 Steve Hay. All rights reserved.

LICENCE

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself, i.e. under the terms of either the GNU General Public License or the Artistic License, as specified in the `LICENCE` file.

VERSION

Version 3.15

DATE

20 March 2023

HISTORY

See the `Changes` file.

Net::FTP

FTP Client class

SYNOPSIS

```
use Net::FTP;

$ftp = Net::FTP->new("some.host.name", Debug => 0)
  or die "Cannot connect to some.host.name: $@";

$ftp->login("anonymous",'-anonymous@')
  or die "Cannot login ", $ftp->message;
```

(continues on next page)

(continued from previous page)

```
$ftp->cwd("/pub")
  or die "Cannot change working directory ", $ftp->message;

$ftp->get("that.file")
  or die "get failed ", $ftp->message;

$ftp->quit;
```

DESCRIPTION

`Net::FTP` is a class implementing a simple FTP client in Perl as described in RFC959. It provides wrappers for the commonly used subset of the RFC959 commands. If *IO::Socket::IP* or IO::Socket::INET6 is installed it also provides support for IPv6 as defined in RFC2428. And with *IO::Socket::SSL* installed it provides support for implicit FTPS and explicit FTPS as defined in RFC4217.

The Net::FTP class is a subclass of Net::Cmd and (depending on avaibility) of IO::Socket::IP, IO::Socket::INET6 or IO::Socket::INET.

Overview

FTP stands for File Transfer Protocol. It is a way of transferring files between networked machines. The protocol defines a client (whose commands are provided by this module) and a server (not implemented in this module). Communication is always initiated by the client, and the server responds with a message and a status code (and sometimes with data).

The FTP protocol allows files to be sent to or fetched from the server. Each transfer involves a **local file** (on the client) and a **remote file** (on the server). In this module, the same file name will be used for both local and remote if only one is specified. This means that transferring remote file `/path/to/file` will try to put that file in `/path/to/file` locally, unless you specify a local file name.

The protocol also defines several standard **translations** which the file can undergo during transfer. These are ASCII, EBCDIC, binary, and byte. ASCII is the default type, and indicates that the sender of files will translate the ends of lines to a standard representation which the receiver will then translate back into their local representation. EBCDIC indicates the file being transferred is in EBCDIC format. Binary (also known as image) format sends the data as a contiguous bit stream. Byte format transfers the data as bytes, the values of which remain the same regardless of differences in byte size between the two machines (in theory - in practice you should only use this if you really know what you're doing). This class does not support the EBCDIC or byte formats, and will default to binary instead if they are attempted.

Class Methods

- `new([$host][, %options])`

  This is the constructor for a new Net::FTP object. `$host` is the name of the remote host to which an FTP connection is required.

`$host` is optional. If `$host` is not given then it may instead be passed as the `Host` option described below.

`%options` are passed in a hash like fashion, using key and value pairs. Possible options are:

Host - FTP host to connect to. It may be a single scalar, as defined for the `PeerAddr` option in *IO::Socket::INET*, or a reference to an array with hosts to try in turn. The "host" method will return the value which was used to connect to the host.

Firewall - The name of a machine which acts as an FTP firewall. This can be overridden by an environment variable `FTP_FIREWALL`. If specified, and the given host cannot be directly connected to, then the connection is made to the firewall machine and the string `@hostname` is appended to the login identifier. This kind of setup is also referred to as an ftp proxy.

FirewallType - The type of firewall running on the machine indicated by **Firewall**. This can be overridden by an environment variable `FTP_FIREWALL_TYPE`. For a list of permissible types, see the description of ftp_firewall_type in *Net::Config*.

BlockSize - This is the block size that Net::FTP will use when doing transfers. (defaults to 10240)

Port - The port number to connect to on the remote machine for the FTP connection

SSL - If the connection should be done from start with SSL, contrary to later upgrade with `starttls`.

SSL_* - SSL arguments which will be applied when upgrading the control or data connection to SSL. You can use SSL arguments as documented in *IO::Socket::SSL*, but it will usually use the right arguments already.

Timeout - Set a timeout value in seconds (defaults to 120)

Debug - debug level (see the debug method in *Net::Cmd*)

Passive - If set to a non-zero value then all data transfers will be done using passive mode. If set to zero then data transfers will be done using active mode. If the machine is connected to the Internet directly, both passive and active mode should work equally well. Behind most firewall and NAT configurations passive mode has a better chance of working. However, in some rare firewall configurations, active mode actually works when passive mode doesn't. Some really old FTP servers might not implement passive transfers. If not specified, then the transfer mode is set by the environment variable `FTP_PASSIVE` or if that one is not set by the settings done by the `libnetcfg` utility. If none of these apply then passive mode is used.

Hash - If given a reference to a file handle (e.g., `\*STDERR`), print hash marks (#) on that filehandle every 1024 bytes. This simply invokes the `hash()` method for you, so that hash marks are displayed for all transfers. You can, of course, call `hash()` explicitly whenever you'd like.

LocalAddr - Local address to use for all socket connections. This argument will be passed to the super class, i.e. *IO::Socket::INET* or *IO::Socket::IP*.

Domain - Domain to use, i.e. AF_INET or AF_INET6. This argument will be passed to the IO::Socket super class. This can be used to enforce IPv4 even with *IO::Socket::IP* which would default to IPv6. **Family** is accepted as alternative name for **Domain**.

If the constructor fails undef will be returned and an error message will be in $@

Object Methods

Unless otherwise stated all methods return either a *true* or *false* value, with *true* meaning that the operation was a success. When a method states that it returns a value, failure will be returned as *undef* or an empty list.

`Net::FTP` inherits from `Net::Cmd` so methods defined in `Net::Cmd` may be used to send commands to the remote FTP server in addition to the methods documented here.

- `login([$login[, $password[, $account]]])`

  Log into the remote FTP server with the given login information. If no arguments are given then the `Net::FTP` uses the `Net::Netrc` package to lookup the login information for the connected host. If no information is found then a login of *anonymous* is used. If no password is given and the login is *anonymous* then *anonymous@* will be used for password.

  If the connection is via a firewall then the `authorize` method will be called with no arguments.

- `starttls()`

  Upgrade existing plain connection to SSL. The SSL arguments have to be given in `new` already because they are needed for data connections too.

- `stoptls()`

  Downgrade existing SSL connection back to plain. This is needed to work with some FTP helpers at firewalls, which need to see the PORT and PASV commands and responses to dynamically open the necessary ports. In this case `starttls` is usually only done to protect the authorization.

- `prot($level)`

  Set what type of data channel protection the client and server will be using. Only `$level`s "C" (clear) and "P" (private) are supported.

- `host()`

  Returns the value used by the constructor, and passed to the IO::Socket super class to connect to the host.

- `account($acct)`

  Set a string identifying the user's account.

- `authorize([$auth[, $resp]])`

  This is a protocol used by some firewall ftp proxies. It is used to authorise the user to send data out. If both arguments are not specified then `authorize` uses `Net::Netrc` to do a lookup.

- `site($args)`

  Send a SITE command to the remote server and wait for a response.

  Returns most significant digit of the response code.

- `ascii()`

  Transfer file in ASCII. CRLF translation will be done if required

- `binary()`

  Transfer file in binary mode. No transformation will be done.

  Hint: If both server and client machines use the same line ending for text files, then it will be faster to transfer all files in binary mode.

- `type([$type])`

  Set or get if files will be transferred in ASCII or binary mode.

- `rename($oldname, $newname)`

  Rename a file on the remote FTP server from `$oldname` to `$newname`. This is done by sending the RNFR and RNTO commands.

- `delete($filename)`

  Send a request to the server to delete `$filename`.

- `cwd([$dir])`

  Attempt to change directory to the directory given in `$dir`. If `$dir` is `"..`", the FTP `CDUP` command is used to attempt to move up one directory. If no directory is given then an attempt is made to change the directory to the root directory.

- `cdup()`

  Change directory to the parent of the current directory.

- `passive([$passive])`

  Set or get if data connections will be initiated in passive mode.

- `pwd()`

  Returns the full pathname of the current directory.

- `restart($where)`

  Set the byte offset at which to begin the next data transfer. Net::FTP simply records this value and uses it when during the next data transfer. For this reason this method will not return an error, but setting it may cause a subsequent data transfer to fail.

- `rmdir($dir[, $recurse])`

  Remove the directory with the name `$dir`. If `$recurse` is *true* then `rmdir` will attempt to delete everything inside the directory.

- `mkdir($dir[, $recurse])`

  Create a new directory with the name `$dir`. If `$recurse` is *true* then `mkdir` will attempt to create all the directories in the given path.

  Returns the full pathname to the new directory.

- `alloc($size[, $record_size])`

  The alloc command allows you to give the ftp server a hint about the size of the file about to be transferred using the ALLO ftp command. Some storage systems use

this to make intelligent decisions about how to store the file. The `$size` argument represents the size of the file in bytes. The `$record_size` argument indicates a maximum record or page size for files sent with a record or page structure.

The size of the file will be determined, and sent to the server automatically for normal files so that this method need only be called if you are transferring data from a socket, named pipe, or other stream not associated with a normal file.

- `ls([$dir])`

  Get a directory listing of `$dir`, or the current directory.

  In an array context, returns a list of lines returned from the server. In a scalar context, returns a reference to a list.

- `dir([$dir])`

  Get a directory listing of `$dir`, or the current directory in long format.

  In an array context, returns a list of lines returned from the server. In a scalar context, returns a reference to a list.

- `get($remote_file[, $local_file[, $where]])`

  Get `$remote_file` from the server and store locally. `$local_file` may be a filename or a filehandle. If not specified, the file will be stored in the current directory with the same leafname as the remote file.

  If `$where` is given then the first `$where` bytes of the file will not be transferred, and the remaining bytes will be appended to the local file if it already exists.

  Returns `$local_file`, or the generated local file name if `$local_file` is not given. If an error was encountered undef is returned.

- `put($local_file[, $remote_file])`

  Put a file on the remote server. `$local_file` may be a name or a filehandle. If `$local_file` is a filehandle then `$remote_file` must be specified. If `$remote_file` is not specified then the file will be stored in the current directory with the same leafname as `$local_file`.

  Returns `$remote_file`, or the generated remote filename if `$remote_file` is not given.

  NOTE: If for some reason the transfer does not complete and an error is returned then the contents that had been transferred will not be remove automatically.

- `put_unique($local_file[, $remote_file])`

  Same as put but uses the `STOU` command.

  Returns the name of the file on the server.

- `append($local_file[, $remote_file])`

  Same as put but appends to the file on the remote server.

  Returns `$remote_file`, or the generated remote filename if `$remote_file` is not given.

- `unique_name()`

  Returns the name of the last file stored on the server using the `STOU` command.

- `mdtm($file)`

  Returns the *modification time* of the given file

- `size($file)`

  Returns the size in bytes for the given file as stored on the remote server.

  NOTE: The size reported is the size of the stored file on the remote server. If the file is subsequently transferred from the server in ASCII mode and the remote server and local machine have different ideas about "End Of Line" then the size of file on the local machine after transfer may be different.

- `supported($cmd)`

  Returns TRUE if the remote server supports the given command.

- `hash([$filehandle_glob_ref[, $bytes_per_hash_mark]])`

  Called without parameters, or with the first argument false, hash marks are suppressed. If the first argument is true but not a reference to a file handle glob, then *STDERR is used. The second argument is the number of bytes per hash mark printed, and defaults to 1024. In all cases the return value is a reference to an array of two: the filehandle glob reference and the bytes per hash mark.

- `feature($name)`

  Determine if the server supports the specified feature. The return value is a list of lines the server responded with to describe the options that it supports for the given feature. If the feature is unsupported then the empty list is returned.

```
if ($ftp->feature( 'MDTM' )) {
  # Do something
}

if (grep { /\bTLS\b/ } $ftp->feature('AUTH')) {
  # Server supports TLS
}
```

The following methods can return different results depending on how they are called. If the user explicitly calls either of the `pasv` or `port` methods then these methods will return a *true* or *false* value. If the user does not call either of these methods then the result will be a reference to a `Net::FTP::dataconn` based object.

- `nlst([$dir])`

  Send an `NLST` command to the server, with an optional parameter.

- `list([$dir])`

  Same as `nlst` but using the `LIST` command

- `retr($file)`

  Begin the retrieval of a file called `$file` from the remote server.

- `stor($file)`

  Tell the server that you wish to store a file. `$file` is the name of the new file that should be created.

- `stou($file)`

  Same as `stor` but using the `STOU` command. The name of the unique file which was created on the server will be available via the `unique_name` method after the data connection has been closed.

- `appe($file)`

  Tell the server that we want to append some data to the end of a file called `$file`. If this file does not exist then create it.

If for some reason you want to have complete control over the data connection, this includes generating it and calling the `response` method when required, then the user can use these methods to do so.

However calling these methods only affects the use of the methods above that can return a data connection. They have no effect on methods `get`, `put`, `put_unique` and those that do not require data connections.

- `port([$port])`
- `eprt([$port])`

  Send a `PORT` (IPv4) or `EPRT` (IPv6) command to the server. If `$port` is specified then it is sent to the server. If not, then a listen socket is created and the correct information sent to the server.

- `pasv()`
- `epsv()`

  Tell the server to go into passive mode (`pasv` for IPv4, `epsv` for IPv6). Returns the text that represents the port on which the server is listening, this text is in a suitable form to send to another ftp server using the `port` or `eprt` method.

The following methods can be used to transfer files between two remote servers, providing that these two servers can connect directly to each other.

- `pasv_xfer($src_file, $dest_server[, $dest_file ])`

  This method will do a file transfer between two remote ftp servers. If `$dest_file` is omitted then the leaf name of `$src_file` will be used.

- `pasv_xfer_unique($src_file, $dest_server[, $dest_file ])`

  Like `pasv_xfer` but the file is stored on the remote server using the STOU command.

- `pasv_wait($non_pasv_server)`

  This method can be used to wait for a transfer to complete between a passive server and a non-passive server. The method should be called on the passive server with the `Net::FTP` object for the non-passive server passed as an argument.

- `abort()`

  Abort the current data transfer.

- `quit()`

  Send the QUIT command to the remote FTP server and close the socket connection.

Methods for the Adventurous

- `quot($cmd[, $args])`

  Send a command, that Net::FTP does not directly support, to the remote server and wait for a response.

  Returns most significant digit of the response code.

  WARNING This call should only be used on commands that do not require data connections. Misuse of this method can hang the connection.

- `can_inet6()`

  Returns whether we can use IPv6.

- `can_ssl()`

  Returns whether we can use SSL.

The dataconn Class

Some of the methods defined in `Net::FTP` return an object which will be derived from the `Net::FTP::dataconn` class. See *Net::FTP::dataconn* for more details.

Unimplemented

The following RFC959 commands have not been implemented:

- `SMNT`

  Mount a different file system structure without changing login or accounting information.

- `HELP`

  Ask the server for "helpful information" (that's what the RFC says) on the commands it accepts.

- `MODE`

  Specifies transfer mode (stream, block or compressed) for file to be transferred.

- `SYST`

  Request remote server system identification.

- `STAT`

  Request remote server status.

- `STRU`

  Specifies file structure for file to be transferred.

- `REIN`

  Reinitialize the connection, flushing all I/O and account information.

EXPORTS

None.

KNOWN BUGS

See https://rt.cpan.org/Dist/Display.html?Status=Active&Queue=libnet.

Reporting Bugs

When reporting bugs/problems please include as much information as possible. It may be difficult for me to reproduce the problem as almost every setup is different.

A small script which yields the problem will probably be of help. It would also be useful if this script was run with the extra options `Debug => 1` passed to the constructor, and the output sent with the bug report. If you cannot include a small script then please include a Debug trace from a run of your program which does yield the problem.

SEE ALSO

Net::Netrc, *Net::Cmd*, *IO::Socket::SSL*;

ftp(1), ftpd(8);

https://www.ietf.org/rfc/rfc959.txt, https://www.ietf.org/rfc/rfc2428.txt, https://www.ietf.org/rfc/rfc4217.txt.

ACKNOWLEDGEMENTS

Henry Gabryjelski <henryg@WPI.EDU> - for the suggestion of creating directories recursively.

Nathan Torkington <gnat@frii.com> - for some input on the documentation.

Roderick Schertler <roderick@gate.net> - for various inputs

AUTHOR

Graham Barr <gbarr@pobox.com>.

Steve Hay <shay@cpan.org> is now maintaining libnet as of version 1.22_02.

COPYRIGHT

Copyright (C) 1995-2004 Graham Barr. All rights reserved.

Copyright (C) 2013-2017, 2020, 2022 Steve Hay. All rights reserved.

LICENCE

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself, i.e. under the terms of either the GNU General Public License or the Artistic License, as specified in the `LICENCE` file.

VERSION

Version 3.15

DATE

20 March 2023

HISTORY

See the Changes file.

Modules

- `Net::FTP::dataconn` - FTP Client data connection class

Net::FTP::dataconn

FTP Client data connection class

SYNOPSIS

```
# Perform IO operations on an FTP client data connection object:

$num_bytes_read = $obj->read($buffer, $size);
$num_bytes_read = $obj->read($buffer, $size, $timeout);

$num_bytes_written = $obj->write($buffer, $size);
$num_bytes_written = $obj->write($buffer, $size, $timeout);

$num_bytes_read_so_far = $obj->bytes_read();

$obj->abort();

$closed_successfully = $obj->close();
```

DESCRIPTION

Some of the methods defined in `Net::FTP` return an object which will be derived from this class. The dataconn class itself is derived from the `IO::Socket::INET` class, so any normal IO operations can be performed. However the following methods are defined in the dataconn class and IO should be performed using these.

- `read($buffer, $size[, $timeout])`

  Read `$size` bytes of data from the server and place it into `$buffer`, also performing any translation necessary. `$timeout` is optional, if not given, the timeout value from the command connection will be used.

  Returns the number of bytes read before any translation.

- `write($buffer, $size[, $timeout])`

  Write `$size` bytes of data from `$buffer` to the server, also performing any translation necessary. `$timeout` is optional, if not given, the timeout value from the command connection will be used.

  Returns the number of bytes written before any translation.

- `bytes_read()`

  Returns the number of bytes read so far.

- `abort()`

  Abort the current data transfer.

- `close()`

  Close the data connection and get a response from the FTP server. Returns *true* if the connection was closed successfully and the first digit of the response from the server was a '2'.

EXPORTS

None.

KNOWN BUGS

None.

AUTHOR

Graham Barr <gbarr@pobox.com>.

Steve Hay <shay@cpan.org> is now maintaining libnet as of version 1.22_02.

COPYRIGHT

Copyright (C) 1997-2010 Graham Barr. All rights reserved.

Copyright (C) 2013-2014, 2020 Steve Hay. All rights reserved.

LICENCE

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself, i.e. under the terms of either the GNU General Public License or the Artistic License, as specified in the `LICENCE` file.

VERSION

Version 3.15

DATE

20 March 2023

HISTORY

See the Changes file.

Net::NNTP

NNTP Client class

SYNOPSIS

```
use Net::NNTP;

$nntp = Net::NNTP->new("some.host.name");
$nntp->quit;

# start with SSL, e.g. nntps
$nntp = Net::NNTP->new("some.host.name", SSL => 1);

# start with plain and upgrade to SSL
$nntp = Net::NNTP->new("some.host.name");
$nntp->starttls;
```

DESCRIPTION

`Net::NNTP` is a class implementing a simple NNTP client in Perl as described in RFC977 and RFC4642. With *IO::Socket::SSL* installed it also provides support for implicit and explicit TLS encryption, i.e. NNTPS or NNTP+STARTTLS.

The Net::NNTP class is a subclass of Net::Cmd and (depending on avaibility) of IO::Socket::IP, IO::Socket::INET6 or IO::Socket::INET.

Class Methods

- `new([$host][, %options])`

  This is the constructor for a new Net::NNTP object. `$host` is the name of the remote host to which a NNTP connection is required. If not given then it may be passed as the `Host` option described below. If no host is passed then two environment variables are checked, first `NNTPSERVER` then `NEWSHOST`, then `Net::Config` is checked, and if a host is not found then `news` is used.

  `%options` are passed in a hash like fashion, using key and value pairs. Possible options are:

  Host - NNTP host to connect to. It may be a single scalar, as defined for the `PeerAddr` option in *IO::Socket::INET*, or a reference to an array with hosts to try in turn. The "host" method will return the value which was used to connect to the host.

Port - port to connect to. Default - 119 for plain NNTP and 563 for immediate SSL (nntps).

SSL - If the connection should be done from start with SSL, contrary to later upgrade with `starttls`. You can use SSL arguments as documented in *IO::Socket::SSL*, but it will usually use the right arguments already.

Timeout - Maximum time, in seconds, to wait for a response from the NNTP server, a value of zero will cause all IO operations to block. (default: 120)

Debug - Enable the printing of debugging information to STDERR

Reader - If the remote server is INN then initially the connection will be to innd, by default `Net::NNTP` will issue a `MODE READER` command so that the remote server becomes nnrpd. If the `Reader` option is given with a value of zero, then this command will not be sent and the connection will be left talking to innd.

LocalAddr and **LocalPort** - These parameters are passed directly to IO::Socket to allow binding the socket to a specific local address and port.

Domain - This parameter is passed directly to IO::Socket and makes it possible to enforce IPv4 connections even if *IO::Socket::IP* is used as super class. Alternatively **Family** can be used.

Object Methods

Unless otherwise stated all methods return either a *true* or *false* value, with *true* meaning that the operation was a success. When a method states that it returns a value, failure will be returned as *undef* or an empty list.

`Net::NNTP` inherits from `Net::Cmd` so methods defined in `Net::Cmd` may be used to send commands to the remote NNTP server in addition to the methods documented here.

- `host()`

  Returns the value used by the constructor, and passed to IO::Socket::INET, to connect to the host.

- `starttls()`

  Upgrade existing plain connection to SSL. Any arguments necessary for SSL must be given in `new` already.

- `article([{$msgid|$msgnum}[, $fh]])`

  Retrieve the header, a blank line, then the body (text) of the specified article.

  If `$fh` is specified then it is expected to be a valid filehandle and the result will be printed to it, on success a true value will be returned. If `$fh` is not specified then the return value, on success, will be a reference to an array containing the article requested, each entry in the array will contain one line of the article.

  If no arguments are passed then the current article in the currently selected newsgroup is fetched.

  `$msgnum` is a numeric id of an article in the current newsgroup, and will change the current article pointer. `$msgid` is the message id of an article as shown in that article's header. It is anticipated that the client will obtain the `$msgid` from a list

provided by the newnews command, from references contained within another article, or from the message-id provided in the response to some other commands.

If there is an error then undef will be returned.

- body([{$msgid|$msgnum}[, [$fh]])

  Like article but only fetches the body of the article.

- head([{$msgid|$msgnum}[, [$fh]])

  Like article but only fetches the headers for the article.

- articlefh([{$msgid|$msgnum}])
- bodyfh([{$msgid|$msgnum}])
- headfh([{$msgid|$msgnum}])

  These are similar to article(), body() and head(), but rather than returning the requested data directly, they return a tied filehandle from which to read the article.

- nntpstat([{$msgid|$msgnum}])

  The nntpstat command is similar to the article command except that no text is returned. When selecting by message number within a group, the nntpstat command serves to set the "current article pointer" without sending text.

  Using the nntpstat command to select by message-id is valid but of questionable value, since a selection by message-id does **not** alter the "current article pointer".

  Returns the message-id of the "current article".

- group([$group])

  Set and/or get the current group. If $group is not given then information is returned on the current group.

  In a scalar context it returns the group name.

  In an array context the return value is a list containing, the number of articles in the group, the number of the first article, the number of the last article and the group name.

- help()

  Request help text (a short summary of commands that are understood by this implementation) from the server. Returns the text or undef upon failure.

- ihave($msgid[, $message])

  The ihave command informs the server that the client has an article whose id is $msgid. If the server desires a copy of that article and $message has been given then it will be sent.

  Returns *true* if the server desires the article and $message was successfully sent, if specified.

  If $message is not specified then the message must be sent using the datasend and dataend methods from *Net::Cmd*

`$message` can be either an array of lines or a reference to an array and must be encoded by the caller to octets of whatever encoding is required, e.g. by using the Encode module's `encode()` function.

- `last()`

  Set the "current article pointer" to the previous article in the current newsgroup.

  Returns the message-id of the article.

- `date()`

  Returns the date on the remote server. This date will be in a UNIX time format (seconds since 1970)

- `postok()`

  `postok` will return *true* if the servers initial response indicated that it will allow posting.

- `authinfo($user, $pass)`

  Authenticates to the server (using the original AUTHINFO USER / AUTHINFO PASS form, defined in RFC2980) using the supplied username and password. Please note that the password is sent in clear text to the server. This command should not be used with valuable passwords unless the connection to the server is somehow protected.

- `authinfo_simple($user, $pass)`

  Authenticates to the server (using the proposed NNTP V2 AUTHINFO SIMPLE form, defined and deprecated in RFC2980) using the supplied username and password. As with "authinfo" the password is sent in clear text.

- `list()`

  Obtain information about all the active newsgroups. The results is a reference to a hash where the key is a group name and each value is a reference to an array. The elements in this array are:- the last article number in the group, the first article number in the group and any information flags about the group.

- `newgroups($since[, $distributions])`

  `$since` is a time value and `$distributions` is either a distribution pattern or a reference to a list of distribution patterns. The result is the same as `list`, but the groups return will be limited to those created after `$since` and, if specified, in one of the distribution areas in `$distributions`.

- `newnews($since[, $groups[, $distributions]])`

  `$since` is a time value. `$groups` is either a group pattern or a reference to a list of group patterns. `$distributions` is either a distribution pattern or a reference to a list of distribution patterns.

  Returns a reference to a list which contains the message-ids of all news posted after `$since`, that are in a groups which matched `$groups` and a distribution which matches `$distributions`.

- `next()`

  Set the "current article pointer" to the next article in the current newsgroup.

Returns the message-id of the article.

- `post([$message])`

  Post a new article to the news server. If `$message` is specified and posting is allowed then the message will be sent.

  If `$message` is not specified then the message must be sent using the `datasend` and `dataend` methods from *Net::Cmd*

  `$message` can be either an array of lines or a reference to an array and must be encoded by the caller to octets of whatever encoding is required, e.g. by using the Encode module's `encode()` function.

  The message, either sent via `datasend` or as the `$message` parameter, must be in the format as described by RFC822 and must contain From:, Newsgroups: and Subject: headers.

- `postfh()`

  Post a new article to the news server using a tied filehandle. If posting is allowed, this method will return a tied filehandle that you can print() the contents of the article to be posted. You must explicitly close() the filehandle when you are finished posting the article, and the return value from the close() call will indicate whether the message was successfully posted.

- `slave()`

  Tell the remote server that I am not a user client, but probably another news server.

- `quit()`

  Quit the remote server and close the socket connection.

- `can_inet6()`

  Returns whether we can use IPv6.

- `can_ssl()`

  Returns whether we can use SSL.

Extension Methods

These methods use commands that are not part of the RFC977 documentation. Some servers may not support all of them.

- `newsgroups([$pattern])`

  Returns a reference to a hash where the keys are all the group names which match `$pattern`, or all of the groups if no pattern is specified, and each value contains the description text for the group.

- `distributions()`

  Returns a reference to a hash where the keys are all the possible distribution names and the values are the distribution descriptions.

- `distribution_patterns()`

  Returns a reference to an array where each element, itself an array reference, consists of the three fields of a line of the distrib.pats list maintained by some NNTP

servers, namely: a weight, a wildmat and a value which the client may use to construct a Distribution header.

- `subscriptions()`

  Returns a reference to a list which contains a list of groups which are recommended for a new user to subscribe to.

- `overview_fmt()`

  Returns a reference to an array which contain the names of the fields returned by `xover`.

- `active_times()`

  Returns a reference to a hash where the keys are the group names and each value is a reference to an array containing the time the groups was created and an identifier, possibly an Email address, of the creator.

- `active([$pattern])`

  Similar to `list` but only active groups that match the pattern are returned. `$pattern` can be a group pattern.

- `xgtitle($pattern)`

  Returns a reference to a hash where the keys are all the group names which match `$pattern` and each value is the description text for the group.

- `xhdr($header, $message_spec)`

  Obtain the header field `$header` for all the messages specified.

  The return value will be a reference to a hash where the keys are the message numbers and each value contains the text of the requested header for that message.

- `xover($message_spec)`

  The return value will be a reference to a hash where the keys are the message numbers and each value contains a reference to an array which contains the overview fields for that message.

  The names of the fields can be obtained by calling `overview_fmt`.

- `xpath($message_id)`

  Returns the path name to the file on the server which contains the specified message.

- `xpat($header, $pattern, $message_spec)`

  The result is the same as `xhdr` except the is will be restricted to headers where the text of the header matches `$pattern`

- `xrover($message_spec)`

  The XROVER command returns reference information for the article(s) specified.

  Returns a reference to a HASH where the keys are the message numbers and the values are the References: lines from the articles

- `listgroup([$group])`

Returns a reference to a list of all the active messages in `$group`, or the current group if `$group` is not specified.

- `reader()`

  Tell the server that you are a reader and not another server.

  This is required by some servers. For example if you are connecting to an INN server and you have transfer permission your connection will be connected to the transfer daemon, not the NNTP daemon. Issuing this command will cause the transfer daemon to hand over control to the NNTP daemon.

  Some servers do not understand this command, but issuing it and ignoring the response is harmless.

Unsupported

The following NNTP command are unsupported by the package, and there are no plans to do so.

```
AUTHINFO GENERIC
XTHREAD
XSEARCH
XINDEX
```

Definitions

- $message_spec

  `$message_spec` is either a single message-id, a single message number, or a reference to a list of two message numbers.

  If `$message_spec` is a reference to a list of two message numbers and the second number in a range is less than or equal to the first then the range represents all messages in the group after the first message number.

  NOTE For compatibility reasons only with earlier versions of Net::NNTP a message spec can be passed as a list of two numbers, this is deprecated and a reference to the list should now be passed

- $pattern

  The `NNTP` protocol uses the `WILDMAT` format for patterns. The WILDMAT format was first developed by Rich Salz based on the format used in the UNIX "find" command to articulate file names. It was developed to provide a uniform mechanism for matching patterns in the same manner that the UNIX shell matches filenames.

  Patterns are implicitly anchored at the beginning and end of each string when testing for a match.

  There are five pattern matching operations other than a strict one-to-one match between the pattern and the source to be checked for a match.

  The first is an asterisk * to match any sequence of zero or more characters.

  The second is a question mark ? to match any single character. The third specifies a specific set of characters.

The set is specified as a list of characters, or as a range of characters where the beginning and end of the range are separated by a minus (or dash) character, or as any combination of lists and ranges. The dash can also be included in the set as a character it if is the beginning or end of the set. This set is enclosed in square brackets. The close square bracket] may be used in a set if it is the first character in the set.

The fourth operation is the same as the logical not of the third operation and is specified the same way as the third with the addition of a caret character ^ at the beginning of the test string just inside the open square bracket.

The final operation uses the backslash character to invalidate the special meaning of an open square bracket [, the asterisk, backslash or the question mark. Two backslashes in sequence will result in the evaluation of the backslash as a character with no special meaning.

- Examples
- `[^]-]`

  matches any single character other than a close square bracket or a minus sign/dash.
- `*bdc`

  matches any string that ends with the string “bdc” including the string “bdc” (without quotes).
- `[0-9a-zA-Z]`

  matches any single printable alphanumeric ASCII character.
- `a??d`

  matches any four character string which begins with a and ends with d.

EXPORTS

None.

KNOWN BUGS

See https://rt.cpan.org/Dist/Display.html?Status=Active&Queue=libnet.

SEE ALSO

Net::Cmd, *IO::Socket::SSL*.

AUTHOR

Graham Barr <gbarr@pobox.com>.

Steve Hay <shay@cpan.org> is now maintaining libnet as of version 1.22_02.

COPYRIGHT

Copyright (C) 1995-1997 Graham Barr. All rights reserved.

Copyright (C) 2013-2016, 2020 Steve Hay. All rights reserved.

LICENCE

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself, i.e. under the terms of either the GNU General Public License or the Artistic License, as specified in the `LICENCE` file.

VERSION

Version 3.15

DATE

20 March 2023

HISTORY

See the `Changes` file.

Net::Netrc

OO interface to users netrc file

SYNOPSIS

```
use Net::Netrc;

$mach = Net::Netrc->lookup('some.machine');
$login = $mach->login;
($login, $password, $account) = $mach->lpa;
```

DESCRIPTION

`Net::Netrc` is a class implementing a simple interface to the .netrc file used as by the ftp program.

`Net::Netrc` also implements security checks just like the ftp program, these checks are, first that the .netrc file must be owned by the user and second the ownership permissions should be such that only the owner has read and write access. If these conditions are not met then a warning is output and the .netrc file is not read.

The `.netrc` File

The .netrc file contains login and initialization information used by the auto-login process. It resides in the user's home directory. The following tokens are recognized; they may be separated by spaces, tabs, or new-lines:

- machine name

  Identify a remote machine name. The auto-login process searches the .netrc file for a machine token that matches the remote machine specified. Once a match is made, the subsequent .netrc tokens are processed, stopping when the end of file is reached or an- other machine or a default token is encountered.

- default

  This is the same as machine name except that default matches any name. There can be only one default token, and it must be after all machine tokens. This is normally used as:

  ```
  default login anonymous password user@site
  ```

  thereby giving the user automatic anonymous login to machines not specified in .netrc.

- login name

  Identify a user on the remote machine. If this token is present, the auto-login process will initiate a login using the specified name.

- password string

  Supply a password. If this token is present, the auto-login process will supply the specified string if the remote server requires a password as part of the login process.

- account string

  Supply an additional account password. If this token is present, the auto-login process will supply the specified string if the remote server requires an additional account password.

- macdef name

  Define a macro. `Net::Netrc` only parses this field to be compatible with *ftp*.

Class Methods

The constructor for a `Net::Netrc` object is not called new as it does not really create a new object. But instead is called `lookup` as this is essentially what it does.

- `lookup($machine[, $login])`

  Lookup and return a reference to the entry for `$machine`. If `$login` is given then the entry returned will have the given login. If `$login` is not given then the first entry in the .netrc file for `$machine` will be returned.

  If a matching entry cannot be found, and a default entry exists, then a reference to the default entry is returned.

If there is no matching entry found and there is no default defined, or no .netrc file is found, then `undef` is returned.

Object Methods

- `login()`

  Return the login id for the netrc entry

- `password()`

  Return the password for the netrc entry

- `account()`

  Return the account information for the netrc entry

- `lpa()`

  Return a list of login, password and account information for the netrc entry

EXPORTS

None.

KNOWN BUGS

See https://rt.cpan.org/Dist/Display.html?Status=Active&Queue=libnet.

SEE ALSO

Net::Cmd.

AUTHOR

Graham Barr <gbarr@pobox.com>.

Steve Hay <shay@cpan.org> is now maintaining libnet as of version 1.22_02.

COPYRIGHT

Copyright (C) 1995-1998 Graham Barr. All rights reserved.

Copyright (C) 2013-2014, 2020 Steve Hay. All rights reserved.

LICENCE

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself, i.e. under the terms of either the GNU General Public License or the Artistic License, as specified in the `LICENCE` file.

VERSION

Version 3.15

DATE

20 March 2023

HISTORY

See the Changes file.

Net::POP3

Post Office Protocol 3 Client class (RFC1939)

SYNOPSIS

```
use Net::POP3;

# Constructors
$pop = Net::POP3->new('pop3host');
$pop = Net::POP3->new('pop3host', Timeout => 60);
$pop = Net::POP3->new('pop3host', SSL => 1, Timeout => 60);

if ($pop->login($username, $password) > 0) {
  my $msgnums = $pop->list; # hashref of msgnum => size
  foreach my $msgnum (keys %$msgnums) {
    my $msg = $pop->get($msgnum);
    print @$msg;
    $pop->delete($msgnum);
  }
}

$pop->quit;
```

DESCRIPTION

This module implements a client interface to the POP3 protocol, enabling a perl5 application to talk to POP3 servers. This documentation assumes that you are familiar with the POP3 protocol described in RFC1939. With *IO::Socket::SSL* installed it also provides support for implicit and explicit TLS encryption, i.e. POP3S or POP3+STARTTLS.

A new Net::POP3 object must be created with the *new* method. Once this has been done, all POP3 commands are accessed via method calls on the object.

The Net::POP3 class is a subclass of Net::Cmd and (depending on avaibility) of IO::Socket::IP, IO::Socket::INET6 or IO::Socket::INET.

Class Methods

- `new([$host][, %options])`

  This is the constructor for a new Net::POP3 object. `$host` is the name of the remote host to which an POP3 connection is required.

  `$host` is optional. If `$host` is not given then it may instead be passed as the `Host` option described below. If neither is given then the `POP3_Hosts` specified in `Net::Config` will be used.

  `%options` are passed in a hash like fashion, using key and value pairs. Possible options are:

  Host - POP3 host to connect to. It may be a single scalar, as defined for the `PeerAddr` option in *IO::Socket::INET*, or a reference to an array with hosts to try in turn. The "host" method will return the value which was used to connect to the host.

  Port - port to connect to. Default - 110 for plain POP3 and 995 for POP3s (direct SSL).

  SSL - If the connection should be done from start with SSL, contrary to later upgrade with `starttls`. You can use SSL arguments as documented in *IO::Socket::SSL*, but it will usually use the right arguments already.

  LocalAddr and **LocalPort** - These parameters are passed directly to IO::Socket to allow binding the socket to a specific local address and port. For compatibility with older versions **ResvPort** can be used instead of **LocalPort**.

  Domain - This parameter is passed directly to IO::Socket and makes it possible to enforce IPv4 connections even if *IO::Socket::IP* is used as super class. Alternatively **Family** can be used.

  Timeout - Maximum time, in seconds, to wait for a response from the POP3 server (default: 120)

  Debug - Enable debugging information

Object Methods

Unless otherwise stated all methods return either a *true* or *false* value, with *true* meaning that the operation was a success. When a method states that it returns a value, failure will be returned as *undef* or an empty list.

`Net::POP3` inherits from `Net::Cmd` so methods defined in `Net::Cmd` may be used to send commands to the remote POP3 server in addition to the methods documented here.

- `host()`

  Returns the value used by the constructor, and passed to IO::Socket::INET, to connect to the host.

- `auth($username, $password)`

  Attempt SASL authentication.

- `user($user)`

  Send the USER command.

- `pass($pass)`

  Send the PASS command. Returns the number of messages in the mailbox.

- `login([$user[, $pass]])`

  Send both the USER and PASS commands. If `$pass` is not given the `Net::POP3` uses `Net::Netrc` to lookup the password using the host and username. If the username is not specified then the current user name will be used.

  Returns the number of messages in the mailbox. However if there are no messages on the server the string `"0E0"` will be returned. This is will give a true value in a boolean context, but zero in a numeric context.

  If there was an error authenticating the user then *undef* will be returned.

- `starttls(%sslargs)`

  Upgrade existing plain connection to SSL. You can use SSL arguments as documented in *IO::Socket::SSL*, but it will usually use the right arguments already.

- `apop([$user[, $pass]])`

  Authenticate with the server identifying as `$user` with password `$pass`. Similar to "login", but the password is not sent in clear text.

  To use this method you must have the Digest::MD5 or the MD5 module installed, otherwise this method will return *undef*.

- `banner()`

  Return the sever's connection banner

- `capa()`

  Return a reference to a hash of the capabilities of the server. APOP is added as a pseudo capability. Note that I've been unable to find a list of the standard capability values, and some appear to be multi-word and some are not. We make an attempt at intelligently parsing them, but it may not be correct.

- `capabilities()`

  Just like capa, but only uses a cache from the last time we asked the server, so as to avoid asking more than once.

- `top($msgnum[, $numlines])`

  Get the header and the first `$numlines` of the body for the message `$msgnum`. Returns a reference to an array which contains the lines of text read from the server.

- `list([$msgnum])`

  If called with an argument the `list` returns the size of the message in octets.

  If called without arguments a reference to a hash is returned. The keys will be the `$msgnum`'s of all undeleted messages and the values will be their size in octets.

- `get($msgnum[, $fh])`

  Get the message `$msgnum` from the remote mailbox. If `$fh` is not given then get returns a reference to an array which contains the lines of text read from the server. If `$fh` is given then the lines returned from the server are printed to the filehandle `$fh`.

- `getfh($msgnum)`

  As per get(), but returns a tied filehandle. Reading from this filehandle returns the requested message. The filehandle will return EOF at the end of the message and should not be reused.

- `last()`

  Returns the highest `$msgnum` of all the messages accessed.

- `popstat()`

  Returns a list of two elements. These are the number of undeleted elements and the size of the mbox in octets.

- `ping($user)`

  Returns a list of two elements. These are the number of new messages and the total number of messages for `$user`.

- `uidl([$msgnum])`

  Returns a unique identifier for `$msgnum` if given. If `$msgnum` is not given `uidl` returns a reference to a hash where the keys are the message numbers and the values are the unique identifiers.

- `delete($msgnum)`

  Mark message `$msgnum` to be deleted from the remote mailbox. All messages that are marked to be deleted will be removed from the remote mailbox when the server connection closed.

- `reset()`

  Reset the status of the remote POP3 server. This includes resetting the status of all messages to not be deleted.

- `quit()`

  Quit and close the connection to the remote POP3 server. Any messages marked as deleted will be deleted from the remote mailbox.

- `can_inet6()`

  Returns whether we can use IPv6.

- `can_ssl()`

  Returns whether we can use SSL.

Notes

If a `Net::POP3` object goes out of scope before `quit` method is called then the `reset` method will called before the connection is closed. This means that any messages marked to be deleted will not be.

EXPORTS

None.

KNOWN BUGS

See https://rt.cpan.org/Dist/Display.html?Status=Active&Queue=libnet.

SEE ALSO

Net::Netrc, *Net::Cmd*, *IO::Socket::SSL*.

AUTHOR

Graham Barr <gbarr@pobox.com>.

Steve Hay <shay@cpan.org> is now maintaining libnet as of version 1.22_02.

COPYRIGHT

Copyright (C) 1995-2004 Graham Barr. All rights reserved.

Copyright (C) 2013-2016, 2020 Steve Hay. All rights reserved.

LICENCE

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself, i.e. under the terms of either the GNU General Public License or the Artistic License, as specified in the `LICENCE` file.

VERSION

Version 3.15

DATE

20 March 2023

HISTORY

See the `Changes` file.

Net::Ping

check a remote host for reachability

SYNOPSIS

```
use Net::Ping;

my $p = Net::Ping->new();
print "$host is alive.\n" if $p->ping($host);
$p->close();

my $p = Net::Ping->new("icmp");
$p->bind($my_addr); # Specify source interface of pings
foreach my $host (@host_array)
{
    print "$host is ";
    print "NOT " unless $p->ping($host, 2);
    print "reachable.\n";
    sleep(1);
}
$p->close();

my $p = Net::Ping->new("icmpv6");
my $ip = "[fd00:dead:beef::4e]";
print "$ip is alive.\n" if $p->ping($ip);

my $p = Net::Ping->new("tcp", 2);
# Try connecting to the www port instead of the echo port
$p->port_number(scalar(getservbyname("http", "tcp")));
while ($stop_time > time())
{
    print "$host not reachable ", scalar(localtime()), "\n"
        unless $p->ping($host);
    sleep(300);
}
undef($p);

# Like tcp protocol, but with many hosts
my $p = Net::Ping->new("syn");
$p->port_number(getservbyname("http", "tcp"));
foreach my $host (@host_array) {
  $p->ping($host);
}
while (my ($host, $rtt, $ip) = $p->ack) {
  print "HOST: $host [$ip] ACKed in $rtt seconds.\n";
}

# High precision syntax (requires Time::HiRes)
my $p = Net::Ping->new();
$p->hires();
my ($ret, $duration, $ip) = $p->ping($host, 5.5);
printf("$host [ip: $ip] is alive (packet return time: %.2f ms)\n",
        1000 * $duration)
```

(continues on next page)

(continued from previous page)

```
  if $ret;
$p->close();

# For backward compatibility
print "$host is alive.\n" if pingecho($host);
```

DESCRIPTION

This module contains methods to test the reachability of remote hosts on a network. A ping object is first created with optional parameters, a variable number of hosts may be pinged multiple times and then the connection is closed.

You may choose one of six different protocols to use for the ping. The "tcp" protocol is the default. Note that a live remote host may still fail to be pingable by one or more of these protocols. For example, www.microsoft.com is generally alive but not "icmp" pingable.

With the "tcp" protocol the ping() method attempts to establish a connection to the remote host's echo port. If the connection is successfully established, the remote host is considered reachable. No data is actually echoed. This protocol does not require any special privileges but has higher overhead than the "udp" and "icmp" protocols.

Specifying the "udp" protocol causes the ping() method to send a udp packet to the remote host's echo port. If the echoed packet is received from the remote host and the received packet contains the same data as the packet that was sent, the remote host is considered reachable. This protocol does not require any special privileges. It should be borne in mind that, for a udp ping, a host will be reported as unreachable if it is not running the appropriate echo service. For Unix-like systems see inetd(8) for more information.

If the "icmp" protocol is specified, the ping() method sends an icmp echo message to the remote host, which is what the UNIX ping program does. If the echoed message is received from the remote host and the echoed information is correct, the remote host is considered reachable. Specifying the "icmp" protocol requires that the program be run as root or that the program be setuid to root.

If the "external" protocol is specified, the ping() method attempts to use the `Net::Ping::External` module to ping the remote host. `Net::Ping::External` interfaces with your system's default `ping` utility to perform the ping, and generally produces relatively accurate results. If `Net::Ping::External` if not installed on your system, specifying the "external" protocol will result in an error.

If the "syn" protocol is specified, the "ping" method will only send a TCP SYN packet to the remote host then immediately return. If the syn packet was sent successfully, it will return a true value, otherwise it will return false. NOTE: Unlike the other protocols, the return value does NOT determine if the remote host is alive or not since the full TCP three-way handshake may not have completed yet. The remote host is only considered reachable if it receives a TCP ACK within the timeout specified. To begin waiting for the ACK packets, use the "ack" method as explained below. Use the "syn" protocol instead the "tcp" protocol to determine reachability of multiple destinations simultaneously by sending parallel TCP SYN packets. It will not block while testing each remote host. This protocol does not require any special privileges.

Functions

- Net::Ping->new([proto, timeout, bytes, device, tos, ttl, family, host, port, bind, gateway, retrans, pingstring, source_verify econnrefused dontfrag IPV6_USE_MIN_MTU IPV6_RECVPATHMTU])

  Create a new ping object. All of the parameters are optional and can be passed as hash ref. All options besides the first 7 must be passed as hash ref.

  `proto` specifies the protocol to use when doing a ping. The current choices are "tcp", "udp", "icmp", "icmpv6", "stream", "syn", or "external". The default is "tcp".

  If a `timeout` in seconds is provided, it is used when a timeout is not given to the ping() method (below). The timeout must be greater than 0 and the default, if not specified, is 5 seconds.

  If the number of data bytes (`bytes`) is given, that many data bytes are included in the ping packet sent to the remote host. The number of data bytes is ignored if the protocol is "tcp". The minimum (and default) number of data bytes is 1 if the protocol is "udp" and 0 otherwise. The maximum number of data bytes that can be specified is 65535, but staying below the MTU (1472 bytes for ICMP) is recommended. Many small devices cannot deal with fragmented ICMP packets.

  If `device` is given, this device is used to bind the source endpoint before sending the ping packet. I believe this only works with superuser privileges and with udp and icmp protocols at this time.

  If <tos> is given, this ToS is configured into the socket.

  For icmp, `ttl` can be specified to set the TTL of the outgoing packet.

  Valid `family` values for IPv4:

```
4, v4, ip4, ipv4, AF_INET (constant)
```

  Valid `family` values for IPv6:

```
6, v6, ip6, ipv6, AF_INET6 (constant)
```

  The `host` argument implicitly specifies the family if the family argument is not given.

  The `port` argument is only valid for a udp, tcp or stream ping, and will not do what you think it does. ping returns true when we get a "Connection refused"! The default is the echo port.

  The `bind` argument specifies the local_addr to bind to. By specifying a bind argument you don't need the bind method.

  The `gateway` argument is only valid for IPv6, and requires a IPv6 address.

  The `retrans` argument the exponential backoff rate, default 1.2. It matches the $def_factor global.

  The `dontfrag` argument sets the IP_DONTFRAG bit, but note that IP_DONTFRAG is not yet defined by Socket, and not available on many systems. Then it is ignored. On linux it also sets IP_MTU_DISCOVER to IP_PMTUDISC_DO but need we don't chunk oversized packets. You need to set $data_size manually.

- $p->ping($host [, $timeout [, $family]]);

  Ping the remote host and wait for a response. $host can be either the hostname or the IP number of the remote host. The optional timeout must be greater than 0 seconds and defaults to whatever was specified when the ping object was created. Returns a success flag. If the hostname cannot be found or there is a problem with the IP number, the success flag returned will be undef. Otherwise, the success flag will be 1 if the host is reachable and 0 if it is not. For most practical purposes, undef and 0 and can be treated as the same case. In array context, the elapsed time as well as the string form of the ip the host resolved to are also returned. The elapsed time value will be a float, as returned by the Time::HiRes::time() function, if hires() has been previously called, otherwise it is returned as an integer.

- $p->source_verify({ 0 | 1 });

  Allows source endpoint verification to be enabled or disabled. This is useful for those remote destinations with multiples interfaces where the response may not originate from the same endpoint that the original destination endpoint was sent to. This only affects udp and icmp protocol pings.

  This is enabled by default.

- $p->service_check({ 0 | 1 });

  Set whether or not the connect behavior should enforce remote service availability as well as reachability. Normally, if the remote server reported ECONNREFUSED, it must have been reachable because of the status packet that it reported. With this option enabled, the full three-way tcp handshake must have been established successfully before it will claim it is reachable. NOTE: It still does nothing more than connect and disconnect. It does not speak any protocol (i.e., HTTP or FTP) to ensure the remote server is sane in any way. The remote server CPU could be grinding to a halt and unresponsive to any clients connecting, but if the kernel throws the ACK packet, it is considered alive anyway. To really determine if the server is responding well would be application specific and is beyond the scope of Net::Ping. For udp protocol, enabling this option demands that the remote server replies with the same udp data that it was sent as defined by the udp echo service.

  This affects the “udp”, “tcp”, and “syn” protocols.

  This is disabled by default.

- $p->tcp_service_check({ 0 | 1 });

  Deprecated method, but does the same as service_check() method.

- $p->hires({ 0 | 1 });

  With 1 causes this module to use Time::HiRes module, allowing milliseconds to be returned by subsequent calls to ping().

- $p->time

  The current time, hires or not.

- $p->socket_blocking_mode($fh, $mode);

  Sets or clears the O_NONBLOCK flag on a file handle.

- $p->IPV6_USE_MIN_MTU

With argument sets the option. Without returns the option value.

- $p->IPV6_RECVPATHMTU

  Notify an according IPv6 MTU.

  With argument sets the option. Without returns the option value.

- $p->IPV6_HOPLIMIT

  With argument sets the option. Without returns the option value.

- $p->IPV6_REACHCONF *NYI*

  Sets ipv6 reachability IPV6_REACHCONF was removed in RFC3542. ping6 -R supports it. IPV6_REACHCONF requires root/admin permissions.

  With argument sets the option. Without returns the option value.

  Not yet implemented.

- $p->bind($local_addr);

  Sets the source address from which pings will be sent. This must be the address of one of the interfaces on the local host. $local_addr may be specified as a hostname or as a text IP address such as "192.168.1.1".

  If the protocol is set to "tcp", this method may be called any number of times, and each call to the ping() method (below) will use the most recent $local_addr. If the protocol is "icmp" or "udp", then bind() must be called at most once per object, and (if it is called at all) must be called before the first call to ping() for that object.

  The bind() call can be omitted when specifying the `bind` option to new().

- $p->message_type([$ping_type]);

  When you are using the "icmp" protocol, this call permit to change the message type to 'echo' or 'timestamp' (only for IPv4, see RFC 792).

  Without argument, it returns the currently used icmp protocol message type. By default, it returns 'echo'.

- $p->open($host);

  When you are using the "stream" protocol, this call pre-opens the tcp socket. It's only necessary to do this if you want to provide a different timeout when creating the connection, or remove the overhead of establishing the connection from the first ping. If you don't call `open()`, the connection is automatically opened the first time `ping()` is called. This call simply does nothing if you are using any protocol other than stream.

  The $host argument can be omitted when specifying the `host` option to new().

- $p->ack([$host]);

  When using the "syn" protocol, use this method to determine the reachability of the remote host. This method is meant to be called up to as many times as ping() was called. Each call returns the host (as passed to ping()) that came back with the TCP ACK. The order in which the hosts are returned may not necessarily be the same order in which they were SYN queued using the ping() method. If the timeout is reached before the TCP ACK is received, or if the remote host is not listening on the port attempted, then the TCP connection will not be established and ack() will

return undef. In list context, the host, the ack time, the dotted ip string, and the port number will be returned instead of just the host. If the optional `$host` argument is specified, the return value will be pertaining to that host only. This call simply does nothing if you are using any protocol other than "syn".

When "new" had a host option, this host will be used. Without `$host` argument, all hosts are scanned.

- $p->nack($failed_ack_host);

  The reason that `host $failed_ack_host` did not receive a valid ACK. Useful to find out why when `ack($fail_ack_host)` returns a false value.

- $p->ack_unfork($host)

  The variant called by "ack" with the "syn" protocol and `$syn_forking` enabled.

- $p->ping_icmp([$host, $timeout, $family])

  The "ping" method used with the icmp protocol. Under Linux under a non-root account this uses now SOCK_DGRAM.

- $p->ping_icmpv6([$host, $timeout, $family])

  The "ping" method used with the icmpv6 protocol. Under Linux under a non-root account this uses now SOCK_DGRAM.

- $p->ping_stream([$host, $timeout, $family])

  The "ping" method used with the stream protocol.

  Perform a stream ping. If the tcp connection isn't already open, it opens it. It then sends some data and waits for a reply. It leaves the stream open on exit.

- $p->ping_syn([$host, $ip, $start_time, $stop_time])

  The "ping" method used with the syn protocol. Sends a TCP SYN packet to host specified.

- $p->ping_syn_fork([$host, $timeout, $family])

  The "ping" method used with the forking syn protocol.

- $p->ping_tcp([$host, $timeout, $family])

  The "ping" method used with the tcp protocol.

- $p->ping_udp([$host, $timeout, $family])

  The "ping" method used with the udp protocol.

  Perform a udp echo ping. Construct a message of at least the one-byte sequence number and any additional data bytes. Send the message out and wait for a message to come back. If we get a message, make sure all of its parts match. If they do, we are done. Otherwise go back and wait for the message until we run out of time. Return the result of our efforts.

- $p->ping_external([$host, $timeout, $family])

  The "ping" method used with the external protocol. Uses Net::Ping::External to do an external ping.

- $p->tcp_connect([$ip, $timeout])

  Initiates a TCP connection, for a tcp ping.

- $p->tcp_echo([$ip, $timeout, $pingstring])

  Performs a TCP echo. It writes the given string to the socket and then reads it back. It returns 1 on success, 0 on failure.

- $p->close();

  Close the network connection for this ping object. The network connection is also closed by "undef $p". The network connection is automatically closed if the ping object goes out of scope (e.g. $p is local to a subroutine and you leave the subroutine).

- $p->port_number([$port_number])

  When called with a port number, the port number used to ping is set to `$port_number` rather than using the echo port. It also has the effect of calling `$p->service_check(1)` causing a ping to return a successful response only if that specific port is accessible. This function returns the value of the port that "ping" will connect to.

- $p->mselect

  A `select()` wrapper that compensates for platform peculiarities.

- $p->ntop

  Platform abstraction over `inet_ntop()`

- $p->checksum($msg)

  Do a checksum on the message. Basically sum all of the short words and fold the high order bits into the low order bits.

- $p->icmp_result

  Returns a list of addr, type, subcode.

- pingecho($host [, $timeout]);

  To provide backward compatibility with the previous version of *Net::Ping*, a `pingecho()` subroutine is available with the same functionality as before. `pingecho()` uses the tcp protocol. The return values and parameters are the same as described for the "ping" method. This subroutine is obsolete and may be removed in a future version of *Net::Ping*.

- wakeonlan($mac, [$host, [$port]])

  Emit the popular wake-on-lan magic udp packet to wake up a local device. See also Net::Wake, but this has the mac address as 1st arg. `$host` should be the local gateway. Without it will broadcast.

  Default host: '255.255.255.255' Default port: 9

```
perl -MNet::Ping=wakeonlan -e'wakeonlan "e0:69:95:35:68:d2"'
```

NOTES

There will be less network overhead (and some efficiency in your program) if you specify either the udp or the icmp protocol. The tcp protocol will generate 2.5 times or more traffic for each ping than either udp or icmp. If many hosts are pinged frequently, you may wish to implement a small wait (e.g. 25ms or more) between each ping to avoid flooding your network with packets.

The icmp and icmpv6 protocols requires that the program be run as root or that it be setuid to root. The other protocols do not require special privileges, but not all network devices implement tcp or udp echo.

Local hosts should normally respond to pings within milliseconds. However, on a very congested network it may take up to 3 seconds or longer to receive an echo packet from the remote host. If the timeout is set too low under these conditions, it will appear that the remote host is not reachable (which is almost the truth).

Reachability doesn't necessarily mean that the remote host is actually functioning beyond its ability to echo packets. tcp is slightly better at indicating the health of a system than icmp because it uses more of the networking stack to respond.

Because of a lack of anything better, this module uses its own routines to pack and unpack ICMP packets. It would be better for a separate module to be written which understands all of the different kinds of ICMP packets.

INSTALL

The latest source tree is available via git:

```
git clone https://github.com/rurban/Net-Ping.git
cd Net-Ping
```

The tarball can be created as follows:

```
perl Makefile.PL ; make ; make dist
```

The latest Net::Ping releases are included in cperl and perl5.

BUGS

For a list of known issues, visit:

https://rt.cpan.org/NoAuth/Bugs.html?Dist=Net-Ping and https://github.com/rurban/Net-Ping/issues

To report a new bug, visit:

https://github.com/rurban/Net-Ping/issues

AUTHORS

```
Current maintainers:
  perl11 (for cperl, with IPv6 support and more)
  p5p    (for perl5)
```

(continues on next page)

(continued from previous page)

```
Previous maintainers:
  bbb@cpan.org (Rob Brown)
  Steve Peters

External protocol:
  colinm@cpan.org (Colin McMillen)

Stream protocol:
  bronson@trestle.com (Scott Bronson)

Wake-on-lan:
  1999-2003 Clinton Wong

Original pingecho():
  karrer@bernina.ethz.ch (Andreas Karrer)
  pmarquess@bfsec.bt.co.uk (Paul Marquess)

Original Net::Ping author:
  mose@ns.ccsn.edu (Russell Mosemann)
```

COPYRIGHT

Copyright (c) 2017-2020, Reini Urban. All rights reserved.

Copyright (c) 2016, cPanel Inc. All rights reserved.

Copyright (c) 2012, Steve Peters. All rights reserved.

Copyright (c) 2002-2003, Rob Brown. All rights reserved.

Copyright (c) 2001, Colin McMillen. All rights reserved.

This program is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

Net::SMTP

Simple Mail Transfer Protocol Client

SYNOPSIS

```
use Net::SMTP;

# Constructors
$smtp = Net::SMTP->new('mailhost');
$smtp = Net::SMTP->new('mailhost', Timeout => 60);
```

DESCRIPTION

This module implements a client interface to the SMTP and ESMTP protocol, enabling a perl5 application to talk to SMTP servers. This documentation assumes that you are familiar with the concepts of the SMTP protocol described in RFC2821. With *IO::Socket::SSL* installed it also provides support for implicit and explicit TLS encryption, i.e. SMTPS or SMTP+STARTTLS.

The Net::SMTP class is a subclass of Net::Cmd and (depending on avaibility) of IO::Socket::IP, IO::Socket::INET6 or IO::Socket::INET.

Class Methods

- `new([$host][, %options])`

  This is the constructor for a new Net::SMTP object. `$host` is the name of the remote host to which an SMTP connection is required.

  On failure `undef` will be returned and `$@` will contain the reason for the failure.

  `$host` is optional. If `$host` is not given then it may instead be passed as the `Host` option described below. If neither is given then the `SMTP_Hosts` specified in `Net::Config` will be used.

  `%options` are passed in a hash like fashion, using key and value pairs. Possible options are:

  Hello - SMTP requires that you identify yourself. This option specifies a string to pass as your mail domain. If not given localhost.localdomain will be used.

  SendHello - If false then the EHLO (or HELO) command that is normally sent when constructing the object will not be sent. In that case the command will have to be sent manually by calling `hello()` instead.

  Host - SMTP host to connect to. It may be a single scalar (hostname[:port]), as defined for the `PeerAddr` option in *IO::Socket::INET*, or a reference to an array with hosts to try in turn. The "host" method will return the value which was used to connect to the host. Format - `PeerHost` from *IO::Socket::INET* new method.

  Port - port to connect to. Default - 25 for plain SMTP and 465 for immediate SSL.

  SSL - If the connection should be done from start with SSL, contrary to later upgrade with `starttls`. You can use SSL arguments as documented in *IO::Socket::SSL*, but it will usually use the right arguments already.

  LocalAddr and **LocalPort** - These parameters are passed directly to IO::Socket to allow binding the socket to a specific local address and port.

  Domain - This parameter is passed directly to IO::Socket and makes it possible to enforce IPv4 connections even if *IO::Socket::IP* is used as super class. Alternatively **Family** can be used.

  Timeout - Maximum time, in seconds, to wait for a response from the SMTP server (default: 120)

  ExactAddresses - If true then all `$address` arguments must be as defined by `addr-spec` in RFC2822. If not given, or false, then Net::SMTP will attempt to extract the address from the value passed.

Debug - Enable debugging information

Example:

```
$smtp = Net::SMTP->new('mailhost',
                       Hello => 'my.mail.domain',
                       Timeout => 30,
                       Debug   => 1,
                      );

# the same
$smtp = Net::SMTP->new(
                       Host => 'mailhost',
                       Hello => 'my.mail.domain',
                       Timeout => 30,
                       Debug   => 1,
                      );

# the same with direct SSL
$smtp = Net::SMTP->new('mailhost',
                       Hello => 'my.mail.domain',
                       Timeout => 30,
                       Debug   => 1,
                       SSL     => 1,
                      );

# Connect to the default server from Net::config
$smtp = Net::SMTP->new(
                       Hello => 'my.mail.domain',
                       Timeout => 30,
                      );
```

Object Methods

Unless otherwise stated all methods return either a *true* or *false* value, with *true* meaning that the operation was a success. When a method states that it returns a value, failure will be returned as *undef* or an empty list.

`Net::SMTP` inherits from `Net::Cmd` so methods defined in `Net::Cmd` may be used to send commands to the remote SMTP server in addition to the methods documented here.

- `banner()`

  Returns the banner message which the server replied with when the initial connection was made.

- `domain()`

  Returns the domain that the remote SMTP server identified itself as during connection.

- `hello($domain)`

  Tell the remote server the mail domain which you are in using the EHLO command (or HELO if EHLO fails). Since this method is invoked automatically when

the Net::SMTP object is constructed the user should normally not have to call it manually.

- `host()`

  Returns the value used by the constructor, and passed to IO::Socket::INET, to connect to the host.

- `etrn($domain)`

  Request a queue run for the `$domain` given.

- `starttls(%sslargs)`

  Upgrade existing plain connection to SSL. You can use SSL arguments as documented in *IO::Socket::SSL*, but it will usually use the right arguments already.

- `auth($username, $password)`

- `auth($sasl)`

  Attempt SASL authentication. Requires Authen::SASL module. The first form constructs a new Authen::SASL object using the given username and password; the second form uses the given Authen::SASL object.

- `mail($address[, %options])`

- `send($address)`

- `send_or_mail($address)`

- `send_and_mail($address)`

  Send the appropriate command to the server MAIL, SEND, SOML or SAML. `$address` is the address of the sender. This initiates the sending of a message. The method `recipient` should be called for each address that the message is to be sent to.

  The `mail` method can take some additional ESMTP `%options` which is passed in hash like fashion, using key and value pairs. Possible options are:

```
Size        => <bytes>
Return      => "FULL" | "HDRS"
Bits        => "7" | "8" | "binary"
Transaction => <ADDRESS>
Envelope    => <ENVID>     # xtext-encodes its argument
ENVID       => <ENVID>     # similar to Envelope, but expects␣
↪argument encoded
XVERP       => 1
AUTH        => <submitter> # encoded address according to RFC␣
↪2554
```

  The `Return` and `Envelope` parameters are used for DSN (Delivery Status Notification).

  The submitter address in `AUTH` option is expected to be in a format as required by RFC 2554, in an RFC2821-quoted form and xtext-encoded, or <> .

- `reset()`

Reset the status of the server. This may be called after a message has been initiated, but before any data has been sent, to cancel the sending of the message.

- `recipient($address[, $address[, ...]][, %options])`

  Notify the server that the current message should be sent to all of the addresses given. Each address is sent as a separate command to the server. Should the sending of any address result in a failure then the process is aborted and a *false* value is returned. It is up to the user to call `reset` if they so desire.

  The `recipient` method can also pass additional case-sensitive `%options` as an anonymous hash using key and value pairs. Possible options are:

```
Notify  => ['NEVER'] or ['SUCCESS','FAILURE','DELAY']  (see
↪below)
ORcpt   => <ORCPT>
SkipBad => 1        (to ignore bad addresses)
```

  If `SkipBad` is true the `recipient` will not return an error when a bad address is encountered and it will return an array of addresses that did succeed.

```
$smtp->recipient($recipient1,$recipient2);  # Good
$smtp->recipient($recipient1,$recipient2, { SkipBad => 1 });  #
↪Good
$smtp->recipient($recipient1,$recipient2, { Notify => ['FAILURE
↪','DELAY'], SkipBad => 1 });  # Good
@goodrecips=$smtp->recipient(@recipients, { Notify => ['FAILURE
↪'], SkipBad => 1 });  # Good
$smtp->recipient("$recipient,$recipient2"); # BAD
```

  Notify is used to request Delivery Status Notifications (DSNs), but your SMTP/ESMTP service may not respect this request depending upon its version and your site's SMTP configuration.

  Leaving out the Notify option usually defaults an SMTP service to its default behavior equivalent to ['FAILURE'] notifications only, but again this may be dependent upon your site's SMTP configuration.

  The NEVER keyword must appear by itself if used within the Notify option and "requests that a DSN not be returned to the sender under any conditions."

```
{Notify => ['NEVER']}

$smtp->recipient(@recipients, { Notify => ['NEVER'], SkipBad =>
↪1 });  # Good
```

  You may use any combination of these three values 'SUCCESS','FAILURE','DELAY' in the anonymous array reference as defined by RFC3461 (see https://www.ietf.org/rfc/rfc3461.txt for more information. Note: quotations in this topic from same.).

  A Notify parameter of 'SUCCESS' or 'FAILURE' "requests that a DSN be issued on successful delivery or delivery failure, respectively."

  A Notify parameter of 'DELAY' "indicates the sender's willingness to receive delayed DSNs. Delayed DSNs may be issued if delivery of a message has been delayed for an unusual amount of time (as determined by the Message Transfer Agent

(MTA) at which the message is delayed), but the final delivery status (whether successful or failure) cannot be determined. The absence of the DELAY keyword in a NOTIFY parameter requests that a "delayed" DSN NOT be issued under any conditions."

```
{Notify => ['SUCCESS','FAILURE','DELAY']}

$smtp->recipient(@recipients, { Notify => ['FAILURE','DELAY'],␣
↪SkipBad => 1 });  # Good
```

ORcpt is also part of the SMTP DSN extension according to RFC3461. It is used to pass along the original recipient that the mail was first sent to. The machine that generates a DSN will use this address to inform the sender, because he can't know if recipients get rewritten by mail servers. It is expected to be in a format as required by RFC3461, xtext-encoded.

- `to($address[, $address[, ...]])`
- `cc($address[, $address[, ...]])`
- `bcc($address[, $address[, ...]])`

  Synonyms for `recipient`.

- `data([$data])`

  Initiate the sending of the data from the current message.

  `$data` may be a reference to a list or a list and must be encoded by the caller to octets of whatever encoding is required, e.g. by using the Encode module's `encode()` function.

  If specified the contents of `$data` and a termination string `".\r\n"` is sent to the server. The result will be true if the data was accepted.

  If `$data` is not specified then the result will indicate that the server wishes the data to be sent. The data must then be sent using the `datasend` and `dataend` methods described in *Net::Cmd*.

- `bdat($data)`
- `bdatlast($data)`

  Use the alternate `$data` command "BDAT" of the data chunking service extension defined in RFC1830 for efficiently sending large MIME messages.

- `expand($address)`

  Request the server to expand the given address Returns an array which contains the text read from the server.

- `verify($address)`

  Verify that `$address` is a legitimate mailing address.

  Most sites usually disable this feature in their SMTP service configuration. Use "Debug => 1" option under new() to see if disabled.

- `help([$subject])`

  Request help text from the server. Returns the text or undef upon failure

- `quit()`

  Send the QUIT command to the remote SMTP server and close the socket connection.

- `can_inet6()`

  Returns whether we can use IPv6.

- `can_ssl()`

  Returns whether we can use SSL.

Addresses

Net::SMTP attempts to DWIM with addresses that are passed. For example an application might extract The From: line from an email and pass that to mail(). While this may work, it is not recommended. The application should really use a module like Mail::Address to extract the mail address and pass that.

If `ExactAddresses` is passed to the constructor, then addresses should be a valid rfc2821-quoted address, although Net::SMTP will accept the address surrounded by angle brackets.

```
funny user@domain       WRONG
"funny user"@domain     RIGHT, recommended
<"funny user"@domain>   OK
```

EXAMPLES

This example prints the mail domain name of the SMTP server known as mailhost:

```
#!/usr/local/bin/perl -w

use Net::SMTP;

$smtp = Net::SMTP->new('mailhost');
print $smtp->domain,"\n";
$smtp->quit;
```

This example sends a small message to the postmaster at the SMTP server known as mailhost:

```
#!/usr/local/bin/perl -w

use Net::SMTP;

my $smtp = Net::SMTP->new('mailhost');

$smtp->mail($ENV{USER});
if ($smtp->to('postmaster')) {
 $smtp->data();
 $smtp->datasend("To: postmaster\n");
```

(continues on next page)

(continued from previous page)

```
 $smtp->datasend("\n");
 $smtp->datasend("A simple test message\n");
 $smtp->dataend();
} else {
 print "Error: ", $smtp->message();
}

$smtp->quit;
```

EXPORTS

None.

KNOWN BUGS

See https://rt.cpan.org/Dist/Display.html?Status=Active&Queue=libnet.

SEE ALSO

Net::Cmd, *IO::Socket::SSL*.

AUTHOR

Graham Barr <gbarr@pobox.com>.

Steve Hay <shay@cpan.org> is now maintaining libnet as of version 1.22_02.

COPYRIGHT

Copyright (C) 1995-2004 Graham Barr. All rights reserved.

Copyright (C) 2013-2016, 2020 Steve Hay. All rights reserved.

LICENCE

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself, i.e. under the terms of either the GNU General Public License or the Artistic License, as specified in the `LICENCE` file.

VERSION

Version 3.15

DATE

20 March 2023

HISTORY

See the Changes file.

Net::SSLeay

Net::SSLeay - Perl bindings for OpenSSL's libssl and libcrypto.

TLS connections, certificates, keys, digests and the rest of OpenSSL, one Perl function per C function: `Net::SSLeay::CTX_new` is `SSL_CTX_new`, `Net::SSLeay::X509_get_subject_name` is `X509_get_subject_name`, with the same arguments, return values and constants. Objects are passed around as plain integers (the C pointers) and freed explicitly, as in C. Most programs use it through IO::Socket::SSL or AnyEvent::TLS, which are built on it; use it directly for what they do not expose.

This is Net::SSLeay 1.96: all 709 functions and 769 constants of the CPAN distribution, and its Perl half (`Net/SSLeay.pm`, `Net/SSLeay/Handle.pm`) unchanged. Because each function is its C namesake, OpenSSL's manual pages (`man SSL_CTX_new`) describe what it does; the overview below says where to look.

Requirements

OpenSSL 3.0 or newer, as `libssl.so.3` and `libcrypto.so.3`, loaded when the module is: nothing is linked into pperl. Without them, `use Net::SSLeay` dies naming the package to install. The module is part of the `native-max` build.

Synopsis

A certificate's subject, expiry and fingerprint - no network needed:

```
use Net::SSLeay;

my $bio  = Net::SSLeay::BIO_new_file('cert.pem', 'r') or die "cert.
 ↪pem: $!";
my $x509 = Net::SSLeay::PEM_read_bio_X509($bio);
Net::SSLeay::BIO_free($bio);

print Net::SSLeay::X509_NAME_oneline(Net::SSLeay::X509_get_subject_
 ↪name($x509)), "\n";
print Net::SSLeay::P_ASN1_TIME_get_isotime(Net::SSLeay::X509_get_
 ↪notAfter($x509)), "\n";
print unpack('H*', Net::SSLeay::X509_digest($x509,
    Net::SSLeay::EVP_get_digestbyname('sha256'))), "\n";
Net::SSLeay::X509_free($x509);
```

A TLS client over a socket of your own:

```
use Socket;
use Net::SSLeay qw(die_now die_if_ssl_error);

my ($host, $port) = ('example.com', 443);
Net::SSLeay::initialize();
```

(continues on next page)

(continued from previous page)

```
socket(my $sock, PF_INET, SOCK_STREAM, getprotobyname('tcp')) or die␣
↪"socket: $!";
connect($sock, sockaddr_in($port, inet_aton($host))) or die
↪"connect: $!";

my $ctx = Net::SSLeay::CTX_new() or die_now("CTX_new: $!");
Net::SSLeay::CTX_set_options($ctx, Net::SSLeay::OP_ALL());
my $ssl = Net::SSLeay::new($ctx) or die_now("new: $!");
Net::SSLeay::set_tlsext_host_name($ssl, $host);    # SNI
Net::SSLeay::set_fd($ssl, fileno $sock);
Net::SSLeay::connect($ssl) == 1 or die_if_ssl_error('connect');
print 'cipher: ', Net::SSLeay::get_cipher($ssl), "\n";

Net::SSLeay::write($ssl, "GET / HTTP/1.0\r\nHost: $host\r\n\r\n");
my $reply = '';
while (defined(my $chunk = Net::SSLeay::read($ssl))) {
    last unless length $chunk;
    $reply .= $chunk;
}
Net::SSLeay::free($ssl);
Net::SSLeay::CTX_free($ctx);
```

This client does not verify the server's certificate. To do so, before `connect`:

```
Net::SSLeay::CTX_set_default_verify_paths($ctx);   # the system's␣
↪CAs
Net::SSLeay::set_verify($ssl, Net::SSLeay::VERIFY_PEER(), undef);
Net::SSLeay::X509_VERIFY_PARAM_set1_host(Net::SSLeay::get0_param(
↪$ssl), $host);
```

`connect` then fails for a certificate that does not chain to a trusted CA or does not name `$host`. IO::Socket::SSL does all of this by default.

The functions, by family

The name says which C object a function works on; strip the prefix the C function had (`SSL_`, sometimes `SSL_CTX_` becomes `CTX_`) and look it up in OpenSSL's manual.

- **Contexts** - `CTX_*` (112): `CTX_new`, the protocol versions, ciphers, certificates and keys a context offers, verification, sessions, and every callback (`CTX_set_verify`, `CTX_set_alpn_select_cb`, `CTX_set_tlsext_servername_callback`).
- **Connections** - functions without a prefix: `new`, `set_fd`, `connect`, `accept`, `read`, `write`, `shutdown`, `free`, and the `get_*`/`set_*` accessors of a connection (`get_peer_certificate`, `get_cipher`, `set_tlsext_host_name`).
- **Sessions** - `SESSION_*`, `get_session`, `set_session`.
- **Certificates** - `X509_*` (162): names, extensions, stores and verification, CRLs, requests; `P_X509_*` are Net::SSLeay's own helpers that return Perl lists (`P_X509_get_crl_distribution_points`, `P_X509_get_ext_key_usage`).
- **Keys and digests** - `EVP_*`, `RSA_*`, `EC_*`, `BN_*`; `RAND_*` for random bytes.

- **Files and encodings** - `PEM_*`, `d2i_*`/`i2d_*` (DER), `BIO_*` (memory and file buffers), `ASN1_*` and `P_ASN1_*` (times and integers).
- **Errors** - `ERR_get_error`, `ERR_error_string`, and the module's `die_now`, `die_if_ssl_error` and `print_errs`.
- **The rest** - `OCSP_*`, `OBJ_*`, `OSSL_PROVIDER_*`, `OPENSSL_*` (version and initialisation).

Constants are functions too, named without their C prefix: `Net::SSLeay::VERIFY_PEER()`, `Net::SSLeay::OP_ALL()`, `Net::SSLeay::ERROR_WANT_READ()`.

Written in Perl

`Net/SSLeay.pm` adds whole-request helpers: `get_https`, `post_https`, `sslcat`, `make_headers`, `make_form`, and `Net::SSLeay::Handle`, a tied filehandle over a connection.

```
my ($page, $response, %headers) = Net::SSLeay::get_https('example.
↪com', 443, '/');
```

Differences from upstream

- Which functions exist is decided by the OpenSSL loaded, not by the one the module was compiled against. Upstream compiles a function only for an OpenSSL that has its feature; here a function is defined only when the loaded library has every C function it needs - so `defined &Net::SSLeay::client_hello_get_extension_order` is false under OpenSSL 3.0 and 3.1, as it is for a Net::SSLeay built there (t/81-xs-native/Net-SSLeay/660-xsub-set-follows-openssl.t). On this machine's OpenSSL it is upstream's set of 709 (600-xsub-set.t).
- The functions an OpenSSL can provide only when built with options off by default (MD2, SSLv3 methods) are not provided, whatever is loaded.
- OpenSSL before 3.0 and LibreSSL are not supported; upstream builds against them too.
- `RAND_bytes`, `RAND_priv_bytes` and `RAND_pseudo_bytes` with a negative count return 0 and leave the buffer empty, where perl exits "Out of memory" (670-or-better.t).

Modules

- `Net::SSLeay::Handle` - Perl module that lets SSL (HTTPS) sockets be handled as standard file handles.

Net::SSLeay::Handle

Perl module that lets SSL (HTTPS) sockets be handled as standard file handles.

SYNOPSIS

```
use Net::SSLeay::Handle qw/shutdown/;
my ($host, $port) = ("localhost", 443);

tie(*SSL, "Net::SSLeay::Handle", $host, $port);

print SSL "GET / HTTP/1.0\r\n";
shutdown(\*SSL, 1);
print while (<SSL>);
close SSL;
```

DESCRIPTION

Net::SSLeay::Handle allows you to request and receive HTTPS web pages using "old-fashion" file handles as in:

```
print SSL "GET / HTTP/1.0\r\n";
```

and

```
print while (<SSL>);
```

If you export the shutdown routine, then the only extra code that you need to add to your program is the tie function as in:

```
my $socket;
if ($scheme eq "https") {
    tie(*S2, "Net::SSLeay::Handle", $host, $port);
    $socket = \*S2;
else {
    $socket = Net::SSLeay::Handle->make_socket($host, $port);
}
print $socket $request_headers;
...
```

FUNCTIONS

- shutdown

  ```
  shutdown(\*SOCKET, $mode)
  ```

  Calls to the main shutdown() don't work with tied sockets created with this module. This shutdown should be able to distinquish between tied and untied sockets and do the right thing.

- debug

  ```
  my $debug = Net::SSLeay::Handle->debug()
  Net::SSLeay::Handle->debug(1)
  ```

Get/set debugging mode. Always returns the debug value before the function call. if an additional argument is given the debug option will be set to this value.

- make_socket

```
my $sock = Net::SSLeay::Handle->make_socket($host, $port);
```

Creates a socket that is connected to $post using $port. It uses $Net::SSLeay::proxyhost and proxyport if set and authentificates itself against this proxy depending on $Net::SSLeay::proxyauth. It also turns autoflush on for the created socket.

USING EXISTING SOCKETS

One of the motivations for writing this module was to avoid duplicating socket creation code (which is mostly error handling). The calls to tie() above where it is passed a $host and $port is provided for convenience testing. If you already have a socket connected to the right host and port, S1, then you can do something like:

```
my $socket \*S1;
if ($scheme eq "https") {
    tie(*S2, "Net::SSLeay::Handle", $socket);
    $socket = \*S2;
}
my $last_sel = select($socket); $| = 1; select($last_sel);
print $socket $request_headers;
...
```

Note: As far as I know you must be careful with the globs in the tie() function. The first parameter must be a glob (*SOMETHING) and the last parameter must be a reference to a glob (*SOMETHING_ELSE) or a scaler that was assigned to a reference to a glob (as in the example above)

Also, the two globs must be different. When I tried to use the same glob, I got a core dump.

EXPORT

None by default.

You can export the shutdown() function.

It is suggested that you do export shutdown() or use the fully qualified Net::SSLeay::Handle::shutdown() function to shutdown SSL sockets. It should be smart enough to distinguish between SSL and non-SSL sockets and do the right thing.

EXAMPLES

```
use Net::SSLeay::Handle qw/shutdown/;
my ($host, $port) = ("localhost", 443);

tie(*SSL, "Net::SSLeay::Handle", $host, $port);

```

(continues on next page)

(continued from previous page)

```
print SSL "GET / HTTP/1.0\r\n";
shutdown(\*SSL, 1);
print while (<SSL>);
close SSL;
```

TODO

Better error handling. Callback routine?

CAVEATS

Tying to a file handle is a little tricky (for me at least).

The first parameter to tie() must be a glob (*SOMETHING) and the last parameter must be a reference to a glob (*SOMETHING_ELSE) or a scaler that was assigned to a reference to a glob ($s = *SOMETHING_ELSE). Also, the two globs must be different. When I tried to use the same glob, I got a core dump.

I was able to associate attributes to globs created by this module (like *SSL above) by making a hash of hashes keyed by the file head1.

CHANGES

Please see Net-SSLeay-Handle-0.50/Changes file.

BUGS

If you encounter a problem with this module that you believe is a bug, please create a new issue in the Net-SSLeay GitHub repository. Please make sure your bug report includes the following information:

- the code you are trying to run;
- your operating system name and version;
- the output of `perl -V`;
- the version of OpenSSL or LibreSSL you are using.

AUTHOR

Originally written by Jim Bowlin.

Maintained by Sampo Kellomäki between July 2001 and August 2003.

Maintained by Florian Ragwitz between November 2005 and January 2010.

Maintained by Mike McCauley between November 2005 and June 2018.

Maintained by Tuure Vartiainen between June 2018 and July 2018.

Maintained by Chris Novakovic and Heikki Vatiainen since June 2018.

COPYRIGHT

Copyright (c) 2001 Jim Bowlin jbowlin@linklint.org

Copyright (c) 2001-2003 Sampo Kellomäki sampo@iki.fi

Copyright (c) 2005-2010 Florian Ragwitz rafl@debian.org

Copyright (c) 2005-2018 Mike McCauley mikem@airspayce.com

Copyright (c) 2018 Tuure Vartiainen vartiait@radiatorsoftware.com

Copyright (c) 2018- Chris Novakovic chris@chrisn.me.uk

Copyright (c) 2018- Heikki Vatiainen hvn@radiatorsoftware.com

All rights reserved.

LICENSE

This module is released under the terms of the Artistic License 2.0. For details, see the `LICENSE` file distributed with Net-SSLeay's source code.

SEE ALSO

Net::SSLeay, perl(1), http://openssl.org/

Net::Time

time and daytime network client interface

SYNOPSIS

```
use Net::Time qw(inet_time inet_daytime);

print inet_time();          # use default host from Net::Config
print inet_time('localhost');
print inet_time('localhost', 'tcp');

print inet_daytime();       # use default host from Net::Config
print inet_daytime('localhost');
print inet_daytime('localhost', 'tcp');
```

DESCRIPTION

`Net::Time` provides subroutines that obtain the time on a remote machine.

Functions

- `inet_time([$host[, $protocol[, $timeout]]])`

  Obtain the time on `$host`, or some default host if `$host` is not given or not defined, using the protocol as defined in RFC868. The optional argument `$protocol` should

define the protocol to use, either `tcp` or udp. The result will be a time value in the same units as returned by time() or *undef* upon failure.

- `inet_daytime([$host[, $protocol[, $timeout]]])`

  Obtain the time on `$host`, or some default host if `$host` is not given or not defined, using the protocol as defined in RFC867. The optional argument `$protocol` should define the protocol to use, either `tcp` or udp. The result will be an ASCII string or *undef* upon failure.

EXPORTS

The following symbols are, or can be, exported by this module:

- Default Exports

  None.

- Optional Exports

  `inet_time`, `inet_daytime`.

- Export Tags

  None.

KNOWN BUGS

None.

AUTHOR

Graham Barr <gbarr@pobox.com>.

Steve Hay <shay@cpan.org> is now maintaining libnet as of version 1.22_02.

COPYRIGHT

Copyright (C) 1995-2004 Graham Barr. All rights reserved.

Copyright (C) 2014, 2020 Steve Hay. All rights reserved.

LICENCE

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself, i.e. under the terms of either the GNU General Public License or the Artistic License, as specified in the `LICENCE` file.

VERSION

Version 3.15

DATE

20 March 2023

HISTORY

See the Changes file.

Net::hostent

by-name interface to Perl's built-in gethost*() functions

SYNOPSIS

```
use Net::hostent;
```

DESCRIPTION

This module's default exports override the core gethostbyname() and gethostbyaddr() functions, replacing them with versions that return "Net::hostent" objects. This object has methods that return the similarly named structure field name from the C's hostent structure from `netdb.h`; namely name, aliases, addrtype, length, and addr_list. The aliases and addr_list methods return array reference, the rest scalars. The addr method is equivalent to the zeroth element in the addr_list array reference.

You may also import all the structure fields directly into your namespace as regular variables using the :FIELDS import tag. (Note that this still overrides your core functions.) Access these fields as variables named with a preceding `h_`. Thus, `$host_obj->name()` corresponds to $h_name if you import the fields. Array references are available as regular array variables, so for example `@{ $host_obj->aliases() }` would be simply @h_aliases.

The gethost() function is a simple front-end that forwards a numeric argument to gethostbyaddr() by way of Socket::inet_aton, and the rest to gethostbyname().

To access this functionality without the core overrides, pass the `use` an empty import list, and then access the functions with their full qualified names. On the other hand, the built-ins are still available via the `CORE::` pseudo-package.

EXAMPLES

```
use Net::hostent;
use Socket;

@ARGV = ('netscape.com') unless @ARGV;

for $host ( @ARGV ) {

   unless ($h = gethost($host)) {
       warn "$0: no such host: $host\n";
```

(continues on next page)

(continued from previous page)

```
        next;
    }

    printf "\n%s is %s%s\n",
           $host,
           lc($h->name) eq lc($host) ? "" : "*really* ",
           $h->name;

    print "\taliases are ", join(", ", @{$h->aliases}), "\n"
                if @{$h->aliases};

    if ( @{$h->addr_list} > 1 ) {
        my $i;
        for $addr ( @{$h->addr_list} ) {
            printf "\taddr #%d is [%s]\n", $i++, inet_ntoa($addr);
        }
    } else {
        printf "\taddress is [%s]\n", inet_ntoa($h->addr);
    }

    if ($h = gethostbyaddr($h->addr)) {
        if (lc($h->name) ne lc($host)) {
            printf "\tThat addr reverses to host %s!\n", $h->name;
            $host = $h->name;
            redo;
        }
    }
}
```

NOTE

While this class is currently implemented using the Class::Struct module to build a struct-like class, you shouldn't rely upon this.

AUTHOR

Tom Christiansen

Net::netent

by-name interface to Perl's built-in getnet*() functions

SYNOPSIS

```
use Net::netent qw(:FIELDS);
getnetbyname("loopback")               or die "bad net";
printf "%s is %08X\n", $n_name, $n_net;
```

(continues on next page)

(continued from previous page)

```
use Net::netent;

$n = getnetbyname("loopback")          or die "bad net";
{ # there's gotta be a better way, eh?
    @bytes = unpack("C4", pack("N", $n->net));
    shift @bytes while @bytes && $bytes[0] == 0;
}
printf "%s is %08X [%d.%d.%d.%d]\n", $n->name, $n->net, @bytes;
```

DESCRIPTION

This module's default exports override the core getnetbyname() and getnetbyaddr() functions, replacing them with versions that return "Net::netent" objects. This object has methods that return the similarly named structure field name from the C's netent structure from `netdb.h`; namely name, aliases, addrtype, and net. The aliases method returns an array reference, the rest scalars.

You may also import all the structure fields directly into your namespace as regular variables using the :FIELDS import tag. (Note that this still overrides your core functions.) Access these fields as variables named with a preceding `n_`. Thus, `$net_obj->name()` corresponds to $n_name if you import the fields. Array references are available as regular array variables, so for example `@{ $net_obj->aliases() }` would be simply @n_aliases.

The getnet() function is a simple front-end that forwards a numeric argument to getnetbyaddr(), and the rest to getnetbyname().

To access this functionality without the core overrides, pass the `use` an empty import list, and then access the functions with their full qualified names. On the other hand, the built-ins are still available via the `CORE::` pseudo-package.

EXAMPLES

The getnet() functions do this in the Perl core:

```
sv_setiv(sv, (I32)nent->n_net);
```

The gethost() functions do this in the Perl core:

```
sv_setpvn(sv, hent->h_addr, len);
```

That means that the address comes back in binary for the host functions, and as a regular perl integer for the net ones. This seems a bug, but here's how to deal with it:

```
use strict;
use Socket;
use Net::netent;

@ARGV = ('loopback') unless @ARGV;
```

(continues on next page)

(continued from previous page)

```
my($n, $net);

for $net ( @ARGV ) {

    unless ($n = getnetbyname($net)) {
       warn "$0: no such net: $net\n";
       next;
    }

    printf "\n%s is %s%s\n",
           $net,
           lc($n->name) eq lc($net) ? "" : "*really* ",
           $n->name;

    print "\taliases are ", join(", ", @{$n->aliases}), "\n"
                if @{$n->aliases};

    # this is stupid; first, why is this not in binary?
    # second, why am i going through these convolutions
    # to make it looks right
    {
       my @a = unpack("C4", pack("N", $n->net));
       shift @a while @a && $a[0] == 0;
       printf "\taddr is %s [%d.%d.%d.%d]\n", $n->net, @a;
    }

    if ($n = getnetbyaddr($n->net)) {
       if (lc($n->name) ne lc($net)) {
           printf "\tThat addr reverses to net %s!\n", $n->name;
           $net = $n->name;
           redo;
       }
    }
}
```

NOTE

While this class is currently implemented using the Class::Struct module to build a struct-like class, you shouldn't rely upon this.

AUTHOR

Tom Christiansen

Net::protoent

by-name interface to Perl's built-in getproto*() functions

SYNOPSIS

```
use Net::protoent;
my $p = getprotobyname(shift || 'tcp') || die "no proto";
printf "proto for %s is %d, aliases are %s\n",
   $p->name, $p->proto, "@{$p->aliases}";

use Net::protoent qw(:FIELDS);
getprotobyname(shift || 'tcp') || die "no proto";
print "proto for $p_name is $p_proto, aliases are @p_aliases\n";
```

DESCRIPTION

This module's default exports override the core getprotoent(), getprotobyname(), and getprotobynumber() functions, replacing them with versions that return "Net::protoent" objects. This object has methods that return the similarly named structure field name from the C's protoent structure from `netdb.h`; namely name, aliases, and proto. The aliases method returns an array reference, the rest scalars.

You may also import all the structure fields directly into your namespace as regular variables using the :FIELDS import tag. (Note that this still overrides your core functions.) Access these fields as variables named with a preceding `p_`. Thus, `$proto_obj->name()` corresponds to $p_name if you import the fields. Array references are available as regular array variables, so for example `@{ $proto_obj->aliases() }` would be simply @p_aliases.

The getproto() function is a simple front-end that forwards a numeric argument to getprotobynumber(), and the rest to getprotobyname().

To access this functionality without the core overrides, pass the `use` an empty import list, and then access the functions with their full qualified names. On the other hand, the built-ins are still available via the `CORE::` pseudo-package.

NOTE

While this class is currently implemented using the Class::Struct module to build a struct-like class, you shouldn't rely upon this.

AUTHOR

Tom Christiansen

Net::servent

by-name interface to Perl's built-in getserv*() functions

SYNOPSIS

```
use Net::servent;
my $s = getservbyname(shift || 'ftp') || die "no service";
printf "port for %s is %s, aliases are %s\n",
   $s->name, $s->port, "@{$s->aliases}";

use Net::servent qw(:FIELDS);
getservbyname(shift || 'ftp') || die "no service";
print "port for $s_name is $s_port, aliases are @s_aliases\n";
```

DESCRIPTION

This module's default exports override the core getservent(), getservbyname(), and getnetbyport() functions, replacing them with versions that return "Net::servent" objects. They take default second arguments of "tcp". This object has methods that return the similarly named structure field name from the C's servent structure from `netdb.h`; namely name, aliases, port, and proto. The aliases method returns an array reference, the rest scalars.

You may also import all the structure fields directly into your namespace as regular variables using the :FIELDS import tag. (Note that this still overrides your core functions.) Access these fields as variables named with a preceding `s_`. Thus, `$serv_obj->name()` corresponds to $s_name if you import the fields. Array references are available as regular array variables, so for example `@{ $serv_obj->aliases()}` would be simply @s_aliases.

The getserv() function is a simple front-end that forwards a numeric argument to getservbyport(), and the rest to getservbyname().

To access this functionality without the core overrides, pass the `use` an empty import list, and then access the functions with their full qualified names. On the other hand, the built-ins are still available via the `CORE::` pseudo-package.

EXAMPLES

```
use Net::servent qw(:FIELDS);

while (@ARGV) {
    my ($service, $proto) = ((split m!/!, shift), 'tcp');
    my $valet = getserv($service, $proto);
    unless ($valet) {
        warn "$0: No service: $service/$proto\n"
        next;
    }
    printf "service $service/$proto is port %d\n", $valet->port;
    print "alias are @s_aliases\n" if @s_aliases;
}
```

NOTE

While this class is currently implemented using the Class::Struct module to build a struct-like class, you shouldn't rely upon this.

AUTHOR

Tom Christiansen

5.2.72 O

Generic interface to Perl Compiler backends

SYNOPSIS

```
perl -MO=[-q,]Backend[,OPTIONS] foo.pl
```

DESCRIPTION

This is the module that is used as a frontend to the Perl Compiler.

If you pass the `-q` option to the module, then the STDOUT filehandle will be redirected into the variable `$O::BEGIN_output` during compilation. This has the effect that any output printed to STDOUT by BEGIN blocks or use'd modules will be stored in this variable rather than printed. It's useful with those backends which produce output themselves (`Deparse`, `Concise` etc), so that their output is not confused with that generated by the code being compiled.

The `-qq` option behaves like `-q`, except that it also closes STDERR after deparsing has finished. This suppresses the "Syntax OK" message normally produced by perl.

CONVENTIONS

Most compiler backends use the following conventions: OPTIONS consists of a comma-separated list of words (no white-space). The `-v` option usually puts the backend into verbose mode. The `-ofile` option generates output to **file** instead of stdout. The `-D` option followed by various letters turns on various internal debugging flags. See the documentation for the desired backend (named `B::Backend` for the example above) to find out about that backend.

IMPLEMENTATION

This section is only necessary for those who want to write a compiler backend module that can be used via this module.

The command-line mentioned in the SYNOPSIS section corresponds to the Perl code

```
use O ("Backend", OPTIONS);
```

The `O::import` function loads the appropriate `B::Backend` module and calls its `compile` function, passing it OPTIONS. That function is expected to return a sub reference which we'll call CALLBACK. Next, the "compile-only" flag is switched on (equivalent to

the command-line option `-c`) and a CHECK block is registered which calls CALLBACK. Thus the main Perl program mentioned on the command-line is read in, parsed and compiled into internal syntax tree form. Since the `-c` flag is set, the program does not start running (excepting BEGIN blocks of course) but the CALLBACK function registered by the compiler backend is called.

In summary, a compiler backend module should be called "B::Foo" for some foo and live in the appropriate directory for that name. It should define a function called `compile`. When the user types

```
perl -MO=Foo,OPTIONS foo.pl
```

that function is called and is passed those OPTIONS (split on commas). It should return a sub ref to the main compilation function. After the user's program is loaded and parsed, that returned sub ref is invoked which can then go ahead and do the compilation, usually by making use of the `B` module's functionality.

BUGS

The `-q` and `-qq` options don't work correctly if perl isn't compiled with PerlIO support : STDOUT will be closed instead of being redirected to `$O::BEGIN_output`.

AUTHOR

Malcolm Beattie, `mbeattie@sable.ox.ac.uk`

5.2.73 ODBM_File

Tie a Perl hash to an old-style dbm database file.

ODBM_File is the tied-hash interface to the original Unix dbm: one database per process, keys and values of bytes. Bind a hash with `tie`; `$h{key}`, `delete`, `keys` and `each` then read and write the file. It has no `exists`.

Synopsis

```
use Fcntl;
use ODBM_File;

tie my %h, 'ODBM_File', 'mydb', O_RDWR|O_CREAT, 0640
    or die "tie failed: $!";
$h{greeting} = 'hello';
untie %h;
```

Opening a database

```
tie %h, 'ODBM_File', $basename, $flags, $mode;
```

With `O_CREAT` in `$flags` the files are created if missing, with the permission `$mode`; without it they must exist, or `tie` croaks `ODBM_FILE: Can't open NAME`. Only one database can be tied at a time in a process: a second `tie` croaks `Old dbm can only open one database` until the first is untied.

DBM filters

`(tied %h)->filter_store_key(sub {...})`, `filter_store_value`, `filter_fetch_key` and `filter_fetch_value` install a sub that rewrites every key or value on its way into, or out of, the database. The sub changes `$_` in place; a store filter works on a copy, so your own variable keeps its value. Each method returns the filter it replaces (or undef), and `undef` as the argument removes the filter. A filter that uses the same database croaks `recursion detected in filter_store_key` (or the filter's own name).

Files

pperl keeps the data in an LMDB file, `NAME.pag`, with `NAME.dir` beside it, as old dbm has the two, and LMDB's lock file `NAME.pag-lock`. The dbm library is not used, and its files cannot be read: tying a real dbm (gdbm) or Berkeley DB file croaks, naming what the file is, and the file is left as it was. A key may be at most 511 bytes, 510 if it starts with a NUL byte; a longer one croaks.

Writes reach the file when the hash is untied: until then the tie holds the file's one writer slot, and another process that writes the file waits for it (readers never wait). Set `$Peta::FFI::LMDB::DBM_WRITE = 'through'` before `tie` to commit every write as it happens; see `Peta::FFI::LMDB`.

Differences from upstream

- The files are LMDB files: real dbm and Berkeley DB files are refused, and pperl's files cannot be read by perl 5's ODBM_File. Covered by `t/82-native/Peta-FFI-LMDB/080-odbm.t`.
- Keys are limited to 511 bytes. Covered by `t/82-native/Peta-FFI-LMDB/080-odbm.t`.
- Writes reach the file at `untie` (or each one, with `$Peta::FFI::LMDB::DBM_WRITE = 'through'`), and a writer holds the file until then. Covered by `t/82-native/Peta-FFI-LMDB/080-odbm.t`.
- `tie` with `O_CREAT` opens a database that already exists, where perl 5 croaks `ODBM_File: Can't create NAME`. Covered by `t/82-native/Peta-FFI-LMDB/080-odbm.t`.
- A `tie` that fails does not keep later ones from opening a database; in perl 5 every later `tie` in the process croaks `Old dbm can only open one database`. Covered by `t/82-native/Peta-FFI-LMDB/080-odbm.t`.

Everything else behaves as with perl 5's ODBM_File. Covered by its own test, `t/81-xs-native/ODBM_File/100-upstream-odbm.t`.

5.2.74 Opcode

Name, group and mask the interpreter's opcodes - the machinery `Safe` compartments are built from.

An *opset* is a bit string with one bit per opcode, `(MAXO + 7) / 8` bytes wide. Operators are named individually (`"print"`, `"backtick"`) or by tag (`":base_core"`, `":subpro-`

cess"); Opcode converts between names, tags and opsets, and installs an opset into PL_op_mask, which the compiler consults to refuse masked ops.

use Opcode gives you the whole interface: the opset constructors and conversions (opset, ops_to_opset, opset_to_ops, opset_to_hex, invert_opset, empty_opset, full_opset, verify_opset), the operator queries (opdesc, opcodes, opdump), the tag builder define_optag, and the mask accessors opmask and opmask_add. Nothing is exported by default - name what you want on the use line. The predefined tags (:base_core, :default, :subprocess and the rest) are defined when the module loads.

```
use Opcode qw(opset opset_to_ops opdesc invert_opset opcodes);
use Safe;

my $io = opset(qw(:filesys_open print));
my @names = opset_to_ops($io);
print scalar(@names), " ops: @names[0..2] ...\n";   # 8 ops: open␣
↪close umask ...
print join(", ", opdesc(qw(print backtick))), "\n";  # print, quoted␣
↪execution (``, qx)
printf "%d opcodes, the rest of the set: %d\n",
    scalar(opcodes()), scalar(() = opset_to_ops(invert_opset($io)));␣
↪  # 429, 421

my $c = Safe->new;
Opcode::deny($c, 'print');
print $c->reval('1 + 2'), "\n";                      # 3
print defined $c->reval('print "x"') ? "printed\n" : "refused: $@";
# refused: 'print' trapped by operation mask at (eval 9) line 1.
```

The opcodes are perl 5.44's, so opsets, tags and masks mean what they mean there, and a Safe compartment refuses the same code.

This is a native implementation of the XS half of Opcode 1.71; its Opcode.pm, which defines the tags and opset_to_hex, opdump and ops_to_opset, is used unchanged.

Differences from upstream

None.

Functions

Opsets

verify_opset

Check that a scalar really is an opset, optionally croaking if it is not.

invert_opset

Return the complement of an opset - every operator it selects deselected, and vice versa.

opset_to_ops

List the operators an opset selects, by name or by description.

opset

Build an opset from operator names, `:tag` names and other opsets.

empty_opset

Return an opset with no operators selected.

full_opset

Return an opset with every operator selected.

Operators and tags

opdesc

Return the human-readable description of each named operator.

define_optag

Give a name to an opset so it can be used as a `:tag` afterwards.

opcodes

Return the number of operators the interpreter knows about.

Masking

opmask_add

Mask the operators an opset selects, on top of whatever is already masked. The compiler refuses a masked operator.

opmask

Return the currently masked operators as an opset.

Safe compartments

permit_only

`Opcode::permit_only($safe, @ops)`, `Opcode::permit($safe, @ops)`, `Opcode::deny_only($safe, @ops)`, `Opcode::deny($safe, @ops)`: change the

operator mask of a compartment object - any blessed hash, such as a `Safe`, whose `Mask` entry is an opset (created when missing). The operators are given as names, `:tags` and opsets; a name with a leading `!` counts inverted.

Also available as `Opcode::permit`, `Opcode::deny_only`, `Opcode::deny`.

permit_only

`Opcode::permit_only($safe, @ops)`, `Opcode::permit($safe, @ops)`, `Opcode::deny_only($safe, @ops)`, `Opcode::deny($safe, @ops)`: change the operator mask of a compartment object - any blessed hash, such as a `Safe`, whose `Mask` entry is an opset (created when missing). The operators are given as names, `:tags` and opsets; a name with a leading `!` counts inverted.

Also available as `Opcode::permit`, `Opcode::deny_only`, `Opcode::deny`.

- `permit` allows the named operators, deny forbids them, both on top of the current mask;
- `permit_only` allows the named operators and forbids every other; `deny_only` forbids the named ones and allows every other.

```
my $c = Safe->new;
Opcode::permit_only($c, ':default');
Opcode::deny($c, 'print');
$c->reval('print "x"');     # 'print' trapped by operation mask
```

`Safe`'s methods of the same names are its own Perl code over `opset`; these are the older XS versions. Anything but a blessed hash dies "Not a Safe object".

5.2.75 PDL

PDL - the Perl Data Language: fast, vectorised, multidimensional arrays for scientific and bulk numeric work.

```
use PDL;
my $x = sequence(3, 2);                          # 3 columns, 2 rows:
↪0 .. 5
print +($x * 2 + 1)->flat, "\n";                  # [1 3 5 7 9 11]
print $x->sumover, " ", $x->max, " ", $x->type, "\n";    # [3 12] 5
↪double
my $col = $x->slice("(1),:");                    # a view of column 1
$col .= -1;
print $x->flat, " ", $x->where($x > 3), "\n";             # [0 -1 2 3
↪-1 5] [5]
print float(1.5) + byte(2), " ", (float(1.5) + byte(2))->type, "\n";
↪  # 3.5 float
```

An ndarray is a blessed scalar holding a dense, typed, multidimensional block of numbers. Operators and functions apply to the whole block at once - `$x * 2 + 1` works on every element without a Perl loop - and slices such as `$x->slice(...)` are live views onto the parent's data rather than copies, so writing through a view writes to the parent. Functions broadcast: an operation on one dimension runs over all the others, and operands of different shapes are matched dimension by dimension.

The element type is chosen per ndarray. The constructors named after each type - `sbyte`, `byte`, `short`, `ushort`, `long`, `ulong`, `indx`, `longlong`, `ulonglong`, `float`, `double`, `ldouble`, `cfloat`, `cdouble`, `cldouble` - both build an ndarray of that type and, called with no arguments, name the type itself.

`use PDL` loads and imports PDL::Core (constructors, dimensions, types), PDL::Ops (the operators), PDL::Primitive (products, selecting, searching, interpolation), PDL::Ufunc (reductions and sorting), PDL::Basic (sequences and coordinates), PDL::Slices (views and indexing), PDL::Bad (bad values), PDL::MatrixOps (linear algebra), PDL::Math (mathematical functions), and the IO modules: PDL::IO::Misc (columns of text, byte swapping), PDL::IO::FITS, PDL::IO::Pic (images, through PDL::IO::Pnm and PDL::ImageRGB) and PDL::IO::Storable. PDL::Lite and PDL::LiteF load fewer; PDL::FFT, PDL::Image2D, PDL::ImageND, PDL::Transform and PDL::Compression are loaded on their own.

This is PDL 2.106: its Perl is upstream's, unchanged, and its compiled kernels are PDL's own C, run by pperl's engine. The XS that binds them is pperl's. `use PDL` needs pperl built with PDL (the `pdl` feature).

As upstream, an operation on large ndarrays (by default from 1 Mi elements) is split across threads: `set_autopthread_targ` and `set_autopthread_size` in PDL::Core control it. The threads are pperl's own pool, sized by `--threads`; `--no-parallel` keeps PDL on one thread.

Differences from upstream

Where perl5 with PDL fails, pperl does better:

- `broadcast` with a dimension the ndarray does not have croaks "Error in broadcastI:Invalid dim N (ndarray has M dims)" when first computed. Upstream reads outside the dimension array: memory before it for a negative dim (a garbage result), and for `broadcast(99)` it runs out of memory (t/81-xs-native/PDL/380-broadcast-methods.t).

Not there yet:

- `isnan` (PDL::Bad) and `casinh` (PDL::Math), both new in PDL 2.105.

t/81-xs-native/PDL/838-open-differences.t holds one case for every difference from upstream, open or closed; a new one goes there first.

Modules

- `PDL::Bad` - Bad values: PDL's marker for missing data, and the functions that find, make and replace them.
- `PDL::Basic` - PDL's basic constructors: counting sequences, coordinate values along a dimension, evenly spaced ranges, distances from a centre and histograms.
- `PDL::Complex`
- `PDL::Compression` - Rice compression of integer ndarrays, as FITS files use it.
- `PDL::Constants` - basic compile time constants for PDL
- `PDL::Core` - Build ndarrays, ask them about themselves, and convert between types.
- `PDL::Exporter` - PDL export control

- PDL::FFT - Fast Fourier transforms of PDL's ndarrays, in place, in one or more dimensions.
- PDL::IO
- PDL::Image2D - Two-dimensional image processing: median filters, polygon filling, connected components, rotation and warping.
- PDL::ImageND - Image processing in any number of dimensions: convolution, rebinning, interpolation, circular masks, contours and graph layout forces.
- PDL::ImageRGB - Utility functions for RGB image data: palette quantisation, palette lookup, conversion to grey, and scaling to bytes.
- PDL::Lite - PDL with methods only: the core modules loaded, four names imported.
- PDL::LiteF - The core PDL modules, with their functions imported.
- PDL::Math - PDL's mathematical functions: trigonometric and hyperbolic, rounding, error and Bessel functions, log gamma, complex square roots, logarithms and inverses, and polynomials.
- PDL::MatrixOps - Linear algebra on PDL's ndarrays: determinants, inverses, LU, singular value and eigen decompositions, linear systems, triangular storage.
- PDL::Ops - PDL's operators: arithmetic, comparison, bitwise and the unary functions, element by element, as the functions behind Perl's operators on ndarrays.
- PDL::Options - simplifies option passing by hash in PerlDL
- PDL::PP
- PDL::Primitive - PDL's primitives: products, selecting by a mask, clipping, searching sorted data, interpolation, set operations on vectors, statistics.
- PDL::Slices - Take pieces of an ndarray, and rearrange the pieces you take.
- PDL::Trans
- PDL::Transform - Coordinate transformations as objects, and the resampling of images through them.
- PDL::Types - define fundamental PDL Datatypes
- PDL::Ufunc - Reductions of PDL's ndarrays: sums, products, means, extrema, medians, percentiles, logical and bitwise folds, running sums and products, and sorting.

Functions

Contours and paths

contour_segments

`($segs, $cnt) = contour_segments($c, $data, $points)` - the line segments of the contour of `$data` at the level `$c`, in the coordinates `$points` gives each data point; `$cnt` is the index of the last segment end in `$segs`.

contour_polylines

`($pathendindex, $paths) = contour_polylines($c, $data, $points)` - the contour of `$data` at the level `$c` as polylines: `$paths` holds the points and `$pathendindex` the index of each polyline's last point, -1 after the last.

path_join

`($pathendindex, $paths) = path_join($edges, $nvertices, $directed)` - the edges `$edges` (pairs of vertex numbers) joined into paths: `$paths` holds the vertices and `$pathendindex` the index of each path's last vertex. `$directed` is true by default.

Graph layout

combcoords

`combcoords($x, $y, $z)` - the three coordinates packed into one float ndarray with a leading dimension of 3.

repulse

`repulse($coords, $boxsize, $dmult, $a, $b, $c, $d)` - the repulsive force on each point of `$coords` from the points near it, for a spring-embedded graph layout. The points are sorted into boxes of side `$boxsize`, `$dmult` of them along each of three dimensions, and only points in neighbouring boxes interact. With `s = 1 / (r + $d)` for two points at distance `r`, each pushes the other along their difference vector, scaled by `$c * sqrt(s) + $b * s + $a * s**2`.

attract

`attract($coords, $from, $to, $strength, $m, $ms)` - the attractive force on each point of `$coords` from the links `$from` to `$to` of the given `$strength`, for a spring-embedded graph layout.

First-dimension reductions

sumover

`sumover($a)` - the sum along the first dimension.

prodover

`prodover($a)` - the product along the first dimension.

dprodover

`dprodover($a)` - the product along the first dimension, computed in double precision.

dsumover

`dsumover($a)` - the sum along the first dimension, computed in double precision.

average

`average($a)` - the mean along the first dimension.

daverage

`daverage($a)` - the mean along the first dimension, computed in double precision.

caverage

`caverage($a)` - the mean along the first dimension as a complex value (`cfloat` for `float` input, `cdouble` otherwise); `cavgover` is the same function.

intover

`intover($a)` - the integral along the first dimension, for a point spacing of one: scale the result by the real spacing.

magnover

`magnover($a)` - the Euclidean length along the first dimension, `sqrt(sumover($a ** 2))`.

minimum

`minimum($a)` - the smallest value along the first dimension.

minimum_ind

`minimum_ind($a)` - the position of the smallest value along the first dimension.

maximum

`maximum($a)` - the largest value along the first dimension.

maximum_ind

`maximum_ind($a)` - the position of the largest value along the first dimension.

medover

`medover($a)` - the median along the first dimension; for an even count, the mean of the two middle values.

oddmedover

`oddmedover($a)` - the median along the first dimension; for an even count, the lower of the two middle values.

modeover

`modeover($data)` - the most frequent value along the first dimension. Meant for integer types: floating-point values are demoted to integers first.

minmaximum

`minmaximum($x)` - `($min, $max, $min_ind, $max_ind)` along the first dimension, broadcasting over the rest.

pctover

`pctover($pdl, $percentile)` - the given percentile along the first dimension, broadcasting over the others. `$percentile` is between 0.0 and 1.0; values outside are clipped to that range. Between data points the result is interpolated, by the variant Microsoft Excel uses and NIST recommends. Bad values are skipped.

oddpctover

`oddpctover($pdl, $percentile)` - like `pctover`, but when the percentile falls between data points the result is the nearest data value rather than an interpolation.

Logical and bitwise reductions

andover

`andover($a)` - 1 where every value along the first dimension is true, else 0.

orover

`orover($a)` - 1 where any value along the first dimension is true, else 0.

xorover

`xorover($a)` - the logical exclusive or along the first dimension.

zcover

`zcover($a)` - 1 where every value along the first dimension is zero, else 0.

bandover

`bandover($a)` - the bitwise and along the first dimension (integer types).

borover

`borover($a)` - the bitwise or along the first dimension (integer types).

bxorover

`bxorover($a)` - the bitwise exclusive or along the first dimension (integer types).

firstnonzeroover

`firstnonzeroover($a)` - the first non-zero value along the first dimension, or 0.

zcheck

`$pdl->zcheck` - 1 if every value is zero, else 0.

and

`$pdl->and` - 1 if every value is true, else 0.

or

`$pdl->or` - 1 if any value is true, else 0.

xorall

`$pdl->xorall` - the logical exclusive or of all the values.

band

`$pdl->band` - the bitwise and of all the values.

bor

`$pdl->bor` - the bitwise or of all the values.

bxor

`$pdl->bxor` - the bitwise exclusive or of all the values.

Cumulative operations

cumusumover

`cumusumover($a)` - the running sum along the first dimension.

dcumusumover

`dcumusumover($a)` - the running sum along the first dimension, in double precision.

cumuprodover

`cumuprodover($a)` - the running product along the first dimension.

dcumuprodover

`dcumuprodover($a)` - the running product along the first dimension, in double precision.

numdiff

`numdiff($x)` - the differences of neighbouring values along the first dimension, the first value kept: the inverse of `cumusumover`. Works in place.

diffcentred

`diffcentred($a)` - the centred difference along the first dimension, `($a[i+1] - $a[i-1]) / 2`, wrapping around at the ends. `partial($f, $dim)` takes it along dimension `$dim`; with `{ dir => 'f' }` or `'b'` it takes the forward or backward difference instead.

diff2

`diff2($a)` - the differences of neighbouring values along the first dimension, `$a[i+1] - $a[i]`: one element shorter than `$a`.

Sorting

qsort

`qsort($a)` - the values sorted along the first dimension, bad values last. Works in place.

qsorti

`qsorti($a)` - the indices that sort the values along the first dimension.

qsortvec

`qsortvec($a)` - a list of vectors sorted lexicographically: each vector runs along the first dimension, the list along the second. Works in place.

qsortveci

`qsortveci($a)` - the indices that would sort a list of vectors lexicographically, as `qsortvec` does.

Whole-array reductions

avg

`$pdl->avg` - the mean of all the values, as a 0-dim ndarray.

sum

`$pdl->sum` - the sum of all the values.

prod

`$pdl->prod` - the product of all the values.

davg

`$pdl->davg` - the mean of all the values, in double precision.

dsum

`$pdl->dsum` - the sum of all the values, in double precision.

dprod

`$pdl->dprod` - the product of all the values, in double precision.

min

`$pdl->min` - the smallest of all the values.

max

`$pdl->max` - the largest of all the values.

median

`$pdl->median` - the median of all the values.

oddmedian

`$pdl->oddmedian` - the median of all the values; for an even count, the lower middle value.

mode

`$pdl->mode` - the most frequent of all the values.

pct

`pct($pdl, $p)` - the given percentile of all the values of `$pdl`, whatever its dimensions, as a 0-dimensional PDL. Interpolated between data points, as `pctover`; bad values are skipped.

oddpct

`oddpct($pdl, $p)` - like `pct`, but when the percentile falls between data points the result is the nearest data value.

Making values bad

setinftobad

`$pdl->setinftobad` - infinities made bad values; works in place.

setnonfinitetobad

`$pdl->setnonfinitetobad` - infinities and NaNs made bad values; works in place.

setnantobad

`$pdl->setnantobad` - NaNs made bad values; works in place.

setbadif

`setbadif($a, $mask)` - `$a` with the elements bad where `$mask` is true.

copybad

`copybad($a, $mask)` - `$a` with the elements bad where `$mask` is bad; works in place.

setvaltobad

`setvaltobad($pdl, $val)` - elements equal to `$val` made bad; works in place.

Finding bad values

isbad

`isbad($a)` - 1 where the element is bad, 0 elsewhere.

isgood

`isgood($a)` - 1 where the element is good, 0 elsewhere.

nbadover

nbadover($a) - the number of bad elements along the first dimension.

ngoodover

ngoodover($a) - the number of good elements along the first dimension.

Replacing bad values

setbadtonan

$pdl->setbadtonan - bad values made NaN; works in place.

locf

locf($a) - each bad value replaced by the last good one before it.

setbadtoval

setbadtoval($pdl, $val) - bad values replaced with $val; works in place.

badmask

badmask($a, $b) - $a with each bad value replaced by $b's; works in place.

Indexing

index2d

index2d($a, $inda, $indb) - the elements of the 2-D $a at the coordinate pairs ($inda, $indb), as a child linked both ways.

index1d

index1d($a, $ind): the elements of $a along dim 0 at each of the indices in $ind, as a child linked both ways.

rangeb

$source->rangeb($index, $size, $boundary) - the kernel under range: the elements, or with $size the rectangular blocks, of $source at the coordinates in $index (its first dimension runs over the dimensions of $source), as a child linked both ways. $boundary says what lies outside $source: forbid (the default), truncate, extend, periodic or mirror, one for all dimensions or one each. range is the same with optional arguments, and indexND is range without a size.

Run-length encoding

rlevec

`($counts, $rows) = rlevec($c)` - run-length encode the rows of `$c`: how often each row repeats, and the rows that start each run.

rleseq

`($counts, $starts) = rleseq($c)` - run-length encode `$c` as runs of consecutive values: the length of each run and the value it starts at.

Views

rotate

`rotate($pdl, $shift)`

make_physdims

Force the ndarray's dimensions to be worked out, and return it.

xchg

`$pdl->xchg($n1, $n2)` - a view with dimensions `$n1` and `$n2` exchanged.

mv

`$pdl->mv($n1, $n2)` - a view with dimension `$n1` moved to position `$n2`, the others shifted.

lags

`$pdl->lags($nthdim, $step, $nlags)` - a view of lagged copies of one dimension, one per lag.

splitdim

`$pdl->splitdim($nthdim, $nsp)` - a view with dimension `$nthdim` split into two, the first of size `$nsp`.

unbroadcast

`$pdl->unbroadcast($atind)` - a view with the broadcast dimensions moved back to position `$atind`.

broadcastI

`$pdl->broadcastI($id, [@dims])` - a view with `@dims` made broadcast dimensions of group `$id`.

Decompositions and solving

svd

`($u, $s, $v) = svd($a)` - singular value decomposition `a = u * diag(s) * transpose(v)`; `svd($a, 1)` for the full decomposition.

simq

`($x, $ips) = simq($a, $b, $flag)` - solve the linear system `a x = b` by LU decomposition; `$a` is replaced by its decomposition.

Triangular storage

squaretotri

Convert the lower triangle of a square matrix to triangular vector storage: `squaretotri($a)` or `squaretotri($a, $b)`.

tritosquare

Convert triangular vector storage to the lower triangle of a square matrix; the upper half of the output is left alone.

tricpy

`tricpy($A, $uplo = 0, $C)`: copy a triangular part of `$A` - uplo -1 strict upper, 0 upper, 1 lower, 2 strict lower.

Building matrices

mstack

Stack two matrices along dim 1: `x(n,m)` and `y(n,p)` give `out(n,m+p)`.

augment

Join two ndarrays along dim 0: `x(n)` and `y(p)` give `out(n+p)`.

Complex numbers

re

`$z->re` - the real part of each complex element; assigning to it writes back into `$z`.

im

`$z->im` - the imaginary part of each complex element; assigning to it writes back into `$z`.

carg

`carg($z)` - the polar angle of each complex element, real.

conj

`conj($z)` - the complex conjugate; `$z->inplace->conj` works.

czip

`czip($re, $im)` - a complex ndarray from its real and imaginary parts.

Arithmetic

plus

`plus($a, $b)` - `$a + $b`, element by element.

minus

`minus($a, $b)` - `$a - $b`.

mult

`mult($a, $b)` - `$a * $b`.

divide

`divide($a, $b)` - `$a / $b`.

power

`power($a, $b)` - `$a ** $b`.

modulo

`modulo($a, $b)` - `$a % $b`, with the sign of `$b` as Perl's `%`.

atan2

`atan2($a, $b)` - the angle of the point `($b, $a)`, element by element.

ipow

`ipow($a, $b)` - `$a` to the integer power `$b`, by repeated multiplication.

pow

`pow($a, $b)` - `$a` to the power `$b`, element by element, the result in the input type (`pow(byte(2), 3)` is a byte); `$a->inplace->pow($b)` works in place.

Comparison

eq

`eq($a, $b)` - `$a == $b`: 1 or 0 per element.

ne

`ne($a, $b)` - `$a != $b`.

gt

`gt($a, $b)` - `$a > $b`.

lt

`lt($a, $b)` - `$a < $b`.

ge

`ge($a, $b)` - `$a >= $b`.

le

`le($a, $b)` - `$a <= $b`.

spaceship

`spaceship($a, $b)` - `$a <=> $b`: -1, 0 or 1 per element.

in

`in($a, $b)` - for each element of `$a`, whether it occurs in `$b`.

approx_artol

`approx_artol($got, $expected, $atol, $rtol)` - 1 where `$got` is within `$atol` of `$expected`, or within `$rtol` relative to it, else 0 (`$atol` 1e-6 and `$rtol` 0 by default). Two NaNs, or two bad values, are approximately equal.

Bitwise

and2

`and2($a, $b)` - `$a & $b`, bitwise.

or2

`or2($a, $b)` - `$a | $b`, bitwise.

xor

`xor($a, $b)` - `$a ^ $b`, bitwise.

shiftleft

`shiftleft($a, $b)` - `$a << $b`.

shiftright

`shiftright($a, $b)` - `$a >> $b`.

bitnot

`bitnot($a)` - bitwise negation.

Unary functions

sqrt

`sqrt($a)` - the element-wise square root.

abs2

`abs2($a)` - the squared absolute value, real.

not

`not($a)` - logical negation: 1 where `$a` is 0, else 0.

sin

`sin($a)`, element-wise.

cos

`cos($a)`, element-wise.

exp

`exp($a)`, element-wise.

log

`log($a)`, element-wise natural logarithm.

Assignment

assgn

`assgn($a, $b)` - copies `$a` into `$b`, broadcasting, and returns `$b` when it was not given: the operation behind `.=`. Into a view the copy reaches the parent, as every write through a view does.

Miscellaneous

hclip

`hclip($a, $b)` - `$a` with every value above `$b` replaced by `$b`: `$b` is the upper bound. Works in place.

lclip

`lclip($a, $b)` - `$a` with every value below `$b` replaced by `$b`: `$b` is the lower bound. Works in place.

axisvalues

`$pdl->inplace->axisvalues` - each element set to its index along the first dimension; broadcasts over the rest.

indadd

`indadd($input, $ind, $sum)` – broadcasting index-add.

broadcastids

`$pdl->broadcastids` - where each broadcast group starts among the dimensions, in list context; how many ids there are, in scalar context.

flags

`$pdl->flags` - the names of the state flags set on the ndarray, such as `PDL_ALLOCATED` or `PDL_PARENTDIMSCHANGED`. For debugging.

dump

`$pdl->dump` - print the engine's view of the ndarray to standard output: its state flags, dimensions, data and the transformations attached to it. For debugging; `PDL::Core::pdump` returns much the same as a string.

broadcastover_n

`PDL::broadcastover_n($code, $pdl1, ...)` - call `$code` once per element of the ndarrays broadcast against each other. Each call gets the number of the outermost dimension that just changed, then the current element of each ndarray as a plain number.

broadcastover

`PDL::broadcastover($code, \@realdims, \@creating, $nothers, $pdl1, ..., @others)` - call `$code` once per broadcast step, with a view of each ndarray that has the dimensions named in `\@realdims`, followed by the `$nothers` trailing arguments. An ndarray whose `\@creating` flag is set is created to fit; its dimensions follow the flags in `\@creating`. This is what `broadcast_define` builds its subs on.

Vectors

eqvec

`eqvec($a, $b)` - whether two vectors are equal.

enumvec

`enumvec($v)` - number each vector by its position within its run of equal vectors.

enumvecg

`enumvecg($v)` - number each vector by the group of equal vectors it belongs to.

vsearchvec

`vsearchvec($find, $which)` - binary search of vectors in a sorted list of vectors.

vcos

`vcos($a, $b)` - the cosine of the angle between each vector of `$a` (its rows along the second dimension) and the vector `$b`.

cmpvec

`cmpvec($a, $b)` - lexicographic comparison of two vectors: -1, 0 or 1.

Searching

vsearch_sample

`vsearch_sample($vals, $x)` - where each value falls in the sorted `$x`, as `vsearch` with `mode => 'sample'`.

vsearch_insert_leftmost

`vsearch_insert_leftmost($vals, $x)` - the leftmost insertion point of each value in the sorted `$x`.

vsearch_insert_rightmost

`vsearch_insert_rightmost($vals, $x)` - the rightmost insertion point of each value in the sorted `$x`.

vsearch_match

`vsearch_match($vals, $x)` - the index of each value in the sorted `$x`, or a negative insertion point where it is missing.

vsearch_bin_inclusive

`vsearch_bin_inclusive($vals, $x)` - the bin of each value, bins closed at their lower edge.

vsearch_bin_exclusive

`vsearch_bin_exclusive($vals, $x)` - the bin of each value, bins open at their lower edge.

Interpolation

pchip_chim

`pchip_chim($x, $f)` - derivatives for a monotone piecewise cubic Hermite interpolant.

pchip_chsp

`pchip_chsp($ic, $vc, $x, $f)` - derivatives for a cubic spline interpolant.

pchip_chfd

`pchip_chfd($x, $f, $d, $xe)` - the Hermite interpolant and its derivative at the points `$xe`.

pchip_chfe

`pchip_chfe($x, $f, $d, $xe)` - the Hermite interpolant at the points `$xe`.

pchip_chia

`pchip_chia($x, $f, $d, $la, $lb)` - the integral of the Hermite interpolant between two abscissae.

pchip_chid

`pchip_chid($x, $f, $d, $ia, $ib)` - the integral of the Hermite interpolant between two data points.

pchip_chbs

`pchip_chbs($x, $f, $d, $knotyp)` - the B-spline representation of the Hermite interpolant.

pchip_bvalu

`pchip_bvalu($t, $a, $ideriv, $x)` - a B-spline or one of its derivatives at `$x`.

pchip_chic

`($d, $ierr) = pchip_chic($ic, $vc, $mflag, $x, $f)` - the derivatives for a monotone piecewise cubic Hermite interpolant, with control over the end conditions (SLATEC PCHIC).

interpolate

`($yi, $err) = interpolate($xi, $x, $y)` - linear interpolation of `$y` over `$x` at the positions `$xi`. `$x` must be monotonic. A position outside `$x` is extrapolated from the two nearest knots, and its `$err` is 1.

Products

inner

`inner($a, $b)` – inner (dot) product along the first dimension.

outer

`outer($a, $b)` - the outer product: `c(n,m) = a(n) * b(m)`.

norm

`norm($vec)` - each vector along the first dimension scaled to unit length. Works in place.

inner2

`inner2($a, $b, $c)` - the inner product of a vector, a matrix and a vector: `$a x $b x $c`.

inner2d

`inner2d($a, $b)` - the inner product over the first two dimensions: the sum of `$a * $b` over a 2-D slice.

inner2t

`inner2t($a, $b, $c)` - the matrix product `$a x $b x $c` for matrices, through a temporary.

innerwt

`innerwt($a, $b, $c)` - the weighted inner product: the sum of `$a * $b * $c` along the first dimension.

crossp

`crossp($a, $b)` - the cross product of two 3-vectors; broadcasts over further dimensions.

Selecting

whichover

`whichover($mask)` - along the first dimension, the positions of the true elements first and -1 for the rest, broadcasting over the other dimensions. Works in place.

index

`index($a, $ind)`: the elements of `$a` along dim 0 at `$ind`, broadcasting over the rest, as a child linked both ways - a write into it reaches `$a`.

Complex functions

csqrt

`csqrt($z)` - the square root, taking real input as complex.

clog

`clog($z)` - the natural logarithm, taking real input as complex.

cacos

`cacos($z)` - the inverse cosine, taking real input as complex.

casin

`casin($z)` - the inverse sine, taking real input as complex.

cacosh

`cacosh($z)` - the inverse hyperbolic cosine, taking real input as complex.

catanh

`catanh($z)` - the inverse hyperbolic tangent, taking real input as complex.

csqrt_up

`csqrt_up($z)` - the square root with its imaginary part non-negative.

Classification

isfinite

`isfinite($pdl)` - 1 where the element is finite (and good), 0 elsewhere, as an int ndarray.

Trigonometric and hyperbolic

asin

`asin($a)` - the arc sine, in radians.

acos

`acos($a)` - the arc cosine, in radians.

atan

`atan($a)` - the arc tangent, in radians.

tan

`tan($a)` - the tangent.

sinh

`sinh($a)` - the hyperbolic sine.

cosh

`cosh($a)` - the hyperbolic cosine.

tanh

`tanh($a)` - the hyperbolic tangent.

asinh

`asinh($a)` - the inverse hyperbolic sine.

acosh

`acosh($a)` - the inverse hyperbolic cosine.

atanh

`atanh($a)` - the inverse hyperbolic tangent.

Rounding

ceil

`ceil($a)` - rounded up to an integer value.

floor

`floor($a)` - rounded down to an integer value.

rint

`rint($a)` - rounded to the nearest integer value, halves to even (C99 `rint`).

Special functions

erf

`erf($a)` - the error function.

erfc

`erfc($a)` - the complementary error function, `1 - erf($a)`.

erfi

`erfi($a)` - the inverse of the error function.

ndtri

`ndtri($a)` - the inverse of the standard normal distribution function: `ndtri(0.975)` is 1.96.

bessj0

`bessj0($a)` - the Bessel function of the first kind, order 0.

bessj1

`bessj1($a)` - the Bessel function of the first kind, order 1.

bessy0

`bessy0($a)` - the Bessel function of the second kind, order 0.

bessy1

`bessy1($a)` - the Bessel function of the second kind, order 1.

lgamma

`lgamma($a)` - the log of the absolute value of the gamma function, and its sign: `($l, $s) = lgamma($a)`.

bessjn

`bessjn($a, $n)` - the Bessel function of the first kind of integer order `$n`.

bessyn

`bessyn($a, $n)` - the Bessel function of the second kind of integer order `$n`.

Constructors

pdl

`$class->pdl(...)`: `$class->new(...)`.

topdl

Core.xs `topdl(klass, arg1, ...)`: `arg1` itself when it is an object; otherwise `klass->new(...)` of all the arguments - several of them, a plain scalar or an array ref.

null

`null()` - create a null (typeless, dimensionless) PDL placeholder.

new

`PDL->new(...)` - an ndarray from a number, a list of numbers, a (nested) array ref, a Matlab-style string (`"[1 2; 3 4]"`), a Math::Complex, or another ndarray (copied); a leading `PDL::Type` object sets the type.

initialize

`$class->initialize`: a new null ndarray blessed into `$class` (a class name, or the class of a reference). `new` calls it on the class it was invoked on, so a subclass - or a subclass's own `initialize` - decides what `My->new(...)` gives back.

zeroes

Core.xs `new_from_specification(invoc, ...)`, `ALIAS: PDL::zeroes = 1`.

Also available as `PDL::new_from_specification`.

set_sv_to_null_pdl

Turn an existing scalar into a null ndarray.

new_around_datasv

`PDL->new_around_datasv($address, $offset)` - an ndarray whose data is the string of the Perl scalar at `$address` (`0 + \$scalar`), from byte `$offset` (default 0) on. The scalar is kept alive, not copied. The result still needs its type and dimensions:

new_around_pointer

`PDL->new_around_pointer($pointer, $nbytes)` - an ndarray whose data is the `$nbytes` bytes of memory at `$pointer`, which it neither copies nor frees. The result still needs its type and dimensions.

Elements and data

get_dataref

Return a reference to the scalar holding the ndarray's raw bytes.

upd_data

Tell the ndarray that the scalar `get_dataref` returned was changed.

update_data_from

Replace the ndarray's contents with the bytes of a scalar.

at

Return a single element of an ndarray as a plain Perl scalar.

set

`set($pdl, @indices, $value)` - set the element at the given indices.

unpdl

Return the ndarray as nested Perl arrayrefs, shape and all.

string

`string($pdl)` - return the PDL's string representation.

sclr

`sclr($pdl)` - return the only element as a plain Perl scalar.

set_donttouchdata

Declare that the ndarray's storage belongs to someone else.

datasv_refcount

Return how many references are held to the ndarray's data scalar.

firstvals_nophys

`$pdl->firstvals_nophys` - up to the first ten elements of the ndarray's own data. Dies when it has none, as a view does.

Dimensions

dims

`dims($pdl)` - return the list of dimension sizes. Also aliased as shape().

getndims

Return how many dimensions the ndarray has.

Also available as `PDL::ndims`.

nelem

Return the total number of elements in the ndarray.

dim

`dim($pdl, $n)` - return the size of dimension `$n`. A negative `$n` counts from the last dimension; one past the last dimension is 1.

Also available as `PDL::getdim`.

dimincs

`dimincs($pdl)` - return the list of stride increments.

isnull

`isnull($pdl)` - return 1 if the PDL is a null placeholder.

setdims

Give an ndarray a new shape, in place.

nbytes

`nbytes($pdl)` - return the number of bytes in the pdl's data buffer.

nelem_nophys

`$pdl->nelem_nophys` - the number of elements as currently recorded, without computing a pending view first.

dims_nophys

`$pdl->dims_nophys` - the dimensions as currently recorded, without computing a pending view first. Always a list.

dimincs_nophys

`$pdl->dimincs_nophys` - the step in elements along each dimension as currently recorded. Always a list.

broadcastids_nophys

`$pdl->broadcastids_nophys` - the broadcast ids as currently recorded. Always a list.

getnbroadcastids

`$pdl->getnbroadcastids` - how many broadcast ids the ndarray has.

getbroadcastid

`$pdl->getbroadcastid($n)` - broadcast id number `$n`: the dimension where that broadcast group starts.

Parallel processing

set_autopthread_targ

`set_autopthread_targ($n)` - the number of threads an operation is split across, at most; 0 turns the splitting off. Returns `$n`.

get_autopthread_targ

`get_autopthread_targ()` - the target set by `set_autopthread_targ`.

set_autopthread_size

`set_autopthread_size($n)` - the size, in Mi elements of the largest ndarray, from which an operation is split across threads (default 1; `PDL_AUTOPTHREAD_SIZE` in the environment sets it when PDL loads). Returns `$n`.

get_autopthread_size

`get_autopthread_size()` - the size set by `set_autopthread_size`.

get_autopthread_actual

`get_autopthread_actual()` - how many threads the most recently set up operation was split across; 0 when it was not split.

get_autopthread_dim

`get_autopthread_dim()` - the dimension the most recently set up operation was split along; -1 when it was not split.

add_threading_magic

`$pdl->add_threading_magic($dim, $nthreads)` - have operations on this ndarray split dimension `$dim` across `$nthreads` threads, whatever the automatic setting would do.

remove_threading_magic

`$pdl->remove_threading_magic` - undo `add_threading_magic`.

Dataflow and views

inplace

`$pdl->inplace` - mark the ndarray so that the next operation on it that can work in place writes its result into it; returns `$pdl`.

flowing

Turn on dataflow, so ndarrays derived from this one track its changes.

sever

Detach a view from its parent, keeping the values it has right now.

dog

`$pdl->dog` - the ndarray as a list of views, one per element of its last dimension; `$pdl->dog({Break => 1})` severs them from it.

readonly

Mark the ndarray read-only and return it.

make_physical

Give the ndarray data of its own - a view (slice, dice, ...) gets its elements computed and stored - and return it.

has_vafftrans

`$pdl->has_vafftrans` - 1 when the ndarray is a view that reads its data from another ndarray's memory, 0 otherwise.

vaffine_from

`$pdl->vaffine_from` - the address of the ndarray a view reads its data from. Dies when the ndarray is not such a view.

make_physvaffine

`$pdl->make_physvaffine` - compute a view far enough that its elements can be read from its parent's memory, without copying them. Returns the ndarray.

trans_parent

`$pdl->trans_parent` - the transformation that produces this ndarray, a `PDL::Trans` object, or undef when it has none.

trans_children

`$pdl->trans_children` - the transformations that read this ndarray, as `PDL::Trans` objects; in scalar context, how many there are.

bind

`$pdl->bind($code)` - have `$code` called, without arguments, after the ndarray's data changed. The calls are collected and made by the next `PDL::dowhenidle()`.

dowhenidle

`PDL::dowhenidle()` - call the subs bound with `bind` to ndarrays that changed since the last call.

Bad values

badflag

`badflag($pdl, $flag?)` - get or set the bad-value flag.

has_badvalue

`$pdl->has_badvalue` - 1 when the ndarray has a bad value of its own instead of its type's, 0 otherwise.

Types

set_datatype

Change the element type of an ndarray in place.

get_datatype

`get_datatype($pdl)` - return the datatype as integer type code.

Headers

sethdr

`$pdl->sethdr($hashref)` - give the ndarray a header: any hash, such as the one `rfits` reads. `sethdr(undef)` removes it.

hdr

`$pdl->hdr` - the header, a hash reference; an ndarray without one gets an empty header first.

gethdr

`$pdl->gethdr` - the header, a hash reference, or undef when the ndarray has none.

hdrcpy

Get or set whether the header travels with derived ndarrays.

Filtering

box2d

`box2d($a, $wx, $wy, $edgezero)` - each pixel replaced by the mean of the `$wx` by `$wy` box around it. The pixels nearer to the edge than half a box are set to 0 if `$edgezero` is true and copied otherwise.

patch2d

`patch2d($a, $bad)` - `$a` with each pixel where `$bad` is true replaced by the mean of its good neighbours in the 3 by 3 box around it; a pixel with no good neighbour is kept.

patchbad2d

`patchbad2d($a)` - `$a` with each bad value replaced by the mean of its good neighbours in the 3 by 3 box around it; one with no good neighbour stays bad.

Resampling

bilin2d

`bilin2d($in, $out)` - `$in` stretched to the size of `$out` by bilinear interpolation and added to `$out`, which must not be smaller.

rescale2d

`rescale2d($in, $out)` - `$in` scaled to the size of `$out`, larger or smaller, each output pixel the area-weighted mean of the input it covers, and added to `$out`.

Other Functions

rice_expand

`rice_expand($in, $len, $dim0, $blocksize?)` - the rows `rice_compress` squeezed into `$in`, restored to `$dim0` elements each.

cquant_c

`cquant_c($a, $b, $c, $psz)` - the kernel under `cquant`: the palette image `$b` and the colour table `$c` (of `$psz` colours) for the RGB byte image `$a`.

wtstat

`wtstat($a, $wt, $avg, $deg)` - the weighted moment of degree `$deg` of `$a` about `$avg` along the first dimension: the sum of `$wt * ($a - $avg) ** $deg` over the sum of `$wt`.

histogram

`histogram($data, $step, $min, $numbins)` - the number of values of `$data` in each of `$numbins` bins of width `$step` starting at `$min`, along the first dimension. A value below the first bin counts in the first, one above the last in the last.

whistogram

`whistogram($data, $weight, $step, $min, $numbins)` - as `histogram`, each value counting with its weight.

histogram2d

`histogram2d($a, $b, $stepa, $mina, $na, $stepb, $minb, $nb)` - the two-dimensional histogram of the pairs (`$a`, `$b`).

whistogram2d

`whistogram2d($a, $b, $weight, $stepa, $mina, $na, $stepb, $minb, $nb)` - as `histogram2d`, each pair counting with its weight.

t_spherical_inv

`t_spherical_inv($in, $origin, $angunit)` - the kernel under the inverse of `t_spherical`: (longitude, latitude, radius) to Cartesian (x, y, z), the angles multiplied by `$angunit` (1 by default) to give radians. Works in place.

max2d_ind

`($val, $x, $y) = max2d_ind($img)` - the largest value of an image and its position.

centroid2d

`($xcen, $ycen) = centroid2d($img, $x, $y, $box)` - the centroid of the `$box`-wide square around `($x, $y)`.

ccNcompt

`ccNcompt($img, $con)` - the connected components of an image, `$con` 4 or 8 for the neighbourhood, each numbered from 1.

rot2d

`rot2d($img, $angle, $bg, $aa)` - a byte image turned by `$angle` degrees (-90 to 90), uncovered pixels set to `$bg`, antialiased when `$aa` is true.

pnminraw

`pnminraw($type, $im, $ms, $ns, $isbin, $fh)` - read a raw PNM image of `$ms` by `$ns` pixels from `$fh`, bits (`$isbin`) or bytes, into `$im`.

pnminascii

`pnminascii($type, $im, $ms, $ns, $format, $fh)` - read an ASCII PNM image of `$ms` by `$ns` pixels from `$fh` into `$im`.

pnmout

`pnmout($im, $israw, $isbin, $fh)` - write the pixels of `$im` to `$fh`, raw or ASCII, as bits (`$isbin`) or bytes.

bswap2

`bswap2($x)` - the bytes of each 2-byte element of `$x` reversed, in place.

bswap4

`bswap4($x)` - the bytes of each 4-byte element reversed, in place.

bswap8

`bswap8($x)` - the bytes of each 8-byte element reversed, in place.

bswap12

`bswap12($x)` - the bytes of each 12-byte element reversed, in place.

bswap16

`bswap16($x)` - the bytes of each 16-byte element reversed, in place.

bswap24

`bswap24($x)` - the bytes of each 24-byte element reversed, in place.

bswap32

`bswap32($x)` - the bytes of each 32-byte element reversed, in place.

PDL::Bad

Bad values: PDL's marker for missing data, and the functions that find, make and replace them.

```
use PDL;
my $x = pdl(1, 2, -1, 4);
$x->inplace->setvaltobad(-1);
print "$x ", $x->nbad, " ", $x->ngood, " ", $x->badflag, "\n";     #␣
↪[1 2 BAD 4] 1 3 1
print isbad($x), " ", $x->setbadtoval(0), " ", locf($x), "\n";   #␣
↪[0 0 1 0] [1 2 0 4] [1 2 2 4]
print $x->sum, " ", $x->avg, "\n";                                # 7␣
↪2.33333333333333
print setbadif(pdl(1, 2, 3), pdl(0, 1, 0)), " ", pdl(1, 9**9**9)->
↪setinftobad, "\n";   # [1 BAD 3] [1 BAD]
```

An element is bad when it holds its type's bad value and the ndarray's bad flag is set (`badflag`). Operations skip bad elements - `sum` and `avg` above count the three good ones - and a result computed from a bad element is bad. Each type has a default bad value, which an ndarray can override for itself: `badvalue` reads or sets either, called on an ndarray or on a type, and `orig_badvalue` gives a type's built-in default. Floating-point types can use NaN instead. nbad and ngood count, `check_badflag` clears the flag of an ndarray that has no bad element left.

This is PDL::Bad of PDL 2.106. The kernels are PDL's own, through pperl's engine; `badvalue`, `orig_badvalue`, `check_badflag`, nbad, ngood and `setbadat` (one element made bad, by position) are upstream's Perl unchanged.

PDL::Basic

PDL's basic constructors: counting sequences, coordinate values along a dimension, evenly spaced ranges, distances from a centre and histograms.

```
use PDL;
print sequence(5), " ", sequence(byte, 2, 2)->type, "\n";         #␣
↪[0 1 2 3 4] byte
print xvals(3, 2)->flat, " ", yvals(3, 2)->flat, "\n";            #␣
↪[0 1 2 0 1 2] [0 0 0 1 1 1]
print zeroes(4)->xlinvals(0, 1), " ", rvals(3), "\n";             #␣
↪[0 0.333333333333333 0.666666666666667 1] [1 0 1]
my ($centres, $counts) = hist(pdl(1, 2, 2, 3), 0, 4, 1);
print "$centres $counts\n";                                       #␣
```

(continues on next page)

(continued from previous page)

```
 ↪[0.5 1.5 2.5 3.5] [0 1 2 1]
print zeroes(2, 3)->ndcoords->dims, " ", ndcoords(2)->dims, "\n";
 ↪# 223 12
```

All of it is upstream's Perl, unchanged: `sequence`, `xvals`, `yvals`, `zvals` and `axisvals` (each element its index along a dimension), `xlinvals`, `ylinvals`, `zlinvals`, `xlogvals`, `ylogvals`, `zlogvals`, `axislinvals`, `axislogvals`, `allaxislinvals` and `allaxislogvals` (evenly spaced, linearly or logarithmically, between two values), `rvals` (the distance of each element from the centre), `ndcoords` (the coordinates of each element), `transpose`, `sec`, `ins` and `similar_assign`. Each takes dimensions, or an ndarray whose dimensions to use, and an optional type first.

`($centres, $counts) = hist($data, $min, $max, $step)` is the histogram of `$data`, the bin centres and the count in each bin; without `$min`, `$max` and `$step` the bins are chosen from the data. In scalar context it returns the counts alone. `whist($data, $weight, ...)` counts each value with its weight.

This is PDL::Basic of PDL 2.106.

PDL::Complex

Namespace containing the following modules:

- `PDL::Complex::Overloads` - subclass of Math::Complex with overload fallbacks

PDL::Complex::Overloads

subclass of Math::Complex with overload fallbacks

SYNOPSIS

```
require PDL::Complex::Overloads;
my $same = PDL::Complex::Overloads::cplx(1, 2) eq '1+2i';
```

DESCRIPTION

This is a subclass whose only purpose is to provide *Math::Complex*'s overloads but with `fallback` true, mainly to allow string-comparison for backwards compatibility.

AUTHOR

Ed J

SEE ALSO

Math::Complex

PDL::Compression

Rice compression of integer ndarrays, as FITS files use it.

```
use PDL;
use PDL::Compression;
my $x = long(1, 2, 3, 4, 5, 4, 3, 2, 1, 0);
my ($y, $n, $blocksize, $len) = $x->rice_compress;
print $y->rice_expand($len, $n), "\n";     # [1 2 3 4 5 4 3 2 1 0]
```

`rice_compress` squeezes each row of a byte, short, ushort or long ndarray; in list context it also returns the row length, the block size and each row's compressed length, which `rice_expand` needs to restore the data. The algorithm is the RICE_1 compression of CFITSIO, so PDL::IO::FITS reads and writes RICE-compressed images with it.

This is PDL::Compression of PDL 2.106: upstream's Perl, unchanged, over PDL's own kernels run by pperl's engine.

PDL::Constants

basic compile time constants for PDL

DESCRIPTION

This module is used to define compile time constant values for PDL. It uses Perl's *constant* pragma for simplicity and availability.

SYNOPSIS

```
use PDL::Constants qw(PI E);
print 'PI is ' . PI . "\n";
print 'E  is ' .  E . "\n";
```

PI

The ratio of a circle's circumference to its diameter

DEGRAD

The number of degrees of arc per radian (180/PI)

DEG2RAD

The constant to convert degrees of arc into radians (PI/180)

```
$radians = DEG2RAD * $degrees;
```

RAD2DEG

The constant to convert radians into degrees of arc (180/PI)

```
$degrees = RAD2DEG * $radians;
```

E

The base of the natural logarithms or Euler's number

COPYRIGHT & LICENSE

Copyright 2010 Chris Marshall (chm at cpan dot org).

This program is free software; you can redistribute it and/or modify it under the terms of either: the GNU General Public License as published by the Free Software Foundation; or the Artistic License.

See http://dev.perl.org/licenses/ for more information.

PDL::Core

Build ndarrays, ask them about themselves, and convert between types.

```
use PDL;
my $x = pdl([1, 2, 3], [4, 5, 6]);
print join(",", $x->dims), " ", $x->nelem, " ", $x->type, " ", $x->
 ↪at(2, 1), "\n";   # 3,2 6 double 6
print $x->info, "\n";                                    # PDL:␣
 ↪Double D [3,2]
print zeroes(byte, 2), ones(2), " ", sequence(3)->convert(float)->
 ↪type, "\n";   # [0 0][1 1] float
my @l = $x->list;
print "@l ", short(1.7), " ", pdl(5)->sclr + 1, "\n";       # 1 2 3␣
 ↪4 5 6 1 6
my $v = $x->slice(":,(0)");
$v .= 0;
print $x->flat, " ", $x->copy->sever->isnull, "\n";          # [0 0␣
 ↪0 4 5 6] 0
print cat(pdl(1, 2), pdl(3, 4))->flat, " ", $x->badflag, "\n";   #␣
 ↪[1 2 3 4] 0
```

`PDL::Core` is the foundation the rest of PDL rests on. It holds the constructors (`pdl`, `null`, `zeroes`, `ones`, `empty`, `cat`), the accessors that report shape and contents (`dims`, `shape`, `ndims`, `nelem`, `at`, `set`, `list`, `unpdl`, `info`), the type system (`type`, `convert`, `get_datatype`, `howbig`), and the bad-value flag (`badflag`; the rest of the bad-value machinery is PDL::Bad).

Two ideas run through all of it. Ndarrays are typed: an ndarray of `byte` and one of `double` hold different things, and `convert` moves between them. And ndarrays can share data: a slice or a reshape is a view onto its parent, so `sever` and `copy` exist for the times you want a snapshot that stops tracking the original.

The types and their constructors, each `type(...)` an ndarray of that type and a bare `type` the type itself (as in `zeroes(float, 3)`): `sbyte`, `byte`, `short`, `ushort`, `long`, `ulong`, `indx`, `ulonglong`, `longlong`, `float`, `double`, `ldouble`, `cfloat`, `cdouble`, `cldouble`. `piddle` is the old name of `pdl`.

The state of an ndarray, as methods returning 0 or 1: `allocated`, `anychgd`, `datachgd`, `dimschgd`, `donttouch`, `fflows`, `is_inplace`, `is_readonly`, `vaffine`; `set_inplace` changes it, and `getdim` and `getndims` are `dim` and `ndims`. `address`, `address_data` and `address_datasv` are the memory addresses behind an ndarray, for debugging, `new_from_specification` is `zeroes`' engine, `is_scalar_SvPOK` is a helper of `pdl`'s, and `howbig_c` is the one of `howbig`.

This is PDL::Core of PDL 2.106. Its XS is pperl's, native; its Perl (`cat`, `ones`, `empty`, `info`, `convert`, `copy`, `reshape`, `squeeze`, `dummy`, `clump`, `mslice`, `barf`, `approx`, `howbig`, `isempty`, `shape`, `type`, `list`, `fhdr` and the rest) is upstream's, unchanged.

Functions

Constructors

alltopdl

`PDL::Core::alltopdl($class, $value, $type)`, see [`alltopdl`].

pdl_avref

Core.xs `pdl_avref(array_ref, class, type)`.

Elements and data

at_bad_c

`PDL::Core::at_bad_c($pdl, [@pos])` (also `at_c`): the element at `@pos`, "BAD" for a bad value - what `$pdl->at(@pos)` is built on.

set_c

`PDL::Core::set_c($pdl, [@pos], $value)`: store `$value` at `@pos` - what `$pdl->set(@pos, $value)` is built on.

listref_c

`PDL::Core::listref_c($pdl)`: the elements as a nested array ref, "BAD" for bad values - what `list` is built on.

Parallel processing

online_cpus

`PDL::Core::online_cpus()` - how many threads PDL can split an operation across.

pthreads_enabled

`PDL::Core::pthreads_enabled()` - 1: operations can be split across threads.

Miscellaneous

seed

`PDL::Core::seed()`: a seed for the random number generators, from the time and the process id.

set_debugging

`PDL::Core::set_debugging($on)` - set the engine's debugging flag and return its previous value.

PDL::Exporter

PDL export control

DESCRIPTION

Implements the standard conventions for import of PDL modules in to the namespace

Hopefully will be extended to allow fine control of which namespace is used.

SYNOPSIS

use PDL::Exporter;

```
use PDL::MyModule;       # Import default function list ':Func'
use PDL::MyModule '';    # Import nothing (OO)
use PDL::MyModule '...'; # Same behaviour as Exporter
```

SUMMARY

`PDL::Exporter` is a drop-in replacement for the *Exporter* module. It confers the standard PDL export conventions to your module. Usage is fairly straightforward and best illustrated by an example. The following shows typical usage near the top of a simple PDL module:

```
package PDL::MyMod;

use strict;

# For Perl 5.6:
use vars qw($VERSION @ISA @EXPORT @EXPORT_OK %EXPORT_TAGS);
# For more modern Perls:
our ($VERSION, @ISA, @EXPORT, @EXPORT_OK, %EXPORT_TAGS);
```

(continues on next page)

(continued from previous page)

```
require PDL::Exporter;

@ISA = qw(PDL::Exporter);
@EXPORT_OK = qw(inc myfunc); # these will be exported by default
%EXPORT_TAGS = (Func=>[@EXPORT_OK],
                Internal => [qw/internfunc1 internfunc2/],
               );

 # ... body of your module

1; # end of simple module
```

SEE ALSO

Exporter

AUTHOR

Copyright (C) Karl Glazebrook (kgb@aaoepp.aao.gov.au). Some docs by Christian Soeller. All rights reserved. There is no warranty. You are allowed to redistribute this software / documentation under certain conditions. For details, see the file COPYING in the PDL distribution. If this file is separated from the PDL distribution, the copyright notice should be included in the file.

PDL::FFT

Fast Fourier transforms of PDL's ndarrays, in place, in one or more dimensions.

```
use PDL;
use PDL::FFT;
my ($re, $im) = (pdl(1, 2, 3, 4), zeroes(4));
fft($re, $im);
print "$re $im\n";                  # [10 -2 -2 -2] [0 2 0 -2]
```

`fft($re, $im)` and `ifft($re, $im)` transform the real and imaginary parts in place along the first dimension, broadcasting over the rest; `realfft`/`realifft` work on real data, `fftnd`/`ifftnd` in every dimension, and `fftconvolve` convolves two ndarrays through the transform. All of these are upstream's Perl, unchanged, over the kernels; this is PDL::FFT of PDL 2.106.

PDL::IO

Namespace containing the following modules:

- `PDL::IO::FITS` - Simple FITS support for PDL
- `PDL::IO::Misc` - Reading and writing ndarrays as text, and byte-order swapping.
- `PDL::IO::Pic` - image I/O for PDL

- `PDL::IO::Pnm` - Reading and writing images in the PNM formats: PBM, PGM and PPM.
- `PDL::IO::Storable` - helper functions to make PDL usable with serialisation packages

PDL::IO::FITS

Simple FITS support for PDL

SYNOPSIS

```
use PDL;
use PDL::IO::FITS;

$x = rfits('foo.fits');          # read a FITS file
$x->wfits('bar.fits');           # write a FITS file
```

DESCRIPTION

This module provides basic FITS support for PDL, in the sense of reading and writing whole FITS files. For more complex operations, such as prefiltering rows out of tables or performing operations on the FITS file in-place on disk, you can use the Astro::FITS::CFITSIO module that is available on CPAN.

Basic FITS image files are supported, along with BINTABLE and IMAGE extensions. ASCII Table support is planned, as are the HEASARC bintable extensions that are recommended in the 1999 FITS standard.

Table support is based on hashes and named columns, rather than the less convenient (but slightly more congruent) technique of perl lists of numbered columns.

The principle interface routines are `rfits` and `wfits`, for reading and writing respectively. FITS headers are returned as perl hashes or (if the module is present) Astro::FITS::Header objects that are tied to perl hashes. Astro::FITS::Header objects provide convenient access through the tied hash interface, but also allow you to control the card structure in more detail using a separate method interface; see the Astro::FITS::Header documentation for details.

AUTHOR

Copyright (C) Karl Glazebrook, Craig DeForest, and Doug Burke, 1997-2010. There is no warranty. You are allowed to redistribute and/or modify this software under certain conditions. For details, see the file COPYING in the PDL distribution. If this file is separated from the PDL distribution, the copyright notice should be pasted into in this file.

FUNCTIONS

rfits()

Simple ndarray FITS reader.

```
$pdl = rfits('file.fits');   # Read a simple FITS image
```

Suffix magic:

```
$pdl = rfits('file.fits.gz'); # Read a file with gunzip(1)
$pdl = rfits('file.fits.Z');  # Read a file with uncompress(1)

$pdl = rfits('file.fits[2]');    # Read 2nd extension
$pdl = rfits('file.fits.gz[3]'); # Read 3rd extension
@pdls = rfits('file.fits');      # Read primary data and extensions
```

Tilde expansion:

```
#expand leading ~ to home directory (using glob())
$pdl = rfits '~/filename.fits';

$hdr = rfits('file.fits',{data=>0});  # Options hash changes␣
↪behavior
```

In list context, `rfits` reads the primary image and all possible extensions, returning them in the same order that they occurred in the file – except that, by default, the primary HDU is skipped if it contains no data. In scalar context, the default is to read the first HDU that contains data. One can read other HDU's by using the [n] syntax. Using the [0] syntax forces a read of the first HDU, regardless of whether it contains data or no. Currently recognized extensions are IMAGE and BINTABLE. (See the addendum on EXTENSIONS for details).

`rfits` accepts several options that may be passed in as a hash ref if desired:

- bscale (default=1)

  Determines whether the data are linearly scaled using the BSCALE/BZERO keywords in the FITS header. To read in the exact data values in the file, set this to 0.

- data (default=1)

  Determines whether to read the data, or just the header. If you set this to 0, you will get back the FITS header rather than the data themselves. (Note that the header is normally returned as the `hdr` field of the returned PDL; this causes it to be returned as a hash ref directly.)

- hdrcpy (default=0)

  Determines whether the *hdrcpy* flag is set in the returned PDL. Setting the flag will cause an explicit deep copy of the header whenever you use the returned PDL in an arithmetic or slicing operation. That is useful in many circumstances but also causes a hit in speed. When two or more PDLs with hdrcpy set are used in an expression, the result gets the header of the first PDL in the expression. See *hdrcpy* for an example.

- expand (default=1)

  Determines whether auto-expansion of tile-compressed images should happen. Tile-compressed images are transmitted as binary tables with particular fields

("ZIMAGE") set. Leaving this alone does what you want most of the time, unpacking such images transparently and returning the data and header as if they were part of a normal IMAGE extension. Setting "expand" to 0 delivers the binary table, rather than unpacking it into an image.

- afh (default=1)

  By default rfits uses Astro::FITS::Header tied-hash objects to contain the FITS header information. This permits explicit control over FITS card information, and conforms well with the FITS specification. But Astro::FITS::Header objects are about 40-60x more memory intensive than comparable perl hashes, and also use ~10x more CPU to manage. For jobs where header processing performance is important (e.g. reading just the headers of 1,000 FITS files), set afh to 0 to use the legacy parser and get a large boost in speed.

FITS image headers are stored in the output PDL and can be retrieved with *hdr* or *gethdr*. The *hdrcpy* flag of the PDL is set so that the header is copied to derived ndarrays by default. (This is inefficient if you are planning to do lots of small operations on the data; clear the flag with "->hcpy(0)" or via the options hash if that's the case.)

The header is a hash whose keys are the keywords in the FITS header. If you have the "Astro::FITS::Header" module installed, the header is actually a tied hash to a FITS header object, which can give you more control over card order, comment fields, and variable types. (see Astro::FITS::Header for details).

The header keywords are converted to *uppercase* per the FITS standard. Access is case-insensitive on the perl side, provided that Astro::FITS::Header is installed.

If Astro::FITS::Header is not installed, then a built-in legacy parser is used to generate the header hash. Keyword-associated comments in the headers are stored under the hash key `<keyword>_COMMENT>`. All HISTORY cards in the header are collected into a single multiline string stored in the `HISTORY` key. All COMMENT cards are similarly collected under the `COMMENT` key.

BSCALE/BZERO

If the BSCALE and/or BZERO keywords are set, they are applied to the image before it is returned. The returned PDL is promoted as necessary to contain the multiplied values, and the BSCALE and BZERO keywords are deleted from the header for clarity. If you don't want this type of processing, set 'bscale=>0' in the options hash.

EXTENSIONS

Sometimes a FITS file contains only extensions and a stub header in the first header/data unit ("primary HDU"). In scalar context, you normally only get back the primary HDU – but in this special case, you get back the first extension HDU. You can force a read of the primary HDU by adding a '[0]' suffix to the file name.

BINTABLE EXTENSIONS

Binary tables are handled. Currently only the following PDL datatypes are supported: byte, short, ushort, long, float, and double. At present ushort() data is written as a long rather than as a short with TSCAL/ZERO; this may change.

The return value for a binary table is a hash ref containing the names of the columns in the table (in UPPER CASE as per the FITS standard). Each element of the hash contains a PDL (for numerical values) or a perl list (for string values). The PDL's 0th dimension runs across rows; the 1st dimension runs across the repeat index within the row (for rows with more than one value). (Note that this is different from standard broadcasting order - but it allows Least Surprise to work when adding more complicated objects such as collections of numbers (via the repeat count) or variable length arrays.)

Thus, if your table contains a column named FOO with type 5D, the expression

```
$x->{FOO}->((2))
```

returns a 5-element double-precision PDL containing the values of FOO from the third row of the table.

The header of the table itself is parsed as with a normal FITS HDU, and is returned in the element 'hdr' of the returned hash. You can use that to preserve the original column order or access the table at a low level, if you like.

Scaling and zero-point adjustment are performed as with BSCALE/BZERO: the appropriate keywords are deleted from the as-returned header. To avoid this behavior, set 'bscale=>0' in the options hash.

As appropriate, TSCAL/ZERO and TUNIT are copied into each column-PDL's header as BSCALE/BZERO and BUNIT.

The main hash also contains the element 'tbl', which is set to 'binary' to distinguish it from an ASCII table.

Because different columns in the table might have identical names in a FITS file, the binary table reader practices collision avoidance. If you have multiple columns named "FOO", then the first one encountered (numerically) gets the name "FOO", the next one gets "FOO_1", and the next "FOO_2", etc. The appropriate TTYPEn fields in the header are changed to match the renamed column fields.

Columns with no name are assigned the name "COL_<n>", where <n> starts at 1 and increments for each no-name column found.

Variable-length arrays are supported for reading. They are unpacked into PDLs that appear exactly the same as the output for fixed-length rows, except that each row is padded to the maximum length given in the extra characters – e.g. a row with TFORM of 1PB(300) will yield an NAXIS2x300 output field in the final hash. The padding uses the TNULn keyword for the column, or 0 if TNULn is not present. The output hash also gets an additional field, "len_<name>", that contains the number of elements in each table row.

TILE-COMPRESSED IMAGES

CFITSIO and several large projects (including NASA's Solar Dynamics Observatory) now support an unofficial extension to FITS that stores images as a collection of individually compressed tiles within a BINTABLE extension. These images are automagically uncompressed by default, and delivered as if they were normal image files. You can override this behavior by supplying the "expand" key in the options hash.

Currently, only Rice compression is supported, though there is a framework in place for adding other compression schemes.

BAD VALUE HANDLING

If a FITS file contains the BLANK keyword (and has BITPIX > 0), the ndarray will have its bad flag set, and those elements which equal the BLANK value will be set bad. For BITPIX < 0, any NaN's are converted to bad (if necessary).

rfitshdr()

Read only the header of a FITS file or an extension within it.

This is syntactic sugar for the data=>0 option to rfits.

See rfits for details on header handling. rfitshdr() runs the same code to read the header, but returns it rather than reading in a data structure as well.

wfits()

Simple PDL FITS writer

```
wfits $pdl, 'filename.fits', [$BITPIX], [$COMPRESSION_OPTIONS];
wfits $hash, 'filename.fits', [$OPTIONS];
$pdl->wfits('foo.fits',-32);
```

Suffix magic:

```
# Automatically compress through pipe to gzip
wfits $pdl, 'filename.fits.gz';
# Automatically compress through pipe to compress
wfits $pdl, 'filename.fits.Z';
```

Tilde expansion:

```
#expand leading ~ to home directory (using glob())
wfits $pdl, '~/filename.fits';
```

- Ordinary (PDL) data handling:

  If the first argument is a PDL, then the PDL is written out as an ordinary FITS file with a single Header/Data Unit of data.

  $BITPIX is then optional and coerces the output data type according to the standard FITS convention for the BITPIX field (with positive values representing integer types and negative values representing floating-point types).

  If $pdl has a FITS header attached to it (actually, any hash that contains a SIMPLE=>T keyword), then that FITS header is written out to the file. The image dimension tags are adjusted to the actual dataset. If there's a mismatch between the dimensions of the data and the dimensions in the FITS header, then the header gets corrected and a warning is printed.

  If $pdl is a slice of another PDL with a FITS header already present (and header copying enabled), then you must be careful. wfits will remove any extraneous NAXISn keywords (per the FITS standard), and also remove the other keywords associated with that axis: CTYPEn, CRPIXn, CRVALn, CDELTn, and CROTAn. This

may cause confusion if the slice is NOT out of the last dimension: `wfits($x(:,(0),:),'file.fits');` and you would be best off adjusting the header yourself before calling `wfits`.

You can tile-compress images according to the CFITSIO extension to the FITS standard, by adding an option hash to the arguments:

- compress

  This can be either unity, in which case Rice compression is used, or a (case-insensitive) string matching the CFITSIO compression type names. Currently supported compression algorithms are:

  * RICE_1 - linear Rice compression

    This uses limited-symbol-length Rice compression, which works well on low entropy image data (where most pixels differ from their neighbors by much less than the dynamic range of the image).

- BLOCKSIZE (RICE_1 only; default 32)

  For RICE_1, indicates the number of pixel samples to use for each compression block within the compression algorithm. The blocksize is independent of the tile dimensions. For RICE compression the pixels from each tile are arranged in normal pixel order (early dims fastest) and compressed as a linear stream.

- Table handling:

  If you feed in a hash ref instead of a PDL, then the hash ref is written out as a binary table extension. The hash ref keys are treated as column names, and their values are treated as the data to be put in each column.

  For numeric information, the hash values should contain PDLs. The 0th dim of the PDL runs across rows, and higher dims are written as multi-value entries in the table (e.g. a 7x5 PDL will yield a single named column with 7 rows and 5 numerical entries per row, in a binary table). Note that this is slightly different from the usual concept of broadcasting, in which dimension 1 runs across rows.

  ASCII tables only allow one entry per column in each row, so if you plan to write an ASCII table then all of the values of `$hash` should have at most one dim.

  All of the columns' 0 dims must agree in the broadcasting sense. That is to say, the 0th dimension of all of the values of `$hash` should be the same (indicating that all columns have the same number of rows). As an exception, if the 0th dim of any of the values is 1, or if that value is a PDL scalar (with 0 dims), then that value is "broadcasted" over – copied into all rows.

  Data dimensions higher than 2 are preserved in binary tables, via the TDIMn field (e.g. a 7x5x3 PDL is stored internally as seven rows with 15 numerical entries per row, and reconstituted as a 7x5x3 PDL on read).

  Non-PDL Perl scalars are treated as strings, even if they contain numerical values. For example, a list ref containing 7 values is treated as 7 rows containing one string each. There is no such thing as a multi-string column in FITS tables, so any non-scalar values in the list are stringified before being written. For example, if you pass in a perl list of 7 PDLs, each PDL will be stringified before being written, just as if you printed it to the screen. This is probably not what you want – you should

use “glue” to connect the separate PDLs into a single one. (e.g. `$x->glue(1,$y,$c)->mv(1,0)`)

The column names are case-insensitive, but by convention the keys of `$hash` should normally be ALL CAPS, containing only digits, capital letters, hyphens, and underscores. If you include other characters, then case is smashed to ALL CAPS, whitespace is converted to underscores, and unrecognized characters are ignored – so if you include the key “Au Purity (%)”, it will be written to the file as a column that is named “AU_PURITY”. Since this is not guaranteed to produce unique column names, subsequent columns by the same name are disambiguated by the addition of numbers.

You can specify the use of variable-length rows in the output, saving space in the file. To specify variable length rows for a column named “FOO”, you can include a separate key “len_FOO” in the hash to be written. The key’s value should be a PDL containing the number of actual samples in each row. The result is a FITS P-type variable length column that, upon read with `rfits()`, will restore to a field named FOO and a corresponding field named “len_FOO”. Invalid data in the final PDL consist of a padding value (which defaults to 0 but which you may set by including a TNULL field in the hdr specificaion). Variable length arrays must be 2-D PDLs, with the variable length in the 1 dimension.

Two further special keys, ‘hdr’ and ‘tbl’, can contain meta-information about the type of table you want to write. You may override them by including an `$OPTIONS` hash with a ‘hdr’ and/or ‘tbl’ key.

The ‘tbl’ key, if it exists, must contain either ‘ASCII’ or ‘binary’ (case-insensitive), indicating whether to write an ascii or binary table. The default is binary. [ASCII table writing is planned but does not yet exist].

You can specify the format of the table quite specifically with the ‘hdr’ key or option field. If it exists, then the ‘hdr’ key should contain fields appropriate to the table extension being used. Any field information that you don’t specify will be filled in automatically, so (for example) you can specify that a particular column name goes in a particular position, but allow `wfits` to arrange the other columns in the usual alphabetical order into any unused slots that you leave behind. The `TFORMn`, `TFIELDS`, `PCOUNT`, `GCOUNT`, `NAXIS`, and `NAXISn` keywords are ignored: their values are calculated based on the hash that you supply. Any other fields are passed into the final FITS header verbatim.

As an example, the following

```
$x = long(1,2,4);
$y = double(1,2,4);
wfits { 'COLA'=>$x, 'COLB'=>$y }, "table1.fits";
```

will create a binary FITS table called `table1.fits` which contains two columns called `COLA` and `COLB`. The order of the columns is controlled by setting the `TTYPEn` keywords in the header array, so

```
$h = { 'TTYPE1'=>'Y', 'TTYPE2'=>'X' };
wfits { 'X'=>$x, 'Y'=>$y, hdr=>$h }, "table2.fits";
```

creates `table2.fits` where the first column is called `Y` and the second column is `X`.

- multi-value handling

  If you feed in a perl array-ref rather than a PDL or a hash, then each element is written out as a separate HDU in the FITS file. Each element of the list must be a PDL or a hash.

- DEVEL NOTES

  ASCII tables are not yet handled but should be.

  Binary tables currently only handle one vector (up to 1-D array) per table entry; the standard allows more, and should be fully implemented.

  Handling multidim arrays implies that perl multidim lists should also be handled.

For integer types (ie `BITPIX > 0`), the `BLANK` keyword is set to the bad value. For floating-point types, the bad value is converted to NaN (if necessary) before writing.

fits_field_cmp

fits_field_cmp

Sorting comparison routine that makes proper sense of the digits at the end of some FITS header fields. Sort your hash keys using "fits_field_cmp" and you will get (e.g.) your "TTYPE" fields in the correct order even if there are 140 of them.

This is a standard perl comparison sub – it uses the magical $a and $b variables, rather than normal argument passing.

_rows()

Return the number of rows in a variable for table entry

You feed in a PDL or a list ref, and you get back the 0th dimension.

_prep_table()

Accept a hash ref containing a table, and return a header describing the table and a string to be written out as the table, or barf.

You can indicate whether the table should be binary or ascii. The default is binary; it can be overridden by the "tbl" field of the hash (if present) or by parameter.

PDL::IO::Misc

Reading and writing ndarrays as text, and byte-order swapping.

```
use PDL;
my $x = short(1, 256);
bswap2($x);
print "$x ", isbigendian() ? "big" : "little", "-endian\n";   # [256␣
 ↪1] little-endian
```

`rcols` reads columns of numbers from a file into ndarrays and `wcols` writes ndarrays as columns; `swcols` writes them to a string, `rgrep` pulls values out of lines matching a pattern, `rasc` reads numbers from an ASCII file into an existing ndarray, and `rcube`

stacks images into a cube. `bswap2`, `bswap4`, `bswap8`, `bswap12`, `bswap16`, `bswap24` and `bswap32` reverse the bytes of each element in place; `isbigendian` reports the host's byte order.

This is PDL::IO::Misc of PDL 2.106: upstream's Perl, unchanged, over PDL's own kernels run by pperl's engine. `use PDL` loads it.

PDL::IO::Pic

image I/O for PDL

DESCRIPTION

This package implements I/O for a number of popular image formats by exploiting the xxxtopnm and pnmtoxxx converters from the netpbm package (which is based on the original pbmplus by Jef Poskanzer).

Netpbm is available at ftp://wuarchive.wustl.edu/graphics/graphics/packages/NetPBM/ Pbmplus (on which netpbm is based) might work as well, I haven't tried it. If you want to read/write JPEG images you additionally need the two converters cjpeg/djpeg which come with the libjpeg distribution (the "official" archive site for this software is ftp://ftp.uu.net/graphics/jpeg).

Image I/O for all formats is established by reading and writing only the PNM format directly while the netpbm standalone apps take care of the necessary conversions. In accordance with netpbm parlance PNM stands here for 'portable any map' meaning any of the PBM/PGM/PPM formats.

As it appeared to be a reasonable place this package also contains the routine wmpeg to write mpeg movies from PDLs representing image stacks (the image stack is first written as a sequence of PPM images into some temporary directory). For this to work you need the program ffmpeg also.

Configuration

The executables from the netpbm package are assumed to be in your path. Problems in finding the executables may show up as PNM format errors when calling wpic/rpic. If you run into this kind of problem run your program with perl `-w` so that perl prints a message if it can't find the filter when trying to open the pipe. [']

FUNCTIONS

rpiccan, wpiccan

Test which image formats can be read/written

```
$im = PDL->rpic('PDL.jpg') if PDL->rpiccan('JPEG');
@wformats = PDL->wpiccan();
```

finds out if PDL::IO::Pic can read/write certain image formats. When called without arguments returns a list of supported formats. When called with an argument returns true if format is supported on your computer (requires appropriate filters in your path), false otherwise.

rpic

Read images in many formats with automatic format detection.

```
$im = rpic $file;
$im = PDL->rpic 'PDL.jpg' if PDL->rpiccan('JPEG');
```

Options

```
FORMAT  =>  'JPEG'   # explicitly read this format
XTRAFLAGS => '-nolut'  # additional flags for converter
```

Reads image files in most of the formats supported by netpbm. You can explicitly specify a supported format by additionally passing a hash containing the FORMAT key as in

```
$im = rpic ($file, {FORMAT => 'GIF'});
```

This is especially useful if the particular format isn't identified by a magic number and doesn't have the 'typical' extension or you want to avoid the check of the magic number if your data comes in from a pipe. The function returns a pdl of the appropriate type upon completion. Option parsing uses the *PDL::Options* module and therefore supports minimal options matching.

You can also read directly into an existing pdl that has to have the right size(!). This can come in handy when you want to read a sequence of images into a datacube, e.g.

```
$stack = zeroes(byte,3,500,300,4);
rpic $stack->slice(':,:,:,(0)'),"PDL.jpg";
```

reads an rgb image (that had better be of size (500,300)) into the first plane of a 3D RGB datacube (=4D pdl datacube). You can also do transpose/inversion upon read that way.

wpic

Write images in many formats with automatic format selection.

```
Usage: wpic($pdl,$filename[,{ options... }])
```

```
wpic $pdl, $file;
$im->wpic('web.gif',{LUT => $lut});
for (@images) {
  $_->wpic($name[0],{CONVERTER => 'ppmtogif'})
}
```

Write out an image file. Function will try to guess correct image format from the filename extension, e.g.

```
$pdl->wpic("image.gif")
```

will write a gif file. The data written out will be scaled to byte if input is of type float/double. Input data that is of a signed integer type and contains negative numbers will be rejected (assuming the user should have the desired conversion to an unsigned type al-

ready). A number of options can be specified (as a hash reference) to get more direct control of the image format that is being written. Valid options are (key => example_value):

```
CONVERTER  => 'ppmtogif',   # explicitly specify pbm converter
FLAGS      => '-interlaced -transparent 0',  # flags for converter
IFORM      => 'PGM',        # explicitly specify intermediate format
XTRAFLAGS  => '-imagename iris', # additional flags to defaultflags
FORMAT     => 'PCX',        # explicitly specify output image format
COLOR      => 'bw',         # specify color conversion
LUT        => $lut,         # use color table information
```

Option parsing uses the *PDL::Options* module and therefore supports minimal options matching. A detailed explanation of supported options follows.

- CONVERTER

  directly specify the converter, you had better know what you are doing, e.g.

  ```
  CONVERTER  => 'ppmtogif',
  ```

- FLAGS

  flags to use with the converter; ignored if !defined($$hints{CONVERTER}), e.g. with the gif format

  ```
  FLAGS      => '-interlaced -transparent 0',
  ```

- IFORM

  intermediate PNM/PPM/PGM/PBM format to use; you can append the strings 'RAW' or 'ASCII' to enforce those modes, eg IFORMAT=>'PGMRAW' or

  ```
  IFORM    => 'PGM',
  ```

- XTRAFLAGS

  additional flags to use with an automatically chosen converter, this example works when you write SGI files (but will give an error otherwise)

  ```
  XTRAFLAGS => '-imagename iris',
  ```

- FORMAT

  explicitly select the format you want to use. Required if wpic cannot figure out the desired format from the file name extension. Supported types are currently TIFF,GIF,SGI,PNM,JPEG,PS,RAST(Sun Raster),IFF,PCX, e.g.

  ```
  FORMAT     => 'PCX',
  ```

- COLOR

  you want black and white (value **bw**), other possible value is **bwdither** which will write a dithered black&white image from the input data, data conversion will be done appropriately, e.g.

```
COLOR       => 'bw',
```

- LUT

  This is a palette image and the value of this key should be a pdl containing an RGB lookup table (3,x), e.g.

```
LUT         => $lut,
```

Using the CONVERTER hint you can also build a pipe and perform several netpbm operations to get the special result you like. Using it this way the first converter/filecommand in the pipe should be specified with the CONVERTER hint and subsequent converters + flags in the FLAGS hint. This is because wpic tries to figure out the required format to be written by wpnm based on the first converter. Be careful when using the PBMBIN var as it will only be prepended to the converter. If more converters are in the FLAGS part specify the full path unless they are in your PATH anyway.

Example:

```
$im->wpic('test.ps',{CONVERTER  => 'pgmtopbm',
                     FLAGS => "-dither8 | pnmtops" })
```

Some of the options may appear silly at the moment and probably are. The situation will hopefully improve as people use the code and the need for different/modified options becomes clear. The general idea is to make the function perl compliant: easy things should be easy, complicated tasks possible.

rim

```
Usage: $x = rim($file);
or        rim($x,$file);
```

Read images in most formats, with improved RGB handling.

You specify a filename and get back a PDL with the image data in it. Any PNM handled format or FITS will work. In the second form, $x is an existing PDL that gets loaded with the image data.

If the image is in one of the standard RGB formats, then you get back data in (,,) format – that is to say, the third dim contains the color information. That allows you to do simple indexing into the image without knowing whether it is color or not – if present, the RGB information is silently broadcasted over. (Contrast "rpic", which munges the information by putting the RGB index in the 0th dim, screwing up subsequent broadcasting operations).

If the image is in FITS format, then you get the data back in exactly the same order as in the file itself.

Images with a ".Z" or ".gz" extension are assumed to be compressed with UNIX "compress" or "gzip", respectively, and are automatically uncompressed before reading.

OPTIONS

The same as "rpic", which is used as an engine:

- FORMAT

  If you don't specify this then formats are autodetected. If you do specify it then only the specified interpreter is tried. For example,

```
$x = rim("foo.gif",{FORMAT=>"JPEG"})
```

  forces JPEG interpretation.

- XTRAFLAGS

  Contains extra command line flags for the pnm interpreter. For example,

```
$x = rim("foo.jpg",{XTRAFLAGS=>"-nolut"})
```

  prevents use of a lookup table in JPEG images.

wim

Write a pdl to an image file with selected type (or using filename extensions)

```
wim $pdl,$file;
$pdl->wim("foo.gif",{LUT=>$lut});
```

Write out an image file. You can specify the format explicitly as an option, or the function will try to guess the correct image format from the filename extension, e.g.

```
$pdl->wim("image.gif");
$pdl->wim("image.fits");
```

will write a gif and a FITS file. The data written out will be scaled to byte if the input if of type float/double. Input data that is of a signed integer type and contains negative numbers will be rejected.

If you append `.gz` or `.Z` to the end of the file name, the final file will be automatically compressed with "gzip" | "compress", respectively.

OPTIONS

You can pass in a hash ref whose keys are options. The code uses the PDL::Options module so unique abbreviations are accepted. Accepted keys are the same as for "wpic", which is used as an engine:

- CONVERTER

  Names the converter program to be used by pbmplus (e.g. "ppmtogif" to output a gif file)

- FLAGS

  Flags that should be passed to the converter (replacing any default flag list) e.g. "-interlaced" to make an interlaced GIF

- IFORM

  Explicitly specifies the intermediate format (e.g. PGM, PPM, or PNM).

- XTRAFLAGS

  Flags that should be passed to the converter (in addition to any default flag list).

- FORMAT

  Explicitly specifies the output image format (allowing pbmplus to pick an output converter)

- COLOR

  Specifies color conversion (e.g. 'bw' converts to black-and-white; see pbmplus for details).

- LUT

  Use color-table information

rmpeg

Read an image sequence (a (3,x,y,n) byte pdl) from an animation.

```
$ndarray = rmpeg('movie.mpg'); # $ndarray is (3,x,y,nframes) byte
```

Reads a stack of RGB images from a movie. While the format generated is nominally MPEG, the file extension is used to determine the video encoder type. It uses the program `ffmpeg`, and throws an exception if not found.

wmpeg

Write an image sequence (a (3,x,y,n) byte pdl) as an animation.

```
$ndarray->wmpeg('movie.mpg'); # $ndarray is (3,x,y,nframes) byte
```

Writes a stack of RGB images as a movie. While the format generated is nominally MPEG, the file extension is used to determine the video encoder type. E.g. `.mpg` for MPEG-1 encoding, `.mp4` for MPEG-4 encoding, `.gif` for GIF animation

`wmpeg` requires a 4-D pdl of type **byte** as input. The first dim **has** to be of size 3 since it will be interpreted as RGB pixel data. `wmpeg` returns 1 on success and undef on failure.

```
use strict; use warnings;
use PDL;
use PDL::IO::Pic;
my ($width, $height, $framecount, $xvel, $maxheight, $ballsize) =
 ↪(320, 80, 100, 15, 60, 8);
my $frames = zeros byte, $width, $height, $framecount;
my $coords = yvals(3, $framecount); # coords for drawing ball, all
 ↪val=frameno
my ($xcoords, $ycoords) = map $coords->slice($_), 0, 1;
$xcoords *= $xvel; # moves $xvel pixels/frame
$xcoords .= $width - abs(($xcoords % (2*$width)) - $width); # back
 ↪and forth
my $sqrtmaxht = sqrt $maxheight;
$ycoords .= indx($maxheight - (($ycoords % (2*$sqrtmaxht)) -
```

(continues on next page)

(continued from previous page)

```
 ↪$sqrtmaxht)**2));
my $val = pdl(byte,250);  # start with white
$frames->range($coords, [$ballsize,$ballsize,1], 't') .= $val;
$frames = $frames->dummy(0, 3)->copy; # now make the movie
$frames->wmpeg('bounce.gif');  # or bounce.mp4, ffmpeg deals OK

# iterate running this with:
rm bounce.gif; perl scriptname.pl && animate bounce.gif
```

Some of the input data restrictions will have to be relaxed in the future but routine serves as a proof of principle at the moment. It uses the program ffmpeg to encode the frames into video. Currently, wmpeg doesn't allow modification of the parameters written through its calling interface. This will change in the future as needed.

In the future it might be much nicer to implement a movie perl object that supplies methods for manipulating the image stack (insert, cut, append commands) and a final movie->make() call would invoke ffmpeg on the picture stack (which will only be held on disk). This should get around the problem of having to hold a huge amount of data in memory to be passed into wmpeg (when you are, e.g. writing a large animation from PDL3D rendered fly-throughs).

imageformat

Figure out the format of an image file from its magic numbers, or else, from its extension.

Currently recognized image formats are: PNM, GIF, TIFF, JPEG, SGI, RAST, IFF, PCX, PS, FITS, PNG, XBM. If the format can not be determined, the string 'UNKNOWN' is returned.

```
$format=imageformat($path); # find out image format of certain file
print "Unknown image format" if $format eq 'UNKNOWN';
$canread=rpiccan($format); # check if this format is readable in␣
 ↪this system
if($canread){
    $pdl=rpic($path) ; # attempt to read image ONLY if we can
} else {
    print "Image can't be read\n"; # skip unreadable file
}
```

BUGS

Currently only a random selection of converters/formats provided by pbmplus/netpbm is supported. It is hoped that the more important formats are covered. Other formats can be added as needed. Please send patches to the author.

AUTHOR

Copyright (C) 1996,1997 Christian Soeller c.soeller@auckland.ac.nz All rights reserved. There is no warranty. You are allowed to redistribute this software / documentation under certain conditions. For details, see the file COPYING in the PDL distribution. If

this file is separated from the PDL distribution, the copyright notice should be included in the file.

PDL::IO::Pnm

Reading and writing images in the PNM formats: PBM, PGM and PPM.

```
use PDL;
use PDL::IO::Pnm;
use File::Temp qw(tempdir);
my $file = tempdir(CLEANUP => 1) . "/img.pgm";
wpnm(byte([[0, 128], [255, 64]]), $file, 'PGM', 1);
my $im = rpnm($file);
print $im->flat, " ", $im->type, "\n";            # [0 128 255 64]␣
 ↪byte
```

`rpnm($file)` reads a PBM, PGM or PPM file, raw or ASCII, into an ndarray - 2-D for grey levels, `(3, width, height)` for colour - and `wpnm($im, $file, $format, $raw)` writes one. `pnminraw`, `pnminascii` and `pnmout` are the kernels underneath, reading and writing through a Perl filehandle.

This is PDL::IO::Pnm of PDL 2.106: upstream's Perl, unchanged, over PDL's own kernels run by pperl's engine. PDL::IO::Pic uses it, so `use PDL` loads it.

PDL::IO::Storable

helper functions to make PDL usable with serialisation packages

SYNOPSIS

```
use Storable;
use PDL::IO::Storable;
$hash = {
          'foo' => 42,
          'bar' => zeroes(23,45),
        };
store $hash, 'perlhash.dat';

use JSON::MaybeXS;
$encoder = JSON::MaybeXS->new(allow_tags => 1);
my $ndarray = xvals (5,2);
my $encoded_json = $encoder->encode ($ndarray);
my $decoded_ndarray = $encoder->decode ($encoded_json);
```

DESCRIPTION

Serialisation packages such as `Storable`, `Sereal`, `JSON::MaybeXS` and `CBOR::XS` implement object persistence for Perl data structures that can contain arbitrary Perl objects. This module implements the relevant methods to be able to store and retrieve ndarrays via `Storable` as well as packages that support the *Types::Serialiser* protocol (currently Sereal, CBOR::XS and JSON packages).

Note that packages supporting the `Types::Serialiser` protocol need to have their respective flags enabled so that the FREEZE and THAW callbacks are used.

Note also that while JSON::MaybeXS is supported, if it has to fall back to *JSON::PP*, it will fail. *JSON::XS* treats the data it gets back from `FREEZE` as items to encode, while *JSON::PP* treats the list it gets as strings already encoded. They are fundamentally incompatible, so this module supports the JSON::XS option.

Finally, data flow is not preserved. If a data structure containing ndarrays connected by data flow is serialised then this will need to be explicitly reinstated on deserialisation.

FUNCTIONS

store

store an ndarray using *Storable*

```
$x = random 12,10;
$x->store('myfile');
```

freeze

freeze an ndarray using *Storable*

```
$x = random 12,10;
$frozen = $x->freeze;
```

AUTHOR

Copyright (C) 2013 Dima Kogan dima@secretsauce.net Copyright (C) 2002 Christian Soeller c.soeller@auckland.ac.nz All rights reserved. There is no warranty. You are allowed to redistribute this software / documentation under certain conditions. For details, see the file COPYING in the PDL distribution. If this file is separated from the PDL distribution, the copyright notice should be included in the file.

PDL::Image2D

Two-dimensional image processing: median filters, polygon filling, connected components, rotation and warping.

```
use PDL;
use PDL::Image2D;
my $im = zeroes(long, 5, 5);
polyfill($im, float([[1, 1], [3, 1], [3, 3], [1, 3]]), 7);
print $im->sum, " ", sequence(5, 4)->med2d(ones(3, 3))->at(2, 2), "\
 ↪n";   # 63 12
```

`med2d` and `med2df` are median filters (by a kernel, or by a box); `polyfill`, `polyfillv`, `pnpoly` fill or select the inside of a polygon; `cc8compt`, `cc4compt` and `ccNcompt` label connected components; `max2d_ind` and `centroid2d` locate features; `rot2d` and `rotnewsz` rotate a byte image; `fitwarp2d`, `applywarp2d`, `warp2d` and `warp2d_kernel` fit and apply a polynomial distortion; `conv2d` convolves with a kernel (`{ Boundary`

`=> 'Reflect' }` and the like choose the edge treatment), `box2d` is a box-mean filter, `patch2d` and `patchbad2d` replace marked or bad pixels by the mean of their neighbours, `bilin2d` and `rescale2d` resample into an ndarray of another size, and `crop` gives the bounding box of the non-zero pixels.

`polyfill_pp`, `pnpoly_pp` and `pnpolyfill_pp` are the kernels under `polyfill`, `pnpoly` and `polyfillv`.

This is PDL::Image2D of PDL 2.106: upstream's Perl, unchanged, over PDL's own kernels and support C run by pperl's engine.

Functions

Other Functions

`rotnewsz`

`($cols, $rows) = rotnewsz($m, $n, $angle)` - the size of an `$m` by `$n` image once `rot2d` has turned it by `$angle` degrees.

PDL::ImageND

Image processing in any number of dimensions: convolution, rebinning, interpolation, circular masks, contours and graph layout forces.

```
use PDL;
use PDL::ImageND;
print convolveND(sequence(5), pdl(1, 1, 1)), "\n";        # [1 3 6 9
 ↪7]
print rebin(sequence(4), 2), " ", kernctr(sequence(4), pdl(1, 2,
 ↪1))->flat, "\n";
# [0.5 2.5] [2 1 0 1]
```

`convolveND($a, $kernel, { boundary => 'periodic' })` convolves in as many dimensions as the kernel has, with a choice of boundary conditions and of the direct or the FFT method; `convolve` is the older form, wrapping around. `rebin($a, @dims)` changes the size by averaging or replicating (`{ Norm => 1 }` keeps the sum), `ninterpol($point, $data, $n)` interpolates linearly in the first `$n` dimensions, `kernctr($image, $kernel)` pads a kernel to the size of an image for the FFT, and `circ_mean` and `circ_mean_p` average around rings. `contour_segments` and `contour_polylines` trace a contour level through a 2-D field, `path_join` and `path_segs` turn edges into paths, `combcoords` packs three coordinates into one ndarray, and `attract` and `repulse` compute the spring and the repulsive forces of a graph layout.

This is PDL::ImageND of PDL 2.106: upstream's Perl, unchanged, over PDL's own kernels run by pperl's engine.

PDL::ImageRGB

Utility functions for RGB image data: palette quantisation, palette lookup, conversion to grey, and scaling to bytes.

```
use PDL;
use PDL::ImageRGB;
print rgbtogr(byte([[10, 20, 30]])), " ", bytescl(pdl(0, 500, 1000),
 ↪100), "\n";   # [18] [0 50 100]
my $lut = byte([[0, 0, 0], [255, 255, 255]]);
print interlrgb(byte(1, 0), $lut)->flat, "\n";        # [255 255 255
 ↪0 0 0]
```

An RGB image here is `(3, width, height)`. `cquant($image, $ncolours)` reduces an RGB byte image to a palette image and its colour table; `interlrgb($palette_image, $lut)` turns one back into RGB; `rgbtogr` converts RGB to grey levels; `bytescl` scales an ndarray into 0..255 (or another top) as bytes.

This is PDL::ImageRGB of PDL 2.106: upstream's Perl, unchanged, over PDL's own `cquant_c` kernel (ppm_quant) run by pperl's engine.

PDL::Lite

PDL with methods only: the core modules loaded, four names imported.

```
use PDL::Lite;
my $x = pdl(1, 2, 3);
print $x->sum, " ", PDL->zeroes(2), " ", defined &main::zeroes ?
 ↪"imported" : "methods only", "\n";   # 6 [0 0] methods only
```

`use PDL::Lite` loads PDL::Core, PDL::Ops, PDL::Primitive, PDL::Ufunc, PDL::Basic, PDL::Slices and PDL::Bad, and imports only `pdl`, `piddle`, `null` and `barf`: everything else is called as a method, or fully qualified. For the same modules with their functions imported, use PDL::LiteF; for the whole of PDL, `use PDL`.

This is PDL::Lite of PDL 2.106: upstream's Perl, unchanged.

PDL::LiteF

The core PDL modules, with their functions imported.

```
use PDL::LiteF;
print sum(pdl(1, 2, 3)), " ", zeroes(2), " ", defined &main::det ?
 ↪"MatrixOps" : "no MatrixOps", "\n";   # 6 [0 0] no MatrixOps
```

`use PDL::LiteF` loads the modules PDL::Lite does - PDL::Core, PDL::Ops, PDL::Primitive, PDL::Ufunc, PDL::Basic, PDL::Slices and PDL::Bad - and imports each one's functions, as `use PDL` does. Unlike `use PDL` it leaves out PDL::Math, PDL::MatrixOps and the IO modules.

This is PDL::LiteF of PDL 2.106: upstream's Perl, unchanged.

PDL::Math

PDL's mathematical functions: trigonometric and hyperbolic, rounding, error and Bessel functions, log gamma, complex square roots, logarithms and inverses, and polynomials.

```
use PDL;
print floor(pdl(1.5, -1.5)), " ", ceil(pdl(1.5, -1.5)), " ",␣
↪rint(pdl(0.5, 1.5, 2.5)), "\n";   # [1 -2] [2 -1] [0 2 2]
printf "%.4f %.4f %.4f\n", asin(pdl(1)), tanh(pdl(1)), erf(pdl(1));␣
↪                  # 1.5708 0.7616 0.8427
print isfinite(pdl(1, 9**9**9, -9**9**9)), "\n";                    ␣
↪              # [1 0 0]
my ($lg, $sign) = lgamma(pdl(5));
print "$lg $sign ", pow(byte(2), 3), "\n";                          ␣
↪               # 3.17805383034795 1 8
print clog(czip(0, 1)), "\n";                                       ␣
↪              # 1.57079637050629i
```

Each function works element by element and broadcasts; most work in place (`$x->inplace->floor`). `sin`, `cos`, `exp`, `log` and `sqrt` are PDL::Ops's, as upstream has them. The `c...` functions take real input as complex, so `csqrt(-4)` is `2i` where `sqrt(-4)` is NaN.

This is PDL::Math of PDL 2.106. The kernels are PDL's own, through pperl's engine; `polyroots` (the roots of a polynomial), `polyfromroots` (its coefficients from the roots) and `polyval` (its value at a point), all with coefficients in decreasing powers, are upstream's Perl unchanged.

PDL::MatrixOps

Linear algebra on PDL's ndarrays: determinants, inverses, LU, singular value and eigen decompositions, linear systems, triangular storage.

```
use PDL;
use PDL::MatrixOps;
my $m = pdl([4, 7], [2, 6]);
print det($m), " ", inv($m)->flat, "\n";                    # 10 [0.6␣
↪-0.7 -0.2 0.4]
print identity(2)->flat, " ", ($m x inv($m))->flat, "\n";  # [1 0 0␣
↪1] [1 0 0 1]
my ($u, $s, $v) = svd(pdl([3, 0], [0, 4]));
print "$s\n";                                               # [4 3]
my ($ev, $e) = eigens_sym(pdl([2, 1], [1, 2]));
print qsort($e), "\n";                                      # [1 3]
```

A matrix is a 2-D ndarray indexed `(column, row)`, and every function broadcasts over further dimensions, so a stack of matrices is one call. `x` is matrix multiplication (PDL::Primitive's `matmult`).

This is PDL::MatrixOps of PDL 2.106. The kernels are PDL's own, through pperl's engine; upstream's Perl is used unchanged for `identity`, `stretcher`, `inv`, `det`, `determinant`, `lu_decomp`, `lu_decomp2`, `lu_backsub`, `eigens`, `eigens_sym`, `gurney` and `tritosym`.

PDL::Ops

PDL's operators: arithmetic, comparison, bitwise and the unary functions, element by element, as the functions behind Perl's operators on ndarrays.

```
use PDL;
my ($a, $b) = (pdl(1, 2, 3), pdl(4, 5, 6));
print $a + $b, " ", $a * $b, " ", $b ** 2, " ", $b % 4, "\n";   # [5␣
↪7 9] [4 10 18] [16 25 36] [0 1 2]
print $a > 1, " ", $a <=> 2, " ", -$a, "\n";                    # [0 1␣
↪1] [-1 0 1] [-1 -2 -3]
print sqrt(pdl(4, 9)), " ", abs(pdl(-2, 3)), " ", log10(pdl(10,␣
↪100)), "\n";   # [2 3] [2 3] [1 2]
print pdl(6) & 3, " ", byte(250) + 10, "\n";                 # 2 4
my $z = czip(3, 4);
print $z, " ", abs($z), " ", $z->re, " ", conj($z), "\n";    # 3+4i␣
↪5 3 3-4i
my $x = $a;
$x += 10;
print "$a\n";                                                # [11 12␣
↪13]
```

Every operation broadcasts: operands of different shapes are matched dimension by dimension, a dimension of size 1 (or a Perl number) repeated to fit. The result has the larger operand type - `byte(250) + 10` stays a byte and wraps. A binary operation takes an output and a swap flag: `minus($a, $b, $c, 1)` writes `$b - $a` into `$c`. With three arguments the third is the swap flag, as perl's overloading passes it - `minus($a, $b, 1)` returns `$b - $a`. With `$a->inplace` the result is written into `$a`.

The operators are overloaded with these functions: `+ - * / ** %`, `== != < > <= >= <=>`, `& | ^ << >> ~ !`, `atan2 sqrt abs sin cos exp log`, every assignment form (`+=` and the rest rewrite the left ndarray, so `$x += 1` reaches every name of it), `++`, `--` and `.=` (assignment into an ndarray or a view of one: `assgn`). Unary minus is perl's `0 - $x`. `eq` compares element-wise and warns when the other operand is neither a number nor an ndarray; the other string operators are not overloaded and die.

This is PDL::Ops of PDL 2.106. The kernels are PDL's own, through pperl's engine. The overload handlers are upstream's Perl (`PDL::overload_plus` and the rest), each with a native fast path in its place that hands anything unusual back to it; `abs` (`_rabs` for real types, `_cabs` for complex), `log10`, `r2C` and `i2C` are upstream's Perl unchanged.

PDL::Options

simplifies option passing by hash in PerlDL

SYNOPSIS

```
use PDL::Options;

%hash = parse( \%defaults, \%user_options);

```

(continues on next page)

(continued from previous page)

```
use PDL::Options ();

$opt = PDL::Options->new;
$opt = PDL::Options->new( \%defaults );

$opt->defaults ( \%defaults );
$opt->synonyms ( { 'COLOR' => 'COLOUR' } );

$hashref = $opt->defaults;

$opt->options ( \%user_options );

$hashref = $opt->options;

$opt->incremental(1);
$opt->full_options(0);
```

DESCRIPTION

Object to simplify option passing for PerlDL subroutines. Allows you to merge a user defined options with defaults. A simplified (non-OO) interface is provided.

Utility functions

ifhref

```
parse({Ext => 'TIF', ifhref($opt)});
```

just return the argument if it is a hashref otherwise return an empty hashref. Useful in conjunction with parse to return just the default values if argument is not a hash ref

NON-OO INTERFACE

A simplified non-object oriented interface is provided. These routines are exported into the callers namespace by default.

- parse(\%defaults, \%user_options)

  This will parse user options by using the defaults. The following settings are used for parsing: The options are case-sensitive, a default synonym table is consulted (see “Default Synonyms”), minimum-matching is turned on, and translation of values is not performed.

  A hash (not hash reference) containing the processed options is returned.

  ```
  %options = parse( { LINE => 1, COLOUR => 'red'}, { COLOR =>
  ↪'blue'});
  ```

- iparse(\%defaults, \%user_options)

  Same as `parse` but matching is case insensitive

Default Synonyms

The following default synonyms are available in the non-OO interface:

```
COLOR  => COLOUR
COLOUR => COLOR
CENTER => CENTRE
CENTRE => CENTER
```

METHODS

The following methods are available to PDL::Options objects.

- new()

  Constructor. Creates the object. With an optional argument can also set the default options.

- extend (\%options)

  This will copy the existing options object and extend it with the requested extra options.

- defaults(\%defaults)

  Method to set or return the current defaults. The argument should be a reference to a hash. The hash reference is returned if no arguments are supplied.

  The current values are reset whenever the defaults are changed.

- add_synonym (\%synonyms)

  Method to add another synonym to an option set The argument should be a reference to a hash.

- add_translation (\%translation)

  Method to add another translation rule to an option set. The argument should be a reference to a hash.

- synonyms(\%synonyms)

  Method to set or return the current synonyms. The argument should be a reference to a hash. The hash reference is returned if no arguments are supplied.

  This allows you to provide alternate keywords (such as allowing 'COLOR' as an option when your defaults uses 'COLOUR').

- current

  Returns the current state of the options. This is returned as a hash reference (although it is not a reference to the actual hash stored in the object). If full_options() is true the full options hash is returned, if full_options() is false only the modified options are returned (as set by the last call to options()).

- clear_current

  This routine clears the 'state' of the `PDL::Options` object so that the next call to current will return an empty list

- translation

  Provide translation of options to more specific values that are recognised by the program. This allows, for example, the automatic translation of the string 'red' to '#ff0000'.

  This method can be used to setup the dictionary and is hash reference with the following structure:

```
OPTIONA => {
            'string1' => decode1,
            'string2' => decode2
            },
OPTIONB => {
            's4' => decodeb1,
           }
etc....
```

  Where OPTION? corresponds to the top level option name as stored in the defaults array (eg LINECOLOR) and the anonymous hashes provide the translation from string1 ('red') to decode1 ('#ff0000').

  An options string will be translated automatically during the main options() processing if autotrans() is set to true. Else translation can be initiated by the user using the translate() method.

- incremental

  Specifies whether the user defined options will be treated as additions to the current state of the object (1) or modifications to the default values only (0).

  Can be used to set or return this value. Default is false.

- full_options

  Governs whether a complete set of options is returned (ie defaults + expanded user options), true, or if just the expanded user options are returned, false (ie the values specified by the user).

  This can be useful when you are only interested in the changes to the options rather than knowing the full state. (For example, if defaults contains keys for COLOUR and LINESTYLE and the user supplied a key of COL, you may simply be interested in the modification to COLOUR rather than the state of LINESTYLE and COLOUR.)

  Default is true.

- casesens

  Specifies whether the user defined options will be processed independent of case (0) or not (1). Default is to be case insensitive.

  Can be used to set or return this value.

- minmatch

  Specifies whether the user defined options will be minimum matched with the defaults (1) or whether the user defined options should match the default keys exactly. Defaults is true (1).

If a particular key matches exactly (within the constraints imposed bby case sensitivity) this key will always be taken as correct even if others are similar. For example COL would match COL and COLOUR but this implementation will always return COL in this case (note that for CO it will return both COL and COLOUR and pick one at random.

Can be used to set or return this value.

- autotrans

  Specifies whether the user defined options will be processed via the translate() method immediately following the main options parsing. Default is to autotranslate (1).

  Can be used to set or return this value.

- casesenstrans

  Specifies whether the keys in the options hash will be matched insensitive of case (0) during translation() or not (1). Default is to be case insensitive.

  Can be used to set or return this value.

- minmatchtrans

  Specifies whether the keys in the options hash will be minimum matched during translation(). Default is false (0).

  If a particular key matches exactly (within the constraints imposed bby case sensitivity) this key will always be taken as correct even if others are similar. For example COL would match COL and COLOUR but this implementation will always return COL in this case (note that for CO it will return both COL and COLOUR and pick one at random.

  Can be used to set or return this value.

- warnonmissing

  Turn on or off the warning message printed when an options is not in the options hash. This can be convenient when a user passes a set of options that has to be parsed by several different option objects down the line.

- debug

  Turn on or off debug messages. Default is off (0). Can be used to set or return this value.

- options

  Takes a set of user-defined options (as a reference to a hash) and merges them with the current state (or the defaults; depends on the state of incremental()).

  The user-supplied keys will be compared with the defaults. Case sensitivity and minimum matching can be configured using the mimatch() and casesens() methods.

  A warning is raised if keys present in the user options are not present in the defaults unless warnonmissing is set.

  A reference to a hash containing the merged options is returned.

```
$merged = $opt->options( { COL => 'red', Width => 1});
```

The state of the object can be retrieved after this by using the current() method or by using the options() method with no arguments. If full_options() is true, all options are returned (options plus overrides), if full_options() is false then only the modified options are returned.

Synonyms are supported if they have been configured via the synonyms() method.

- translate

  Translate the current option values (eg those set via the options() method) using the provided translation().

  This method updates the current state of the object and returns the updated options hash as a reference.

```
$ref = $opt->translate;
```

EXAMPLE

Two examples are shown. The first uses the simplified interface and the second uses the object-oriented interface.

Non-OO

```
use PDL::Options (':Func');

%options = parse( {
                LINE => 1,
                COLOUR => 'red',
               },
               {
                COLOR => 'blue'
               }
             );
```

This will return a hash containing

```
%options = (
             LINE => 1,
             COLOUR => 'blue'
           )
```

Object oriented

The following example will try to show the main points:

```
use PDL::Options ();

# Create new object and supply defaults
```

(continues on next page)

(continued from previous page)

```
$opt = PDL::Options->new(    { Colour => 'red',
                               LineStyle => 'dashed',
                               LineWidth => 1
                             }
                         );

# Create synonyms
$opt->synonyms( { Color => 'Colour' } );

# Create translation dictionary
$opt->translation( { Colour => {
                        'blue' => '#0000ff',
                        'red'  => '#ff0000',
                        'green'=> '#00ff00'
                               },
                     LineStyle => {
                      'solid' => 1,
                      'dashed' => 2,
                      'dotted' => 3
                      }
                   }
                 );

# Generate and parse test hash
$options = $opt->options( { Color => 'green',
                            lines => 'solid',
                          }
                        );
```

When this code is run, $options will be the reference to a hash containing the following:

```
Colour => '#00ff00',
LineStyle => 1,
LineWidth => 1
```

If full_options() was set to false (0), $options would be a reference to a hash containing:

```
Colour => '#00ff00',
LineStyle => 1
```

Minimum matching and case insensitivity can be configured for both the initial parsing and for the subsequent translating. The translation can be turned off if not desired.

Currently synonyms are not available for the translation although this could be added quite simply.

AUTHOR

Copyright (C) Tim Jenness 1998 (t.jenness@jach.hawaii.edu). All rights reserved. There is no warranty. You are allowed to redistribute this software / documentation under certain conditions. For details, see the file COPYING in the PDL distribution. If this file is separated from the PDL distribution, the copyright notice should be included in the file.

PDL::PP

Namespace containing the following modules:

- `PDL::PP::Signature` - Internal module to handle signatures

PDL::PP::Signature

Internal module to handle signatures

DESCRIPTION

Internal module to handle signatures

SYNOPSIS

```
use PDL::PP::Signature;
```

AUTHOR

Copyright (C) Tuomas J. Lukka 1997 (lukka@husc.harvard.edu) and by Christian Soeller (c.soeller@auckland.ac.nz). All rights reserved. There is no warranty. You are allowed to redistribute this software / documentation under certain conditions. For details, see the file COPYING in the PDL distribution. If this file is separated from the PDL distribution, the copyright notice should be included in the file.

PDL::Primitive

PDL's primitives: products, selecting by a mask, clipping, searching sorted data, interpolation, set operations on vectors, statistics.

```
use PDL;
my ($a, $b) = (pdl(1, 2, 3), pdl(4, 5, 6));
print inner($a, $b), " ", crossp($a, $b), " ", norm(pdl(3, 4)), "\n
↪";   # 32 [-3 6 -3] [0.6 0.8]
print $a->clip(1.5, 2.5), " ", which($a > 1), " ", $a->where($a >␣
↪1), "\n";   # [1.5 2 2.5] [1 2] [2 3]
print uniq(pdl(3, 1, 3)), " ", append($a, 9), " ", vsearch(2.5, $a),␣
↪"\n";   # [1 3] [1 2 3 9] 2
print +(pdl([1, 2], [3, 4]) x pdl([1], [1]))->flat, "\n";       # [3␣
↪7]
my ($mean, $prms) = stats($b);
```

(continues on next page)

(continued from previous page)

```
print "$mean $prms ", interpol(1.5, $a, $b), "\n";                # 5
 ↪1 4.5
```

`where` returns a view: `$x->where($x < 0) .= 0` zeroes the negative elements of `$x`. `x` is matrix multiplication (`matmult`).

This is PDL::Primitive of PDL 2.106. The kernels are PDL's own, through pperl's engine; upstream's Perl is used unchanged for `matmult`, `append`, `glue`, `clip`, `hclip`, `lclip`, `stats`, `statsover`, `fibonacci`, `one2nd`, `uniq`, `uniqind`, `uniqvec`, `vsearch`, `interpND`, `whichND`, `whichND_both`, `whereND`, `whereND_both`, `where_both`, `setops`, `intersect`, the `...vec` and `..._sorted` set operations (`unionvec`, `intersectvec`, `setdiffvec`, `union_sorted`, `intersect_sorted`, `setdiff_sorted`), `conv1d`, and the random numbers: `random` (uniform in [0, 1)), `randsym` (uniform in (0, 1)), `grandom` (normal, mean 0 and standard deviation 1) and `srandom` (seeds them).

PDL::Slices

Take pieces of an ndarray, and rearrange the pieces you take.

```
use PDL;
my $m = sequence(4, 3);
print $m->slice("1:2,(0)"), " ", $m->slice(":,(-1)"), "\n";       #
 ↪[1 2] [8 9 10 11]
print $m->dice([0, 3], [2])->flat, " ", $m->xchg(0, 1)->dims, "\n";
 ↪ # [8 11] 34
print sequence(5)->index(pdl(4, 0)), " ", sequence(5)->rotate(1), "\
 ↪n";   # [4 0] [4 0 1 2 3]
my $row = $m->slice(":,1");
$row .= 0;
print $m->sum, " ", $m->clump(-1)->dims, "\n";                     #
 ↪44 12
my ($len, $val) = rle(pdl(5, 5, 7));
print "$len $val ", rld($len, $val), "\n";                        # [2
 ↪1] [5 7] [5 5 7]
```

Most of these return a view, not a copy: the result shares storage with the ndarray it came from, so writing into the view writes into the original - `$row .= 0` above zeroes a row of `$m`. Call `sever` or `copy` when the two should part company.

`slice` takes, per dimension, a range `"start:end:step"`, a single index in parentheses `"(i)"` (which drops the dimension) or nothing for all; negative indices count from the end. `dice` and `dice_axis` pick elements by number, `index`, `index1d` and `index2d` by an index ndarray, `range` gathers neighbourhoods around coordinates (and `indexND`, one element at each), `diagonal` takes a diagonal. The rest rearrange without changing values: `xchg`, `mv`, `reorder`, `flat`, `clump`, `splitdim`, `rotate`, `lags`, and `broadcast`/`unbroadcast` for broadcasting control. `rle`/`rld` and their `vec`, `seq` and `ND` forms encode and decode runs.

This is PDL::Slices of PDL 2.106. The kernels are PDL's own, through pperl's engine; upstream's Perl is used unchanged for `slice`, `dice`, `dice_axis`, `diagonal`, `reorder`, `meshgrid`, `using`, `rle`, `rld`, `rleND`, `rldND`, `rldvec`, `rldseq`, `indexND` and `indexNDb`.

PDL::Trans

Modules

- `PDL::Trans::VTable`

Functions

Dataflow and views

parents

`$trans->parents` - the ndarrays the transformation reads.

children

`$trans->children` - the ndarrays the transformation produces.

address

`$trans->address` - the memory address of the transformation, the number its reference holds. For debugging.

bvalflag

`$trans->bvalflag` - whether the transformation handles bad values.

flags

`$trans->flags` - the names of the flags set on the transformation, such as `PDL_ITRANS_ISAFFINE`.

vtable

`$trans->vtable` - the description of the operation behind the transformation, a `PDL::Trans::VTable` object.

affine

`$trans->affine` - 1 when the transformation is a view expressed as an offset and a step per dimension, 0 otherwise.

offs

`$trans->offs` - the offset of an affine view into its parent's data.

incs

`$trans->incs` - the step per dimension of an affine view. Dies for a transformation that is not affine.

trans_children_indices

`$trans->trans_children_indices` - for each parent, the slot the transformation occupies in that parent's list of children.

ind_sizes

`$trans->ind_sizes` - the size of each named dimension of the operation's signature.

inc_sizes

`$trans->inc_sizes` - the step in elements for each dimension of each parameter of the operation.

PDL::Trans::VTable

Functions

Dataflow and views

name

`$vtable->name` - the name of the operation, such as `PDL::Ops::plus`.

flags

`$vtable->flags` - the names of the flags of the operation, such as `PDL_TRANS_DO_BROADCAST`.

par_names

`$vtable->par_names` - two array references: the names of the operation's input parameters, and of its outputs.

dump

`$vtable->dump` - print the description of an operation to standard output.

PDL::Transform

Coordinate transformations as objects, and the resampling of images through them.

```
use PDL;
use PDL::Transform;
my $t = t_linear(scale => 2, pre => pdl(1, 0));
```

(continues on next page)

(continued from previous page)

```
print $t->apply(pdl(1, 1)), " ", $t->invert(pdl(4, 2)), "\n";     #␣
↪[4 2] [1 1]
print sequence(3, 3)->map(t_linear(rot => 90), { method => 'sample'␣
↪})->slice(':,(0)'), "\n";
# [2 5 8]
```

A transformation is made by a constructor - `t_linear` (scale, rotation, offset, matrix), `t_scale`, `t_offset`, `t_rot`, `t_identity`, `t_lookup` (a table), `t_fits` (the world coordinates of a FITS header), `t_code` (your own subs), `t_cylindrical`, `t_radial`, `t_spherical`, `t_quadratic`, `t_cubic`, `t_quartic`, `t_projective` - and combined: `$a x $b` or `t_compose` applies `$b` first, `!$t` or `t_inverse` reverses, `$t ** $n` repeats, `t_wrap` conjugates. `$t->apply($coords)` and `$t->invert($coords)` transform coordinates (the first dimension runs over the axes). `$image->map($t, ...)` resamples an image through a transformation and unmap through its inverse, with the interpolation `method` (`sample`, `linear`, `cubic`, `fft`, the anti-aliasing `jacobian`, `hanning`, `gaussian`) and the boundary condition among their options.

This is PDL::Transform of PDL 2.106: upstream's Perl, unchanged, over PDL's own kernels run by pperl's engine.

Modules

- `PDL::Transform::Cartography` - Useful cartographic projections

PDL::Transform::Cartography

Useful cartographic projections

> **Note**
>
> The plotting lines in the examples (`pgswin()` and `plot`) come from PDL::Graphics::Simple, which is not bundled with pperl. The projections and the data functions (`graticule`, `earth_coast`, `earth_image`, `earth_shape`, `clean_lines`) work without it.

SYNOPSIS

```
# make a Mercator map of Earth
use PDL::Transform::Cartography;
use PDL::Graphics::Simple;
$x = earth_coast();
$x = graticule(10,2)->glue(1,$x);
$t = t_mercator;
$w = pgswin();
$w->plot(with=>'polylines', $t->apply($x)->clean_lines);
```

DESCRIPTION

PDL::Transform::Cartography includes a variety of useful cartographic and observing projections (mappings of the surface of a sphere), including reprojected observer coordinates. See *PDL::Transform* for more information about image transforms in general.

Cartographic transformations are used for projecting not just terrestrial maps, but also any nearly spherical surface including the Sun, the Celestial sphere, various moons and planets, distant stars, etc. They also are useful for interpreting scientific images, which are themselves generally projections of a sphere onto a flat focal plane (e.g. the "t_gnomonic" projection).

Unless otherwise noted, all the transformations in this file convert from (theta,phi) coordinates on the unit sphere (e.g. (lon,lat) on a planet or (RA,dec) on the celestial sphere) into some sort of projected coordinates, and have inverse transformations that convert back to (theta,phi). This is equivalent to working from the equidistant cylindrical (or "plate carree") projection, if you are a cartography wonk.

The projected coordinates are generally in units of body radii (radians), so that multiplying the output by the scale of the map yields physical units that are correct wherever the scale is correct for that projection. For example, areas should be correct everywhere in the authalic projections; and linear scales are correct along meridians in the equidistant projections and along the standard parallels in all the projections.

The transformations that are authalic (equal-area), conformal (equal-angle), azimuthal (circularly symmetric), or perspective (true perspective on a focal plane from some viewpoint) are marked. The first two categories are mutually exclusive for all but the "unit sphere" 3-D projection.

Extra dimensions tacked on to each point to be transformed are, in general, ignored. That is so that you can add on an extra index to keep track of pen color. For example, "earth_coast" returns a 3x<n> ndarray containing (lon, lat, pen) at each list location. Transforming the vector list retains the pen value as the first index after the dimensional directions.

GENERAL NOTES ON CARTOGRAPHY

Unless otherwise noted, the transformations and miscellaneous information in this section are taken from Snyder & Voxland 1989: "An Album of Map Projections", US Geological Survey Professional Paper 1453, US Printing Office (Denver); and from Snyder 1987: "Map Projections - A Working Manual", US Geological Survey Professional Paper 1395, US Printing Office (Denver, USA). You can obtain your own copy of both by contacting the U.S. Geological Survey, Federal Center, Box 25425, Denver, CO 80225 USA.

The mathematics of cartography have a long history, and the details are far trickier than the broad overview. For terrestrial (and, in general, planetary) cartography, the best reference datum is not a sphere but an oblate ellipsoid due to centrifugal force from the planet's rotation. Furthermore, because all rocky planets, including Earth, have randomly placed mass concentrations that affect the gravitational field, the reference gravitational isosurface (sea level on Earth) is even more complex than an ellipsoid and, in general, different ellipsoids have been used for different locations at the same time and for the same location at different times.

The transformations in this package use a spherical datum and hence include global distortion at about the 0.5% level for terrestrial maps (Earth's oblateness is ~1/300). This

is roughly equal to the dimensional precision of physical maps printed on paper (due to stretching and warping of the paper) but is significant at larger scales (e.g. for regional maps). If you need more precision than that, you will want to implement and use the ellipsoidal transformations from Snyder 1987 or another reference work on geodesy. A good name for that package would be `...::Cartography::Geodetic`.

GENERAL NOTES ON PERSPECTIVE AND SCIENTIFIC IMAGES

Cartographic transformations are useful for interpretation of scientific images, as all cameras produce projections of the celestial sphere onto the focal plane of the camera. A simple (single-element) optical system with a planar focal plane generates gnomonic images – that is to say, gnomonic projections of a portion of the celestial sphere near the paraxial direction. This is the projection that most consumer grade cameras produce.

Magnification in an optical system changes the angle of incidence of the rays on the focal plane for a given angle of incidence at the aperture. For example, a 10x telescope with a 2 degree field of view exhibits the same gnomonic distortion as a simple optical system with a 20 degree field of view. Wide-angle optics typically have magnification less than 1 ('fisheye lenses'), reducing the gnomonic distortion considerably but introducing "equidistant azimuthal" distortion – there's no such thing as a free lunch!

Because many solar-system objects are spherical, PDL::Transform::Cartography includes perspective projections for producing maps of spherical bodies from perspective views. Those projections are "t_vertical" and "t_perspective". They map between (lat,lon) on the spherical body and planar projected coordinates at the viewpoint. "t_vertical" is the vertical perspective projection given by Snyder, but "t_perspective" is a fully general perspective projection that also handles magnification correction.

TRANSVERSE & OBLIQUE PROJECTIONS; STANDARD OPTIONS

Oblique projections rotate the sphere (and graticule) to an arbitrary angle before generating the projection; transverse projections rotate the sphere exactly 90 degrees before generating the projection.

Most of the projections accept the following standard options, useful for making transverse and oblique projection maps.

- o, origin, Origin [default (0,0,0)]

  The origin of the oblique map coordinate system, in (old-theta, old-phi) coordinates.

- r, roll, Roll [default 0.0]

  The roll angle of the sphere about the origin, measured CW from (N = up) for reasonable values of phi and CW from (S = up) for unreasonable values of phi. This is equivalent to observer roll angle CCW from the same direction.

- u, unit, Unit [default 'degree']

  This is the name of the angular unit to use in the lon/lat coordinate system.

- b, B

  The "B" angle of the body – used for extraterrestrial maps. Setting this parameter is exactly equivalent to setting the phi component of the origin, and in fact overrides

it.

- l,L

  The longitude of the central meridian as observed – used for extraterrestrial maps. Setting this parameter is exactly equivalent to setting the theta component of the origin, and in fact overrides it.

- p,P

  The "P" (or position) angle of the body – used for extraterrestrial maps. This parameter is a synonym for the roll angle, above.

- bad, Bad, missing, Missing [default nan]

  This is the value that missing points get. Mainly useful for the inverse transforms. (This should work fine if set to BAD).

EXAMPLES

Draw a Mercator map of the world on-screen:

```
$w = pgswin();
$w->plot(with=>'polylines', earth_coast->apply(t_mercator)->clean_
 ↪lines);
```

Here, `earth_coast()` returns a 3xn ndarray containing (lon, lat, pen) values for the included world coastal outline; `t_mercator` converts the values to projected Mercator coordinates, and `clean_lines` breaks lines that cross the 180th meridian.

Draw a Mercator map of the world, with lon/lat at 10 degree intervals:

```
$w = pgswin();
$x = earth_coast()->glue(1,graticule(10,1));
$w->plot(with=>'polylines', $x->apply(t_mercator)->clean_lines);
```

This works just the same as the first example, except that a map graticule has been applied with interline spacing of 10 degrees lon/lat and inter-vertex spacing of 1 degree (so that each meridian contains 181 points, and each parallel contains 361 points).

NOTES

Currently angular conversions are rather simpleminded. A list of common conversions is present in the main constructor, which inserts a conversion constant to radians into the {params} field of the new transform. Something like Math::Convert::Units should be used instead to generate the conversion constant.

A cleaner higher-level interface is probably needed (see the examples); for example, earth_coast could return a graticule if asked, instead of needing one to be glued on.

The class structure is somewhat messy because of the varying needs of the different transformations. PDL::Transform::Cartography is a base class that interprets the origin options and sets up the basic machinery of the Transform. The conic projections have their own subclass, PDL::Transform::Conic, that interprets the standard parallels. Since the cylindrical and azimuthal projections are pretty simple, they are not subclassed.

AUTHOR

Copyright 2002, Craig DeForest (deforest@boulder.swri.edu). This module may be modified and distributed under the same terms as PDL itself. The module comes with NO WARRANTY.

The included digital world map is derived from the 1987 CIA World Map, translated to ASCII in 1988 by Joe Dellinger (geojoe@freeusp.org) and simplified in 1995 by Kirk Johnson (tuna@indra.com) for the program XEarth. The map comes with NO WARRANTY. An ASCII version of the map, and a sample PDL function to read it, may be found in the `examples` subdirectory of the PDL source distribution.

FUNCTIONS

The module exports both transform constructors ('t_<foo>') and some auxiliary functions (no leading 't_').

graticule

```
$lonlatp     = graticule(<grid-spacing>,<line-segment-size>);
$lonlatp     = graticule(<grid-spacing>,<line-segment-size>,1);
```

(Cartography) PDL constructor - generate a lat/lon grid.

Returns a grid of meridians and parallels as a list of vectors suitable for sending to PDL::Graphics::Simple for plotting. The grid is in degrees in (theta, phi) coordinates – this is (E lon, N lat) for terrestrial grids or (RA, dec) for celestial ones. You must then transform the graticule in the same way that you transform the map.

You can attach the graticule to a vector map using the syntax:

```
$out = graticule(10,2)->glue(1,$map);
```

In array context you get back a 2-element list containing an ndarray of the (theta,phi) pairs and an ndarray of the pen values (1 or 0) suitable for calling PDL::Graphics::Simple. In scalar context the two elements are combined into a single ndarray.

The pen values associated with the graticule are negative, which will cause PDL::Graphics::Simple to plot them as hairlines.

If a third argument is given, it is a hash of options, which can be:

- nan - if true, use two columns instead of three, and separate lines with a 'nan' break
- lonpos - if true, all reported longitudes are positive (0 to 360) instead of (-180 to 180).
- dup - if true, the meridian at the far boundary is duplicated.

earth_coast

```
$x = earth_coast()
```

(Cartography) PDL constructor - coastline map of Earth

Returns a vector coastline map based on the 1987 CIA World Coastline database (see author information). The vector coastline data are in plate carree format so they can be converted to other projections via the *apply* method and cartographic transforms, and are suitable for plotting with the polylines plot type in the PDL::Graphics::Simple output library: the first dimension is (X,Y,pen) with breaks having a pen value of 0 and hairlines having negative pen values. The second dimension broadcasts over all the points in the data set.

The vector map includes lines that pass through the antipodean meridian, so if you want to plot it without reprojecting, you should run it through "clean_lines" first:

```
$w = pgswin();
$w->plot(with=>'polylines',
  earth_coast->clean_lines);     # plot plate carree map of world
$w->plot(with=>'polylines',
  earth_coast->apply(t_gnomonic)->clean_lines)# plot gnomonic map of␣
↪world
```

`earth_coast` is just a quick-and-dirty way of loading the file "earth_coast.vec.fits" that is part of the normal installation tree.

earth_image

```
$rgb = earth_image()
```

(Cartography) PDL constructor - RGB pixel map of Earth

Returns an RGB image of Earth based on data from the MODIS instrument on the NASA EOS/Terra satellite. (You can get a full-resolution image from http://earthobservatory.nasa.gov/Newsroom/BlueMarble/). The image is a plate carree map, so you can convert it to other projections via the *map* method and cartographic transforms.

This is just a quick-and-dirty way of loading the earth-image files that are distributed along with PDL.

earth_shape

```
$fits_shape = earth_shape()
```

(Cartography) PDL constructor - height map of Earth

Returns a height map of Earth based on data from the General Bathymetric Chart of the Oceans (GEBCO) produced by the British Oceanographic Data Centre. (You can get a full-resolution image from http://visibleearth.nasa.gov/view.php?id=73934). The image is a plate carree map, so you can convert it to other projections via the *map* method and cartographic transforms. The data is from 8-bit grayscale (so only 256 levels), but is returned as float, values in Earth radii, in dimensions (2048,1024), like "earth_image". The range represents a span of 6400m, so Everest and the Marianas Trench are not accurately represented.

```
Value       Hex value   Float    From centre in km   Float as radius
Base        00          0.0      6370.69873km        0.99995
```

(continues on next page)

(continued from previous page)

```
Sea level   0C          0.04705  6371km              1.0
Highest     FF          1.0      6377.09863km        1.00096
```

Code:

```
$shape = earth_shape();
```

raster2fits

```
$pdl_fits = raster2fits($pdl, \@axislabels, \@axisunits, \
 ↪@axisranges);
$pdl_fits = raster2fits($pdl, @PDL::Transform::Cartography::PLATE_
 ↪CARREE);
```

Convert a raster ([3,]x,y) to FITS (x,y[,3]), with a suitable header for the given parameters.

clean_lines

```
$x = clean_lines(t_mercator->apply(scalar(earth_coast())));
$x = $lines->clean_lines; # same as above, both "first (scalar) form
 ↪"
$x = clean_lines($line_pen[,threshold][,opt]); # also same but␣
 ↪threshold given
$x = clean_lines($line,$pen[,threshold][,opt]); # "second (list)␣
 ↪form"
```

(Cartography) PDL method - remove projection irregularities

`clean_lines` massages vector data to remove jumps due to singularities in the transform.

In the first (scalar) form, `$line_pen` contains both (X,Y) points and pen values in the 3rd column suitable to be fed to the polylines plot type in the PDL::Graphics::Simple.

In the second (list) form, `$lines` contains the (X,Y) points and `$pen` contains the pen values, in one less dimension than the `$lines`.

`clean_lines` assumes that all the outline polylines are local – that is to say, there are no large jumps. Any jumps larger than a threshold size are broken by setting the appropriate pen values to 0.

The `threshold` parameter sets the relative size of the largest jump, relative to the map range (as determined by a min/max operation). The default size is 0.1. If `threshold` is greater than or equal to 1, lines will not be broken based on point separation.

The following options are interpreted:

- or, orange, output_range, Output_Range

  ```
  $lp = $lp->clean_lines(1.1,{or=>[[178.9,179.7], [62.8,64.5]]})
  ```

This sets the window of output space, similar to that in *"map" in PDL::Transform*. As there, it specifies a quadrilateral in output space. Any points not in that will be removed from the output, and line breaks (0 pen values) will be inserted before.

Because this returns a selection of the inputs, it will not broadcast.

- fn, filter_nan

  Defaults to true. Will break before any points with bad/NaN/Inf coordinates, and remove those points from the output.

  Because this returns a selection of the inputs, it will not broadcast.

NOTES

This almost never catches stuff near the apex of cylindrical maps, because the anomalous vectors get arbitrarily small. This could be improved somewhat by looking at individual runs of the pen and using a relative length scale that is calibrated to the rest of each run. it is probably not worth the computational overhead.

t_unit_sphere

```
$t = t_unit_sphere(<options>);
```

(Cartography) 3-D globe projection (conformal; authalic)

This is similar to the inverse of *t_spherical*, but the inverse transform projects 3-D coordinates onto the unit sphere, yielding only a 2-D (lon/lat) output. Similarly, the forward transform deprojects 2-D (lon/lat) coordinates onto the surface of a unit sphere.

The cartesian system has its Z axis pointing through the pole of the (lon,lat) system, and its X axis pointing through the equator at the prime meridian.

Unit sphere mapping is unusual in that it is both conformal and authalic. That is possible because it properly embeds the sphere in 3-space, as a notional globe.

This is handy as an intermediate step in lots of transforms, as Cartesian 3-space is cleaner to work with than spherical 2-space.

Higher dimensional indices are preserved, so that "rider" indices (such as pen value) are propagated.

There is no oblique transform for t_unit_sphere, largely because it's so easy to rotate the output using t_linear once it's out into Cartesian space. In fact, the other projections implement oblique transforms by *wrapping t_linear* with "t_unit_sphere".

OPTIONS:

- radius, Radius (default 1.0)

  The radius of the sphere, for the inverse transform. (Radius is ignored in the forward transform). Defaults to 1.0 so that the resulting Cartesian coordinates are in units of "body radii".

t_rot_sphere

```
$t = t_rot_sphere({origin=>[<theta>,<phi>],roll=>[<roll>]});
```

(Cartography) Generate oblique projections

You feed in the origin in (theta,phi) and a roll angle, and you get back out (theta', phi') coordinates. This is useful for making oblique or transverse projections: just compose t_rot_sphere with your favorite projection and you get an oblique one.

Most of the projections automagically compose themselves with t_rot_sphere if you feed in an origin or roll angle.

t_rot_sphere converts the base plate carree projection (straight lon, straight lat) to a Cassini projection.

OPTIONS

- STANDARD POSITIONAL OPTIONS

t_orthographic

```
$t = t_orthographic(<options>);
```

(Cartography) Ortho. projection (azimuthal; perspective)

This is a perspective view as seen from infinite distance. You can specify the sub-viewer point in (lon,lat) coordinates, and a rotation angle of the map CW from (north=up). This is equivalent to specify viewer roll angle CCW from (north=up).

t_orthographic is a convenience interface to t_unit_sphere – it is implemented as a composition of a t_unit_sphere call, a rotation, and a slice.

[*] In the default case where the near hemisphere is mapped, the inverse exists. There is no single inverse for the whole-sphere case, so the inverse transform superimposes everything on a single hemisphere. If you want an invertible 3-D transform, you want "t_unit_sphere".

OPTIONS

- STANDARD POSITIONAL OPTIONS
- m, mask, Mask, h, hemisphere, Hemisphere [default 'near']

  The hemisphere to keep in the projection (see *PDL::Transform::Cartography*).

NOTES

Alone of the various projections, this one does not use "t_rot_sphere" to handle the standard options, because the cartesian coordinates of the rotated sphere are already correctly projected – t_rot_sphere would put them back into (theta', phi') coordinates.

t_carree

```
$t = t_carree(<options>);
```

(Cartography) Plate Carree projection (cylindrical; equidistant)

This is the simple Plate Carree projection – also called a "lat/lon plot". The horizontal axis is theta; the vertical axis is phi. This is a no-op if the angular unit is radians; it is a simple scale otherwise.

OPTIONS

- STANDARD POSITIONAL OPTIONS
- s, std, standard, Standard (default 0)

  The standard parallel where the transformation is conformal. Conformality is achieved by shrinking of the horizontal scale to match the vertical scale (which is correct everywhere).

t_mercator

```
$t = t_mercator(<options>);
```

(Cartography) Mercator projection (cylindrical; conformal)

This is perhaps the most famous of all map projections: meridians are mapped to parallel vertical lines and parallels are unevenly spaced horizontal lines. The poles are shifted to +/- infinity. The output values are in units of globe-radii for easy conversion to kilometers; hence the horizontal extent is -pi to pi.

You can get oblique Mercator projections by specifying the `origin` or `roll` options; this is implemented via "t_rot_sphere".

OPTIONS

- STANDARD POSITIONAL OPTIONS
- c, clip, Clip (default 75 [degrees])

  The north/south clipping boundary of the transformation. Because the poles are displaced to infinity, many applications require a clipping boundary. The value is in whatever angular unit you set with the standard 'units' option. The default roughly matches interesting landforms on Earth. For no clipping at all, set c=>0. For asymmetric clipping, use a 2-element list ref or ndarray.

- s, std, Standard (default 0)

  This is the parallel at which the map has correct scale. The scale is also correct at the parallel of opposite sign.

t_utm

```
$t = t_utm(<zone>,<options>);
```

(Cartography) Universal Transverse Mercator projection (cylindrical)

This is the internationally used UTM projection, with 2 subzones (North/South). The UTM zones are parametrized individually, so if you want a Zone 30 map you should use `t_utm(30)`. By default you get the northern subzone, so that locations in the southern

hemisphere get negative Y coordinates. If you select the southern subzone (with the "subzone=>-1" option), you get offset southern UTM coordinates.

The 20-subzone military system is not yet supported. If/when it is implemented, you will be able to enter "subzone=>[a-t]" to select a N/S subzone.

Note that UTM is really a family of transverse Mercator projections with different central meridia. Each zone properly extends for three degrees of longitude on either side of its appropriate central meridian, with Zone 1 being centered at -177 degrees longitude (177 west). Properly speaking, the zones only extend from 80 degrees south to 84 degrees north; but this implementation lets you go all the way to 90 degrees. The default UTM coordinates are meters. The origin for each zone is on the equator, at an easting of -500,000 meters.

The default output units are meters, assuming that you are wanting a map of the Earth. This will break for bodies other than Earth (which have different radii and hence different conversions between lat/lon angle and meters).

The standard UTM projection has a slight reduction in scale at the prime meridian of each zone: the transverse Mercator projection's standard "parallels" are 180km e/w of the central meridian. However, many Europeans prefer the "Gauss-Kruger" system, which is virtually identical to UTM but with a normal tangent Mercator (standard parallel on the prime meridian). To get this behavior, set "gk=>1".

Like the rest of the PDL::Transform::Cartography package, t_utm uses a spherical datum rather than the "official" ellipsoidal datums for the UTM system.

This implementation was derived from the rather nice description by Denis J. Dean, located on the web at: http://www.cnr.colostate.edu/class_info/nr502/lg3/datums_coordinates/utm.html

OPTIONS

- STANDARD OPTIONS

  (No positional options – Origin and Roll are ignored)

- ou, ounit, OutputUnit (default 'meters')

  (This is likely to become a standard option in a future release) The unit of the output map. By default, this is 'meters' for UTM, but you may specify 'deg' or 'km' or even (heaven help us) 'miles' if you prefer.

- sz, subzone, SubZone (default 1)

  Set this to -1 for the southern hemisphere subzone. Ultimately you should be able to set it to a letter to get the corresponding military subzone, but that's too much effort for now.

- gk, gausskruger (default 0)

  Set this to 1 to get the (European-style) tangent-plane Mercator with standard parallel on the prime meridian. The default of 0 places the standard parallels 180km east/west of the prime meridian, yielding better average scale across the zone. Setting gk=>1 makes the scale exactly 1.0 at the central meridian, and >1.0 everywhere else on the projection. The difference in scale is about 0.3%.

t_sin_lat

```
$t = t_sin_lat(<options>);
```

(Cartography) Cyl. equal-area projection (cyl.; authalic)

This projection is commonly used in solar Carrington plots; but not much for terrestrial mapping.

OPTIONS

- STANDARD POSITIONAL OPTIONS
- s,std, Standard (default 0)

  This is the parallel at which the map is conformal. It is also conformal at the parallel of opposite sign. The conformality is achieved by matched vertical stretching and horizontal squishing (to achieve constant area).

t_sinusoidal

```
$t = t_sinusoidal(<options>);
```

(Cartography) Sinusoidal projection (authalic)

Sinusoidal projection preserves the latitude scale but scales longitude according to cos(lat); in this respect it is the companion to "t_sin_lat", which is also authalic but preserves the longitude scale instead.

OPTIONS

- STANDARD POSITIONAL OPTIONS

t_conic

```
$t = t_conic(<options>)
```

(Cartography) Simple conic projection (conic; equidistant)

This is the simplest conic projection, with parallels mapped to equidistant concentric circles. It is neither authalic nor conformal. This transformation is also referred to as the "Modified Transverse Mercator" projection in several maps of Alaska published by the USGS; and the American State of New Mexico re-invented the projection in 1936 for an official map of that State.

OPTIONS

- STANDARD POSITIONAL OPTIONS
- s, std, Standard (default 29.5, 45.5)

  The locations of the standard parallel(s) (where the cone intersects the surface of the sphere). If you specify only one then the other is taken to be the nearest pole. If you specify both of them to be one pole then you get an equidistant azimuthal map. If you specify both of them to be opposite and equidistant from the equator you get a Plate Carree projection.

t_albers

```
$t = t_albers(<options>)
```

(Cartography) Albers conic projection (conic; authalic)

This is the standard projection used by the US Geological Survey for sectionals of the 48 contiguous United States of America.

The projection reduces to the Lambert equal-area conic (infrequently used and not to be confused with the Lambert conformal conic, "t_lambert"!) if the pole is used as one of the two standard parallels.

Notionally, this is a conic projection onto a cone that intersects the sphere at the two standard parallels; it works best when the two parallels straddle the region of interest.

OPTIONS

- STANDARD POSITIONAL OPTIONS
- s, std, standard, Standard (default (29.5,45.5))

  The locations of the standard parallel(s). If you specify only one then the other is taken to be the nearest pole and a Lambert Equal-Area Conic map results. If you specify both standard parallels to be the same pole, then the projection reduces to the Lambert Azimuthal Equal-Area map as a special case. (Note that "t_lambert" is Lambert's Conformal Conic, the most commonly used of Lambert's projections.)

  The default values for the standard parallels are those chosen by Adams for maps of the lower 48 US states: (29.5,45.5). The USGS recommends (55,65) for maps of Alaska and (8,18) for maps of Hawaii – these latter are chosen to also include the Canal Zone and Philippine Islands farther south, which is why both of those parallels are south of the Hawaiian islands.

  The transformation reduces to the cylindrical equal-area (sin-lat) transformation in the case where the standard parallels are opposite and equidistant from the equator, and in fact this is implemented by a call to t_sin_lat.

t_lambert

```
$t = t_lambert(<options>);
```

(Cartography) Lambert conic projection (conic; conformal)

Lambert conformal conic projection is widely used in aeronautical charts and state base maps published by the USA's FAA and USGS. It's especially useful for mid-latitude charts. In particular, straight lines approximate (but are not exactly) great circle routes of up to ~2 radians.

The default standard parallels are 33 and 45 to match the USGS state 1:500,000 base maps of the United States. At scales of 1:500,000 and larger, discrepancies between the spherical and ellipsoidal projections become important; use care with this projection on spheres.

OPTIONS

- STANDARD POSITIONAL OPTIONS

- s, std, standard, Standard (default (33,45))

  The locations of the standard parallel(s) for the conic projection. The transform reduces to the Mercator projection in the case where the standard parallels are opposite and equidistant from the equator, and in fact this is implemented by a call to t_mercator.

- c, clip, Clip (default [-75,75])

  Because the transform is conformal, the distant pole is displaced to infinity. Many applications require a clipping boundary. The value is in whatever angular unit you set with the standard 'unit' option. For consistency with "t_mercator", clipping works the same way even though in most cases only one pole needs it. Set this to 0 for no clipping at all.

t_stereographic

```
$t = t_stereographic(<options>);
```

(Cartography) Stereographic projection (az.; conf.; persp.)

The stereographic projection is a true perspective (planar) projection from a point on the spherical surface opposite the origin of the map.

OPTIONS

- STANDARD POSITIONAL OPTIONS
- c, clip, Clip (default 120)

  This is the angular distance from the center to the edge of the projected map. The default 120 degrees gives you most of the opposite hemisphere but avoids the hugely distorted part near the antipodes.

t_gnomonic

```
$t = t_gnomonic(<options>);
```

(Cartography) Gnomonic (focal-plane) projection (az.; persp.)

The gnomonic projection projects a hemisphere onto a tangent plane. It is useful in cartography for the property that straight lines are great circles; and it is useful in scientific imaging because it is the projection generated by a simple optical system with a flat focal plane.

OPTIONS

- STANDARD POSITIONAL OPTIONS
- c, clip, Clip (default 75)

  This is the angular distance from the center to the edge of the projected map. The default 75 degrees gives you most of the hemisphere but avoids the hugely distorted part near the horizon.

t_az_eqd

```
$t = t_az_eqd(<options>);
```

(Cartography) Azimuthal equidistant projection (az.; equi.)

Basic azimuthal projection preserving length along radial lines from the origin (meridians, in the original polar aspect). Hence, both azimuth and distance are correct for journeys beginning at the origin.

Applied to the celestial sphere, this is the projection made by fisheye lenses; it is also the projection into which `t_vertical` puts perspective views.

The projected plane scale is normally taken to be planetary radii; this is useful for cartographers but not so useful for scientific observers. Setting the 't=>1' option causes the output scale to shift to camera angular coordinates (the angular unit is determined by the standard 'Units' option; default is degrees).

OPTIONS

- STANDARD POSITIONAL OPTIONS
- c, clip, Clip (default 180 degrees)

  The largest angle relative to the origin. Default is the whole sphere.

t_az_eqa

```
$t = t_az_eqa(<options>);
```

(Cartography) Azimuthal equal-area projection (az.; auth.)

OPTIONS

- STANDARD POSITIONAL OPTIONS
- c, clip, Clip (default 180 degrees)

  The largest angle relative to the origin. Default is the whole sphere.

t_hammer, t_aitoff

(Cartography) Hammer/Aitoff elliptical projection (az.; auth.)

The Hammer/Aitoff projection is often used to display the Celestial sphere. It is mathematically related to the Lambert Azimuthal Equal-Area projection ("t_az_eqa"), and maps the sphere to a 2:1 ellipse, with vertical radius sqrt(2) and horizontal radius of 2 sqrt(2).

OPTIONS

- STANDARD POSITIONAL OPTIONS

t_vertical, t_zenithal

```
$t = t_vertical(<options>);
```

(Cartography) Vertical perspective projection (az.; persp.)

Vertical (also called "zenithal") perspective projection is a generalization of gnomonic and stereographic projection, and a special case of perspective projection. It is a projection from the sphere onto a tangent plane from a point at the camera location.

OPTIONS

- STANDARD POSITIONAL OPTIONS
- m, mask, Mask, h, hemisphere, Hemisphere [default 'near']

  The hemisphere to keep in the projection (see *PDL::Transform::Cartography*).
- r0, R0, radius, d, dist, distance [default 2.0]

  The altitude of the focal plane above the center of the sphere. The default places the point of view one radius above the surface.
- t, telescope, Telescope, cam, Camera (default '')

  If this is set, then the central scale is in telescope or camera angular units rather than in planetary radii. The angular units are parsed as with the normal 'u' option for the lon/lat specification. If you specify a non-string value (such as 1) then you get telescope-frame radians, suitable for working on with other transformations.
- f, fish, fisheye (default '')

  If this is set then the output is in azimuthal equidistant coordinates instead of in tangent-plane coordinates. This is a convenience function for '(t_az_eqd) x !(t_gnomonic) x (t_vertical)'.

t_perspective

```
$t = t_perspective(<options>);
```

(Cartography) Arbitrary perspective projection

Perspective projection onto a focal plane from an arbitrary location within or without the sphere, with an arbitrary central look direction, and with correction for magnification within the optical system.

In the forward direction, t_perspective generates perspective views of a sphere given (lon/lat) mapping or vector information. In the reverse direction, t_perspective produces (lon/lat) maps from aerial or distant photographs of spherical objects.

Viewpoints outside the sphere treat the sphere as opaque by default, though you can use the 'm' option to specify either the near or far surface (relative to the origin). Viewpoints below the surface treat the sphere as transparent and undergo a mirror reversal for consistency with projections that are special cases of the perspective projection (e.g. t_gnomonic for r0=0 or t_stereographic for r0=-1).

Magnification correction handles the extra edge distortion due to higher angles between the focal plane and focused rays within the optical system of your camera. If you do not

happen to know the magnification of your camera, a simple rule of thumb is that the magnification of a reflective telescope is roughly its focal length (plate scale) divided by its physical length; and the magnification of a compound refractive telescope is roughly twice its physical length divided by its focal length. Simple optical systems with a single optic have magnification = 1. Fisheye lenses have magnification < 1.

This transformation was derived by direct geometrical calculation rather than being translated from Voxland & Snyder.

OPTIONS

- STANDARD POSITIONAL OPTIONS

  As always, the 'origin' field specifies the sub-camera point on the sphere.

  The 'roll' option is the roll angle about the sub-camera point, for consistency with the other projections.

- p, ptg, pointing, Pointing (default (0,0,0))

  The pointing direction, in (horiz. offset, vert. offset, roll) of the camera relative to the center of the sphere. This is a spherical coordinate system with the origin pointing directly at the sphere and the pole pointing north in the pre-rolled coordinate system set by the standard origin. It's most useful for space-based images taken some distance from the body in question (e.g. images of other planets or the Sun).

  Be careful not to confuse 'p' (pointing) with 'P' (P angle, a standard synonym for roll).

- c, cam, camera, Camera (default undef)

  Alternate way of specifying the camera pointing, using a spherical coordinate system with poles at the zenith (positive) and nadir (negative) – this is useful for aerial photographs and such, where the point of view is near the surface of the sphere. You specify (azimuth from N, altitude from horizontal, roll from vertical=up). If you specify pointing by this method, it overrides the 'pointing' option, above. This coordinate system is most useful for aerial photography or low-orbit work, where the nadir is not necessarily the most interesting part of the scene.

- r0, R0, radius, d, dist, distance [default 2.0]

  The altitude of the point of view above the center of the sphere. The default places the point of view 1 radius above the surface. Do not confuse this with 'r', the standard origin roll angle! Setting r0 < 1 gives a viewpoint inside the sphere. In that case, the images are mirror-reversed to preserve the chirality of the perspective. Setting r0=0 gives gnomonic projections; setting r0=-1 gives stereographic projections. Setting r0 < -1 gives strange results.

- iu, im_unit, image_unit, Image_Unit (default 'degrees')

  This is the angular units in which the viewing camera is calibrated at the center of the image.

- mag, magnification, Magnification (default 1.0)

  This is the magnification factor applied to the optics – it affects the amount of tangent-plane distortion within the telescope. 1.0 yields the view from a simple optical system; higher values are telescopic, while lower values are wide-angle

(fisheye). Higher magnification leads to higher angles within the optical system, and more tangent-plane distortion at the edges of the image. The magnification is applied to the incident angles themselves, rather than to their tangents (simple two-element telescopes magnify tan(theta) rather than theta itself); this is appropriate because wide-field optics more often conform to the equidistant azimuthal approximation than to the tangent plane approximation. If you need more detailed control of the relationship between incident angle and focal-plane position, use mag=1.0 and compose the transform with something else to tweak the angles.

- m, mask, Mask, h, hemisphere, Hemisphere [default 'near']

  'hemisphere' is by analogy to other cartography methods although the two regions to be selected are not really hemispheres.

- f, fov, field_of_view, Field_Of_View [default 60 degrees]

  The field of view of the telescope – sets the crop radius on the focal plane. If you pass in a scalar, you get a circular crop. If you pass in a 2-element list ref, you get a rectilinear crop, with the horizontal 'radius' and vertical 'radius' set separately.

EXAMPLES

Model a camera looking at the Sun through a 10x telescope from Earth (~230 solar radii from the Sun), with an 0.5 degree field of view and a solar P (roll) angle of 30 degrees, in February (sub-Earth solar latitude is 7 degrees south). Convert a solar FITS image taken with that camera to a FITS lon/lat map of the Sun with 20 pixels/degree latitude:

```
# Define map output header (no need if you don't want a FITS output␣
 ↪map)
$maphdr = {NAXIS1=>7200,NAXIS2=>3600,              # Size of image
           CTYPE1=>longitude,CTYPE2=>latitude,    # Type of axes
           CUNIT1=>deg,CUNIT2=>deg,               # Unit of axes
           CDELT1=>0.05,CDELT2=>0.05,             # Scale of axes
           CRPIX1=>3601,CRPIX2=>1801,             # Center of map
           CRVAL1=>0,CRVAL2=>0                    # (lon,lat) of␣
 ↪center
           };

# Set up the perspective transformation, and apply it.
$t = t_perspective(r0=>229,fov=>0.5,mag=>10,P=>30,B=>-7);
$map = $im->map( $t , $maphdr );
```

Draw an aerial-view map of the Chesapeake Bay, as seen from a sounding rocket at an altitude of 100km, looking NNE from ~200km south of Washington (the radius of Earth is 6378 km; Washington D.C. is at roughly 77W,38N). Superimpose a linear coastline map on a photographic map.

```
$x = graticule(1,0.1)->glue(1,earth_coast());
$t = t_perspective(r0=>6478/6378.0,fov=>60,cam=>[22.5,-20],o=>[-77,
 ↪36])
$w = pgswin(size=>[10,6],J=>1);
$w->plot(with=>'fits', earth_image()->map($t,[800,500],{m=>linear}),
  with=>'polylines', $x->apply($t)->clean_lines);
```

Model a 5x telescope looking at Betelgeuse with a 10 degree field of view (since

the telescope is looking at the Celestial sphere, r is 0 and this is just an expensive modified-gnomonic projection).

```
$t = t_perspective(r0=>0,fov=>10,mag=>5,o=>[88.79,7.41])
```

PDL::Types

define fundamental PDL Datatypes

SYNOPSIS

```
use PDL::Types;

$pdl = ushort( 2.0, 3.0 );
print "The actual c type used to store ushort's is '" .
   $pdl->type->realctype() . "'\n";
The actual c type used to store ushort's is 'unsigned short'
```

DESCRIPTION

Internal module - holds all the PDL Type info. The type info can be accessed easily using the PDL::Type object returned by the *type* method as shown in the synopsis.

Skip to the end of this document to find out how to change the set of types supported by PDL.

FUNCTIONS

A number of functions are available for module writers to get/process type information. These are used in various places (e.g. PDL::PP, PDL::Core) to generate the appropriate type loops, etc.

typesrtkeys

Returns an array of keys of typehash sorted in order of type complexity

```
pdl> @typelist = PDL::Types::typesrtkeys;
pdl> print @typelist;
PDL_SB PDL_B PDL_S PDL_US PDL_L PDL_UL PDL_IND PDL_ULL PDL_LL PDL_F␣
 ↪PDL_D PDL_LD PDL_CF PDL_CD PDL_CLD
```

ppdefs

Returns an array of pp symbols for all real types. This informs the default GenericTypes for pp_def functions, making support for complex types require an “opt-in”.

```
pdl> print PDL::Types::ppdefs
A B S U L K N P Q F D E
```

ppdefs_complex

Returns an array of pp symbols for all complex types.

```
pdl> print PDL::Types::ppdefs_complex
G C H
```

ppdefs_all

Returns an array of pp symbols for all types including complex.

```
pdl> print PDL::Types::ppdefs_all
A B S U L K N P Q F D E G C H
```

typesynonyms

return type related synonym definitions to be included in pdl.h . This routine must be updated to include new types as required. Mostly the automatic updating should take care of the vital things.

PDL TYPES OVERVIEW

As of 2.065, PDL supports these types:

- SByte

  Signed 8-bit value.

- Byte

  Unsigned 8-bit value.

- Short

  Signed 16-bit value.

- UShort

  Unsigned 16-bit value.

- Long

  Signed 32-bit value.

- ULong

  Unsigned 32-bit value.

- Indx

  Signed value, same size as a pointer on the system in use.

- ULongLong

  Unsigned 64-bit value.

- LongLong

  Signed 64-bit value.

- Float

  IEEE 754 single-precision real floating-point value.

- Double

  IEEE 754 double-precision real value.

- LDouble

  A C99 “long double”, defined as “at least as precise as a double”, but often more precise.

- CFloat

  A C99 complex single-precision floating-point value.

- CDouble

  A C99 complex double-precision floating-point value.

- CLDouble

  A C99 complex “long double” - see above for description.

As of 2.099, documentation for PDL::Type is separate. See there for more.

DEVELOPER NOTES ON ADDING/REMOVING TYPES

You can change the types that PDL knows about by editing entries in the definition of the variable `@types` that appears close to the top of the file `lib/PDL/Types.pm`.

Format of a type entry

Each entry in the `@HASHES` array is a hash reference. Here is an example taken from the actual code that defines the `ushort` type:

```
{
  identifier => 'US',
  onecharident => 'U',   # only needed if different from identifier
  ctype => 'PDL_Ushort',
  realctype => 'unsigned short',
  ppforcetype => 'ushort',
  usenan => 0,
  packtype => 'S*',
  defbval => 'USHRT_MAX',
  real=>1,
  integer=>1,
  unsigned=>1,
},
```

Before we start to explain the fields please take this important message on board: *entries must be listed in order of increasing complexity*. This is critical to ensure that PDL’s type conversion works correctly. Basically, a less complex type will be converted to a more complex type as required.

Fields in a type entry

Each type entry has a number of required and optional entry.

A list of all the entries:

- identifier

  Required. A short sequence of uppercase letters that identifies this type uniquely. More than three characters is probably overkill.

- onecharident

  Optional. Only required if the `identifier` has more than one character. This should be a unique uppercase character that will be used to reference this type in PP macro expressions of the `TBSULFD` type - see *"$T" in PDL::PP*.

- ctype

  Required. The `typedefed` name that will be used to access this type from C code.

- realctype

  Required. The C compiler type that is used to implement this type. For portability reasons this one might be platform dependent.

- ppforcetype

  Required. The type name used in PP signatures to refer to this type.

- usenan

  Required. Flag that signals if this type has to deal with NaN issues. Generally only required for floating point types.

- packtype

  Required. The Perl pack type used to pack Perl values into the machine representation for this type. For details see `perldoc -f pack`.

- integer

  Required. Boolean - is this an integer type?

- unsigned

  Required. Boolean - is this an unsigned type?

- real

  Required. Boolean - is this a real (not complex) type?

- realversion

  String - the real version of this type (e.g. cdouble -> 'D').

- complexversion

  String - the complex version of this type (e.g. double -> 'C').

Also have a look at the entries at the top of `lib/PDL/Types.pm`.

Other things you need to do

You need to check modules that do I/O (generally in the IO part of the directory tree). In the future we might add fields to type entries to automate this. This requires changes to those IO modules first though.

You may also need to update any type macros in PP files (i.e. `$TBSULFD...`) to reflect the new type - PP will throw an error if you have a `$T...` macro which misses types supported by the operation.

PDL::Ufunc

Reductions of PDL's ndarrays: sums, products, means, extrema, medians, percentiles, logical and bitwise folds, running sums and products, and sorting.

```
use PDL;
my $m = pdl([3, 1, 2], [9, 7, 8]);
print $m->sum, " ", $m->avg, " ", $m->max, "\n";         # 30 5 9
print sumover($m), " ", minimum($m), "\n";               # [6 24] [1␣
 ↪7]
print $m->transpose->sumover, "\n";                      # [12 8 10]
print qsort(pdl(3, 1, 2)), " ", qsorti(pdl(3, 1, 2)), "\n";   # [1 2␣
 ↪3] [1 2 0]
my ($min, $max) = $m->minmax;
print "$min $max ", cumusumover(pdl(1, 2, 3)), "\n";     # 1 9 [1 3␣
 ↪6]
print pct(pdl(1 .. 9), 0.5), " ", medover(pdl(1, 2, 3, 4)), "\n";
 ↪# 5 2.5
```

The functions come in two shapes. The `...over` functions reduce the FIRST dimension and broadcast over the others: a 3x2 matrix gives 2 sums. To reduce another dimension, move it first (`transpose`, `mv`, `xchg`). The whole-array functions (`sum`, `avg`, `min`, `median`, `and`, ...) reduce every value to one, a 0-dim ndarray. Bad values are skipped throughout.

This is PDL::Ufunc of PDL 2.106. The kernels are PDL's own, through pperl's engine; upstream's Perl is used unchanged for `minmax`, `minimum_n_ind`/`maximum_n_ind` (the n smallest or largest) and the synonyms `minover`, `maxover`, `minover_ind`, `maxover_ind`, `minover_n_ind`, `maxover_n_ind`, `minmaxover`, `avgover`, `davgover`, `any` and `all`.

inner

`inner($a, $b)` – inner (dot) product along the first dimension.

Given two ndarrays with matching first-axis length n, computes `c = sum_i a_i * b_i`. For 1-D inputs this is the standard dot product, a 0-dimensional result; for higher-rank inputs the contraction is over axis 0 and the remaining axes broadcast.

Signature

```
$c = inner($a, $b);        # function form
$c = $a->inner($b);        # method form
inner($a, $b, $c);         # writes into $c, returns nothing
```

Bad values

If any element of `$a` or `$b` along the first dimension is bad, that result is bad.

Examples

```
use PDL;
my $a = pdl([1, 2, 3]);
my $b = pdl([4, 5, 6]);
print inner($a, $b);            # 32   (1*4 + 2*5 + 3*6)
print $a->inner($b);            # 32   (method form)

# Scalar-on-vector also works (scalar broadcasts to each element)
print inner(pdl([1, 2, 3]), 2); # 12   (1*2 + 2*2 + 3*2)
```

Errors

Croaks when the dimensions do not match, or with the usage when given fewer than two or more than three arguments.

zeroes

Core.xs `new_from_specification(invoc, ...)`, `ALIAS: PDL::zeroes = 1`.

Also available as `PDL::new_from_specification`.

`ST(0)` is ALWAYS the invocant - that is the whole character of this sub and the reason upstream keeps a second, lenient one in Core.pm. `PDL::zeroes(4,4)` therefore means "the class 4, one dimension of 4", and upstream reaches `Can't locate object method "initialize" via package "4"`. It is not a synonym for `zeroes(4,4)`, and treating it as one - which we did, by skipping the invocant only when it was the literal string "PDL" - silently accepted a call perl5 refuses.

get_dataref

Return a reference to the scalar holding the ndarray's raw bytes.

my $ref = $pdl->get_dataref; # SCALAR ref

The scalar holds the ndarray's data in machine format - the same bytes you would find in a file written by `PDL::IO::FastRaw`. For all but the smallest ndarrays (up to 32 bytes of data) it IS the data: writing into `$$ref` writes the ndarray. Call `$pdl->upd_data` afterwards in either case; it tells views of `$pdl` about the change, and copies the bytes back for a small ndarray, whose scalar is a copy.

The ndarray is made physical first, so a slice or other view yields the bytes of its own elements rather than its parent's.

upd_data

Tell the ndarray that the scalar `get_dataref` returned was changed.

$pdl->upd_data; # after writing into the scalar $pdl->upd_data(1); # after replacing the scalar's buffer

Any views derived from `$pdl` are told their parent changed, so they see the new data. For a small ndarray, whose scalar is a copy, the bytes are copied back and the scalar is released; a true argument keeps the scalar and makes it the ndarray's data instead.

For an ndarray whose data is the scalar, the ndarray is pointed at the scalar's buffer again - which is what makes a buffer that moved, such as one a file was mapped into, take effect.

Croaks with `Trying to touch dataref of magical (mmaped?) pdl` for an ndarray whose data is owned elsewhere, and with `Trying to upd_data but datasv now length N instead of M` if a small ndarray's scalar was resized.

at

Return a single element of an ndarray as a plain Perl scalar.

`@pos` is a coordinate list whose length matches the number of dimensions of `$pdl`. Indices may be negative (counted from the far end, so `-1` is the last element along that axis). Missing trailing indices default to 0; extra trailing indices past `$pdl->ndims` are ignored.

The value you get back is current even if `$pdl` is a view onto another ndarray that has since been written to. Integer-typed ndarrays give you an integer, float-typed ones a float.

Signature

```
$z = at($pdl, @pos);        # function form
$z = $pdl->at(@pos);        # method form
```

Broadcasting

`at` does not broadcast. It is a scalar-valued accessor: supply exactly one coordinate tuple.

Bad values

If `$pdl` has its bad-value flag set and the element fetched is the bad value, `at` returns the string `"BAD"`. With the flag clear the raw number is returned regardless of what it happens to be.

Examples

```
use PDL;
my $x = sequence(3, 4);      # 3x4, values 0..11
print $x->at(1, 2);          # 7
print $x->at(-1, -1);        # 11 (last element via negative
 ↪indexing)

# BAD value propagation
my $b = pdl([1, 2, 3]);
$b->badflag(1);
$b->inplace->setbadat(1);
print $b->at(1);             # "BAD"
```

Errors

Croaks with `"Position out of range"` when:

- `$pdl` is a null ndarray, fresh from `PDL->null` and not yet given dimensions;
- `$pdl` has zero elements, as `pdl([])` does;
- any supplied index, after negative indices are resolved, falls outside `[0, dim_size)` for its axis.

set_autopthread_targ

`set_autopthread_targ($n)` - the number of threads an operation is split across, at most; 0 turns the splitting off. Returns `$n`.

An operation is split along one of its broadcast dimensions, the one that divides most evenly, when its largest ndarray has at least `get_autopthread_size` Mi elements. The threads are pperl's own pool, so `--no-parallel` keeps every operation on one thread whatever this says. `PDL_AUTOPTHREAD_TARG` in the environment sets it when PDL loads; the default is `online_cpus`.

flowing

Turn on dataflow, so ndarrays derived from this one track its changes.

my $b = $a->flowing->slice('0:4'); $a->set(2, 99); print $b->at(2); # 99 - $b followed the change

Returns the ndarray, so it chains. Without `flowing`, a derived ndarray keeps whatever values it had when it was made.

Views built by slicing and other index arithmetic follow their parent. Results of computed expressions such as `$a + $a` do not yet: the expression is evaluated once and the link is not retained, so a later write to `$a` will not refresh them.

sever

Detach a view from its parent, keeping the values it has right now.

my $view = $big->slice('0:9'); $view->sever; # now an ndarray in its own right

Returns the ndarray. A slice or other view normally shares its parent's data; after `sever` it owns a copy of the elements it was showing, and neither side sees the other's writes. Useful when you want to keep a small window and let a large parent be freed.

Severing an ndarray that has no parent does nothing.

set_datatype

Change the element type of an ndarray in place.

$pdl->set_datatype($typecode);

`$typecode` is the integer type code, as `get_datatype` and a type's `enum` return; it must be in range or the call croaks with `set_datatype: datatype out of range`.

A null ndarray that has no data yet simply takes on the new type. Otherwise the existing values are converted, exactly as `convert` would, but the ndarray itself is modified rather than a new one returned. `PDL::IO::Storable` uses this when reconstituting an ndarray it has read back.

set_donttouchdata

Declare that the ndarray's storage belongs to someone else.

$pdl->set_donttouchdata; $pdl->set_donttouchdata($nbytes);

After this the ndarray will not free its own data when it goes away - the caller who supplied that memory is responsible for it. With a size argument, the ndarray also records that many bytes as its extent. Returns 1.

This is how a memory-mapped ndarray is set up, by `set_data_by_file_map` and hence by `PDL::IO::FastRaw::mapfraw`. Setting it on an ndarray whose data PDL did allocate leaks that data.

setdims

Give an ndarray a new shape, in place.

$pdl->setdims([3, 4]);

The argument is an arrayref of dimension sizes; anything else means no dimensions. The ndarray is modified rather than a new one returned, and storage is allocated to match the new shape.

This does not rearrange existing values into the new shape - use `reshape` for that.

set_sv_to_null_pdl

Turn an existing scalar into a null ndarray.

PDL::set_sv_to_null_pdl($sv);

The scalar is made to refer to a freshly created ndarray with no dimensions and no data, as `PDL->null` returns; whatever it held before is replaced.

Deserialisation is what this exists for. `PDL::IO::Storable` gets an empty blessed stub back from `Storable` and needs to make a real ndarray out of it before filling in the dimensions and data.

hdrcpy

Get or set whether the header travels with derived ndarrays.

$pdl->hdrcpy(1); my $on = $pdl->hdrcpy;

With the flag on, ndarrays made from `$pdl` by an operation get a deep copy of its header, and the flag; with it off - the default - they start without a header. It returns the setting, after any change.

5.2.76 POSIX

The POSIX (IEEE Std 1003.1) C standard library, exposed to Perl.

Reach for `POSIX` when you need something the Perl core does not give you directly: a `strftime` format, `setlocale` to switch numeric conventions, `sigaction` for signal handling, wait-status macros like `WIFEXITED` / `WEXITSTATUS`, `mkfifo`, or any of the hundreds of `E*` / `O_*` / `F_*` / `S_*` constants from `<errno.h>`, `<fcntl.h>`, and friends.

```
use POSIX qw(strftime floor ceil fmod :sys_wait_h);
use POSIX qw(:math_h_c99);

print strftime("%Y-%m-%d %H:%M:%S", gmtime 86400 * 365), "\n";   #␣
↪1971-01-01 00:00:00
printf "%s %s %s %s\n", floor(-1.5), ceil(-1.5), fmod(7, -3),␣
↪round(2.5);   # -2 -1 1 3

system $^X, '-e', 'exit 3';
print "exited ", WEXITSTATUS($?), "\n" if WIFEXITED($?);          #␣
↪exited 3

printf "EINTR is %d, PATH_MAX %d\n", POSIX::EINTR, POSIX::PATH_MAX; ␣
↪ # 4, 4096
```

The functions below are the C library's, called directly, so they behave as the host's C library does (glibc or musl on Linux). This is a native implementation of the XS half of POSIX 2.26; its `POSIX.pm` is used unchanged. That `.pm` also provides the many POSIX names that are perl built-ins under another name (`POSIX::printf` is `printf`, `POSIX::exit` is `exit`, `POSIX::getpid` is `$$` ...), the classes `POSIX::SigSet`, `POSIX::SigAction` and `POSIX::Termios`, and a helpful `die` for the C functions that make no sense in Perl (`POSIX::malloc`, `POSIX::strcpy` ...).

`use POSIX` with no list exports 587 names - most of the C89 library and its constants. Ask for less: a list of names, or one of the tags named after the C header it comes from (`:sys_wait_h`, `:fcntl_h`, `:math_h`, `:math_h_c99` for the C99 math functions, `:termios_h`, `:signal_h`, ...). The C99 functions are never exported by default.

Every macro the platform defines is a constant sub. Any other name dies when called: one POSIX knows but the platform lacks with "Your vendor has not defined POSIX macro OPEN_MAX, used at FILE line N", one that is no POSIX macro with "tcgetattr is not a valid POSIX macro at FILE line N" - the location being the caller's.

Differences from upstream

One, where perl 5.44 with POSIX.xs fails: a NaN payload of 2**64 or more (`nan(1e20)`, `setpayload($x, 1e20)`) gives the largest payload and warns "payload lost bits". POSIX.xs's conversion of such a number is undefined C: on x86-64 it gives payload 0 and no warning, losing every bit silently (t/81-xs-native/POSIX/380-or-better.t).

Modules

- `POSIX::SigSet`
- `POSIX::Termios`

Functions

Math

acos

`acos($x)`: the arc cosine of `$x`, in radians; NaN outside -1..1. Exported by default.

acosh

`acosh($x)`: the inverse hyperbolic cosine; NaN below 1.

asin

`asin($x)`: the arc sine of `$x`, in radians; NaN outside -1..1. Exported by default.

asinh

`asinh($x)`: the inverse hyperbolic sine.

atan

`atan($x)`: the arc tangent of `$x`, in radians, between -pi/2 and pi/2. Perl's own `atan2($y, $x)` covers the full circle. Exported by default.

atanh

`atanh($x)`: the inverse hyperbolic tangent; infinite at -1 and 1, NaN outside them.

cbrt

`cbrt($x)`: the cube root, negative for negative `$x`: `cbrt(-27)` is -3.

ceil

`ceil($x)`: the smallest integral value not below `$x`, as a floating-point number: `ceil(-1.5)` is -1. Exported by default.

cosh

`cosh($x)`: the hyperbolic cosine. Exported by default.

erf

`erf($x)`: the error function.

erfc

`erfc($x)`: the complementary error function, `1 - erf($x)`, computed without the loss of precision for large `$x`.

exp2

`exp2($x)`: 2 to the power `$x`.

expm1

`expm1($x)`: `exp($x) - 1`, accurate for `$x` near 0.

floor

`floor($x)`: the largest integral value not above `$x`, as a floating-point number: `floor(-1.5)` is -2. Exported by default.

j0

`j0($x)`: the Bessel function of the first kind of order 0; `j1` is order 1, `jn` any order.

j1

`j1($x)`: the Bessel function of the first kind of order 1.

lgamma

`lgamma($x)`: the natural logarithm of the absolute value of the gamma function: `lgamma(5)` is log(24).

log10

`log10($x)`: the base-10 logarithm. Exported by default.

log1p

`log1p($x)`: `log(1 + $x)`, accurate for `$x` near 0.

log2

`log2($x)`: the base-2 logarithm.

logb

`logb($x)`: the binary exponent of `$x`, as a floating-point number: `logb(8)` is 3. `ilogb` returns it as an integer.

nearbyint

`nearbyint($x)`: `$x` rounded to an integral value in the current rounding mode (see `fesetround`), without raising the inexact floating-point exception.

rint

`rint($x)`: `$x` rounded to an integral value in the current rounding mode; in the default mode halves go to the even neighbour: `rint(2.5)` is 2.

round

`round($x)`: `$x` rounded to the nearest integral value, halves away from zero: `round(2.5)` is 3, `round(-2.5)` is -3.

sinh

`sinh($x)`: the hyperbolic sine. Exported by default.

tan

`tan($x)`: the tangent of `$x` radians. Exported by default.

tanh

`tanh($x)`: the hyperbolic tangent. Exported by default.

tgamma

`tgamma($x)`: the gamma function: `tgamma(5)` is 24.

trunc

`trunc($x)`: `$x` with its fraction dropped, towards zero: `trunc(-2.7)` is -2.

y0

`y0($x)`: the Bessel function of the second kind of order 0; `y1` is order 1, `yn` any order.

y1

`y1($x)`: the Bessel function of the second kind of order 1.

copysign

`copysign($x, $y)`: the magnitude of `$x` with the sign of `$y`, negative zero included: `copysign(3, -0.0)` is -3.

fdim

`fdim($x, $y)`: `$x - $y` when that is positive, else 0.

fmax

`fmax($x, $y)`: the larger of the two; a NaN argument is ignored: `fmax(1, "nan")` is 1.

fmin

`fmin($x, $y)`: the smaller of the two; a NaN argument is ignored.

fmod

`fmod($x, $y)`: the floating-point remainder of `$x / $y`, with the sign of `$x`: `fmod(7, -3)` is 1. Unlike %, it does not truncate its operands to integers. Exported by default.

hypot

`hypot($x, $y)`: `sqrt($x**2 + $y**2)`, without overflow or underflow in the squares: `hypot(3, 4)` is 5.

nextafter

`nextafter($x, $y)`: the next representable floating-point number after `$x` in the direction of `$y`.

nexttoward

`nexttoward($x, $y)`: as `nextafter`; C's version takes `$y` as a `long double`, which for a Perl number makes no difference.

remainder

`remainder($x, $y)`: `$x - $n * $y` with `$n` the integer nearest to `$x / $y` (halves to even), so the result can be negative: `remainder(7, 3)` is 1, `remainder(8, 3)` is -1.

frexp

Split a floating-point number into mantissa and exponent.

modf

Split a number into integer and fractional parts, both as doubles.

ldexp

Compute `$x * 2**$exp` with a single rounding step.

Synopsis

```
my $y = POSIX::ldexp(1.5, 10);   # 1536
```

Differences from upstream

Fully compatible with upstream POSIX.

scalbn

Compute `$x * FLT_RADIX**$n` efficiently. On IEEE-754 systems `scalbn` is equivalent to `ldexp`.

Differences from upstream

Fully compatible with upstream POSIX.

fma

Fused multiply-add: `$x * $y + $z` with a single rounding.

nan

Build a NaN, optionally carrying a payload.

remquo

Floating-point remainder with partial quotient.

jn

Bessel function of the first kind, integer order `$n`.

yn

Bessel function of the second kind, integer order `$n`.

Float classification

getpayload

`getpayload($nv)` - the payload carried by the NaN `$nv`, as a number.

setpayload

`setpayload($nv, $payload)` - make `$nv` a quiet NaN carrying `$payload`.

setpayloadsig

`setpayloadsig($nv, $payload)` - make `$nv` a SIGNALING NaN carrying `$payload`, writing the result back into the first argument exactly as `setpayload` does.

issignaling

`issignaling($nv)` - true when `$nv` is a signaling NaN.

fpclassify

Classify a floating-point number.

ilogb

Exponent of `$x` as an integer, `trunc(log2(|$x|))`.

Edge cases

- `ilogb(0)` returns `FP_ILOGB0` (typically `INT_MIN`).
- `ilogb(NaN)` returns `FP_ILOGBNAN`.
- `ilogb(inf)` returns `INT_MAX`.

Differences from upstream

Fully compatible with upstream POSIX.

isfinite

`isfinite($x)`: 1 when `$x` is neither infinite nor NaN, else 0.

isinf

`isinf($x)`: 1 when `$x` is infinite, of either sign, else 0.

isnan

`isnan($x)`: 1 when `$x` is NaN, else 0.

isnormal

`isnormal($x)`: 1 when `$x` is a normal number - not zero, subnormal, infinite or NaN: `isnormal(1e-310)` is 0.

signbit

`signbit($x)`: 1 when the sign bit of `$x` is set, which includes negative zero: `signbit(-0.0)` is 1.

lrint

Round `$x` to the nearest integer, respecting the current rounding mode, and return it as an integer.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- `lround` - always rounds half-away-from-zero regardless of mode.
- `fegetround` / `fesetround` - query or change the rounding mode.

lround

Round `$x` half-away-from-zero to the nearest integer.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- `lrint` - rounds according to the current FP rounding mode.

isgreater

`isgreater($x, $y)`: 1 when `$x > $y`, else 0 - and 0 when either is NaN, as for `isgreaterequal`, `isless`, `islessequal` and `islessgreater`; `isunordered` tells that case apart.

isgreaterequal

`isgreaterequal($x, $y)`: 1 when `$x >= $y`, else 0.

isless

`isless($x, $y)`: 1 when `$x < $y`, else 0.

islessequal

`islessequal($x, $y)`: 1 when `$x <= $y`, else 0.

islessgreater

`islessgreater($x, $y)`: 1 when `$x` is less or greater than `$y`, 0 when they are equal or either is NaN.

isunordered

`isunordered($x, $y)`: 1 when either is NaN, so that they cannot be compared, else 0.

fegetround

Current floating-point rounding direction.

What you get back

An integer matching one of the `FE_TONEAREST`, `FE_DOWNWARD`, `FE_UPWARD`, `FE_TOWARDZERO` constants.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- `fesetround` - change the mode.

FLT_ROUNDS

The current rounding direction as C's `FLT_ROUNDS` reports it.

fesetround

Change the floating-point rounding direction.

Wait status

WEXITSTATUS

Exit code of a child that terminated normally.

WIFEXITED

True if the child terminated normally (via `exit` or `return` from `main`).

Differences from upstream

Fully compatible with upstream POSIX.

WIFSIGNALED

True if the child was killed by a signal.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- `WTERMSIG` - the signal number, when this is true.

WIFSTOPPED

True if the child is stopped (usable with `waitpid(..., WUNTRACED)`).

Differences from upstream

Fully compatible with upstream POSIX.

WSTOPSIG

Signal number that stopped the child, meaningful only when `WIFSTOPPED($status)` is true.

Differences from upstream

Fully compatible with upstream POSIX.

WTERMSIG

Signal number that killed the child, meaningful only when `WIFSIGNALED($status)` is true.

Differences from upstream

Fully compatible with upstream POSIX.

Strings and printing

mblen

`mblen($s, $n = ~0)` - the number of bytes in the first multibyte character of `$s`, examining at most `$n` bytes.

mbtowc

`mbtowc($pwc, $s, $n = ~0)` - decode the first multibyte character of `$s` into `$pwc` and return how many bytes it occupied.

wctomb

`wctomb($s, $wchar)` - encode the wide character `$wchar` into `$s` and return the number of bytes written.

strtold

`strtold($str)` - parse a floating-point number off the front of `$str`.

strtod

Parse a double from the start of a string, locale-aware.

strtol

Parse a signed integer from the start of a string, in a given base.

strtoul

Parse an unsigned integer from the start of a string, in a given base.

strcoll

Compare two strings using the current `LC_COLLATE` ordering.

What you get back

A negative, zero, or positive integer, suitable for use as a `cmp`-like sort return. The ordering depends on the locale set via `setlocale(LC_COLLATE, ...)`.

Differences from upstream

Fully compatible with upstream POSIX.

strxfrm

Locale-transformed form of a string, for repeated collation.

System and process

`cuserid`

`cuserid($s = 0)` - the login name associated with the process's effective user ID, or `undef` when it cannot be determined.

`setuid`

Change the real user ID of the current process.

What you get back

1 on success, `0` on failure (and `$!` is set). Permission is required; only privileged processes can switch to arbitrary UIDs.

Differences from upstream

Fully compatible with upstream POSIX.

`setgid`

Change the real group ID of the current process.

What you get back

1 on success, `0` on failure.

Differences from upstream

Fully compatible with upstream POSIX.

`setpgid`

Put process `$pid` into process group `$pgid`.

`setsid`

Create a new session and become its leader.

What you get back

The new session ID on success, `-1` on failure. The caller must not already be a process group leader.

Differences from upstream

Fully compatible with upstream POSIX.

`uname`

Kernel and machine identification as a five-element list.

sysconf

Runtime system configuration value for the given `_SC_*` name.

_exit

Terminate the process immediately with the given status, skipping `END` blocks, destructors, and stdio flushing.

abort

Abort the process abnormally (typically producing a core dump).

Differences from upstream

Fully compatible with upstream POSIX.

nice

Adjust the current process's scheduling niceness by `$incr`.

pause

Suspend the process until any signal is delivered.

What you get back

Always `-1` (with `$!` set to `EINTR`) once a signal handler returns - `pause` has no "success" exit path.

Differences from upstream

Fully compatible with upstream POSIX.

sleep

Suspend the process for `$secs` seconds, or until a signal arrives.

What you get back

The number of seconds left unslept - `0` for a full sleep, or a positive remainder when a signal cut the sleep short.

Differences from upstream

Fully compatible with upstream POSIX.

File and I/O

ttyname

Device path of the terminal attached to `$fd`, or `undef` if none.

Differences from upstream

Fully compatible with upstream POSIX.

ctermid

Path of the controlling terminal.

What you get back

On most systems, the string `"/dev/tty"`.

Differences from upstream

Fully compatible with upstream POSIX.

getcwd

Current working directory as an absolute path.

What you get back

The path as a byte string, or `undef` if the call fails (for example, if the process has no read permission on some ancestor directory).

Differences from upstream

Fully compatible with upstream POSIX.

See also

- `Cwd::getcwd` - the recommended Perl-level interface.

access

Check the caller's permissions on a path.

lchown

Change the owner/group of a symlink itself (without following it).

Synopsis

```
POSIX::lchown($uid, $gid, $path);
```

What you get back

1 on success, `0` on failure.

Differences from upstream

Fully compatible with upstream POSIX.

mkfifo

Create a FIFO (named pipe) at `$path` with permission bits `$mode`.

What you get back

1 on success, `0` on failure.

Differences from upstream

Fully compatible with upstream POSIX.

open

Open a file by path, returning a raw file descriptor.

close

Close a raw file descriptor.

What you get back

`"0 but true"` on success, `undef` on failure. A negative `$fd` is rejected with `$!` set to `EBADF`.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- open, dup, dup2 - where fds come from.

dup

Duplicate a file descriptor, returning the lowest-numbered unused fd.

What you get back

The new fd as an integer on success, `undef` on failure. A negative `$fd` is rejected with `$!` set to `EBADF`.

Differences from upstream

Fully compatible with upstream POSIX.

dup2

Duplicate `$fd1` onto `$fd2`, closing `$fd2` first if necessary.

lseek

Reposition the read/write offset of an open fd.

pipe

Create an anonymous pipe and return its two fds.

read

Read up to `$nbytes` bytes from `$fd` into `$buffer`.

write

Write bytes from `$buffer` to `$fd`.

Time

strftime

Format a broken-down time as a string using a format template.

mktime

Convert a broken-down local time into epoch seconds.

difftime

Difference between two epoch times, `$t1 - $t2`, as a double.

Differences from upstream

Fully compatible with upstream POSIX.

tzset

Refresh the process's timezone information from the `TZ` environment variable.

Differences from upstream

Fully compatible with upstream POSIX.

tzname

Short names of the current standard and daylight timezones.

clock

Approximate processor time used by the program, in `CLOCKS_PER_SEC` units.

What you get back

An integer count of clock ticks. Divide by `POSIX::CLOCKS_PER_SEC` for seconds. On Linux this counts user+system CPU time of the whole process.

Differences from upstream

Fully compatible with upstream POSIX.

times

Raw wall + CPU times of the process and its children, as a five-element list.

asctime

Canonical English timestamp string from a broken-down time.

`ctime`

Canonical English timestamp string for an epoch time, in local time.

Locale

`setlocale`

Query or change the program's locale setting.

`localeconv`

Numeric and monetary formatting rules for the active locale.

Errno and path limits

`fpathconf`

Runtime limit for a configuration variable on an open fd.

`pathconf`

Runtime limit for a configuration variable on a pathname.

What you get back

The limit value, or `-1` if unsupported/unlimited.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- `fpathconf` - same query on an already-open fd.

Terminal control

`tcdrain`

Block until all queued output on `$fd` has been transmitted.

What you get back

`"0 but true"` on success, `undef` on failure. A negative `$fd` sets `$!` to `EBADF`.

Differences from upstream

Fully compatible with upstream POSIX.

`tcflow`

Suspend or resume transmission or reception on terminal `$fd`.

`tcflush`

Discard buffered input, output, or both on terminal `$fd`.

`tcsendbreak`

Send a continuous break condition on terminal `$fd` for roughly `$duration` tenths of a second (0 means the implementation default).

What you get back

`"0 but true"` on success, `undef` on failure. Negative `$duration` sets `$!` to `EINVAL`.

Differences from upstream

Fully compatible with upstream POSIX.

`tcgetpgrp`

Process group ID of the foreground process group for terminal `$fd`.

What you get back

The pgid as an integer, or `-1` on error (with `$!` set).

Differences from upstream

Fully compatible with upstream POSIX.

`tcsetpgrp`

Make `$pgrp` the foreground process group of terminal `$fd`.

What you get back

`"0 but true"` on success, `undef` on failure.

Differences from upstream

Fully compatible with upstream POSIX.

Signals

`sigprocmask`

Change, query, or both the set of signals blocked for this process.

`sigpending`

Fill `$sigset` with the signals currently pending for delivery.

What you get back

`"0 but true"` on success, `undef` on failure.

Differences from upstream

Fully compatible with upstream POSIX.

sigsuspend

Temporarily replace the signal mask with `$sigset` and suspend the process until any non-blocked signal arrives.

What you get back

Always `undef` (with `$!` set to `EINTR`) once a signal handler has run - there is no success exit path.

Differences from upstream

Fully compatible with upstream POSIX.

sigaction

Install or inspect a signal handler for signal `$sig`.

POSIX::SigSet

Functions

Signals

new

Create a `POSIX::SigSet` object, optionally pre-populated with the given signal numbers.

addset

Add signal `$sig` to the set.

What you get back

`"0 but true"` on success, `undef` on failure.

Differences from upstream

Fully compatible with upstream POSIX.

delset

Remove signal `$sig` from the set.

What you get back

`"0 but true"` on success, `undef` on failure.

Differences from upstream

Fully compatible with upstream POSIX.

emptyset

Remove every signal from the set.

Differences from upstream

Fully compatible with upstream POSIX.

fillset

Add every signal to the set.

Differences from upstream

Fully compatible with upstream POSIX.

ismember

Test whether signal `$sig` is in the set.

What you get back

1 if the signal is a member, `0` if it is not, `-1` on error.

Differences from upstream

Fully compatible with upstream POSIX.

new

Create a `POSIX::SigSet` object, optionally pre-populated with the given signal numbers.

Synopsis

```
use POSIX qw(SIGINT SIGTERM);
my $set = POSIX::SigSet->new(SIGINT, SIGTERM);
```

What you get back

A blessed `POSIX::SigSet` reference, with `sigemptyset` applied first and then each argument added via `sigaddset`.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- `addset`, `delset`, `emptyset`, `fillset`, `ismember` - mutate or query the set.
- `sigprocmask` - install a `SigSet` as the process mask.

POSIX::Termios

Functions

Terminal control

new

Create a fresh `POSIX::Termios` object, all fields zeroed.

getattr

Fill the `POSIX::Termios` object with the current settings of `$fd`.

setattr

Apply the `POSIX::Termios` settings to `$fd`.

getispeed

Input baud rate encoded in the termios structure.

What you get back

The encoded speed as an integer (compare against `B9600`, `B115200`, etc., which this module also exports).

Differences from upstream

Fully compatible with upstream POSIX.

getospeed

Output baud rate encoded in the termios structure.

Differences from upstream

Fully compatible with upstream POSIX.

setispeed

Encode an input baud rate into the termios structure.

What you get back

`"0 but true"` on success, `undef` on failure. The change is not applied to the terminal until you call `setattr`.

Differences from upstream

Fully compatible with upstream POSIX.

`setospeed`

Encode an output baud rate into the termios structure.

Differences from upstream

Fully compatible with upstream POSIX.

`getcc`

Value of a special control character in the termios structure.

`setcc`

Set a special control character in the termios structure.

Edge cases

- Indices beyond `NCCS` croak with `Bad setcc subscript`.
- The change is not applied until you call `setattr`.

Differences from upstream

Fully compatible with upstream POSIX.

`new`

Create a fresh `POSIX::Termios` object, all fields zeroed.

Synopsis

```
use POSIX;
my $t = POSIX::Termios->new;
$t->getattr(fileno(STDIN));
```

What you get back

A blessed `POSIX::Termios` reference. To fill it with the current terminal settings, follow up with `$t->getattr($fd)`.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- `getattr`, `setattr` - populate and apply terminal settings.

getattr

Fill the `POSIX::Termios` object with the current settings of `$fd`.

Synopsis

```
$t->getattr(fileno(STDIN));
```

What you get back

`"0 but true"` on success, `undef` on failure. `$fd` defaults to `0` (stdin).

Differences from upstream

Fully compatible with upstream POSIX.

setattr

Apply the `POSIX::Termios` settings to `$fd`.

Synopsis

```
use POSIX qw(TCSANOW TCSADRAIN);
$t->setattr(fileno(STDIN), TCSANOW);
```

What you get back

`"0 but true"` on success, `undef` on failure. `$fd` defaults to `0`, `$optional_actions` defaults to `TCSANOW` (apply immediately).

Edge cases

- Negative `$optional_actions` sets `$!` to `EINVAL`.

Differences from upstream

Fully compatible with upstream POSIX.

getcc

Value of a special control character in the termios structure.

Synopsis

```
use POSIX qw(VINTR VEOF);
my $intr = $t->getcc(VINTR);   # Ctrl-C byte
```

Edge cases

- Indices beyond the `NCCS` bound croak with `Bad getcc subscript`.

Differences from upstream

Fully compatible with upstream POSIX.

frexp

Split a floating-point number into mantissa and exponent.

Synopsis

```
my ($frac, $exp) = POSIX::frexp($x);
# $x == $frac * 2**$exp, with 0.5 <= |$frac| < 1 (or $frac == 0).
```

What you get back

A two-element list: the fractional part as a double and the exponent as an integer.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- `ldexp` - the inverse operation: reassemble `$x` from `$frac` and `$exp`.
- `modf` - split into integer and fractional parts instead.

modf

Split a number into integer and fractional parts, both as doubles.

Synopsis

```
my ($frac, $int) = POSIX::modf(3.75);   # (0.75, 3.0)
```

What you get back

A two-element list `($frac, $int)`. Both parts carry the sign of the input.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- `frexp` - split into mantissa and exponent instead.

fma

Fused multiply-add: `$x * $y + $z` with a single rounding.

Synopsis

```
my $r = POSIX::fma($x, $y, $z);
```

What you get back

A double. Because there is only one rounding step, the result can be more accurate than the naive two-step expression, especially for catastrophic-cancellation patterns like `$a * $b - $c * $d` rewritten as `fma($a, $b, -($c * $d))`.

Differences from upstream

Fully compatible with upstream POSIX.

nan

Build a NaN, optionally carrying a payload.

Synopsis

```
my $q = POSIX::nan();     # the platform's default NaN
my $p = POSIX::nan(42);   # NaN carrying payload 42
```

What you get back

A NaN whose low mantissa bits hold the payload, readable again with `POSIX::getpayload`. The payload is truncated to the 51 bits an IEEE double has room for.

Edge cases

- No argument yields the default NaN, which is not necessarily the zero payload on every platform.
- The argument is a NUMBER, not a string of digits: `nan("0x2a")` numifies to 0, and `nan("42abc")` to 42, exactly as any other numeric context would.
- A negative payload wraps rather than clamping, so `nan(-1)` gives the all-ones payload.

Differences from upstream

Payloads at or above 2**64 only: see `nv_to_uv_c`. Every payload an IEEE double can actually represent matches upstream exactly.

remquo

Floating-point remainder with partial quotient.

Synopsis

```
my ($rem, $quo) = POSIX::remquo($x, $y);
```

What you get back

A two-element list: the IEEE remainder and the low bits of the quotient as an integer (typically at least three bits, enough for argument reduction of trigonometric functions).

Differences from upstream

Fully compatible with upstream POSIX.

jn

Bessel function of the first kind, integer order `$n`.

Synopsis

```
my $y = POSIX::jn($n, $x);
```

Differences from upstream

Fully compatible with upstream POSIX.

See also

- `j0`, `j1` - fixed-order variants for `$n == 0` and `$n == 1`.
- yn - Bessel function of the second kind.

yn

Bessel function of the second kind, integer order `$n`.

Synopsis

```
my $y = POSIX::yn($n, $x);
```

Edge cases

- `yn($n, 0)` returns negative infinity (the function has a singularity at zero).

Differences from upstream

Fully compatible with upstream POSIX.

See also

- `y0`, `y1` - fixed-order variants.
- `jn` - Bessel function of the first kind.

fpclassify

Classify a floating-point number.

Synopsis

```
use POSIX qw(fpclassify FP_NAN FP_INFINITE FP_ZERO FP_SUBNORMAL FP_
 ↪NORMAL);
my $k = fpclassify($x);
```

What you get back

An integer matching one of the `FP_*` constants. Compare against the symbolic names, not a hardcoded number - the values are libc implementation details.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- `isnan`, `isinf`, `isfinite`, `isnormal` - narrower single-class tests.

FLT_ROUNDS

The current rounding direction as C's `FLT_ROUNDS` reports it.

What you get back

1 for round-to-nearest, which is the default. The other values are `0` toward zero, `2` toward +infinity, `3` toward -infinity.

Differences from upstream

None. `FLT_ROUNDS` is a FUNCTION here, not a constant, because it is one upstream too: it takes no arguments and raises a usage error if given any, and its value tracks `fesetround` instead of being fixed once and for all.

fesetround

Change the floating-point rounding direction.

Synopsis

```
use POSIX qw(fesetround FE_DOWNWARD);
fesetround(FE_DOWNWARD);
```

What you get back

`0` on success, non-zero if the requested mode is not supported.

Differences from upstream

Fully compatible with upstream POSIX.

WEXITSTATUS

Exit code of a child that terminated normally.

Synopsis

```
system($cmd);
my $code = POSIX::WEXITSTATUS($?);
```

What you get back

The low 8 bits of the child's `exit()` argument. Only meaningful when `WIFEXITED($status)` is true.

Differences from upstream

Fully compatible with upstream POSIX.

strtod

Parse a double from the start of a string, locale-aware.

Synopsis

```
use POSIX qw(strtod);
my ($num, $unparsed) = strtod("3.14abc");   # (3.14, 3)
```

What you get back

A two-element list: the parsed value and the number of bytes that did not form part of the number. When `$unparsed == 0`, the whole string was consumed.

Edge cases

- Input that doesn't start with a number: returns `(0, length)`.
- Overflow: returns `HUGE_VAL` and sets `$!` to `ERANGE`.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- `strtol`, `strtoul` - integer variants.

strtol

Parse a signed integer from the start of a string, in a given base.

Synopsis

```
use POSIX qw(strtol);
my ($n, $unparsed) = strtol("0x2a rest", 0);   # (42, 5)
```

What you get back

A two-element list: the parsed integer and the number of bytes that were not consumed. Base `0` auto-detects from prefix (`0x`, `0b`, leading `0`); otherwise `$base` must be 2-36.

Edge cases

- Invalid base: returns `(undef, undef)` and sets `$!` to `EINVAL`.
- Overflow: returns `LONG_MIN` / `LONG_MAX` and sets `$!` to `ERANGE`.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- `strtoul` - unsigned variant.
- `strtod` - floating-point variant.

strtoul

Parse an unsigned integer from the start of a string, in a given base.

Synopsis

```
use POSIX qw(strtoul);
my ($n, $unparsed) = strtoul("255 end", 10);
```

Edge cases

- Invalid base: returns `(undef, undef)` and sets `$!` to `EINVAL`.
- Negative input: parsed and two's-complement-cast to unsigned, matching C `strtoul`.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- `strtol` - signed variant.

strxfrm

Locale-transformed form of a string, for repeated collation.

Synopsis

```
use POSIX qw(strxfrm);
my @sorted = map { $_->[1] }
             sort { $a->[0] cmp $b->[0] }
             map { [ strxfrm($_), $_ ] } @words;
```

What you get back

A byte string whose plain `cmp` ordering equals `strcoll`'s ordering of the originals under the current `LC_COLLATE` locale. Worth it when the same string is compared many times, as in a sort.

Differences from upstream

Fully compatible with upstream POSIX.

setpgid

Put process `$pid` into process group `$pgid`.

Synopsis

```
POSIX::setpgid($pid, $pgid);
```

What you get back

1 on success, `0` on failure. A `$pid` of `0` means the calling process; a `$pgid` of `0` means use `$pid` as the new group ID.

Differences from upstream

Fully compatible with upstream POSIX.

uname

Kernel and machine identification as a five-element list.

Synopsis

```
use POSIX qw(uname);
my ($sysname, $nodename, $release, $version, $machine) = uname();
```

What you get back

Five strings in order: sysname (e.g. `"Linux"`), nodename (hostname), release (kernel release), version (kernel version string), and machine (architecture, e.g. `"x86_64"`).

Edge cases

- On `uname` failure, returns an empty list.

Differences from upstream

Fully compatible with upstream POSIX.

sysconf

Runtime system configuration value for the given _SC_* name.

Synopsis

```
use POSIX qw(sysconf _SC_OPEN_MAX _SC_PAGESIZE);
my $max_fd = sysconf(_SC_OPEN_MAX);
```

What you get back

The configured value, or -1 if the option is not supported (in which case $! may or may not be set - distinguish by checking $! before and after).

Differences from upstream

Fully compatible with upstream POSIX.

access

Check the caller's permissions on a path.

Synopsis

```
use POSIX qw(access R_OK W_OK);
print "readable\n" if access($path, R_OK);
```

What you get back

The string "0 but true" when the access check succeeds, or undef when it fails (with $! set). The "0 but true" return matches core Perl's convention for system calls whose success value is 0.

Edge cases

- Check uses real (not effective) UID/GID, so is vulnerable to TOCTOU races; prefer trying the operation and handling failure.

Differences from upstream

Fully compatible with upstream POSIX.

open

Open a file by path, returning a raw file descriptor.

Synopsis

```
use POSIX qw(open O_RDONLY O_CREAT O_WRONLY);
my $fd = open($path, O_RDONLY)
    or die "open: $!";
```

What you get back

A non-negative integer fd on success, `undef` on failure (with `$!` set). This is the raw system fd, not a Perl filehandle - pair it with `POSIX::close` or convert via `IO::Handle::fdopen`.

Differences from upstream

Fully compatible with upstream POSIX. Third argument `$mode` defaults to `0666` when opening with `O_CREAT` and no mode given.

See also

- `close`, dup, `lseek`, `read`, `write` - the rest of the raw fd API.
- Core open - the Perl-native filehandle interface.

dup2

Duplicate `$fd1` onto `$fd2`, closing `$fd2` first if necessary.

Synopsis

```
POSIX::dup2($fd, 1);   # redirect stdout to $fd
```

What you get back

The target fd on success, `undef` on failure. Either argument being negative is rejected with `$!` set to EBADF.

Differences from upstream

Fully compatible with upstream POSIX.

lseek

Reposition the read/write offset of an open fd.

Synopsis

```
use POSIX qw(lseek SEEK_SET SEEK_CUR SEEK_END);
my $pos = lseek($fd, 0, SEEK_CUR);    # current offset
```

What you get back

The new absolute offset on success, `-1` on failure (with `$!` set).

Differences from upstream

Fully compatible with upstream POSIX.

pipe

Create an anonymous pipe and return its two fds.

Synopsis

```
use POSIX qw(pipe);
my ($rd, $wr) = pipe();
```

What you get back

A two-element list `($read_fd, $write_fd)` on success. On failure, a single `undef` is returned and `$!` is set.

Differences from upstream

Fully compatible with upstream POSIX.

read

Read up to `$nbytes` bytes from `$fd` into `$buffer`.

Synopsis

```
use POSIX qw(read);
my $buf = "";
my $n = read($fd, $buf, 4096);
```

What you get back

On success, the number of bytes read (or `"0 but true"` for EOF); on failure, `undef` (with `$!` set). The data is written into `$buffer` in place - truncated or grown to match the actual read length.

Edge cases

- Partial reads are normal on pipes, sockets, and terminals; always check the return length.

Differences from upstream

Fully compatible with upstream POSIX.

write

Write bytes from `$buffer` to `$fd`.

Synopsis

```
POSIX::write($fd, $data, length($data));
```

What you get back

The number of bytes actually written. A value less than `$nbytes` is not an error - common for pipes and sockets - and the caller must loop to write the rest.

Differences from upstream

Fully compatible with upstream POSIX.

strftime

Format a broken-down time as a string using a format template.

Synopsis

```
use POSIX qw(strftime);
my @tm = localtime;
my $str = strftime("%Y-%m-%d %H:%M:%S", @tm);
```

What you get back

A byte string formatted according to the template. Argument order matches `localtime` / `gmtime`: second, minute, hour, mday, mon (0-11), year (years since 1900), wday, yday, isdst - the last three optional and defaulting to `-1`.

Edge cases

- Fewer than seven arguments (format plus the first six tm fields): returns the empty string.
- Output truncated at 1024 bytes.

- Month is 0-11 and year is years-since-1900, the `struct tm` convention, *not* the 1-12 / 4-digit-year human convention.
- `$wday` and `$yday` are accepted and IGNORED, as upstream ignores them. Both are derived from the date, so passing a wrong weekday cannot produce a wrong `%A`; passing none is the normal call.
- Out-of-range fields normalise rather than being echoed back: month 12 is January of the following year.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- `mktime` - go the other way: broken-down time to epoch seconds.
- `asctime`, `ctime` - simpler canned-format alternatives.

mktime

Convert a broken-down local time into epoch seconds.

Synopsis

```
use POSIX qw(mktime);
my $t = mktime($sec, $min, $hour, $mday, $mon, $year - 1900);
```

What you get back

The epoch seconds as an integer on success. The sentinel value `"0 but true"` is returned when the epoch genuinely is `0` (so you can distinguish it from failure). `undef` is returned on failure (when the input cannot be represented).

Edge cases

- The `tm` fields are normalised in place by libc, so values outside the usual ranges (e.g. `mon = 13`) are accepted and carry into adjacent fields.
- `wday` and `yday` are ignored on input; `isdst` defaults to `-1` (let libc figure it out) when omitted.

Differences from upstream

Fully compatible with upstream POSIX.

tzname

Short names of the current standard and daylight timezones.

Synopsis

```
use POSIX qw(tzname);
my ($std, $dst) = tzname();
```

What you get back

A two-element list. Both strings are derived from `TZ` - the call also implicitly refreshes the timezone cache.

Differences from upstream

Fully compatible with upstream POSIX.

times

Raw wall + CPU times of the process and its children, as a five-element list.

Synopsis

```
my ($real, $user, $sys, $cuser, $csys) = POSIX::times();
```

What you get back

Five integer tick counts, in `sysconf(_SC_CLK_TCK)` units - raw, not divided by the ticks-per-second constant. Divide if you want seconds.

Differences from upstream

Fully compatible with upstream POSIX.

asctime

Canonical English timestamp string from a broken-down time.

Synopsis

```
use POSIX qw(asctime);
my @tm = localtime;
print asctime(@tm);   # "Wed Apr 22 14:30:00 2026\n"
```

What you get back

A 25-byte string **including the trailing newline**, or `undef` if nine arguments are not supplied.

Differences from upstream

Fully compatible with upstream POSIX. The trailing newline is preserved as-is; do not assume this function trims it.

ctime

Canonical English timestamp string for an epoch time, in local time.

Synopsis

```
use POSIX qw(ctime);
print ctime(time);   # "Wed Apr 22 14:30:00 2026\n"
```

What you get back

A 25-byte string including the trailing newline, or `undef` on conversion failure.

Differences from upstream

Fully compatible with upstream POSIX. The trailing newline is preserved.

See also

- `asctime` - same formatting, but from a broken-down time list.

_exit

Terminate the process immediately with the given status, skipping `END` blocks, destructors, and stdio flushing.

Synopsis

```
POSIX::_exit(0);
```

`POSIX::exit` is a different function: perl's own `exit`, which does run `END` blocks and destructors.

Differences from upstream

Fully compatible with upstream POSIX.

nice

Adjust the current process's scheduling niceness by `$incr`.

Synopsis

```
POSIX::nice(5);   # lower priority
```

What you get back

The new niceness value. When the new value genuinely is `0`, returns the string `"0 but true"` to distinguish it from error. On error, returns `undef` and sets `$!`.

Edge cases

- Positive increments lower priority; only root can pass negative values to raise priority.

Differences from upstream

Fully compatible with upstream POSIX.

setlocale

Query or change the program's locale setting.

Synopsis

```
use POSIX qw(setlocale LC_ALL LC_NUMERIC);
setlocale(LC_ALL, "");              # honour environment
my $cur = setlocale(LC_NUMERIC);    # query current
```

What you get back

The resulting locale name as a string, or `undef` if the category/locale combination is invalid. Calling with one argument queries; calling with two sets. Passing `""` as the new locale asks libc to pick from `LC_*` / `LANG` environment variables.

Edge cases

- Changing a category can affect formatting of numbers, collation, date strings, and error messages throughout the process.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- `localeconv` - inspect the active locale's numeric and monetary conventions.

localeconv

Numeric and monetary formatting rules for the active locale.

Synopsis

```
use POSIX qw(localeconv);
my $lc = localeconv();
print "Decimal: $lc->{decimal_point}\n";
```

What you get back

A hashref whose keys are the `struct lconv` field names - `decimal_point`, `thousands_sep`, `grouping`, `int_curr_symbol`, `currency_symbol`, `mon_decimal_point`, `mon_thousands_sep`, `mon_grouping`, `positive_sign`, `negative_sign`, and the monetary char fields (`int_frac_digits`, `frac_digits`, `p_cs_precedes`, `p_sep_by_space`, `n_cs_precedes`, `n_sep_by_space`, `p_sign_posn`, `n_sign_posn`).

Edge cases

- Char fields equal to `CHAR_MAX` mean "not available in this locale"; all numeric char fields are returned unconditionally.
- Returns `undef` if libc `localeconv()` fails (rare).

Differences from upstream

Fully compatible with upstream POSIX.

See also

- `setlocale` - pick which locale those rules come from.

fpathconf

Runtime limit for a configuration variable on an open fd.

Synopsis

```
use POSIX qw(fpathconf _PC_NAME_MAX);
my $max = fpathconf($fd, _PC_NAME_MAX);
```

What you get back

The limit value, or `-1` if the variable is unsupported or unlimited (check `$!` to distinguish).

Differences from upstream

Fully compatible with upstream POSIX.

tcflow

Suspend or resume transmission or reception on terminal `$fd`.

Synopsis

```
use POSIX qw(tcflow TCOOFF TCOON);
tcflow($fd, TCOOFF);   # pause output
```

What you get back

`"0 but true"` on success, `undef` on failure. Negative `$action` sets `$!` to `EINVAL`.

Differences from upstream

Fully compatible with upstream POSIX.

tcflush

Discard buffered input, output, or both on terminal `$fd`.

Synopsis

```
use POSIX qw(tcflush TCIFLUSH TCOFLUSH TCIOFLUSH);
tcflush($fd, TCIOFLUSH);   # drop pending input and output
```

What you get back

`"0 but true"` on success, `undef` on failure. Negative `$queue_selector` sets `$!` to `EINVAL`.

Differences from upstream

Fully compatible with upstream POSIX.

sigprocmask

Change, query, or both the set of signals blocked for this process.

Synopsis

```
use POSIX qw(sigprocmask SIG_BLOCK SIG_UNBLOCK SIG_SETMASK);
my $mask = POSIX::SigSet->new(SIGINT);
sigprocmask(SIG_BLOCK, $mask);
```

What you get back

`"0 but true"` on success, `undef` on failure. When `$oldsigset` is supplied, it is filled with the mask in effect before the call.

Edge cases

- Passing a non-`POSIX::SigSet` reference for either set croaks with `sigset is not of type POSIX::SigSet` / `oldsigset is not of type POSIX::SigSet`.

Differences from upstream

Fully compatible with upstream POSIX.

See also

- `sigpending`, `sigsuspend`, `sigaction` - the rest of the POSIX signal API.

sigaction

Install or inspect a signal handler for signal `$sig`.

Synopsis

```
use POSIX qw(sigaction SIGINT);
my $action = POSIX::SigAction->new("IGNORE");
my $old = POSIX::SigAction->new;
sigaction(SIGINT, $action, $old);
```

What you get back

`"0 but true"` on success, `undef` on failure. When `$oldaction` is supplied, it is populated with the previous `HANDLER`, `MASK`, `FLAGS`, and `SAFE` fields.

Edge cases

- Negative `$sig` croaks with `Negative signals are not allowed`.
- An action hash without a `HANDLER` key croaks with `Can't supply an action without a HANDLER`.

See also

- `sigprocmask` - change the blocked-signal mask.
- `POSIX::SigSet` - build the `MASK` field of an action.

5.2.77 Package

Namespace containing the following modules:

- `Package::Stash`

Package::Stash

Namespace containing the following modules:

- `Package::Stash::XS` - The compiled implementation of Package::Stash: reading and changing a package's symbol table - its scalars, arrays, hashes, subs and IO handles - by name.

Package::Stash::XS

The compiled implementation of Package::Stash: reading and changing a package's symbol table - its scalars, arrays, hashes, subs and IO handles - by name.

A variable is named with its sigil (`'$foo'`, `'@ISA'`, `'%h'`, `'&f'`; no sigil means the IO handle) or given as `{ name => 'foo', type => 'SCALAR' }`. `add_symbol` installs a value, `get_symbol` returns a reference to it, `get_or_add_symbol` vivifies it first, `has_symbol` and `remove_symbol` test and remove one slot, `remove_glob` the whole glob, and `list_all_symbols` / `get_all_symbols` list them, optionally of one type. The object may also wrap an anonymous stash (a hash ref).

This is a native implementation of Package::Stash::XS 0.32's `XS.xs`; its `Package/Stash/XS.pm` is used unchanged, and Package::Stash, which picks it as its implementation, comes from CPAN.

Synopsis

```
use Package::Stash;                    # uses this one when it can

my $stash = Package::Stash->new('Foo');
$stash->add_symbol('%foo', { bar => 1 });
$stash->add_symbol('&hello', sub { "Hello" });
print ${ $stash->get_symbol('%foo') }{bar}, "\n";
print join(' ', sort $stash->list_all_symbols('CODE')), "\n";
$stash->remove_symbol('&hello');
```

Differences from upstream

None.

Functions

Symbol tables

new

A stash object for the package named `$package`, or for the anonymous stash `\%stash`.

name

The package's name.

Edge cases

- Called on the class dies "Can't call name as a class method".
- An anonymous stash has none: dies "Can't get the name of an anonymous package".

namespace

A reference to the stash itself: `\%Foo::Bar::`, created when there is none yet. The object keeps it as a weak reference.

Edge cases

- Called on the class dies "Can't call namespace as a class method".

add_symbol

Install `$initial` in a slot: a reference installs its referent, a plain value a copy of it, nothing an empty value. Under the debugger (`$^P & 0x10`) a sub also gets its `%DB::sub` entry, from `filename`, `first_line_num` and `last_line_num` options.

remove_glob

Delete the whole glob `$name` (no sigil) from the stash.

has_symbol

Whether the slot holds something: 1 or 0, undef when the stash has no such entry at all.

Edge cases

- An entry that is not yet a glob (a sub stub or constant) counts as a sub and as nothing else.

get_symbol

A reference to the slot's value, or undef when it has none.

get_or_add_symbol

A reference to the slot's value, created empty when there is none.

Edge cases

- &name dies "Don't know how to vivify CODE variables" whenever the entry exists.

remove_symbol

Empty one slot of the glob; for an entry that is not yet a glob, & deletes it.

list_all_symbols

The names in the stash, or with a type the names whose slot of that type holds something.

Synopsis

```
my @all  = $stash->list_all_symbols;
my @subs = $stash->list_all_symbols('CODE');
```

Edge cases

- A type that is not a string dies "vartype must be a string".

get_all_symbols

A hash of the stash: name => glob, or with a type name => a reference to that slot's value, for the names whose slot holds something. Entries that are not yet globs are made globs first.

Synopsis

```
my $subs = $stash->get_all_symbols('CODE');   # { name => \&code }
```

new

A stash object for the package named $package, or for the anonymous stash \%stash.

Synopsis

```
my $s = Package::Stash::XS->new('Foo::Bar');
my $a = Package::Stash::XS->new({});          # an anonymous stash
```

Edge cases

- A name that is not a module name dies “NAME is not a module name”.
- Anything but a string or a hash ref dies “Package::Stash->new must be passed the name of the package to access”.

add_symbol

Install `$initial` in a slot: a reference installs its referent, a plain value a copy of it, nothing an empty value. Under the debugger (`$^P & 0x10`) a sub also gets its `%DB::sub` entry, from `filename`, `first_line_num` and `last_line_num` options.

Synopsis

```
$stash->add_symbol('$x', \1);
$stash->add_symbol('@ISA', ['Base']);
$stash->add_symbol('&f', sub { 42 }, filename => 'gen.pl', first_
↪line_num => 1);
```

Edge cases

- A value of the wrong kind dies “VALUE is not of type TYPE”.
- `&name` with no value dies “Don’t know how to vivify CODE variables”.
- A name containing `::` dies “Variable names may not contain ::”.

5.2.78 PadWalker

Reach into another subroutine’s `my` and `our` variables - read them, change them, find out their names - from anywhere on the call stack.

`PadWalker` is the tool of choice for debuggers, introspection utilities, and any code that needs to look at variables it did not declare. It is what Perl’s own built-in debugger uses, and it is the modern replacement for `Devel::LexAlias` when you want to alias or substitute a lexical in a caller’s pad.

Every function that walks the stack takes a `$level` argument interpreted just like the argument to `caller`:

- `0` - the current scope.
- `1` - the scope of whoever called the current sub.
- `2` - two frames up, and so on.

File scope is a valid destination: asking for a `$level` deeper than the call stack supports raises `Not nested deeply enough`.

Returned hashes key on the variable name *including* its sigil - so `$x` appears under the key `'$x'`, `@items` under `'@items'`. Values are always references to the live variable, so assigning through the reference updates the original.

```
use PadWalker qw(peek_my closed_over var_name set_closed_over);

sub inner {
    my $vars = peek_my(1);                          # the caller's␣
↪lexicals
    ${ $vars->{'$count'} }++;                       # change the caller
↪'s $count
    return join ",", sort keys %$vars;
}
sub outer {
    my $count = 41;
    my @items = (1, 2);
    my $names = inner();
    print "$names; count is now $count\n";      # $count,@items;␣
↪count is now 42
}
outer();

my $x = 10;
my $add = sub { $_[0] + $x };
print join(",", keys %{ closed_over($add) }), " | ", var_name(0, \
↪$x), "\n";   # $x | $x
set_closed_over($add, { '$x' => \100 });
print $add->(1), "\n";                              # 101
```

The functions are peek_my, peek_our, peek_sub (the lexicals a sub declares), closed_over and set_closed_over (the variables a closure captured), and var_name (the name of a variable, given a reference to it). This is a native implementation of PadWalker 2.5; its PadWalker.pm is used unchanged, and nothing is exported by default.

Differences from upstream

None.

Functions

Lexical inspection

peek_my

Return every my variable in scope at a given call-stack depth as a hash of name-to-reference pairs.

peek_our

Return every our variable in scope at a given call-stack depth as a hash of name-to-reference pairs.

peek_sub

Return the `my` variables declared inside a given subroutine, whether or not the sub is currently on the call stack.

closed_over

Return the lexicals that a subroutine closes over - the variables it references but did not itself declare.

var_name

Given a reference to a variable, return the name that variable has in a specified subroutine's pad - or `undef` if it is not there.

Lexical modification

set_closed_over

Replace the lexicals that a subroutine closes over with a new set of references.

peek_my

Return every `my` variable in scope at a given call-stack depth as a hash of name-to-reference pairs.

Synopsis

```
use PadWalker qw(peek_my);
my $h = peek_my($level);
```

What you get back

A hash reference. Each key is a variable name *with* its sigil - `'$x'`, `'@list'`, `'%env'`. Each value is a reference to the live variable, so dereferencing and assigning through it changes the original. `our` variables are not included; use `peek_our` for those.

Examples

Read a `my` variable in the current scope:

```
my $x = 12;
my $h = peek_my(0);
print ${ $h->{'$x'} };      # 12
```

Modify the caller's `my` variable:

```
sub bump_x {
    my $h = peek_my(1);
```

(continues on next page)

(continued from previous page)

```
    ${ $h->{'$x'} }++;
}
my $x = 5;
bump_x();
print $x;                   # 6
```

A `my` declared deeper than the target frame is not in the returned hash - only variables visible at the call site come back.

Edge cases

- Called with other than one argument: croaks with `Usage: PadWalker::peek_my(uplevel)`.
- `$level` deeper than the call stack croaks with `Not nested deeply enough`.
- Negative `$level` is not supported and croaks.
- Variables declared inside an `eval BLOCK` between the target frame and the call site are folded in at the sequence number of the call - they become visible alongside the target's own lexicals.

Differences from upstream

Fully compatible with upstream PadWalker 2.5.

See also

- `peek_our` - same idea for `our` variables.
- `peek_sub` - inspect a CV's pad directly, without walking frames.
- `var_name` - given a reference, find the name it had in a pad.

peek_our

Return every `our` variable in scope at a given call-stack depth as a hash of name-to-reference pairs.

Synopsis

```
use PadWalker qw(peek_our);
my $h = peek_our($level);
```

What you get back

A hash reference keyed on sigiled variable names, with values that are references to the package variables those `our` declarations alias. Dereferencing reads from the package slot; assigning through the reference writes back to it.

Examples

```
package Main;
our $config = 'ok';
my $h = peek_our(0);
print ${ $h->{'$config'} };  # ok
```

Inspect a caller's our variables:

```
sub show_our {
    my $h = peek_our(1);
    print join ",", sort keys %$h;
}
```

Edge cases

- `$level` works exactly as in `peek_my` - `0` for the current scope, `1` for the caller, and so on.
- `our` declarations are resolved through the package stash in effect at the declaration site, not the calling site.
- A package with an `our` name but no value yet appears in the hash with a reference to `undef`.

Differences from upstream

Fully compatible with upstream PadWalker 2.5.

See also

- `peek_my` - the sibling for `my` variables.
- `var_name` - resolve a reference back to its declared name.

peek_sub

Return the `my` variables declared inside a given subroutine, whether or not the sub is currently on the call stack.

Synopsis

```
use PadWalker qw(peek_sub);
my $h = peek_sub(\&some_sub);
```

What you get back

A hash reference keyed on sigiled variable names. The references point at whatever values the pad currently holds. If the sub is not active, those values will typically be `undef` for scalars or empty for aggregates - the sub has not yet run to populate them. If the sub *is* active, the references point at the live values of the most recent invocation.

Examples

```
my $x = "Hello!";
my $r = peek_sub(sub { $x })->{'$x'};
print $$r;                    # Hello!
```

Inspect the pad of a named sub:

```
sub counter { my $n = 0; sub { $n++ } }
my $h = peek_sub(\&counter);
print join ",", sort keys %$h;   # $n
```

Edge cases

- If the sub declares several `my` variables with the same name, only the last one is returned. There is no way to distinguish shadowed entries; this is a known upstream limitation.
- A sub with no pad of its own - one implemented natively rather than written in Perl - croaks with `PadWalker: cv has no padlist`.
- A sub name (a string) is looked up as `\&{"name"}` would be; anything else that is not code croaks with `PadWalker::peek_sub: cv is not a CODE reference`.

Differences from upstream

Fully compatible with upstream PadWalker 2.5.

See also

- `closed_over` - the strict subset of `peek_sub` covering only variables the sub closes over.
- `peek_my` - walk the live call stack instead of a named sub.
- `var_name` - name-from-reference lookup against the same pad.

closed_over

Return the lexicals that a subroutine closes over - the variables it references but did not itself declare.

Synopsis

```
use PadWalker qw(closed_over);
my $h = closed_over(\&some_sub);
my ($h, $slots) = closed_over(\&some_sub);
```

What you get back

In scalar context, a hash reference keyed on sigiled variable names with references to the enclosing variables as values. In list context, that hash reference plus a second hash reference keyed on the sub's pad slot indices (as numbers) - useful for tools that need to match up pad positions with values.

Only captured lexicals appear; `my` variables declared inside the sub itself and `our` variables are both excluded.

Examples

Classic closure:

```
my $n = 41;
my $adder = sub { $n + $_[0] };
my $h = closed_over($adder);
${ $h->{'$n'} } = 100;
print $adder->(1);            # 101
```

List form for serialisers that need pad indices:

```
my ($names, $slots) = closed_over($sub);
for my $idx (sort { $a <=> $b } keys %$slots) {
    print "slot $idx -> ", ${ $slots->{$idx} }, "\n";
}
```

Edge cases

- A sub that captures nothing returns an empty hash reference.
- A sub with no pad of its own - one implemented natively rather than written in Perl - returns empty hashes rather than croaking.
- A sub name (a string) is looked up as `\&{"name"}` would be; anything else that is not code croaks with `PadWalker::closed_over: cv is not a CODE reference`.

Differences from upstream

Fully compatible with upstream PadWalker 2.5.

See also

- `set_closed_over` - substitute the captured variables wholesale.
- `peek_sub` - full pad, not just the captures.
- `var_name` - given one of the returned references, find its name.

set_closed_over

Replace the lexicals that a subroutine closes over with a new set of references.

Synopsis

```
use PadWalker qw(set_closed_over);
set_closed_over(\&some_sub, \%replacements);
```

What you get back

Nothing useful - the function is called for its side effect. For every name in `%replacements` that matches one of the sub's captured pad slots, the corresponding slot is replaced by the referent of `$replacements{$name}`.

Examples

Swap a closure's captured variable for a different scalar:

```
my $n = 1;
my $say = sub { print "$n\n" };
my $other = 42;
set_closed_over($say, { '$n' => \$other });
$say->();                    # 42
```

Round-trip through `closed_over` is the idiomatic pattern:

```
my $refs = closed_over($sub);
$refs->{'$x'} = \my $fresh;
set_closed_over($sub, $refs);
```

Edge cases

- Names in `%replacements` that the sub does not close over are silently ignored.
- The reftype of each replacement must match the original - scalar for scalar, array for array, and so on. A mismatch croaks with `Incorrect reftype for variable NAME (got X expected Y)`.
- A non-reference value in `%replacements` croaks with `The variable for NAME is not a reference`.
- A second argument that is not a hash reference croaks with `PadWalker::set_closed_over: pad is not a HASH reference`.

Differences from upstream

Upstream PadWalker 2.5 crashes when the first argument is not a code reference, or is a sub with no pad of its own (one implemented natively rather than written in Perl); here both are a no-op.

See also

- `closed_over` - read the captures this function writes.
- `peek_sub` - broader inspection, read-only.

var_name

Given a reference to a variable, return the name that variable has in a specified subroutine's pad - or `undef` if it is not there.

Synopsis

```
use PadWalker qw(var_name);
my $name = var_name($level, \$var);
my $name = var_name(\&sub,  \$var);
```

What you get back

A string - the sigiled pad name, e.g. `'$foo'` or `'@items'`. If the reference does not correspond to any pad slot in the target sub, returns `undef`.

The first argument selects the sub to search: a non-negative integer is interpreted the same as the argument to `peek_my` (walk the call stack), a code reference is searched directly.

Examples

Look up the name of a local variable by reference:

```
my $foo;
print var_name(0, \$foo);        # $foo
```

Wrap it for use from helper subs:

```
sub my_name {
    return var_name(1, shift);
}
my $bar;
print my_name(\$bar);            # $bar
```

Edge cases

- References to globals, package variables, or anonymous SVs return `undef` - only pad entries are searched.
- If several pad slots point at the same SV, the last one wins (same caveat as `peek_sub`).
- The second argument must be a reference; passing a plain value croaks with `Usage: PadWalker::var_name(sub, var_ref)`.

Differences from upstream

Fully compatible with upstream PadWalker 2.5.

See also

- peek_my - the inverse direction, names to references.
- peek_sub - all names in a sub's pad regardless of value.
- closed_over - captured lexicals only.

5.2.79 Params

Namespace containing the following modules:

- Params::Check - A generic input parsing/checking mechanism.
- Params::Util - Check parameters: each function returns its argument if it is a value of the kind the name says, and undef if not.

Params::Check

A generic input parsing/checking mechanism.

SYNOPSIS

```
use Params::Check qw[check allow last_error];

sub fill_personal_info {
    my %hash = @_;
    my $x;

    my $tmpl = {
        firstname   => { required   => 1, defined => 1 },
        lastname    => { required   => 1, store => \$x },
        gender      => { required   => 1,
                         allow      => [qr/M/i, qr/F/i],
                       },
        married     => { allow      => [0,1] },
        age         => { default    => 21,
                         allow      => qr/^\d+$/,
                       },

        phone       => { allow => [ sub { return 1 if /$valid_re/ },
                                    '1-800-PERL' ]
                       },
        id_list     => { default        => [],
                         strict_type    => 1
                       },
        employer    => { default => 'NSA', no_override => 1 },
    };
```

(continues on next page)

(continued from previous page)

```
    ### check() returns a hashref of parsed args on success ###
    my $parsed_args = check( $tmpl, \%hash, $VERBOSE )
                        or die qw[Could not parse arguments!];

    ... other code here ...
}

my $ok = allow( $colour, [qw|blue green yellow|] );

my $error = Params::Check::last_error();
```

DESCRIPTION

Params::Check is a generic input parsing/checking mechanism.

It allows you to validate input via a template. The only requirement is that the arguments must be named.

Params::Check can do the following things for you:

- Convert all keys to lowercase
- Check if all required arguments have been provided
- Set arguments that have not been provided to the default
- Weed out arguments that are not supported and warn about them to the user
- Validate the arguments given by the user based on strings, regexes, lists or even subroutines
- Enforce type integrity if required

Most of Params::Check's power comes from its template, which we'll discuss below:

Template

As you can see in the synopsis, based on your template, the arguments provided will be validated.

The template can take a different set of rules per key that is used.

The following rules are available:

- default

  This is the default value if none was provided by the user. This is also the type `strict_type` will look at when checking type integrity (see below).

- required

  A boolean flag that indicates if this argument was a required argument. If marked as required and not provided, check() will fail.

- strict_type

This does a `ref()` check on the argument provided. The `ref` of the argument must be the same as the `ref` of the default value for this check to pass.

This is very useful if you insist on taking an array reference as argument for example.

- defined

  If this template key is true, enforces that if this key is provided by user input, its value is `defined`. This just means that the user is not allowed to pass `undef` as a value for this key and is equivalent to: allow => sub { defined $_[0] && OTHER TESTS }

- no_override

  This allows you to specify `constants` in your template. ie, they keys that are not allowed to be altered by the user. It pretty much allows you to keep all your `configurable` data in one place; the `Params::Check` template.

- store

  This allows you to pass a reference to a scalar, in which the data will be stored:

```
my $x;
my $args = check(foo => { default => 1, store => \$x }, $input);
```

  This is basically shorthand for saying:

```
my $args = check( { foo => { default => 1 }, $input );
my $x    = $args->{foo};
```

  You can alter the global variable $Params::Check::NO_DUPLICATES to control whether the `store`'d key will still be present in your result set. See the "Global Variables" section below.

- allow

  A set of criteria used to validate a particular piece of data if it has to adhere to particular rules.

  See the `allow()` function for details.

Functions

check(\%tmpl, \%args, [$verbose]);

This function is not exported by default, so you'll have to ask for it via:

```
use Params::Check qw[check];
```

or use its fully qualified name instead.

`check` takes a list of arguments, as follows:

- Template

  This is a hash reference which contains a template as explained in the `SYNOPSIS` and `Template` section.

- Arguments

  This is a reference to a hash of named arguments which need checking.

- Verbose

  A boolean to indicate whether `check` should be verbose and warn about what went wrong in a check or not.

  You can enable this program wide by setting the package variable `$Params::Check::VERBOSE` to a true value. For details, see the section on `Global Variables` below.

`check` will return when it fails, or a hashref with lowercase keys of parsed arguments when it succeeds.

So a typical call to check would look like this:

```
my $parsed = check( \%template, \%arguments, $VERBOSE )
                or warn q[Arguments could not be parsed!];
```

A lot of the behaviour of `check()` can be altered by setting package variables. See the section on `Global Variables` for details on this.

allow($test_me, \@criteria);

The function that handles the `allow` key in the template is also available for independent use.

The function takes as first argument a key to test against, and as second argument any form of criteria that are also allowed by the `allow` key in the template.

You can use the following types of values for allow:

- string

  The provided argument MUST be equal to the string for the validation to pass.

- regexp

  The provided argument MUST match the regular expression for the validation to pass.

- subroutine

  The provided subroutine MUST return true in order for the validation to pass and the argument accepted.

  (This is particularly useful for more complicated data).

- array ref

  The provided argument MUST equal one of the elements of the array ref for the validation to pass. An array ref can hold all the above values.

It returns true if the key matched the criteria, or false otherwise.

last_error()

Returns a string containing all warnings and errors reported during the last time `check` was called.

This is useful if you want to report then some other way than `carp`'ing when the verbose flag is on.

It is exported upon request.

Global Variables

The behaviour of Params::Check can be altered by changing the following global variables:

$Params::Check::VERBOSE

This controls whether Params::Check will issue warnings and explanations as to why certain things may have failed. If you set it to 0, Params::Check will not output any warnings.

The default is 1 when *warnings* are enabled, 0 otherwise;

$Params::Check::STRICT_TYPE

This works like the `strict_type` option you can pass to `check`, which will turn on `strict_type` globally for all calls to `check`.

The default is 0;

$Params::Check::ALLOW_UNKNOWN

If you set this flag, unknown options will still be present in the return value, rather than filtered out. This is useful if your subroutine is only interested in a few arguments, and wants to pass the rest on blindly to perhaps another subroutine.

The default is 0;

$Params::Check::STRIP_LEADING_DASHES

If you set this flag, all keys passed in the following manner:

```
function( -key => 'val' );
```

will have their leading dashes stripped.

$Params::Check::NO_DUPLICATES

If set to true, all keys in the template that are marked as to be stored in a scalar, will also be removed from the result set.

Default is false, meaning that when you use `store` as a template key, `check` will put it both in the scalar you supplied, as well as in the hashref it returns.

$Params::Check::PRESERVE_CASE

If set to true, *Params::Check* will no longer convert all keys from the user input to lowercase, but instead expect them to be in the case the template provided. This is useful when you want to use similar keys with different casing in your templates.

Understand that this removes the case-insensitivity feature of this module.

Default is 0;

$Params::Check::ONLY_ALLOW_DEFINED

If set to true, *Params::Check* will require all values passed to be `defined`. If you wish to enable this on a 'per key' basis, use the template option `defined` instead.

Default is 0;

$Params::Check::SANITY_CHECK_TEMPLATE

If set to true, *Params::Check* will sanity check templates, validating for errors and unknown keys. Although very useful for debugging, this can be somewhat slow in hot-code and large loops.

To disable this check, set this variable to `false`.

Default is 1;

$Params::Check::WARNINGS_FATAL

If set to true, *Params::Check* will `croak` when an error during template validation occurs, rather than return `false`.

Default is 0;

$Params::Check::CALLER_DEPTH

This global modifies the argument given to `caller()` by `Params::Check::check()` and is useful if you have a custom wrapper function around `Params::Check::check()`. The value must be an integer, indicating the number of wrapper functions inserted between the real function call and `Params::Check::check()`.

Example wrapper function, using a custom stacktrace:

```
sub check {
    my ($template, $args_in) = @_;

    local $Params::Check::WARNINGS_FATAL = 1;
    local $Params::Check::CALLER_DEPTH = $Params::Check::CALLER_
↪DEPTH + 1;
    my $args_out = Params::Check::check($template, $args_in);

    my_stacktrace(Params::Check::last_error) unless $args_out;
```

(continues on next page)

(continued from previous page)

```
    return $args_out;
}
```

Default is 0;

Acknowledgements

Thanks to Richard Soderberg for his performance improvements.

BUG REPORTS

Please report bugs or other issues to bug-params-check@rt.cpan.org.

AUTHOR

This module by Jos Boumans kane@cpan.org.

COPYRIGHT

This library is free software; you may redistribute and/or modify it under the same terms as Perl itself.

Params::Util

Check parameters: each function returns its argument if it is a value of the kind the name says, and `undef` if not.

`_STRING` is a defined, non-empty, non-reference value; `_NUMBER` one that is or looks like a number. `_SCALAR`, `_ARRAY`, `_HASH` and `_CODE` are unblessed references to such a thing, the plain names non-empty, the `0` forms (`_SCALAR0`, `_ARRAY0`, `_HASH0`) empty too. `_ARRAYLIKE`, `_HASHLIKE` and `_CODELIKE` also accept objects that overload `@{}`, `%{}` or `&{}`. `_REGEX` is a compiled regex, `_INSTANCE($obj, $class)` an object that `isa $class`.

These are the compiled functions of Params::Util 1.102; its `Params/Util.pm` and `Params/Util/PP.pm` are used unchanged, and the functions it has no compiled form of (`_IDENTIFIER`, `_CLASS`, `_POSINT`, `_SET`, `_HANDLE`, ...) come from `Params::Util::PP`.

Synopsis

```
use Params::Util qw(_STRING _HASH _INSTANCE);

sub new {
    my $class = shift;
    my $name  = _STRING(shift)         or die "no name";
    my $opts  = _HASH(shift) // {};
    my $log   = _INSTANCE(shift, 'Log') or die "no logger";
    ...
}
```

Differences from upstream

None.

Modules

- `Params::Util::PP` - PurePerl Params::Util routines

Functions

Value checks

_STRING

The value itself if it is a defined, non-reference string of at least one character (a number counts, in its string form), else undef.

Synopsis

```
my $name = _STRING($in) or die "need a name";
```

_NUMBER

The value itself if it is a number, or a non-empty string that looks like one (looks_like_number), else undef.

Synopsis

```
my $n = _NUMBER($in) // die "not a number";
```

Reference checks

_SCALAR0

An unblessed reference to a scalar, whatever it holds, else undef.

_SCALAR

An unblessed reference to a scalar holding a non-empty string, else undef.

Synopsis

```
my $text = _SCALAR($ref) or die "need a reference to some text";
```

_REGEX

A compiled regex (qr//), else undef.

_ARRAY0

An unblessed array reference, empty or not, else undef.

_ARRAY

An array reference with at least one element, else undef.

Synopsis

```
my $list = _ARRAY($in) or die "need a non-empty list";
```

_ARRAYLIKE

An array reference, or an object that overloads @{}, else undef.

_HASH0

A hash reference, empty or not, else undef.

_HASH

A hash reference with at least one key, else undef.

Synopsis

```
my $opts = _HASH($in) || {};
```

_HASHLIKE

A hash reference, or an object that overloads %{}, else undef.

_CODE

A code reference, else undef.

_CODELIKE

A code reference, or an object that overloads &{}, else undef.

_INSTANCE

An object that isa the class given (its own isa method decides), else undef; an empty class name is never matched.

Synopsis

```
my $dbh = _INSTANCE($in, 'DBI::db') or die "need a database handle";
```

Params::Util::PP

PurePerl Params::Util routines

5.2.80 Parse

Namespace containing the following modules:

- `Parse::CPAN`

Parse::CPAN

Namespace containing the following modules:

- `Parse::CPAN::Meta` - Parse META.yml and META.json CPAN metadata files

Parse::CPAN::Meta

Parse META.yml and META.json CPAN metadata files

VERSION

version 2.150013

SYNOPSIS

```
################################################
# In your file

---
name: My-Distribution
version: 1.23
resources:
  homepage: "http://example.com/dist/My-Distribution"


################################################
# In your program

use Parse::CPAN::Meta;

my $distmeta = Parse::CPAN::Meta->load_file('META.yml');

# Reading properties
my $name     = $distmeta->{name};
my $version  = $distmeta->{version};
my $homepage = $distmeta->{resources}{homepage};
```

DESCRIPTION

Parse::CPAN::Meta is a parser for `META.json` and `META.yml` files, using *JSON::PP* and/or *CPAN::Meta::YAML*.

Parse::CPAN::Meta provides three methods: `load_file`, `load_json_string`, and `load_yaml_string`. These will read and deserialize CPAN metafiles, and are described below in detail.

Parse::CPAN::Meta provides a legacy API of only two functions, based on the YAML functions of the same name. Wherever possible, identical calling semantics are used. These may only be used with YAML sources.

All error reporting is done with exceptions (die'ing).

Note that META files are expected to be in UTF-8 encoding, only. When converted string data, it must first be decoded from UTF-8.

METHODS

load_file

```
my $metadata_structure = Parse::CPAN::Meta->load_file('META.json');

my $metadata_structure = Parse::CPAN::Meta->load_file('META.yml');
```

This method will read the named file and deserialize it to a data structure, determining whether it should be JSON or YAML based on the filename. The file will be read using the ":utf8" IO layer.

load_yaml_string

```
my $metadata_structure = Parse::CPAN::Meta->load_yaml_string($yaml_
↪string);
```

This method deserializes the given string of YAML and returns the first document in it. (CPAN metadata files should always have only one document.) If the source was UTF-8 encoded, the string must be decoded before calling `load_yaml_string`.

load_json_string

```
my $metadata_structure = Parse::CPAN::Meta->load_json_string($json_
↪string);
```

This method deserializes the given string of JSON and the result. If the source was UTF-8 encoded, the string must be decoded before calling `load_json_string`.

load_string

```
my $metadata_structure = Parse::CPAN::Meta->load_string($some_
↪string);
```

If you don't know whether a string contains YAML or JSON data, this method will use some heuristics and guess. If it can't tell, it assumes YAML.

yaml_backend

```
my $backend = Parse::CPAN::Meta->yaml_backend;
```

Returns the module name of the YAML serializer. See "ENVIRONMENT" for details.

json_backend

```
my $backend = Parse::CPAN::Meta->json_backend;
```

Returns the module name of the JSON serializer. If `CPAN_META_JSON_BACKEND` is set, this will be whatever that's set to. If not, this will either be *JSON::PP* or *JSON*. If `PERL_JSON_BACKEND` is set, this will return *JSON* as further delegation is handled by the *JSON* module. See "ENVIRONMENT" for details.

json_decoder

```
my $decoder = Parse::CPAN::Meta->json_decoder;
```

Returns the module name of the JSON decoder. Unlike "json_backend", this is not necessarily a full *JSON*-style module, but only something that will provide a `decode_json` subroutine. If `CPAN_META_JSON_DECODER` is set, this will be whatever that's set to. If not, this will be whatever has been selected as "json_backend". See "ENVIRONMENT" for more notes.

FUNCTIONS

For maintenance clarity, no functions are exported by default. These functions are available for backwards compatibility only and are best avoided in favor of `load_file`.

Load

```
my @yaml = Parse::CPAN::Meta::Load( $string );
```

Parses a string containing a valid YAML stream into a list of Perl data structures.

LoadFile

```
my @yaml = Parse::CPAN::Meta::LoadFile( 'META.yml' );
```

Reads the YAML stream from a file instead of a string.

ENVIRONMENT

CPAN_META_JSON_DECODER

By default, *JSON::PP* will be used for deserializing JSON data. If the `CPAN_META_JSON_DECODER` environment variable exists, this is expected to be the name of a loadable module that provides a `decode_json` subroutine, which will then be used for deserialization. Relying only on the existence of said subroutine allows for maximum compatibility, since this API is provided by all of *JSON::PP*, *JSON::XS*, Cpanel::JSON::XS, JSON::MaybeXS, JSON::Tiny, and Mojo::JSON.

CPAN_META_JSON_BACKEND

By default, *JSON::PP* will be used for deserializing JSON data. If the `CPAN_META_JSON_BACKEND` environment variable exists, this is expected to be the name of a loadable module that provides the *JSON* API, since downstream code expects to be able to call `new` on this class. As such, while *JSON::PP*, *JSON::XS*, Cpanel::JSON::XS and JSON::MaybeXS will work for this, to use Mojo::JSON or JSON::Tiny for decoding requires setting "CPAN_META_JSON_DECODER".

PERL_JSON_BACKEND

If the `CPAN_META_JSON_BACKEND` environment variable does not exist, and if `PERL_JSON_BACKEND` environment variable exists, is true and is not "JSON::PP", then the *JSON* module (version 2.5 or greater) will be loaded and used to interpret `PERL_JSON_BACKEND`. If *JSON* is not installed or is too old, an exception will be thrown. Note that at the time of writing, the only useful values are 1, which will tell *JSON* to guess, or *JSON::XS* - if you want to use a newer JSON module, see "CPAN_META_JSON_BACKEND".

PERL_YAML_BACKEND

By default, *CPAN::Meta::YAML* will be used for deserializing YAML data. If the `PERL_YAML_BACKEND` environment variable is defined, then it is interpreted as a module to use for deserialization. The given module must be installed, must load correctly and must implement the `Load()` function or an exception will be thrown.

AUTHORS

- David Golden dagolden@cpan.org
- Ricardo Signes cpan@semiotic.systems
- Adam Kennedy adamk@cpan.org

COPYRIGHT AND LICENSE

This software is copyright (c) 2010 by David Golden, Ricardo Signes, Adam Kennedy and Contributors.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

5.2.81 Perl

Namespace containing the following modules:

- `Perl::OSType` - Map Perl operating system names to generic types

Perl::OSType

Map Perl operating system names to generic types

VERSION

version 1.010

SYNOPSIS

```
use Perl::OSType ':all';

$current_type = os_type();
$other_type = os_type('dragonfly'); # gives 'Unix'
```

DESCRIPTION

Modules that provide OS-specific behaviors often need to know if the current operating system matches a more generic type of operating systems. For example, 'linux' is a type of 'Unix' operating system and so is 'freebsd'.

This module provides a mapping between an operating system name as given by $^0 and a more generic type. The initial version is based on the OS type mappings provided in Module::Build and *ExtUtils::CBuilder*. (Thus, Microsoft operating systems are given the type 'Windows' rather than 'Win32'.)

USAGE

No functions are exported by default. The export tag ":all" will export all functions listed below.

os_type()

```
$os_type = os_type();
$os_type = os_type('MSWin32');
```

Returns a single, generic OS type for a given operating system name. With no arguments, returns the OS type for the current value of $^0. If the operating system is not recognized, the function will return the empty string.

is_os_type()

```
$is_windows = is_os_type('Windows');
$is_unix    = is_os_type('Unix', 'dragonfly');
```

Given an OS type and OS name, returns true or false if the OS name is of the given type. As with `os_type`, it will use the current operating system as a default if no OS name is provided.

SEE ALSO

- Devel::CheckOS

SUPPORT

Bugs / Feature Requests

Please report any bugs or feature requests through the issue tracker at https://github.com/Perl-Toolchain-Gang/Perl-OSType/issues. You will be notified automatically of any progress on your issue.

Source Code

This is open source software. The code repository is available for public review and contribution under the terms of the license.

https://github.com/Perl-Toolchain-Gang/Perl-OSType

```
git clone https://github.com/Perl-Toolchain-Gang/Perl-OSType.git
```

AUTHOR

David Golden dagolden@cpan.org

CONTRIBUTORS

- Chris 'BinGOs' Williams chris@bingosnet.co.uk
- David Golden xdg@xdg.me
- Graham Ollis plicease@cpan.org
- Jonas B. Nielsen jonasbn@hoarfrost.local
- Owain G. Ainsworth oga@nicotinebsd.org
- Paul Green Paul.Green@stratus.com
- Piotr Roszatycki piotr.roszatycki@gmail.com

COPYRIGHT AND LICENSE

This software is copyright (c) 2016 by David Golden.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

5.2.82 PerlIO

On demand loader for PerlIO layers and root of PerlIO::* name space

SYNOPSIS

```
# support platform-native and CRLF text files
open(my $fh, "<:crlf", "my.txt") or die "open failed: $!";

# append UTF-8 encoded text
open(my $fh, ">>:encoding(UTF-8)", "some.log")
  or die "open failed: $!";

# portably open a binary file for reading
open(my $fh, "<", "his.jpg") or die "open failed: $!";
binmode($fh) or die "binmode failed: $!";

Shell:
  PERLIO=:perlio perl ....
```

DESCRIPTION

When an undefined layer 'foo' is encountered in an `open` or `binmode` layer specification then C code performs the equivalent of:

```
use PerlIO 'foo';
```

The Perl code in PerlIO.pm then attempts to locate a layer by doing

```
require PerlIO::foo;
```

Otherwise the `PerlIO` package is a place holder for additional PerlIO related functions.

Layers

Generally speaking, PerlIO layers (previously sometimes referred to as "disciplines") are an ordered stack applied to a filehandle (specified as a space- or colon-separated list, conventionally written with a leading colon). Each layer performs some operation on any input or output, except when bypassed such as with `sysread` or `syswrite`. Read operations go through the stack in the order they are set (left to right), and write operations in the reverse order.

There are also layers which actually just set flags on lower layers, or layers that modify the current stack but don't persist on the stack themselves; these are referred to as pseudo-layers.

When opening a handle, it will be opened with any layers specified explicitly in the open() call (or the platform defaults, if specified as a colon with no following layers).

If layers are not explicitly specified, the handle will be opened with the layers specified by the *${^OPEN}* variable (usually set by using the *open* pragma for a lexical scope, or the `-C` command-line switch or `PERL_UNICODE` environment variable for the main program scope).

If layers are not specified in the open() call or `${^OPEN}` variable, the handle will be opened with the default layer stack configured for that architecture; see "Defaults and how to override them".

Some layers will automatically insert required lower level layers if not present; for example `:perlio` will insert `:unix` below itself for low level IO, and `:encoding` will insert the platform defaults for buffered IO.

The `binmode` function can be called on an opened handle to push additional layers onto the stack, which may also modify the existing layers. `binmode` called with no layers will remove or unset any existing layers which transform the byte stream, making the handle suitable for binary data.

The following layers are currently defined:

- :unix

  Lowest level layer which provides basic PerlIO operations in terms of UNIX/POSIX numeric file descriptor calls (open(), read(), write(), lseek(), close()). It is used even on non-Unix architectures, and most other layers operate on top of it.

- :stdio

  Layer which calls `fread`, `fwrite` and `fseek`/`ftell` etc. Note that as this is "real" stdio it will ignore any layers beneath it and go straight to the operating system via the C library as usual. This layer implements both low level IO and buffering, but is rarely used on modern architectures.

- :perlio

  A from scratch implementation of buffering for PerlIO. Provides fast access to the buffer for `sv_gets` which implements Perl's readline/<> and in general attempts to minimize data copying.

  `:perlio` will insert a `:unix` layer below itself to do low level IO.

- :crlf

  A layer that implements DOS/Windows like CRLF line endings. On read converts pairs of CR,LF to a single "\n" newline character. On write converts each "\n" to a CR,LF pair. Note that this layer will silently refuse to be pushed on top of itself.

  It currently does *not* mimic MS-DOS as far as treating of Control-Z as being an end-of-file marker.

  On DOS/Windows like architectures where this layer is part of the defaults, it also acts like the `:perlio` layer, and removing the CRLF translation (such as with `:raw`) will only unset the CRLF translation flag. Since Perl 5.14, you can also apply another `:crlf` layer later, such as when the CRLF translation must occur after an encoding layer. On other architectures, it is a mundane CRLF translation layer and can be added and removed normally.

```
# translate CRLF after encoding on Perl 5.14 or newer
binmode $fh, ":raw:encoding(UTF-16LE):crlf"
  or die "binmode failed: $!";
```

- :utf8

  Pseudo-layer that declares that the stream accepts Perl's *internal* upgraded encoding of characters, which is approximately UTF-8 on ASCII machines, but UTF-EBCDIC on EBCDIC machines. This allows any character Perl can represent to be read from or written to the stream.

  This layer (which actually sets a flag on the preceding layer, and is implicitly set by any `:encoding` layer) does not translate or validate byte sequences.

  CAUTION: Do not use this layer to translate from UTF-8 bytes, as invalid UTF-8 or binary data will result in malformed Perl strings. It is unlikely to produce invalid UTF-8 when used for output, though it will instead produce UTF-EBCDIC on EBCDIC systems. The `:encoding(UTF-8)` layer (hyphen is significant) is preferred as it will ensure translation between valid UTF-8 bytes and valid Unicode characters.

- :bytes

  This is the inverse of the `:utf8` pseudo-layer. It turns off the flag on the layer below so that data read from it is considered to be Perl's internal downgraded encoding, thus interpreted as the native single-byte encoding of Latin-1 or EBCDIC. Likewise on output Perl will warn if a "wide" character (a codepoint not in the range 0..255) is written to a such a stream.

  This is very dangerous to push on a handle using an `:encoding` layer, as such a layer assumes to be working with Perl's internal upgraded encoding, so you will likely get a mangled result. Instead use `:raw` or `:pop` to remove encoding layers.

- :raw

  The `:raw` pseudo-layer is *defined* as being identical to calling `binmode($fh)` - the stream is made suitable for passing binary data, i.e. each byte is passed as-is. The stream will still be buffered (but this was not always true before Perl 5.14).

  In Perl 5.6 and some books the `:raw` layer is documented as the inverse of the `:crlf` layer. That is no longer the case - other layers which would alter the binary nature of the stream are also disabled. If you want UNIX line endings on a platform that normally does CRLF translation, but still want UTF-8 or encoding defaults, the appropriate thing to do is to add `:perlio` to the PERLIO environment variable, or open the handle explicitly with that layer, to replace the platform default of `:crlf`.

  The implementation of `:raw` is as a pseudo-layer which when "pushed" pops itself and then any layers which would modify the binary data stream. (Undoing `:utf8` and `:crlf` may be implemented by clearing flags rather than popping layers but that is an implementation detail.)

  As a consequence of the fact that `:raw` normally pops layers, it usually only makes sense to have it as the only or first element in a layer specification. When used as the first element it provides a known base on which to build e.g.

```
open(my $fh,">:raw:encoding(UTF-8)",...)
  or die "open failed: $!";
```

will construct a "binary" stream regardless of the platform defaults, but then enable UTF-8 translation.

- :pop

  A pseudo-layer that removes the top-most layer. Gives Perl code a way to manipulate the layer stack. Note that :pop only works on real layers and will not undo the effects of pseudo-layers or flags like :utf8. An example of a possible use might be:

```
open(my $fh,...) or die "open failed: $!";
...
binmode($fh,":encoding(...)") or die "binmode failed: $!";
# next chunk is encoded
...
binmode($fh,":pop") or die "binmode failed: $!";
# back to un-encoded
```

  A more elegant (and safer) interface is needed.

Custom Layers

Some custom layers come with the Perl distribution.

- :encoding

  Use :encoding(ENCODING) to transparently do character set and encoding transformations, for example from Shift-JIS to Unicode. Note that an :encoding also enables :utf8. See *PerlIO::encoding* for more information.

- :mmap

  A layer which implements "reading" of files by using mmap() to make a (whole) file appear in the process's address space, and then using that as PerlIO's "buffer". This *may* be faster in certain circumstances for large files, and may result in less physical memory use when multiple processes are reading the same file.

  Files which are not mmap()-able revert to behaving like the :perlio layer. Writes also behave like the :perlio layer, as mmap() for write needs extra house-keeping (to extend the file) which negates any advantage.

  The :mmap layer will not exist if the platform does not support mmap(). See *PerlIO::mmap* for more information.

- :via

  :via(MODULE) allows a transformation to be applied by an arbitrary Perl module, for example compression / decompression, encryption / decryption. See *PerlIO::via* for more information.

- :scalar

  A layer implementing "in memory" files using scalar variables, automatically used in place of the platform defaults for IO when opening such a handle. As such, the scalar is expected to act like a file, only containing or storing bytes. See *PerlIO::scalar* for more information.

Alternatives to raw

To get a binary stream an alternate method is to use:

```
open(my $fh,"<","whatever") or die "open failed: $!";
binmode($fh) or die "binmode failed: $!";
```

This has the advantage of being backward compatible with older versions of Perl that did not use PerlIO or where `:raw` was buggy (as it was before Perl 5.14).

To get an unbuffered stream specify an unbuffered layer (e.g. `:unix`) in the open call:

```
open(my $fh,"<:unix",$path) or die "open failed: $!";
```

Defaults and how to override them

If the platform is MS-DOS like and normally does CRLF to "\n" translation for text files then the default layers are:

```
:unix:crlf
```

Otherwise if `Configure` found out how to do "fast" IO using the system's stdio (not common on modern architectures), then the default layers are:

```
:stdio
```

Otherwise the default layers are

```
:unix:perlio
```

Note that the "default stack" depends on the operating system and on the Perl version, and both the compile-time and runtime configurations of Perl. The default can be overridden by setting the environment variable PERLIO to a space or colon separated list of layers, however this cannot be used to set layers that require loading modules like `:encoding`.

This can be used to see the effect of/bugs in the various layers e.g.

```
cd .../perl/t
PERLIO=:stdio  ./perl harness
PERLIO=:perlio ./perl harness
```

For the various values of PERLIO see "PERLIO" in perlrun.

The following table summarizes the default layers on UNIX-like and DOS-like platforms and depending on the setting of `$ENV{PERLIO}`:

```
PERLIO     UNIX-like                   DOS-like
------     ---------                   --------
unset / "" :unix:perlio / :stdio [1]   :unix:crlf
:stdio     :stdio                      :stdio
:perlio    :unix:perlio                :unix:perlio
```

(continues on next page)

(continued from previous page)

```
# [1] ":stdio" if Configure found out how to do "fast stdio"␣
 ↪(depends
# on the stdio implementation) and in Perl 5.8, else ":unix:perlio"
```

Querying the layers of filehandles

The following returns the **names** of the PerlIO layers on a filehandle.

```
my @layers = PerlIO::get_layers($fh); # Or FH, *FH, "FH".
```

The layers are returned in the order an open() or binmode() call would use them, and without colons.

By default the layers from the input side of the filehandle are returned; to get the output side, use the optional `output` argument:

```
my @layers = PerlIO::get_layers($fh, output => 1);
```

(Usually the layers are identical on either side of a filehandle but for example with sockets there may be differences.)

There is no set_layers(), nor does get_layers() return a tied array mirroring the stack, or anything fancy like that. This is not accidental or unintentional. The PerlIO layer stack is a bit more complicated than just a stack (see for example the behaviour of `:raw`). You are supposed to use open() and binmode() to manipulate the stack.

Implementation details follow, please close your eyes.

The arguments to layers are by default returned in parentheses after the name of the layer, and certain layers (like `:utf8`) are not real layers but instead flags on real layers; to get all of these returned separately, use the optional `details` argument:

```
my @layer_and_args_and_flags = PerlIO::get_layers($fh, details =>␣
 ↪1);
```

The result will be up to be three times the number of layers: the first element will be a name, the second element the arguments (unspecified arguments will be `undef`), the third element the flags, the fourth element a name again, and so forth.

You may open your eyes now.

AUTHOR

Nick Ing-Simmons nick@ing-simmons.net

SEE ALSO

"binmode" in perlfunc, *"open" in perlfunc*, perlunicode, perliol, *Encode*

Modules

- `PerlIO::encoding` - Open a filehandle with a transparent character-set encoding filter.
- `PerlIO::mmap` - The `:mmap` PerlIO layer: read a file through a memory mapping instead of `read(2)` calls.
- `PerlIO::scalar` - in-memory IO, scalar IO
- `PerlIO::utf8_strict` - A UTF-8 I/O layer that refuses malformed input.
- `PerlIO::via` - Write a PerlIO layer in Perl: open with `:via(MyClass)` and I/O dispatches to your class.

PerlIO::encoding

Open a filehandle with a transparent character-set encoding filter.

`PerlIO::encoding` provides the `:encoding(NAME)` layer that open and binmode accept. Bytes read through the layer are decoded from NAME into Perl character strings; character strings written through it are encoded back into NAME before reaching the layer below. Both directions go through `Encode`, so every encoding `Encode` knows is available:

```
open my $fh, '<:encoding(UTF-8)', $file or die $!;
binmode STDOUT, ':encoding(iso-8859-1)';
```

The encoding name is resolved with `Encode::find_encoding`. A name it does not know warns `Cannot find encoding "NAME"` and the layer fails to be pushed, with `$!` set to `EINVAL`.

`$PerlIO::encoding::fallback` holds the CHECK value handed to `Encode`'s `encode` and `decode` for every conversion the layer makes. It defaults to `Encode::PERLQQ | Encode::WARN_ON_ERR | Encode::ONLY_PRAGMA_WARNINGS`, so malformed data is substituted and warned about rather than being fatal. Each handle captures the value in effect when its layer is pushed, so changing it afterwards does not affect handles that are already open. It must be an integer: a reference croaks with `PerlIO::encoding::fallback must be an integer`. Whatever it is set to, the layer always adds `Encode::STOP_AT_PARTIAL` and always clears `Encode::LEAVE_SRC`.

A partial character stranded in the buffer is an error wherever it cannot be represented: closing such a handle croaks with `Close with partial character`, and `tell` on one croaks with `Cannot tell at partial character`.

Encodings that report `needs_lines` - the stateful ones that have to see whole lines - make the layer split writes at newlines and flush each line on its own.

This is a native implementation of PerlIO::encoding 0.31; its `PerlIO/encoding.pm` is used unchanged, and the layer loads itself (and `Encode`) the first time `:encoding(...)` is named.

Differences from upstream

None.

PerlIO::mmap

The `:mmap` PerlIO layer: read a file through a memory mapping instead of `read(2)` calls.

```
open my $fh, '<:mmap', $file or die "open: $!";
my @lines = <$fh>;
print join(",", PerlIO::get_layers($fh)), "\n";   # unix,perlio,mmap
```

On the first read the layer maps the rest of the file, from the current position to its end, and reads come straight out of the mapping instead of being copied into a buffer. Only a regular file is mapped; anything else (a pipe, a terminal) reads as an ordinary buffered handle, and writing always goes through an ordinary buffer. Nothing needs to be loaded: naming the layer in open or `binmode` loads it.

This is a native implementation of PerlIO::mmap 0.017.

Differences from upstream

None.

PerlIO::scalar

in-memory IO, scalar IO

SYNOPSIS

```
my $scalar = '';
...
open my $fh, "<",  \$scalar or die;
open my $fh, ">",  \$scalar or die;
open my $fh, ">>", \$scalar or die;
```

or

```
my $scalar = '';
...
open my $fh, "<:scalar",  \$scalar or die;
open my $fh, ">:scalar",  \$scalar or die;
open my $fh, ">>:scalar", \$scalar or die;
```

DESCRIPTION

A filehandle is opened but the file operations are performed "in-memory" on a scalar variable. All the normal file operations can be performed on the handle. The scalar is considered a stream of bytes. Currently fileno($fh) returns -1.

Attempting to open a read-only scalar for writing will fail, and if warnings are enabled, produce a warning.

IMPLEMENTATION NOTE

`PerlIO::scalar` only exists as a stub. One does not need to explicitly `use PerlIO::scalar`.

PerlIO::utf8_strict

A UTF-8 I/O layer that refuses malformed input.

`:utf8_strict` decodes UTF-8 like `:encoding(UTF-8)`, much faster, and like it unlike `:utf8` it validates: a read that meets an ill-formed sequence, a surrogate, a non-shortest form, a code point beyond U+10FFFF or a noncharacter dies. Writing through it encodes as `:utf8` does.

Synopsis

```
open my $fh, '<:utf8_strict', $file or die $!;
my $line = <$fh>;            # dies on malformed UTF-8

open my $lax, '<:utf8_strict(allow_noncharacters)', $file or die $!;
binmode STDOUT, ':utf8_strict';
```

Options

A comma-separated list in parentheses relaxes the check: `allow_surrogates`, `allow_noncharacters`, `allow_nonshortest`, or `loose` for all three; `strict`, the default, for none. An unknown word dies with `Unknown argument to :utf8_strict`.

The errors are `Can't decode ill-formed UTF-8 octet sequence <..>`, with the bytes in hex and `at end of file` when the input ends inside a sequence, and `Can't interchange noncharacter code point U+...`. The handle's error flag is set as well.

This is a native implementation of PerlIO::utf8_strict 0.010's `utf8_strict.xs`; its `PerlIO/utf8_strict.pm` is used unchanged, and perl loads it by itself the first time `:utf8_strict` is named.

Differences from upstream

None.

PerlIO::via

Write a PerlIO layer in Perl: open with `:via(MyClass)` and I/O dispatches to your class.

`PerlIO::via` provides the `:via(CLASS)` layer. A handle opened or binmoded with it routes every I/O operation to a method call on CLASS, so the whole behaviour of a layer - buffering, filtering, transcoding - is defined by an ordinary Perl class:

```
package PerlIO::via::UpperCase;
sub PUSHED { my ($class, $mode, $fh) = @_; bless {}, $class }
sub WRITE  { my ($self, $buf, $fh) = @_; print $fh uc $buf; length
```

(continues on next page)

(continued from previous page)

```
 ↪$buf }
sub FILL   { my ($self, $fh) = @_; scalar <$fh> }

package main;
open my $out, '>:via(UpperCase)', $file or die $!;
print $out "hello\n";                 # the file gets HELLO
```

CLASS is looked up as `PerlIO::via::CLASS` first and as CLASS itself second, so both spellings of the layer name work. If neither package exists the layer warns `Cannot find package 'CLASS'` and fails to be pushed; `:via()` with no name at all warns `No package specified`.

`PUSHED` is the only method a class must supply. It is called with the class name and the open mode. If it returns an object, that object becomes the invocant for every later call; a true non-object return is taken as a failure code and aborts the push. Every other method is optional - when one is missing, the layer falls back to the default behaviour for that operation. The full set is `PUSHED`, `POPPED`, `OPEN`, `FDOPEN`, `SYSOPEN`, `GETARG`, `FILENO`, `READ`, `WRITE`, `FILL`, `CLOSE`, `SEEK`, `TELL`, `UNREAD`, `FLUSH`, `SETLINEBUF`, `CLEARERR`, `ERROR`, `EOF`, `BINMODE` and `UTF8`.

Each method receives the object, then its own arguments, then - when there is a layer below - a filehandle onto that layer. That handle is how a class reads and writes the data passing through it.

Reading goes through `FILL` when the class defines one: the layer buffers what `FILL` returns and hands it upwards a piece at a time. With no `FILL`, `READ` is called instead and must return how many bytes it put in the buffer; a count outside the requested range warns `Invalid return from CLASS::READ = N, expected undef or 0 to M` and fails with `EINVAL`.

`UTF8` is called with a true value when the layer below already delivers characters; returning true marks this layer's data as characters too. `BINMODE` returning `undef`, or not being defined at all, pops the layer.

This is a native implementation of PerlIO::via 0.21; its `PerlIO/via.pm` is used unchanged.

Differences from upstream

None.

Modules

- `PerlIO::via::QuotedPrint` - PerlIO layer for quoted-printable strings

PerlIO::via::QuotedPrint

PerlIO layer for quoted-printable strings

SYNOPSIS

```
use PerlIO::via::QuotedPrint;

open(my $in, '<:via(QuotedPrint)', 'file.qp') or
    die "Can't open file.qp for reading: $!\n";

open(my $out, '>:via(QuotedPrint)', 'file.qp') or
    die "Can't open file.qp for writing: $!\n";
```

DESCRIPTION

This module implements a PerlIO layer that works on files encoded in the quoted-printable format. It will decode from quoted-printable while reading from a handle, and it will encode as quoted-printable while writing to a handle.

EXPORTS

None.

KNOWN BUGS

None.

FEEDBACK

Patches, bug reports, suggestions or any other feedback is welcome.

Patches can be sent as GitHub pull requests at https://github.com/steve-m-hay/PerlIO-via-QuotedPrint/pulls.

Bug reports and suggestions can be made on the CPAN Request Tracker at https://rt.cpan.org/Public/Bug/Report.html?Queue=PerlIO-via-QuotedPrint.

Currently active requests on the CPAN Request Tracker can be viewed at https://rt.cpan.org/Public/Dist/Display.html?Status=Active;Queue=PerlIO-via-QuotedPrint.

Please test this distribution. See CPAN Testers Reports at https://www.cpantesters.org/ for details of how to get involved.

Previous test results on CPAN Testers Reports can be viewed at https://www.cpantesters.org/distro/P/PerlIO-via-QuotedPrint.html.

Please rate this distribution on CPAN Ratings at https://cpanratings.perl.org/rate/?distribution=PerlIO-via-QuotedPrint.

SEE ALSO

PerlIO::via, *MIME::QuotedPrint*.

ACKNOWLEDGEMENTS

Based on an example in the standard library module MIME::QuotedPrint in Perl (version 5.8.0).

AVAILABILITY

The latest version of this module is available from CPAN (see "CPAN" in perlmodlib for details) at

https://metacpan.org/release/PerlIO-via-QuotedPrint or

https://www.cpan.org/authors/id/S/SH/SHAY/ or

https://www.cpan.org/modules/by-module/PerlIO/.

The latest source code is available from GitHub at https://github.com/steve-m-hay/PerlIO-via-QuotedPrint.

INSTALLATION

See the `INSTALL` file.

AUTHOR

Elizabeth Mattijsen <liz@dijkmat.nl>.

Steve Hay <shay@cpan.org> is now maintaining PerlIO::via::QuotedPrint as of version 0.08.

COPYRIGHT

Copyright (C) 2002-2004, 2012 Elizabeth Mattijsen. All rights reserved.

Copyright (C) 2015, 2020 Steve Hay. All rights reserved.

LICENCE

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself, i.e. under the terms of either the GNU General Public License or the Artistic License, as specified in the `LICENCE` file.

VERSION

Version 0.10

DATE

22 May 2022

HISTORY

See the Changes file.

5.2.83 Peta

Namespace containing the following modules:

- `Peta::Dict` - a fast dictionary bound to a Perl hash with `tie`
- `Peta::FFI` - Call C library functions from Perl with no build step and no binding crate.
- `Peta::ReadLine` - Interactive line editing, command history, and configurable key bindings, with the Term::ReadLine::Gnu interface.
- `Peta::Shell` - ipps, the interactive mode of pperl
- `Peta::WebGPU` - compute and draw on the graphics card
- `Peta::XS` - the interpreter's capability layer for pure Perl

Peta::Dict

Peta::Dict - a fast dictionary bound to a Perl hash with `tie`.

```
use Peta::Dict;

tie my %count, 'Peta::Dict';
$count{$_}++ for qw(a b a c a);
print join(", ", map { "$_=$count{$_}" } sort keys %count), "\n";
 ↪# a=3, b=1, c=1
delete $count{b};
print exists $count{b} ? "b still there\n" : "b gone\n";
 ↪# b gone
```

It behaves as a plain hash does - the same keys, values, `exists`, `delete`, `each` and `keys` - with its storage built for speed: under the JIT an element access compiles to a direct table probe. Iteration order is the table's own. Peta::Dict is pperl's own module, version 0.2.0.

%h itself stays a faithful perl5 hash; what `tie` binds it to is a separate stratum: an open-addressed table (node's `Map` class, ~one cache line per access) whose element accesses the JIT compiles to an in-register probe. That is the same "native store + JIT probe" shape that carries Memoize, applied to a general container - a dictionary is a language facility every pole ships, and this one is stratified beside %h rather than changing it.

The tie protocol is perl5's, untouched (`pp_sys.c pp_tie`, the `PERL_MAGIC_tied` node, the per-access `PERL_MAGIC_tiedelem` proxy, `mg.c Perl_magic_methcall`); the class merely implements the methods natively, so even the interpreted path never runs Perl code:

- `FETCH(obj, key)` returns a copy of the stored value, as the pure-Perl class does - so a direct `tied(%h)->FETCH($k)` cannot change the entry through its result. A missing key returns undef, unconditionally (perl calls FETCH for every rvalue read; there is no existence pre-check to skip).

- `STORE(obj, key, val)` copies `val` in (a `sv_setsv` without get-magic and without stealing: `val` is the throwaway proxy, or the already-copied RHS of `%h = (...)`, and `$x = $h{$k} = 5` reads the proxy again after the STORE). `$h{$k}++` is exactly one FETCH and one STORE on one proxy (`pp_preinc` -> `sv_inc` -> `SvSETMAGIC`).
- `EXISTS/DELETE/CLEAR/FIRSTKEY/NEXTKEY/SCALAR` are each exactly one synchronous call from the responsible C function (`magic_existspack`, `magic_clearpack`, `magic_wipepack`, `magic_nextpack`, `magic_scalarpack`). `delete` returns the value. Iteration order is whatever this class produces (perl imposes none on a tied hash); `keys %h` never FETCHes.

KEYS. A key is the integer or the string perl would stringify it to, so an integer key and its decimal string name the same entry, exactly as in a plain hash (`$h{5}` is `$h{"5"}`; `"05"` is another key). An IOK-only key is taken as its integer; anything else is stringified by perl's own `sv_2pv_flags` (which warns for undef as perl does) and an exact canonical decimal (no sign on zero, no leading zero, in IV range) is folded back to an integer. Integer keys live in the open-addressed table the JIT probes; string keys live in a side map that is correct and native but not JIT-probed. Iteration returns integer keys as IVs, whose string form is that same decimal. A string key is held as a hash holds it: a UTF-8 key that fits in Latin-1 is the same entry as its byte string, and one that does not keeps its UTF-8 flag through `keys`.

The pure-Perl `assets/modules/user/Peta/Dict.pm` is the fallback perl5 (the t/TEST oracle) loads: a plain hash behind the same protocol, identical output, merely not fast. It is not embedded: pperl's registry answers the `require` first, and a build without this class stops there with the capability hint.

The JIT seam: `[DictTable]` is `#[repr(C)]` with the probed fields first (`slots/state/mask/shift`), so compiled code reaches an entry with offset arithmetic from the table pointer. A HIT (an existing integer key) is served in-register; a MISS (an insert, which may grow the table) goes through the same Rust functions the methods use, after which the probe reloads the arrays from the struct.

Peta::FFI

Call C library functions from Perl with no build step and no binding crate.

`Peta::FFI` is the dynamic FFI layer pperl ships for reaching into shared libraries at runtime. You open a library with `dlopen`, call any of its exported functions through `call` by giving a type signature string, and release the handle with `dlclose`. `scan` finds the libraries in a list of directories, which is useful for finding the right file before you open it.

This is how Perl code here reaches a C library: pperl loads no compiled extension modules, so anything living in a `.so` is called either through this module or through one of the bindings built on it. pperl's *FFI::Platypus* is one of them: its calls, types, records and closures are this module's, so whatever FFI::Platypus can express, Peta::FFI can, in its own signature language.

Reach for it when:

- The function you need lives in a system library (`libm`, `libc`, `libuuid`, `libcrypto`).
- You already know the C prototype and want a one-liner, not a build step.

- You are calling into a third-party `.so` that no pre-baked wrapper covers.

C calling back into Perl is supported: `closure` wraps a coderef as a function pointer you can hand to any C API that wants one (see the Callbacks category). C structs are records: `record` declares one as a Perl class, and a signature passes and returns it by value or by pointer (see the Records category). A library's global variables are reached with `symbol`, `fetch`/`store` and `variable`.

What it does not do is **derive types from a header**: you read the C prototype and write the matching signature by hand, and for a struct its members in order, since a signature that disagrees with the real prototype is undefined behaviour rather than an error.

The two sibling modules `Peta::FFI::Libc` and `Peta::FFI::UUID` are pre-baked convenience wrappers built on top of this layer - they handle the `dlopen` / `call` / `dlclose` dance for you for a fixed set of functions. If your need is covered by one of them, prefer it over hand-rolled FFI.

A call that can block for good - `statfs` on a dead NFS mount - runs on a *Peta::FFI::Pool* thread instead, with a deadline, and reports how it went as an event. A library that calls back on threads of its own gets a queued closure, whose calls wait in a *Peta::FFI::Queue* for Perl's event loop.

Signature strings

Every `call`, `function` and `closure` takes a signature string of the form `(args)ret`, where each position is one type:

Type	C type	Perl side
v	void	return only; 1 is returned
i	int	integer
l	long	integer
L	unsigned long / size_t	unsigned integer
i8 i16 i32 i64	int8_t … int64_t	integer
u8 u16 u32 u64	uint8_t … uint64_t	unsigned integer
b	bool	perl's true or false
d	double	float
f	float	float
D	long double	float (rounded to a double)
cf cd	float / double complex	[re, im]; a number is [n, 0]
p	const char *	input string; a returned string is copied
s	char * the caller frees	a result, copied, then freed; *s an out argument
P	mutable buffer	output buffer; \$buf is written
o	opaque pointer	raw address as integer
<Class>	a C object's pointer	an object of a declared handle type
{Class}	struct by value	an object of a declared record class
*{Class}	struct *	the same object, by pointer
*T	T *	\$scalar in and out; undef is NULL
*<Class>	T ** C stores a handle in	\$scalar that gets the object
T[N] T[]	T * to an array	\@array in and out
T[0]	T *[] ended by NULL	\@array of o, p or handles
*T[r] *T[#k] *T[0]	T ** to an array C allocates	\$scalar or \@array it fills
*P[r] *P[#k]	char ** to bytes C allocates	\$scalar that gets the bytes
&T[r] &T[#k] &T[0]	const T ** to an array C keeps	the same, copied, never freed
&P[r] &P[#k]	const char ** to bytes C keeps	\$scalar that gets a copy
result &T[#k] &T[0]	const T * to an array C keeps	an array reference of copies
result &P[#k]	const char * to bytes C keeps	a copy of the bytes

A signature with no arguments is written ()ret, e.g. ()i for an int-returning no-arg function. Types may be separated by blanks: (u64 <My::Pool>)b. A variadic function lists its named parameters, a ;, and the types this call passes through its ...: (iL;o)i is ioctl(int, unsigned long, ...) passing one pointer.

An unsigned type reads its argument as an unsigned integer, so 2**64-1 passes as itself, not as -1, and returns above 2**63 come back unsigned. s is for C functions whose returned string the caller owns; function names the free function that releases it. <Class> is a typed handle declared with handle_type, {Class} a record declared with record. An argument is read once, as an XS function reads it, so a regex capture or a tied scalar passes its value.

`*T` and `T[N]` take the numbers, `o`, `p` (an array of C strings, `char **`) and `b`: `(d*i)d` is `frexp(double, int *)`, called with `\my $exp`, and `(i32[]LLo)v` is `qsort` sorting `\@numbers` in place. `T[N]` also takes handles and records, and C's changes to the records are written back: `({My::Point}[]LLo)v` sorts `\@points`. `T[0]` is an array of `o`, `p` or handles that C reads up to a NULL the call appends, as `execv`'s `argv` or a NULL-terminated list of handles.

Some C functions hand over what they allocate through a pointer the caller passes. `*<Class>` is a handle stored there, as `sqlite3_open` stores its `sqlite3 *`: called with `\my $db`, `$db` becomes the object, owned by the caller and destroyed with it; a handle already in `$db` goes in, and stays the same object if C leaves it. `*s` is a string stored there, copied into the scalar and freed.

`*T[r]`, `*T[#k]` and `*T[0]` are an array of `T` the callee allocates and stores there: as long as the function's integer result (`[r]`), as argument `k` (`[#k]`, an integer or an `*` integer, counted from 1), or up to a NULL element (`[0]`, for `o`, `p`, `s` and handles). `(<My::Conn>*<My::Item>[r]u32)i` lists a connection's items: called with `\my $items`, `$items` becomes an array reference of objects, each the caller's; a reference to an array is filled instead. Strings (`*s[...]`) are copied and each freed, numbers and `o` read, records copied. A negative length is the callee's error and leaves `undef`. The array itself is freed with `function`'s `free` - or, for records, with its `release`, `(array, count)`, the library's function that also frees what the records point to. Records keep the array until the last of them is gone, so a record's `p` member reads its string for as long as the record lives; a bare copy of its bytes does not keep it.

`*P[r]` and `*P[#k]` are bytes the callee allocates and stores there, with the length as for an array: one byte string, NULs included, where `*u8[#k]` would give an array of numbers. librados's `rados_mon_command` hands over its reply this way, as a `char **` and a `size_t *` freed with `rados_buffer_free`: `(op[]LPL*P[#7]*L*P[#9]*L)i` with `free => "rados_buffer_free"`, the input buffer passed as `\$inbuf` and the reply taken with `\my $out, \my $outlen, \my $status, \my $statuslen`. A NULL leaves `undef`, and so does a negative length; the buffer is freed with `function`'s `free`. A buffer ended by a NUL is an `*s`. Bytes the callee only lends are `&P[r]` and `&P[#k]`, and `&P[#k]` is also a result: copied, never freed, as the lent arrays below.

`&T[r]`, `&T[#k]` and `&T[0]` are the same for an array the callee only lends - a pointer into memory it keeps, as libxkbcommon's `xkb_keymap_key_get_syms_by_level` hands out a keymap's keysyms: `(<Keymap>u32u32u32&u32[r])i`. The elements are copied and the array is never freed. They are numbers, `o`, `p`, `b` or records; an `s`, which the caller frees, and a handle, whose object owns its pointer, cannot be lent.

`&T[#k]` and `&T[0]` are also a result: the array C returns, as libxkbcommon's `xkb_compose_table_entry_sequence` returns an entry's keysyms and stores their count through a `size_t *`: `(o*L)&u32[#2]`, called as `$f->($entry, \my $n)`. The call returns an array reference of the copies, or `undef` for a negative length or a NULL with elements. The length is read after the call, from an integer or an `*` integer argument; `[r]` is no form here, since the result is the array itself.

`P` passes the bytes of the scalar it references, which hold what the callee wrote afterwards: `my $buf = "\0" x 64; call(..., \$buf)`. Given a string or a number instead, it passes a zeroed buffer of that length or that many bytes (at least 16) that nobody reads back.

`o` is the handle type for C APIs built on opaque structs (`PGconn*`, `sqlite3*`, `MYSQL*`): the raw address travels as a Perl integer, is never dereferenced by the runtime, and `undef`/`0`

passes NULL. Never use p for such pointers - p returns copy-until-NUL string semantics, which is wrong and unsafe for non-string data.

Memory primitives

For C APIs that traffic in caller-owned memory (out-parameters, pointer arrays, length-counted buffers) the module provides the pointer toolbox: alloc/free, peek/peek_cstr, poke, and pack_ptr/unpack_ptr for bytes, fetch/store for a value of any type at an address, and sizeof/alignof. Addresses are ordinary Perl integers as with o. A pointer array (e.g. char *argv[]) is built portably as join "", map pack_ptr($_), @addrs poked into an alloced block, or passed as a p[] array.

Synopsis

```
use Peta::FFI;

my $libm = Peta::FFI::dlopen("libm.so.6");
my $root = Peta::FFI::call($libm, "sqrt", "(d)d", 2.0);
Peta::FFI::dlclose($libm);
print $root, "\n";       # 1.4142135623731

Peta::FFI::record("Div", quot => "i", rem => "i");      # div_t
my $d = Peta::FFI::call(undef, "div", "(ii){Div}", 17, 5);
print $d->quot, " r ", $d->rem, "\n";                   # 3 r 2
```

Modules

- Peta::FFI::EventSource - events from a C library's own threads, drained on Perl's
- Peta::FFI::LMDB - Lightning memory-mapped databases: transactional key-value stores in a file, over the system's liblmdb.
- Peta::FFI::Libc - Call common libc functions from Perl without writing a single line of FFI glue.
- Peta::FFI::PCRE2 - Compile and run PCRE2 regular expressions via the system's libpcre2-8 library.
- Peta::FFI::Pool - Run C functions on worker threads, so that a call which hangs cannot hang the program.
- Peta::FFI::Queue - Callbacks a C library makes from threads of its own, queued for Perl's event loop.
- Peta::FFI::UUID - Generate and parse RFC 4122 UUIDs via the system's libuuid library.

Functions

Library loading

dlopen

Open a shared library and return a handle suitable for passing to `call`.

dlclose

Release a library handle previously returned by `dlopen`.

Function calls

call

Invoke a C function from an opened library and return its result.

function

Prepare a C function once and return it as a code reference.

attach

Prepare a C function once and install it as a named Perl sub.

errno

errno as the last C call through Peta::FFI left it.

Symbol introspection

symbol

The address of a symbol - a function or a variable - in a library.

variable

A library's global variable as a Perl scalar.

scan

Find the shared libraries in a list of directories.

Handles

handle_type

Declare a Perl class for a C object type, with its destructor.

`release`

Destroy a handle's C object now, ahead of its last reference.

Records

`record`

Declare a record class: a C struct, passed by value or by pointer.

`union`

Declare a union class: a C union, reached by pointer and in memory.

`sizeof`

The size of a type in bytes, as C's `sizeof`.

`alignof`

The alignment of a type in bytes, as C's `_Alignof`.

Callbacks

`closure`

Wrap a Perl coderef as a C-callable function pointer.

`closure_free`

Release a closure created by `closure`.

Memory

`fetch`

Read the value of a type at an address.

`store`

Write a value of a type at an address.

`peek`

Read `$len` raw bytes at address `$ptr` into a Perl byte string.

peek_cstr

Read the NUL-terminated C string at address `$ptr`.

poke

Copy a Perl byte string into memory at address `$ptr`.

alloc

Allocate `$len` zeroed bytes of C memory; returns the address.

free

Release memory obtained from `alloc`.

pack_ptr

Pack an address into a native pointer-sized byte string.

unpack_ptr

Unpack a native pointer-sized byte string into an address.

Peta::FFI::EventSource

events from a C library's own threads, drained on Perl's

SYNOPSIS

```
use Peta::FFI;
use Peta::FFI::EventSource;

my $lib = Peta::FFI::dlopen('/opt/example/lib/libexample.so');
Peta::FFI::handle_type('Example::Monitor', $lib, 'example_monitor_
↪free');
my $open = Peta::FFI::function($lib, 'example_monitor_open', '(p)
↪<Example::Monitor>');
my $fd   = Peta::FFI::function($lib, 'example_monitor_fd',   '(
↪<Example::Monitor>)i32');
my $next = Peta::FFI::function($lib, 'example_monitor_next', '(
↪<Example::Monitor>)s',
                               free => 'example_string_free');

my $source = Peta::FFI::EventSource->new($open->('block'), fd =>
↪$fd, next => $next);

# With AnyEvent (and so with EV underneath):
my $guard = $source->watch(sub { handle_event($_) for @_ });
```

(continues on next page)

(continued from previous page)

```
AnyEvent->condvar->recv;

# With Mojo::IOLoop, which takes filehandles:
Mojo::IOLoop->singleton->reactor->io($source->fh, sub { handle_
↪event($_) for $source->drain })
    ->watch($source->fh, 1, 0);

# With a loop of your own:
vec(my $rin = '', $source->fd, 1) = 1;
while (select(my $rout = $rin, undef, undef, undef)) {
    handle_event($_) for $source->drain;
}
```

DESCRIPTION

A library that produces events on threads of its own - a device monitor, a netlink listener - must not call into Perl from those threads: Perl runs on the interpreter's thread only, and a *Peta::FFI* closure called from anywhere else stops the program. The library queues the events instead, and signals that there are some through a file descriptor. This module watches the descriptor and, when it reads ready, takes the events one by one until the queue is empty.

One source is one handle of the library's: a library with several independent event streams opens a handle for each, and each becomes a source of its own.

A *Peta::FFI::Pool*, which runs C functions on threads of its own, is such a source too, with its own `fd` and `next`, and so is a *Peta::FFI::Queue*, where the calls of queued closures wait; their events are hash references:

```
my $source = Peta::FFI::EventSource->new($pool,
    fd => \&Peta::FFI::Pool::fd, next => \&Peta::FFI::Pool::next);
```

The library's half

The library provides, for its source handle type:

- a file descriptor function, `int fd(source *)`

  returning a descriptor that reads ready while events are queued - an eventfd or the read end of a pipe. Peta::FFI calls it once, in `new`.

- a next-event function, `char *next(source *)`

  returning the oldest queued event as a string the caller frees, or NULL when the queue is empty. Declared with the `s` result type and the library's `free` function, so Perl receives a copy and the library's string is released at once.

- a destructor

  declared with `handle_type`, which stops the library's thread and closes the descriptor. It runs once, when the last reference to the handle goes; the source holds one.

Clearing the descriptor is the library's job: when `next` returns NULL, the library resets its eventfd or empties its pipe. Peta::FFI only waits for readability, level-triggered, and drains until NULL, so it never needs to know which kind of descriptor it is. A full queue is the library's to report, typically as an event of its own saying how many were lost; such an event passes through like any other.

METHODS

- new($handle, fd => $fd_function, next => $next_function)

  The source for `$handle`. Both functions are code references from `Peta::FFI::function` taking the handle; `$fd_function` is called once, here. Croaks when the library returns no descriptor.

- fd

  The library's descriptor, for an event loop that takes descriptor numbers (*EV*, `select`). Never close it: it is the library's.

- fh

  A read filehandle on a duplicate of the descriptor, for an event loop that takes filehandles (Mojo::IOLoop, *IO::Select*). It becomes readable exactly when the descriptor does, and closing it - or letting it go - closes only the duplicate, so it may outlive the source. The same filehandle is returned on every call.

- drain

  The queued events, oldest first, each as `next` returned it - the string a library returned, a pool's hash reference; the empty list when there are none.

- watch($callback)

  Watches the descriptor with *AnyEvent* (loaded on first use) and calls `$callback` with the drained events whenever there are some. Returns the watcher: the watch lasts as long as it is referenced.

- handle

  The library handle the source was made for.

SEE ALSO

Peta::FFI - `function`, `handle_type` and the `s` result type.

Peta::FFI::Pool - C functions run on worker threads, their calls reported as events.

Peta::FFI::Queue - closures a library calls on its own threads, their calls queued for the event loop.

Peta::FFI::LMDB

Lightning memory-mapped databases: transactional key-value stores in a file, over the system's liblmdb.

An environment (`Peta::FFI::LMDB::Env`) is a file, or a directory, holding one unnamed database and, with `maxdbs`, named ones. Everything happens in a transaction (`Peta::FFI::LMDB::Txn`): read transactions see a snapshot and never block; one write

transaction at a time commits atomically, and nothing it wrote survives an abort or a crash before the commit. A database handle (`Peta::FFI::LMDB::DB`) names one database; a cursor (`Peta::FFI::LMDB::Cursor`) walks one in key order.

liblmdb is the system's library, loaded when first needed; a call without it croaks naming the library it looked for.

Synopsis

```
use Peta::FFI::LMDB qw(:flags);
my $env = Peta::FFI::LMDB::Env->new('data.lmdb', flags => MDB_
 ↪NOSUBDIR,
                                    mapsize => 1 << 30, maxdbs => 4);
my $txn = $env->begin;
my $db  = $txn->open_db('users', MDB_CREATE);
$txn->put($db, alice => 'admin');
$txn->commit;

my $r = $env->begin(MDB_RDONLY);
print $r->get($db, 'alice'), "\n";
my $c = $r->cursor($db);
while (my ($k, $v) = $c->next) { print "$k=$v\n" }
```

Keys and values

Keys and values are byte strings; a string with characters over 255 croaks `Wide character`. In an `MDB_INTEGERKEY` database the keys are unsigned integers, and in an `MDB_DUPSORT|MDB_INTEGERDUP` one the values. A key may be at most `$env->maxkeysize` bytes, 511 with the usual liblmdb. A missing key is undef, not an error. `$txn->get_view($db, $key, $var)` makes `$var` the value without copying it: a read-only scalar that becomes undef when its transaction ends or, in a write transaction, at the next write.

Errors

LMDB's errors croak with a `Peta::FFI::LMDB::Error` object: `$@->code` is LMDB's code, to compare with the `MDB_*` constants of the `:errors` tag, `$@->message` its text, which the object also stringifies to.

```
eval { $txn->put($db, $key, $value) };
if (ref $@ && $@->code == MDB_MAP_FULL) { ... }
```

Misuse croaks with a plain message naming the problem: using a transaction that has ended, a closed environment or cursor, a database of another environment, an option `new` does not know.

Ordering

A database keeps its keys in byte order (integer order with `MDB_INTEGERKEY`) unless `$db->set_compare` gives it a sort sub; an `MDB_DUPSORT` database orders a key's values the same way, or by `$db->set_dupsort`'s sub. The sub is called as a `sort` block is, with the two items in `$a` and `$b` of its package. The file does not remember it: set it every time the environment is opened.

Constants

`use Peta::FFI::LMDB qw(:flags)` imports the flags of `new`, `begin`, `open_db`, `put`, `del` and `copy` (`MDB_NOSUBDIR`, `MDB_RDONLY`, `MDB_CREATE`, `MDB_DUPSORT`, `MDB_NOOVERWRITE`, `MDB_CP_COMPACT` and the rest); `:errors` the error codes (`MDB_NOTFOUND`, `MDB_MAP_FULL` ...); `:cursor_ops` the operations of `$cursor->get` (`MDB_FIRST`, `MDB_SET_RANGE` ...); `:all` all of them. They are also callable as `Peta::FFI::LMDB::MDB_CREATE` without importing.

The DBM modules' write policy

NDBM_File, ODBM_File, GDBM_File and DB_File (its Hash and Btree databases) keep their files with LMDB too, without transactions of their own. `$Peta::FFI::LMDB::DBM_WRITE`, read when a file is tied, says when their writes reach the file:

- `'back'` (the default): the writes of a tie collect in one LMDB transaction, committed at `sync` and when the hash is untied. Fast, as gdbm is; other processes see the writes after the commit, and a crash loses what was not committed. Like a gdbm writer, the tie holds the file's one writer slot until then: another process that writes the file waits. Readers never wait.
- `'through'`: every write is committed and synced at once - visible to other processes immediately, and a disk sync per write. Use it for files that several processes write.

GDBM_File's own sync request, `GDBM_SYNC` or `sync_mode(1)`, makes its tie write-through.

The DBM modules' files

A DBM module's file is an LMDB environment of its own (one file, `MDB_NOSUBDIR`, with LMDB's lock file `NAME-lock` beside it), which Peta::FFI::LMDB can open, with `maxdbs => 1` or more, when the module has it closed. Its main database holds two records: the named database `data`, with the keys and values, and `pperl-store`, whose value says what they are - `version=1 dups=0`, or `dups=1` when a key may hold several values (DB_File's `R_DUP`). Such a key is stored as the key followed by an 8-byte big-endian sequence number, the order its values were stored in. A key that is empty or starts with a NUL byte is stored behind one more NUL byte. The modules refuse an LMDB file without the `pperl-store` record.

Modules

- `Peta::FFI::LMDB::Cursor` - A cursor: a position in a database that walks its keys in order, finds the first key at or after a given one, and writes or deletes where it stands.
- `Peta::FFI::LMDB::DB` - A database handle: one database of an LMDB environment, as `$txn->open_db` returns it, to pass to the transaction's and cursor's methods.
- `Peta::FFI::LMDB::Env` - An LMDB environment: the file (or directory) a set of databases lives in, and what every transaction starts from.
- `Peta::FFI::LMDB::Txn` - A transaction: everything read from or written to an LMDB environment happens in one.

Functions

Errors

`strerror`

LMDB's message for an error code: `Peta::FFI::LMDB::strerror(MDB_MAP_FULL)` is `MDB_MAP_FULL: Environment mapsize limit reached`. A positive code is a system error, and gives the system's message.

Library

`version`

The version of the system's liblmdb: `Peta::FFI::LMDB::version()` gives its version string, such as `LMDB 0.9.33: (May 21, 2024)`; in list context the string and its major, minor and patch numbers.

Peta::FFI::LMDB::Cursor

A cursor: a position in a database that walks its keys in order, finds the first key at or after a given one, and writes or deletes where it stands.

`$txn->cursor($db)` opens one. Every move returns the key and value it lands on as a two-element list, or an empty list when there is nothing there, so a loop reads naturally:

```
my $c = $txn->cursor($db);
while (my ($k, $v) = $c->next) { print "$k=$v\n" }
```

A new cursor stands nowhere: `next` starts at the first key, `prev` at the last. In an `MDB_DUPSORT` database the cursor visits every value of a key; the _dup moves stay within the current key and the _nodup moves skip to the next key.

A cursor ends with its transaction. A read transaction's cursor can move on to another read transaction with `renew`.

Functions

Moving

get

Move the cursor by any of LMDB's cursor operations: `$c->get($op)`, `$c->get($op, $key)`, `$c->get($op, $key, $value)`, with `$op` an `MDB_*` constant of the `:cursor_ops` tag (`MDB_SET_RANGE` and the like). The key and value it lands on, or an empty list. The named moves below do the same more readably.

first

To the first key (and its first value): `($k, $v) = $c->first`.

last

To the last key (and its last value).

next

To the next item - the next value of the key in an `MDB_DUPSORT` database, else the next key; from a new cursor, the first. An empty list after the last.

prev

To the previous item; from a new cursor, the last. An empty list before the first.

first_dup

To the current key's first value (`MDB_DUPSORT` databases).

last_dup

To the current key's last value (`MDB_DUPSORT` databases).

next_dup

To the current key's next value; an empty list after its last (`MDB_DUPSORT` databases).

prev_dup

To the current key's previous value; an empty list before its first (`MDB_DUPSORT` databases).

next_nodup

To the first value of the next key, skipping the current key's other values.

prev_nodup

To the last value of the previous key.

current

The key and value the cursor stands on, without moving.

set

To the key itself: `($k, $v) = $c->set($key)`. An empty list when the key is not stored.

set_range

To the first key at or after the one given, in the database's order: `($k, $v) = $c->set_range('ab')` finds ab, or the first key after it - the start of a range or prefix walk. An empty list when no key follows.

get_both

To a key holding a given value: `$c->get_both($key, $value)` (MDB_DUPSORT databases). An empty list when the key does not hold it.

get_both_range

To a key's first value at or after the one given: `$c->get_both_range($key, $value)` (MDB_DUPSORT databases).

count

How many values the current key holds (MDB_DUPSORT databases).

Writing

put

Store a key and value and leave the cursor on them: `$c->put($key, $value)`. `$c->put($key, $value, MDB_CURRENT)` replaces the value the cursor stands on (the key must be the cursor's). The flags MDB_NOOVERWRITE, MDB_NODUPDATA, MDB_APPEND and MDB_APPENDDUP work as for `$txn->put`. True when stored, false when MDB_NOOVERWRITE or MDB_NODUPDATA refused. MDB_RESERVE and MDB_MULTIPLE croak.

del

Delete the item the cursor stands on: `$c->del`. In an MDB_DUPSORT database, `$c->del(MDB_NODUPDATA)` deletes all the key's values. The next `next` returns the item after the deleted one.

Ending

renew

Move a read transaction's cursor to another read transaction: `$c->renew($txn)`, cheaper than opening a new cursor. It then stands nowhere. A write transaction croaks `renew needs a read transaction`.

close

Close the cursor now, rather than when it goes out of scope or its transaction ends. Using it afterwards croaks `the cursor is closed or its transaction has ended`.

Peta::FFI::LMDB::DB

A database handle: one database of an LMDB environment, as `$txn->open_db` returns it, to pass to the transaction's and cursor's methods.

A handle belongs to its environment, not to the transaction that opened it: once that transaction has committed, every later transaction of the environment can use it. Handles of one environment cannot be used with another's transactions: that croaks `Peta::FFI::LMDB: the database belongs to another environment`.

A database keeps its keys in LMDB's byte order (or integer order, with `MDB_INTEGERKEY`) unless given a sort sub with `set_compare`; an `MDB_DUPSORT` database orders a key's values the same way, or by `set_dupsort`'s sub.

Functions

Ordering

set_compare

Order the database's keys by a sort sub of your own.

set_dupsort

Order the values of each key of an `MDB_DUPSORT` database by a sort sub, called as `set_compare`'s is: `$db->set_dupsort(sub { $a <=> $b })`. Set it every time the environment is opened, before the first read or write.

Information

dbi

LMDB's number for the database within its environment.

set_compare

Order the database's keys by a sort sub of your own.

Synopsis

```
$db->set_compare(sub { lc $a cmp lc $b });
```

What you get back

Nothing. From then on every transaction uses the sub to place and find keys in this database.

Edge cases

- The sub sees the two keys in `$a` and `$b` of its own package, as a `sort` block does, and returns less than, equal to or greater than zero.
- The database does not remember the sub: set the same one every time the environment is opened, before the first read or write. Data stored under one order and read under another is not found.
- A sub that dies makes the operation that called it croak with the sub's error, after LMDB has finished it.
- A plain value instead of a code reference croaks `Peta::FFI::LMDB: set_compare needs a code reference`.

See also

- `set_dupsort` - the order of a key's values
- `cmp` (a `Peta::FFI::LMDB::Txn` method) - compares two keys this way

Peta::FFI::LMDB::Env

An LMDB environment: the file (or directory) a set of databases lives in, and what every transaction starts from.

`Peta::FFI::LMDB::Env->new` opens one; `begin` starts a transaction in it. An environment holds an unnamed database and, when opened with `maxdbs`, that many named ones. Its memory map has a fixed size, `mapsize`: a write that would grow the data past it fails with `MDB_MAP_FULL`, so open with room to spare - the size reserves address space, not disk or memory.

A file can be open in one environment per process: a second `Peta::FFI::LMDB::Env->new` of the same file croaks while the first is open. The environment closes when its last reference goes, or at `close`.

Synopsis

```
use Peta::FFI::LMDB qw(:flags);

my $env = Peta::FFI::LMDB::Env->new('data.lmdb', flags => MDB_
↪NOSUBDIR,
                                    mapsize => 1 << 30, maxdbs => 4);
```

(continues on next page)

(continued from previous page)

```
my $txn = $env->begin;
# ...
$txn->commit;
print $env->stat->{entries}, "\n";
```

Functions

Environments

new

Open an environment: the file or directory `$path`, with options.

begin

Start a transaction: `$env->begin` for a write transaction, `$env->begin(MDB_RDONLY)` for a read one.

stat

The unnamed database's statistics, as a hash reference: `entries` (the number of keys), `depth` of its tree, `branch_pages`, `leaf_pages`, `overflow_pages`, and the page size `psize`. A named database's are `$txn->stat($db)`.

info

The environment's state, as a hash reference: `mapsize`, the map's address `mapaddr`, the last page used `last_pgno`, the last committed transaction's ID `last_txnid`, and the readers allowed and live, `maxreaders` and `numreaders`.

sync

Flush committed data to disk: needed only when the environment was opened with `MDB_NOSYNC` or `MDB_MAPASYNC`, as commits otherwise sync themselves. `$env->sync(1)` forces the flush even then.

copy

Copy the environment to `$path` while it is in use - a consistent backup of the last commit. `$path` is a directory, or with `MDB_NOSUBDIR` the copy's file name, which must not exist. `$env->copy($path, MDB_CP_COMPACT)` leaves the free pages out, making a smaller copy.

path

The path the environment was opened with.

flags

The environment's MDB_* flags, as a number to test with &.

set_flags

$env->set_flags($flags, $on): switch flags on (true $on) or off while open. Only MDB_NOSYNC, MDB_NOMETASYNC, MDB_MAPASYNC and MDB_NOMEMINIT can change; others croak Invalid argument.

maxkeysize

The longest key the environment takes, in bytes (511 with the usual liblmdb), which is also the longest value of an MDB_DUPSORT database. A longer one croaks MDB_BAD_VALSIZE.

reader_check

Free the reader slots of processes that died with a read transaction open; returns how many it freed. Such slots keep pages from being reused until cleared.

close

Close the environment now, rather than when its last reference goes. No transaction may be live: that croaks Peta::FFI::LMDB: the environment has live transactions. Using the environment afterwards croaks the environment is closed.

new

Open an environment: the file or directory $path, with options.

Synopsis

```
my $env = Peta::FFI::LMDB::Env->new($path, %options);
```

Options

- flags - MDB_* environment flags OR'd together, chiefly MDB_NOSUBDIR ($path is the data file itself, with the lock file $path-lock beside it; without it $path is a directory holding data.mdb and lock.mdb), MDB_RDONLY (read only), MDB_NOSYNC, MDB_NOMETASYNC, MDB_MAPASYNC (sync less, risking the last commits on a crash), MDB_NOTLS, MDB_NORDAHEAD, MDB_NOMEMINIT, MDB_WRITEMAP.
- mode - the permission of files it creates (default 0644).
- mapsize - the memory map's size in bytes, the most data the environment can hold (LMDB's default is 1 MiB).
- maxdbs - how many named databases open_db may open.
- maxreaders - how many read transactions may be live at once (default 126).

What you get back

A `Peta::FFI::LMDB::Env` object. Failures croak: an LMDB error with a `Peta::FFI::LMDB::Error`, an unknown option with `Peta::FFI::LMDB::Env->new: unknown option 'NAME'`, a file already open in the process with `... is already open in this process`.

Examples

```
my $env = Peta::FFI::LMDB::Env->new('cache.lmdb', flags => MDB_
↪NOSUBDIR);
```

A directory environment with named databases:

```
mkdir 'app.db';
my $env = Peta::FFI::LMDB::Env->new('app.db', maxdbs => 8, mapsize␣
↪=> 1 << 32);
```

Edge cases

- The directory form needs the directory to exist; a file it creates.
- A second environment on a file already open in the process croaks rather than open it: LMDB's locks would break under two.

See also

- `begin` - starts a transaction
- `close` - closes the environment now

begin

Start a transaction: `$env->begin` for a write transaction, `$env->begin(MDB_RDONLY)` for a read one.

Synopsis

```
my $txn = $env->begin;
my $r   = $env->begin(MDB_RDONLY);
```

What you get back

A `Peta::FFI::LMDB::Txn`. A read transaction sees the data as it was when it began, however long it lives, and never waits. A write transaction waits while another one - in this process or another - is live, as there is one writer at a time.

Edge cases

- A transaction that goes out of scope without `commit` is aborted: its writes are discarded.
- A long-lived read transaction keeps the pages it sees from being reused, so the file grows while writers work; end read transactions (or `reset` them) when done.

See also

- `Peta::FFI::LMDB::Txn` - what a transaction does
- `stat` - the size of the unnamed database

Peta::FFI::LMDB::Txn

A transaction: everything read from or written to an LMDB environment happens in one.

`$env->begin` starts a write transaction, `$env->begin(MDB_RDONLY)` a read one. A read transaction sees the data as it was when it began, never waits and never blocks a writer. A write transaction sees its own writes; nothing it writes is visible elsewhere until `commit` makes all of it durable at once, and `abort` - or the transaction going out of scope - discards all of it. There is one write transaction at a time across all processes using the file.

Keys and values are byte strings: a string with characters over 255 croaks `Wide character`. In a database opened with `MDB_INTEGERKEY` the keys are unsigned integers, and with `MDB_DUPSORT|MDB_INTEGERDUP` the values.

Synopsis

```
my $txn = $env->begin;
my $db  = $txn->open_db('users', MDB_CREATE);
$txn->put($db, alice => 'admin');
print $txn->get($db, 'alice'), "\n";    # admin
$txn->commit;
```

Edge cases

- After `commit` or `abort` the transaction is over: using it croaks `Peta::FFI::LMDB: the transaction has ended`.
- While a child transaction (`begin` on a transaction) is live, using its parent croaks `the transaction has a live child transaction`.

Functions

Reading and writing

get

Look up a key; the value, or undef when the key is not stored.

get_view

Look up a key and make a variable the value itself, without copying it: for large values read once.

put

Store a value under a key.

del

Delete a key - in an `MDB_DUPSORT` database all its values - or, with a value, just that one value: `$txn->del($db, $key)`, `$txn->del($db, $key, $value)`. True when something was deleted, false when there was nothing to delete.

cursor

Open a cursor on a database: `$txn->cursor($db)`, a `Peta::FFI::LMDB::Cursor` that walks the keys in order. It ends with the transaction; a read transaction's cursor can be moved to another read transaction with `renew`.

Databases

open_db

Open a database of the environment, for this and later transactions.

stat

A database's statistics, as a hash reference: `entries` (the number of items), `depth`, `branch_pages`, `leaf_pages`, `overflow_pages`, `psize`.

drop

Empty a database: `$txn->drop($db)`. `$txn->drop($db, 1)` deletes the database itself, after which its handle is no longer valid.

db_flags

The `MDB_*` flags a database was created with, as a number to test with &.

cmp

Compare two keys as the database orders them - by its `set_compare` sub, or LMDB's order for its flags: `-1`, `0` or `1`, as `cmp` returns.

dcmp

Compare two values as an `MDB_DUPSORT` database orders a key's values - by its `set_dupsort` sub, or LMDB's order: `-1`, `0` or `1`.

Ending a transaction

commit

Make the transaction's writes durable and visible, and end it. A failure croaks; the transaction is over either way. Committing a read transaction ends it, keeping the databases it opened.

abort

Discard the transaction's writes and end it. Ending a transaction that has already ended does nothing. A transaction that goes out of scope is aborted.

reset

Let a read transaction's snapshot go, keeping the transaction for `renew`: cheaper than ending and beginning, for a reader that pauses. A write transaction croaks `reset needs a read transaction`.

renew

Give a `reset` read transaction a fresh snapshot of the data as it is now.

Nesting

begin

Start a child transaction inside a write transaction: `$txn->begin`. Its writes join the parent's at its `commit`, or vanish at its `abort`, while the parent's other writes stand. The parent cannot be used until the child ends. A read transaction cannot have children.

Information

id

The transaction's ID: for a write transaction its own, for a read one the ID of the commit its snapshot shows.

env

The transaction's environment, a `Peta::FFI::LMDB::Env`.

get

Look up a key; the value, or undef when the key is not stored.

Synopsis

```
my $value = $txn->get($db, $key);
```

What you get back

A copy of the value as a string (an integer in an `MDB_INTEGERDUP` database), or undef. In an `MDB_DUPSORT` database, the key's first value.

Edge cases

- A key too long for the environment croaks with `MDB_BAD_VALSIZE`; `maxkeysize` gives the limit.
- In a write transaction, it sees the transaction's own writes.

See also

- `get_view` - the same without copying the value
- `Peta::FFI::LMDB::Cursor` - walk keys in order, find a range

get_view

Look up a key and make a variable the value itself, without copying it: for large values read once.

Synopsis

```
my $view;
if ($txn->get_view($db, $key, $view)) { print length $view }
```

What you get back

True when the key is stored, with `$view` a read-only string over the value in the database's memory map; false when not, with `$view` undef.

Edge cases

- `$view` becomes undef when the transaction ends, and in a write transaction at its next write: copy what must outlive that (`my $copy = $view`).
- Assigning to `$view` croaks `Modification of a read-only value attempted`; a later `get_view` into the same variable replaces it.
- The view must be a plain scalar variable, given as the argument: a returned value would be copied by the assignment.

See also

- `get` - a copy of the value

put

Store a value under a key.

Synopsis

```
$txn->put($db, $key, $value);
my $added = $txn->put($db, $key, $value, MDB_NOOVERWRITE);
```

Flags

- none - store, replacing the key's value; in an `MDB_DUPSORT` database, add the value to the key's values.
- `MDB_NOOVERWRITE` - store only a key not yet there.
- `MDB_NODUPDATA` - in an `MDB_DUPSORT` database, store only a value the key does not hold yet.
- `MDB_APPEND`, `MDB_APPENDDUP` - the key (value) goes after all the others: a fast load of sorted data; out of order it croaks `MDB_KEYEXIST`.

What you get back

True when stored, false when `MDB_NOOVERWRITE` or `MDB_NODUPDATA` found the key (value) already there.

Edge cases

- In a read transaction it croaks with LMDB's `Permission denied`.
- `MDB_RESERVE` is refused with a croak.
- When the data outgrows the environment's `mapsize` it croaks `MDB_MAP_FULL`, and the transaction can only be aborted.

See also

- `del` - removes a key or one value
- `Peta::FFI::LMDB::Cursor` - `put` at a cursor

open_db

Open a database of the environment, for this and later transactions.

Synopsis

```
my $main  = $txn->open_db;                          # the unnamed␣
↪database
my $users = $txn->open_db('users', MDB_CREATE);
```

What you get back

A `Peta::FFI::LMDB::DB` handle. It belongs to the environment: once the transaction that opened it commits (a read one, too), every later transaction can use it.

Flags

- `MDB_CREATE` - create a named database that does not exist (in a write transaction).
- `MDB_DUPSORT` - keys may hold several values, kept sorted; `MDB_DUPFIXED`, `MDB_INTEGERDUP`, `MDB_REVERSEDUP` refine it.
- `MDB_INTEGERKEY` - keys are unsigned integers; `MDB_REVERSEKEY` - keys compare from their last byte.

Edge cases

- A named database needs the environment opened with `maxdbs`; one more than that croaks `MDB_DBS_FULL`.
- A name that does not exist, without `MDB_CREATE`, croaks `MDB_NOTFOUND`.
- A handle opened in a transaction that aborts is gone with it.

See also

- `drop` - empties or deletes a database
- `Peta::FFI::LMDB::DB` - a database's own order

Peta::FFI::Libc

Call common libc functions from Perl without writing a single line of FFI glue.

Pre-baked libc bindings: each function here calls the C library directly, so there is no handle to open and no signature string to write. Reach for `Peta::FFI` when you need a function this module does not cover.

```
use Peta::FFI::Libc qw(getpid strlen strerror gethostname uname␣
↪access R_OK);

print getpid() == $$ ? "getpid is \$\$\n" : "?\n";
print strlen("hello"), " ", strerror(2), "\n";   # 5 No such file or␣
↪directory
my ($sys, $node) = uname();
print "$sys on $node\n" if gethostname() eq $node;
```

(continues on next page)

(continued from previous page)

```
print access("/etc/passwd", R_OK) == 0 ? "readable\n" : "not␣
↪readable: $!\n";
```

The functions keep C's conventions: the ones that report success return 0 for it and -1 for failure, with `$!` set to the C `errno`. Nothing is exported by default; name what you want.

Function	Returns
`getpid`, `getppid`	the process id, the parent's
`getuid`, `getgid`	the real user and group id
`geteuid`, `getegid`	the effective user and group id
`strlen($s)`	the length of `$s`; a NUL byte in it dies
`strerror($errno)`	the system's message for an error number
`getenv($name)`	the value, or undef when unset
`setenv($name, $value, $over)`	0 / -1; `$over` false keeps an existing value
`unsetenv($name)`	0 / -1
`abs($n)`, `labs($n)`	the absolute value, as an int / a long
`sleep($s)`, `usleep($us)`	seconds left unslept / 0 or -1
`gethostname()`	the host name
`uname()`	(sysname, nodename, release, version, machine)
`access($path, $mode)`	0 / -1, `$mode` an OR of `F_OK`, `R_OK`, `W_OK`, `X_OK`
`unlink`, `rmdir($path)`	0 / -1
`mkdir($path, $mode)`	0 / -1
`chmod($path, $mode)`	0 / -1
`chown($path, $uid, $gid)`	0 / -1
`time()`	seconds since the epoch
`clock()`	processor time used, in `CLOCKS_PER_SEC` units

Peta::FFI::Libc is pperl's own module, version 0.2.0.

Peta::FFI::PCRE2

Compile and run PCRE2 regular expressions via the system's libpcre2-8 library.

The library is located at run time, so pperl needs no build-time dependency on it. If `libpcre2-8.so` is not installed, `use Peta::FFI::PCRE2` still succeeds, the `PCRE2_*` constants still work, and every other call croaks with an instruction telling you which package to install.

`Peta::FFI::PCRE2` is an explicit binding, not a regex-engine plugin: a compiled pattern is a blessed object and matching is a method call. Perl's own `m//` is untouched. Reach for it when you want PCRE2's JIT specifically - each method is one native call, and the scan loop behind `count` runs entirely on the C side.

A pattern whose scalar carries the UTF8 flag is compiled with `PCRE2_UTF`. A subject is matched in the pattern's encoding: against a UTF pattern a byte string is matched as its UTF-8 upgrade, and against a byte pattern a UTF-8 string that fits in Latin-1 is matched as its bytes - so `"\xe9"` matches `"\xe9"` whichever way either is stored. Every

reported offset is a byte offset into the subject in that encoding. A start offset inside a UTF-8 character croaks. A subject with characters above 0xFF against a byte pattern is matched as its UTF-8 bytes. `substitute`'s result is UTF-8 exactly when the subject it matched was.

Synopsis

```
use Peta::FFI::PCRE2 qw(PCRE2_CASELESS);

my $re = Peta::FFI::PCRE2->new("ab+c", PCRE2_CASELESS);
my ($start, $end) = $re->match("xxABBBCxx");   # byte offsets, () on
↪no match
my $n   = $re->count($genome);                  # non-overlapping
↪match count
my $out = $re->substitute($seq, "<2>");         # global replace,
↪new string
```

Compile constants

Pass these to `new`, OR'd together. All are exportable on request; none are exported by default.

- `PCRE2_CASELESS` - case-insensitive matching, like `/i`.
- `PCRE2_MULTILINE` - `^` and `$` match at internal newlines, like `/m`.
- `PCRE2_DOTALL` - `.` matches newline too, like `/s`.
- `PCRE2_EXTENDED` - ignore unescaped whitespace and `#` comments in the pattern, like `/x`. `PCRE2_EXTENDED_MORE` additionally ignores spaces and tabs inside character classes, like `/xx`.
- `PCRE2_UNGREEDY` - invert greediness: quantifiers are lazy unless followed by `?`.
- `PCRE2_LITERAL` - treat the whole pattern as a literal string, with no metacharacters at all.
- `PCRE2_DUPNAMES` - permit several capture groups to share one name.
- `PCRE2_ANCHORED` - match only at the start offset.
- `PCRE2_UTF` - interpret pattern and subject as UTF-8. Set automatically when the pattern scalar is UTF8-flagged, so you rarely need it by hand.

Substitution constants

Pass these to `substitute`, whose default is `PCRE2_SUBSTITUTE_GLOBAL` alone.

- `PCRE2_SUBSTITUTE_GLOBAL` - replace every match, not just the first.
- `PCRE2_SUBSTITUTE_EXTENDED` - enable backslash escapes and conditional forms such as `${1:-default}` in the replacement.

Differences from upstream

There is no CPAN module this mirrors. The closest, `re::engine::PCRE2`, is a different design: it replaces perl's regex engine inside a lexical scope so that ordinary `m//` runs under PCRE2, and it offers no substitution entry point. This module never touches perl's engine and exposes PCRE2 directly instead, which is why matching is a method call rather than an operator.

Functions

Construction

new

Compile a pattern into a `Peta::FFI::PCRE2` object.

Matching

match

Match the pattern against a subject string once.

count

Count non-overlapping matches of the pattern in a subject string.

Substitution

substitute

Replace matches of the pattern in a subject, returning a new string.

Introspection

capture_count

Number of capturing groups in the compiled pattern.

Synopsis

```
Peta::FFI::PCRE2->new("(a)(b(c))?")->capture_count;   # 3
```

jit

Whether this pattern was successfully JIT-compiled.

jit_available

Whether the loaded libpcre2-8 was built with JIT support.

version

Version string of the loaded libpcre2-8 (e.g. `"10.47 2025-06-30"`).

new

Compile a pattern into a `Peta::FFI::PCRE2` object.

`$options` is a bitwise OR of the `PCRE2_*` compile constants (default 0). A UTF8-flagged pattern automatically adds `PCRE2_UTF|PCRE2_NO_UTF_CHECK`. The pattern is JIT-compiled when the library supports it; `jit` reports whether that succeeded. Croaks with PCRE2's message and the byte offset on a bad pattern.

Synopsis

```
my $re = Peta::FFI::PCRE2->new("a[act]ggtaaa|tttacc[agt]t");
my $ci = Peta::FFI::PCRE2->new("hello", PCRE2_CASELESS);
```

match

Match the pattern against a subject string once.

Returns a list of byte-offset pairs: `(start, end)` of the whole match followed by one pair per capture group (`undef, undef` for groups that did not participate), or the empty list on no match. An optional `$startoffset` (byte offset, default 0) begins the match part-way into the subject; an offset past the end of the subject returns the empty list.

Call it in list context. In scalar context you get the last value pushed, which is the end offset of the final pair, not a count and not a boolean.

Synopsis

```
my ($s, $e, @captures) = $re->match($subject);
while (my ($s, $e) = $re->match($subject, $pos)) {
    $pos = $e;
}
```

count

Count non-overlapping matches of the pattern in a subject string.

The whole scan loop runs natively: one method call per subject, not one per match. Empty matches are counted the way a global match ought to: after an empty match the scan retries at the same offset with `PCRE2_NOTEMPTY_ATSTART|PCRE2_ANCHORED` and then advances one character (the pcre2demo.c idiom), stepping over a whole UTF-8 character or CRLF pair where applicable.

Synopsis

```
my $n = Peta::FFI::PCRE2->new("agggtaaa|tttaccct", PCRE2_CASELESS)
                    ->count($genome);
```

substitute

Replace matches of the pattern in a subject, returning a new string.

Global by default (PCRE2_SUBSTITUTE_GLOBAL); pass an explicit $options value to override; it replaces the default rather than adding to it, so include PCRE2_SUBSTITUTE_GLOBAL yourself if you still want all occurrences. The replacement uses PCRE2 substitution syntax ($1, ${name}; \ escapes with PCRE2_SUBSTITUTE_EXTENDED). The subject is not modified, and a UTF8-flagged subject yields a UTF8-flagged result.

Synopsis

```
my $clean = Peta::FFI::PCRE2->new(">.*\\n|\\n")->substitute($fasta,
 ↪"");
```

Peta::FFI::Pool

Run C functions on worker threads, so that a call which hangs cannot hang the program.

Some C calls can block inside the kernel for as long as it likes: `statfs`, `stat` or open on an NFS, CIFS or CephFS mount whose server is gone waits uninterruptibly, and no signal or alarm gets the thread back. perl5 programs fork a child with a timeout for every such probe. A pool runs the call on a thread of its own instead, and reports how it went as an event: the event loop stays free, and a hung mount costs one parked thread, not a process per probe.

`Peta::FFI::Pool->new(threads => 4)` makes a pool of at most four threads. `$pool->call($function, $deadline, @args)` runs a function prepared with *Peta::FFI*'s `function` or `attach` with `@args`, and returns the call's id at once. When the call returns, an event `{ id => $id, state => 'done', result => ..., errno => ... }` is queued, and what the function wrote through its arguments (a P buffer, an `*{Class}` record, an `*i`) has been written into the caller's scalars by the time the event is taken. When `$deadline` seconds pass first, the event is `{ id => $id, state => 'timeout' }` instead. The thread stays with the call until it returns, since nothing can cancel it, and what the call then produces is thrown away: the caller's scalars are left alone.

A pool never queues a call. When all its threads are taken, by calls running or by calls hung past their deadline, the call's event is `{ id => $id, state => 'busy' }`, at once.

The events are taken with `next`, oldest first, and `fd` reads ready while there are some, so the pool is a *Peta::FFI::EventSource* for any event loop.

Synopsis

```
use Peta::FFI;
use Peta::FFI::Pool;
use Peta::FFI::EventSource;

my $libc   = Peta::FFI::dlopen("libc.so.6");
my $statfs = Peta::FFI::function($libc, "statfs", "(pP)i");

my $pool   = Peta::FFI::Pool->new(threads => 4);
my $source = Peta::FFI::EventSource->new($pool,
    fd => \&Peta::FFI::Pool::fd, next => \&Peta::FFI::Pool::next);

my $st = "\0" x 120;                    # struct statfs, x86-64
my $probe = $pool->call($statfs, 2, "/mnt/share", \$st);
my $guard = $source->watch(sub {
    for my $event (@_) {
        next unless $event->{id} == $probe;
        if ($event->{state} eq 'done') {
            print $event->{result} == 0 ? "mounted\n" : "error
 ↪$event->{errno}\n";
        }
        else {
            print "/mnt/share does not answer\n";    # timeout, or␣
 ↪busy
        }
    }
});
```

What a call takes

A call's arguments are read and copied before `call` returns, so the thread never reads a Perl value, and changing them meanwhile changes nothing. A function can take numbers, `b`, `o`, `p`, `P`, `*T` of a number or `o`, handles and records (`{Class}` and `*{Class}`). It can return `v`, a number other than a long double or a complex one, `b`, `o`, `p`, `s`, a handle or a record. Arrays, the out pointers and arrays of `*T[...]`, `*<Class>`, `*s` and `*P[...]`, and closures are refused: a call that needs them runs on Perl's own thread, with `$function->(@args)`. A closure passed as an `o` still cannot be called on a pool's thread, which stops the program as any thread of a library's own does.

A handle passed to a call is kept alive until the call returns, even past its deadline, so its destructor cannot free what the thread is still using. A handle a late call returns is destroyed, and a string it returns freed; a plain number is lost, so a file descriptor a late open returns stays open.

Threads, signals and fork

A pool's threads start when a call needs one, up to `threads`, and block every signal, which Perl's thread handles. They belong to the process that made the pool: after `fork`, the child makes a pool of its own, and `call` croaks on the parent's.

Functions

Pools

new

`Peta::FFI::Pool->new(threads => $n)`: a pool of at most `$n` threads (4 when not given), none started yet.

call

`$pool->call($function, $deadline, @args)`: run `$function`, a code reference from *Peta::FFI*'s `function` or `attach`, with `@args` on one of the pool's threads, and return the call's id. Its event is done when it returns, `timeout` when `$deadline` seconds (a fraction is fine; `undef` is no deadline) pass first, and `busy` at once when every thread is taken.

Events

fd

`$pool->fd`: a descriptor that reads ready while events are queued, for an event loop. Never close it: it is the pool's.

next

`$pool->next`: the oldest event, a hash reference, or undef when there is none. `id` is the call's, `state` is done, `timeout` or `busy`, and a done event has the function's `result` and the `errno` the call left; what the function wrote through its arguments is in the caller's scalars when it is returned.

Peta::FFI::Queue

Callbacks a C library makes from threads of its own, queued for Perl's event loop.

Some libraries call back on a thread of theirs: librados calls its monitor log callback and its watch and notify callbacks that way. Perl cannot run there, so a *Peta::FFI* closure called from such a thread stops the program. A queued closure runs no Perl when it is called: it copies its arguments, as its signature reads them, into a queue and returns at once, and Perl runs its sub later, when the event loop takes the queue's events.

`Peta::FFI::Queue->new` makes a queue, and `Peta::FFI::closure($sig, $sub, queue => $queue)` a closure that queues into it. Every call of the closure, from any thread - Perl's own included - becomes one event, `{ handler => $sub, args => [...], closure => $address }`; `dispatch` runs each queued event's handler with its arguments, oldest first. The queue reads ready on `fd` while it holds events, so it is a *Peta::FFI::EventSource* for any event loop.

A queued closure returns `v`, or a number, `b` or `o` given once as `returns => $value` (0 when not given), since nothing can compute a value at the time of the call. Its arguments are what any closure takes: numbers, `b`, `o`, `p` (copied up to its NUL) and records, copied, and bytes or an array of numbers C passes with their length in another argument, `&P[#k]` and `&T[#k]`. A `P` argument, an address valid only during the call, is not.

Synopsis

```
use Peta::FFI;
use Peta::FFI::Queue;
use Peta::FFI::EventSource;

my $queue = Peta::FFI::Queue->new;

# librados's rados_log_callback2_t
my $log = Peta::FFI::closure('(opppp u64u64u64 pp)v', sub {
    my (undef, $line, $channel, $who, $name, $sec, $nsec, $seq,
 ↪$level, $msg) = @_;
    print "$level $who: $msg\n";
}, queue => $queue);
$monitor_log2->($cluster, "info", $log, undef);

# rados_watchcb2_t: the notify's payload, data_len bytes
my $watch = Peta::FFI::closure('(ou64u64u64&P[#6]L)v', sub {
    my (undef, $notify_id, $cookie, $notifier, $payload) = @_;
    ...
}, queue => $queue);

my $source = Peta::FFI::EventSource->new($queue,
    fd => \&Peta::FFI::Queue::fd, next => \&Peta::FFI::Queue::next);
my $guard = $source->watch(sub { $_->{handler}->(@{ $_->{args} })
 ↪for @_ });

# or, with a loop of your own
vec(my $rin = '', $queue->fd, 1) = 1;
$queue->dispatch while select(my $rout = $rin, undef, undef, undef);
```

Bounds and lifetimes

A queue holds at most `limit` events (`Peta::FFI::Queue->new(limit => $n)`, 65536 when not given), so a library calling faster than Perl takes them cannot fill the memory: a call that finds the queue full is dropped, and `lost` counts it. An event whose closure was freed with `closure_free` before Perl took it is dropped too. A queued closure keeps its queue's memory alive, so a library may call it after the queue object is gone; those calls are dropped. As for any closure, `closure_free` comes only after the library has been told to stop calling.

Functions

Queues

new

`Peta::FFI::Queue->new(limit => $n)`: a queue of at most `$n` events, 65536 when not given.

dispatch

`$queue->dispatch`: run the handler of every queued call with its arguments, oldest first, and return how many ran. A handler that dies stops it there, with the error; the calls after it stay queued.

lost

`$queue->lost`: how many calls were dropped because the queue was full.

Events

fd

`$queue->fd`: a descriptor that reads ready while events are queued, for an event loop. Never close it: it is the queue's.

next

`$queue->next`: the oldest queued call as a hash reference - its closure's sub as `handler`, the arguments as the sub would have got them as `args`, the closure's address as `closure` - or undef when there is none.

Peta::FFI::UUID

Generate and parse RFC 4122 UUIDs via the system's libuuid library.

The library is located at run time, so pperl needs no build-time dependency on it. If `libuuid.so` is not installed, `use Peta::FFI::UUID` still succeeds and every call croaks with an instruction telling you which package to install.

```
use Peta::FFI::UUID qw(uuid_generate_random uuid_parse uuid_unparse␣
↪uuid_is_null);

my $text  = uuid_generate_random();     # "2d6c8b1e-0f4a-4c3e-9b7d-
↪5a1e2f3c4d5e"
my $bytes = uuid_parse($text);          # the same UUID as 16 raw␣
↪bytes
print uuid_unparse($bytes) eq $text ? "round trip\n" : "differs\n";
print uuid_is_null("\0" x 16) ? "nil UUID\n" : "not nil\n";
```

The generators return the text form; `uuid_parse` and `uuid_unparse` convert between it and the binary form. Nothing is exported by default. Peta::FFI::UUID is pperl's own module, version 0.2.0.

Functions

Other Functions

uuid_generate

`uuid_generate()`: a new UUID as its 36-character text form - random (version 4) when the system has a good random source, else time-based (version 1).

uuid_generate_random

`uuid_generate_random()`: a new random (version 4) UUID, as text: `"2d6c8b1e-0f4a-4c3e-9b7d-5a1e2f3c4d5e"`.

uuid_generate_time

`uuid_generate_time()`: a new time-based (version 1) UUID, as text. It carries the generation time and a node identifier, so it reveals when and where it was made.

uuid_parse

`uuid_parse($text)`: the 16-byte binary form of a UUID's text form, or undef when `$text` is not one.

uuid_is_null

`uuid_is_null($bytes)`: 1 when the 16-byte binary UUID is all zeros (the nil UUID), else 0. Anything but 16 bytes dies "requires a 16-byte UUID".

uuid_unparse

`uuid_unparse($bytes)`: the 36-character text form of a 16-byte binary UUID, in lower case. Anything but 16 bytes dies "requires a 16-byte UUID".

closure

Wrap a Perl coderef as a C-callable function pointer.

Synopsis

```
# int (*)(void*) - a progress handler
my $ptr = Peta::FFI::closure('(o)i', sub { my ($ud) = @_; 0 });
Peta::FFI::call($lib, 'sqlite3_progress_handler', '(oioo)v',
                $db, 100, $ptr, undef);
...
Peta::FFI::closure_free($ptr);
```

The signature grammar is `call`'s. A closure takes and returns the numbers, `o`, `p`, `P`, `b` and records - `{Class}` by value, and as an argument `*{Class}` by pointer, which the callback gets as a read-only copy. As an argument it also takes what C lends with its length in another argument: `&P[#k]` bytes, as a string, and `&T[#k]` numbers or `o`, as an array reference - `(o&P[#3]L)v` is a callback handed `void *data, size_t len`. There is no `;`, `s`, handle type, other array, pointer to a number, long double or complex number: C calls the closure through a fixed prototype and owns nothing it hands over.

The returned value is the raw address of the trampoline, to be passed wherever C wants the function pointer.

Options

- `queue => $queue` - a queued closure, which C may call on any thread: each call copies its arguments into the *Peta::FFI::Queue* `$queue` and returns at once, and the sub runs when Perl takes the queue's events. It takes no `P`, an address valid only during the call, and returns `v`, a number, `b` or `o`.
- `returns => $value` - what a queued closure returns to C, read once, here; 0 when not given.

Threads

A closure may be called only on the interpreter's own thread, during a C call Perl made - a progress handler, a comparator, a visitor. A library that calls it from a thread of its own stops the process with a message saying so, since Perl cannot run there. A library that calls back on its own threads takes a queued closure instead, and events a library queues itself reach Perl through a file descriptor the event loop watches, as a *Peta::FFI::EventSource*.

Lifetime

The closure stays alive until `closure_free` is called with the address this returned. Freeing it while C still holds the pointer leaves that library calling into released memory, so free it only after the C side has been told to stop - for a driver, that means tying it to the handle and releasing it at disconnect.

Errors in the callback

A callback that dies does NOT propagate: the error would have to unwind through the C caller's stack frame. `$@` is set and the closure returns a zero of its declared return type; check `$@` after the C call that triggered it.

dlopen

Open a shared library and return a handle suitable for passing to `call`.

Synopsis

```
my $lib = Peta::FFI::dlopen("libm.so.6");
my $lib = Peta::FFI::dlopen("/usr/lib/libuuid.so.1");   # absolute␣
↪path
```

What you get back

An opaque integer handle. Treat it as a token: pass it to `call` and later to `dlclose`, but do not do arithmetic on it. Store it in a lexical and keep it alive as long as you want to call into the library.

The argument is either a bare soname (e.g. `"libm.so.6"`) that the dynamic linker resolves through the usual `LD_LIBRARY_PATH` / `ld.so.cache` machinery, or an absolute path to an `.so` file. Both `RTLD_NOW` and `RTLD_GLOBAL` are set - symbols are resolved eagerly and become visible to subsequently loaded libraries.

Edge cases

- **Library not found** - croaks with the message reported by `dlerror`, e.g. `Peta::FFI::dlopen: libnope.so: cannot open shared object file: No such file or directory`.
- **Name contains a null byte** - croaks with `Peta::FFI::dlopen: invalid library name (contains null byte)`.
- **No argument** - croaks with `Peta::FFI::dlopen: requires library name`.
- **Calling dlopen repeatedly with the same name** - returns the same (reference-counted) handle each time. Pair every successful `dlopen` with a matching `dlclose` if you want the library to eventually unload.

call

Invoke a C function from an opened library and return its result.

Synopsis

```
my $n = Peta::FFI::call($lib, "strlen",   "(p)L", "hello");      # 5
my $r = Peta::FFI::call($lib, "sqrt",     "(d)d", 2.0);          #␣
↪1.4142...
my $s = Peta::FFI::call($lib, "getenv",   "(p)p", "PATH");       # /
↪usr/bin:...
Peta::FFI::call($lib, "srand", "(i)v", 42);                      #␣
↪void return
```

Arguments

- `$lib` - handle returned by `dlopen`.
- `$func` - symbol name to look up in the library, or the address of a function: a pointer C handed out, such as an entry of a table of callbacks, which has no name to look up.
- `$sig` - signature string `(args)ret`; see the module-level type-code table.
- the remaining arguments - one value per type code in `args`, coerced to the matching C type before the call.

What you get back

A single scalar whose shape depends on the return type:

- the integer types, `b` - an integer; perl's true or false.
- `d` / `f` / `D` - a float (a long double rounded to one).
- `cf` / `cd` - `[re, im]`.
- `p` - the returned `const char *` copied into a Perl string; `undef` if the C function returned `NULL`.
- `o` - the address; `0` for `NULL`.
- `v` - 1, so the call chains naturally in boolean contexts.
- `P` - pointer returned as a string (same rules as `p`).
- `*T` - a reference to the value it points to; `T[N]` - a reference to an array of the `N` values; `undef` for `NULL`.
- `{Class}` - a new record object; `*{Class}` - a copy of the record it points to, `undef` for `NULL`.

Output buffers (P)

A `P` argument that is a reference to a scalar passes that scalar's own bytes, and after the call the scalar holds what the callee wrote there - as many bytes as it had: `my $buf = "\0" x 64; call(..., \$buf)`. Any other `P` argument is a zeroed buffer the callee may write and nobody reads, as long as the string passed (minimum 16 bytes) or as many bytes as the number says.

Pointers and arrays

A `*T` argument is a reference to a scalar, which the callee reads and writes - an out-parameter, `\my $n` - or `undef` for `NULL`; a reference to an array passes its elements. A `T[N]` argument is a reference to an array of `N` values, and `T[]` one of any length; after the call the array holds what the callee left there.

Edge cases

- **Wrong argument count** - croaks with `Peta::FFI::call FNAME: expected N args, got M`.
- **Bad signature string** - croaks with `FFI signature error: ...` describing the offending character or missing `)`.
- **Symbol not found** - croaks with `Peta::FFI::call: symbol not found: FNAME` or the message reported by `dlerror`.
- **Incorrect type codes** - calling a function with a signature that does not match its real prototype is undefined behaviour. Check the C header before writing the signature.
- **String argument contains a null byte** - croaks with `Peta::FFI::call FNAME: CString error: ...`.

- **A handle of the wrong class, or a released one** - croaks before the call.

`call` parses the signature, looks the symbol up and prepares the call interface every time; a function called more than a few times is better prepared once with `function` or `attach`.

function

Prepare a C function once and return it as a code reference.

Synopsis

```
my $move_to = Peta::FFI::function($cairo, "cairo_move_to", "(odd)v
 ↪");
$move_to->($cr, 10, 20);

my $validate = Peta::FFI::function($lib, "schema_validate", "(
 ↪<My::Schema>p)s",
                                   free => "string_free");
my $errors = $validate->($schema, $json);
```

Arguments

- `$lib`, `$name`, `$signature` - as for `call`.
- `free => $function` - the library function that frees what the callee hands over: an `s` result, an `*s` argument, an array the callee allocates (`*T[r]`, `*T[#k]`, `*T[0]`) and the strings in one, and bytes it allocates (`*P[r]`, `*P[#k]`). By name (looked up in `$lib`) or by address. Without it these are freed with the C library's `free()`, which is right for C code that allocates with `malloc`, and wrong for anything else - a Rust library must name its own.
- `release => $function` - for an array of records the callee allocates: the library function `(array, count)` that frees it and whatever its records point to (libvirt's `virTypedParamsFree`). It runs when the last record taken from the array is gone.

What you get back

A code reference. Calling it with the arguments the signature lists returns what `call` would. The signature is parsed, the symbol resolved and the call interface prepared once, here, so each call costs only its argument and result conversion. A function whose arguments and result are integers, `f`, `d` or `o` - at most eight arguments, six of them integers or `o`, and not variadic - is also compiled into machine code of its own that calls it directly, once it has been called 8192 times, when the compile has paid for itself. `PPERL_FFI_STUBS=N` compiles after N calls instead, and `PPERL_FFI_STUBS=0` sends every call through the general path.

The function keeps its library loaded for as long as the code reference lives, whatever `dlclose` is called on.

Edge cases

- **Symbol not found**, **bad signature** - croak here, as `call` would on its first use.
- **Wrong argument count** - croaks with `NAME: expected N args, got M`, NAME being the C function's.
- **`free` on a function that hands nothing over**, **`release` on one without an allocated array of records** - croak.

attach

Prepare a C function once and install it as a named Perl sub.

Synopsis

```
package My::Zlib;
my $z = Peta::FFI::dlopen("libz.so.1");
Peta::FFI::attach($z, "zlibVersion", "()p");          #␣
↪My::Zlib::zlibVersion
Peta::FFI::attach($z, "crc32", "(Lpu32)L", as => "crc");
print zlibVersion(), "\n";
```

Arguments

As `function`, plus `as => $sub_name`: the Perl name to install, qualified with the calling package unless it contains `::`. Without it the C function's own name is used.

What you get back

A reference to the installed sub. Redefining the sub (another `attach` of the same name) releases the function it replaced.

errno

errno as the last C call through Peta::FFI left it.

Synopsis

```
my $fd = Peta::FFI::call($libc, "open", "(pi;u32)i", $path, 0x40,␣
↪0644);
if ($fd < 0) {
    local $! = Peta::FFI::errno();
    die "open $path: $!";
}
```

What you get back

An integer: the value errno had the moment the C function returned, read before Perl could change it. As in C, it means something only when the function reported a failure. Assigning it to `$!` gives its message.

handle_type

Declare a Perl class for a C object type, with its destructor.

Synopsis

```
Peta::FFI::handle_type("My::Schema", $lib, "schema_free");
my $compile = Peta::FFI::function($lib, "schema_compile", "(p)
 ↪<My::Schema>");

my $schema = $compile->($schema_json) or die "invalid schema";
...                                    # schema_free runs when
 ↪$schema goes
```

What it does

After this, a signature may name the type as `<My::Schema>`:

- **As a result**, the returned pointer becomes an object of class `My::Schema`; a NULL pointer becomes `undef`.
- **As an argument**, a `My::Schema` object passes its pointer, and `undef` passes NULL. Anything else - another class's handle, a plain number, a released handle - croaks rather than reaching C.
- When the last reference to an object goes, the destructor is called with its pointer, exactly once. No `DESTROY` method is involved, so the class is free for the program's own methods and subclasses.

`$destructor` is a function name in `$lib`, or an address. The library stays loaded for as long as the program runs, since objects may outlive every `dlclose`.

Edge cases

- **Declared again with the same destructor** - no effect.
- **Declared again with another destructor** - croaks.

release

Destroy a handle's C object now, ahead of its last reference.

Synopsis

```
Peta::FFI::release($schema);    # schema_free runs here
```

What you get back

True when the object was destroyed; false when `$handle` is no handle or was released already. A released handle stays an object of its class, and passing it to C croaks.

record

Declare a record class: a C struct, passed by value or by pointer.

Synopsis

```
Peta::FFI::record("My::Point", x => "d", y => "d");
Peta::FFI::record("My::Rect",  origin => "{My::Point}", size => "
 ↪{My::Point}");

my $p = My::Point->new(x => 1.5, y => -2);
print $p->x, "\n";                                  # 1.5
$p->y(4);                                           # set

# struct Point point_add(struct Point, struct Point);
my $sum = Peta::FFI::call($lib, "point_add", "({My::Point}
 ↪{My::Point}){My::Point}", $p, $p);
# void point_scale(struct Point *, double);
Peta::FFI::call($lib, "point_scale", "(*{My::Point}d)v", $p, 2);
```

Members

Each member is a name and its type, in the order of the C struct: the integer types, `f`, d, b, o, p (a `const char *`), a fixed array of numbers or o such as `u8[16]`, and another record by value, `{Class}`. The layout - each member's offset and the size and alignment of the whole - is the C compiler's for the platform, as libffi computes it.

The class

`Class->new(member => value, ...)` makes a zeroed record and sets the members given. Each member has an accessor: `$r->name` reads it, `$r->name($value)` sets it; an array member also takes `($index)`, `($index, $value)` and an array reference; a record member reads as a copy, an object of its class, and is set from one. A record is a reference to a string of the struct's bytes, so `$$r` is them and a record can be built from bytes C produced: `bless \$bytes, "My::Point"`.

A `p` member reads the string it points to, `undef` for `NULL`. Set from Perl, it points to the record's own copy of the string, which lives as long as the record does or until the member is set again; `undef` sets `NULL`. A string C stored in the member is C's: the record never frees it. Where C moves the strings between records - an array of them it sorts, written back - each goes with the record that points to it, and a record stored into another

as its member gives it copies of its own. A bare copy of the record's bytes points to the same strings but does not keep them.

In a signature `{Class}` passes and returns the struct by value and `*{Class}` a pointer to it - the record's own bytes, which the callee may change in place; `undef` is `NULL`.

Edge cases

- **A class declared twice** - croaks.
- **A member type a struct cannot hold here** (`s`, `P`, a handle, a pointer type, a long double or complex member) - croaks naming it.

union

Declare a union class: a C union, reached by pointer and in memory.

Synopsis

```
Peta::FFI::record("Keysym", scancode => "i32", sym => "i32", mod =>
 ↪"u16", unused => "u32");
Peta::FFI::record("KeyEvent", type => "u32", timestamp => "u32",␣
 ↪windowID => "u32",
                  state => "u8", repeat => "u8", pad => "u8[2]",␣
 ↪keysym => "{Keysym}");
Peta::FFI::union("Event", type => "u32", key => "{KeyEvent}",␣
 ↪padding => "u8[56]");

my $ev = Event->new;                          # SDL_Event
while (Peta::FFI::call($sdl, "SDL_PollEvent", "(*{Event})i", $ev)) {
    print $ev->key->keysym->sym, "\n" if $ev->type == 0x300;   #␣
 ↪SDL_KEYDOWN
}
```

As `record`, with every member at offset 0: the union is as large as its largest member, rounded to its strictest alignment. A member that is a record - in a union or a struct - reads as a copy, an object of its class, and is set from one.

A union passes by pointer, `*{Class}`, and is read and written whole by `fetch`, `store` and as a member of a record. Passing it by value

- as `{Class}`, or inside a struct passed by value - croaks: libffi has no union type, and a struct of the same size passes differently on some ABIs, which would be a wrong call rather than an error.

sizeof

The size of a type in bytes, as C's `sizeof`.

Synopsis

```
my $n = Peta::FFI::sizeof("{My::Point}");   # 16
my $buf = Peta::FFI::alloc(Peta::FFI::sizeof("u32[8]"));
```

Any one type of the signature language: a number type, o, p (a pointer), `T[N]`, `*T` and a record. `T[]` is 0.

symbol

The address of a symbol - a function or a variable - in a library.

Synopsis

```
my $environ = Peta::FFI::symbol($libc, "environ");   # char ***
my $first = Peta::FFI::peek_cstr(Peta::FFI::fetch(Peta::FFI::fetch(
 ↪$environ, "o"), "o"));
```

`$lib` is a dlopen handle, or `undef` for everything loaded. Croaks when the library has no such symbol.

fetch

Read the value of a type at an address.

Synopsis

```
my $n  = Peta::FFI::fetch($ptr, "i32");
my $xy = Peta::FFI::fetch($ptr, "d[2]");           # [1.5, -2]
my $pt = Peta::FFI::fetch($ptr, "{My::Point}");    # a My::Point␣
 ↪copy
```

What `call` would return for that type, read from the memory at `$addr`: the numbers, b, o (`0` for `NULL`), D as a float, `cf`/`cd` as `[re, im]`, p - the C string the `char *` there points to, `T[N]` as an array reference and `{Class}` as a record. Reading memory the process does not own is undefined behaviour.

store

Write a value of a type at an address.

Synopsis

```
Peta::FFI::store($ptr, "i32", 42);
Peta::FFI::store($ptr, "{My::Point}", My::Point->new(x => 1));
```

The numbers, b, o, D, `cf`/`cd` (`[re, im]`), `T[N]` from an array reference and `{Class}` from a record of that class. A string is not stored: memory C keeps would point into a Perl value that moves - `alloc` and poke its bytes, and store the address as o.

variable

A library's global variable as a Perl scalar.

Synopsis

```
my $optind = Peta::FFI::variable($libc, "optind", "i");
print $$optind, "\n";        # reads the C variable
$$optind = 1;                # writes it
```

A reference to a scalar bound to the variable: reading `$$var` reads the C variable at that moment, assigning to it writes the variable, for any type `store` can write; a `p` variable is read-only. The library stays loaded while the scalar lives. Croaks when the library has no such symbol.

dlclose

Release a library handle previously returned by `dlopen`.

Synopsis

```
Peta::FFI::dlclose($lib);
```

What you get back

Always 1. The return value exists so the call composes cleanly in expressions; inspecting it tells you nothing about success.

Edge cases

- **No argument** - croaks with `Peta::FFI::dlclose: requires library handle`.
- **Zero or null handle** - silently ignored. Passing `0` is safe and commonly used as an "already closed" sentinel.
- **Double close** - passing a handle that has already been released is undefined behaviour at the `dlclose` level; keep a single owner per handle or null out your copy after closing.
- **Outstanding calls** - closing a library while a pointer or string it returned is still in use is undefined behaviour. Copy everything you need out of the library first.

scan

Find the shared libraries in a list of directories.

Synopsis

```
my $libs = Peta::FFI::scan;
print $libs->{"libm.so.6"}, "\n";          # /usr/lib/libm.so.6
my @names = grep { /^libssl/ } keys %$libs;

my $ours = Peta::FFI::scan("/opt/PetaTech/lib");
my $core = Peta::FFI::dlopen($ours->{"libpeta_core.so"});
```

What you get back

A hashref mapping each library's file name to its path: every `lib*.so` and `lib*.so.*` file in the directories searched that this process can load - an ELF file of its own architecture. A linker script (`/usr/lib/libc.so` is one), a 32-bit library on a 64-bit system and any other file are left out. A name found in more than one directory maps to the first one's file.

Where it looks

- **`scan(@dirs)`** - those directories, in that order, and no others.
- **`scan()`** - the directories in `PETA_FFI_PATH` (colon-separated, like `PATH`), then where the dynamic linker looks for a bare name: `LD_LIBRARY_PATH`, the directories `/etc/ld.so.conf` configures and the system's own (`/lib`, `/usr/lib` and their 64-bit forms).

The search happens only here. `dlopen` with a bare name asks the dynamic linker, which never reads `PETA_FFI_PATH`; a library installed in a directory of its own is found by `scan` and opened by the path it returns.

Edge cases

- **A directory that does not exist or cannot be read** - skipped.
- **One directory reached twice** (`/lib64` is often a link to `/usr/lib`) - searched once. A library in a system directory is reported under the directory's real path; one in a directory you named, under the name you gave.

peek

Read `$len` raw bytes at address `$ptr` into a Perl byte string.

Synopsis

```
my $bytes = Peta::FFI::peek($ptr, 16);
```

What you get back

A byte string of exactly `$len` bytes copied from `$ptr`; `undef` when `$ptr` is `0`/`undef`. `$len` of `0` returns the empty string.

Edge cases

- **Fewer than 2 arguments** - croaks.
- **Address or length wrong** - reading memory the process does not own is undefined behaviour; the length must come from the same C API that produced the pointer (e.g. `PQgetlength`).

peek_cstr

Read the NUL-terminated C string at address `$ptr`.

Synopsis

```
my $str = Peta::FFI::peek_cstr($ptr);
```

What you get back

The bytes at `$ptr` up to (excluding) the first NUL; `undef` when `$ptr` is `0`/`undef`. This is the reader for `o`-typed returns that point at C strings (`PQgetvalue`, `mysql_error`, ...).

poke

Copy a Perl byte string into memory at address `$ptr`.

Synopsis

```
my $buf = Peta::FFI::alloc(8);
Peta::FFI::poke($buf, pack("l", 42));
```

What you get back

The number of bytes written (the string's byte length).

Edge cases

- **Fewer than 2 arguments or NULL pointer** - croaks; poking through NULL is never meaningful.
- **Buffer too small** - writing past an allocation is undefined behaviour; the destination must be at least as large as the string, which the caller controls via `alloc`.

alloc

Allocate `$len` zeroed bytes of C memory; returns the address.

Synopsis

```
my $out = Peta::FFI::alloc(8);       # e.g. a sqlite3** out-param
↪cell
Peta::FFI::call($lib, "sqlite3_open", "(po)i", ":memory:", $out);
my $db = Peta::FFI::unpack_ptr(Peta::FFI::peek($out, 8));
Peta::FFI::free($out);
```

What you get back

The address of a zeroed `calloc` block, as an integer for `o` arguments. The caller owns it: every `alloc` needs a matching `free`. A `$len` of `0` still returns a valid 1-byte allocation.

free

Release memory obtained from `alloc`.

Synopsis

```
Peta::FFI::free($ptr);
```

What you get back

Always 1.

Edge cases

- **Zero or undef pointer** - silently ignored, like `free(NULL)`.
- **Double free / foreign pointers** - undefined behaviour; only pass addresses obtained from `alloc`, exactly once. Memory a C library allocated is released by that library's own function (`PQclear`, `mysql_free_result`, ...), never by `free`.

pack_ptr

Pack an address into a native pointer-sized byte string.

Synopsis

```
# Build a char*[] for PQexecParams-style APIs:
my $vec = join "", map { Peta::FFI::pack_ptr($_) } @addrs;
```

What you get back

The address as native-endian bytes, sized to the platform pointer width - the in-memory representation a C pointer slot holds.

unpack_ptr

Unpack a native pointer-sized byte string into an address.

Synopsis

```
my $db = Peta::FFI::unpack_ptr(Peta::FFI::peek($out, 8));
```

What you get back

The address stored in the string's first pointer-sized bytes, as an integer for o arguments. Croaks when the string is shorter than the platform pointer width.

Peta::ReadLine

Interactive line editing, command history, and configurable key bindings, with the Term::ReadLine::Gnu interface.

Install a readline instance, hand it a prompt, and you get the usual bash-style editing: cursor motion, word-wise kill and yank, incremental history search, filename completion, and an inputrc-controlled key map. The module has Term::ReadLine::Gnu's whole surface - nearly one hundred entry points expose readline's C-level interface directly, organised here by the same section names the GNU Readline manual uses - and `Term::ReadLine::Gnu` is this module under its old name.

A typical script calls `new` to obtain the readline object, then `readline` in a loop with `add_history` after each accepted line. Scripts that customise behaviour reach for `rl_parse_and_bind` and `rl_variable_bind` for inputrc-style configuration, or install completion by binding `rl_completion_entry_function` via the `Attribs` tied hash.

```
use Term::ReadLine;

my $term = Term::ReadLine->new('demo', \*STDIN, \*STDOUT);
print $term->ReadLine, "\n";                    # Term::ReadLine::Gnu
$term->add_history($_) for 'ls -l', 'make test', 'git status';
my @h = $term->GetHistory;
print scalar(@h), " in history, last '$h[-1]'\n";   # 3 in history,␣
↪last 'git status'

my $attribs = $term->Attribs;                   # readline's␣
↪variables, as a hash
$attribs->{completion_entry_function} = $attribs->{list_completion_
↪function};
$attribs->{completion_word} = [qw(alpha beta gamma)];   # Tab␣
↪completes these
```

(continues on next page)

(continued from previous page)

```
my $line = $term->readline('> ');              # one line, with␣
↪editing and history
```

Programs normally `use Term::ReadLine`, which picks `Term::ReadLine::Gnu` when it is available, as it is here, and gets this module; a program that names `Term::ReadLine::Gnu` gets it under that name, and `ReadLine` answers with the name it was asked by. The library behind the interface is pperl's own line editor, built in: no readline or curses library is loaded, and the terminal is described from the terminfo database. It behaves as GNU Readline 8.3 does - the same commands, key bindings, init file, history and completion, and the same screen after every key - and says what it is: `$attribs->{library_version}` is `8.3 (Peta::ReadLine 0.2.0)`, the interface level first, where programs look for a number, and `$attribs->{gnu_readline_p}` is 0.

A few methods are Perl in `ReadLine/XS.pm` over private XSUBs, and so are not among the entries below: `$term->rl_message($fmt, @args)` shows `sprintf($fmt, @args)` in place of the prompt until `rl_clear_message`; `$term->rl_discard_keymap($map)` frees the bindings of a keymap (by name or handle) and returns it, still usable; `$term->get_history_event($string, $cindex [, $qchar])` expands the history event at `$cindex` and returns `($text, $cindex)`, the index where scanning stopped, so a string of several `!`-events can be walked one at a time.

This is a native implementation of the XS half of Term::ReadLine::Gnu 1.47; `ReadLine.pm` and `ReadLine/XS.pm` are its `Gnu.pm` and `Gnu/XS.pm` under this module's name.

Differences from upstream

The module is `Peta::ReadLine`; `Term::ReadLine::Gnu` is an alias of it, so a program written for Term::ReadLine::Gnu runs unchanged and sees that name, while `Peta::ReadLine->new` reports `Peta::ReadLine`.

Modules

- `Peta::ReadLine::XS`

Peta::ReadLine::XS

Functions

Parse and configuration

`rl_get_function_name`

Return the name under which a readline command function is registered, or `undef` if it has none.

rl_get_all_function_names

Return the name of every readline command currently known, including any added with `rl_add_defun`.

rl_parse_and_bind

Apply a single line of inputrc syntax (`"\\C-x\\C-r": re-read-init-file`, `set editing-mode vi`, ...).

rl_read_init_file

Load and apply an inputrc file. Defaults to `$ENV{INPUTRC}` or `~/.inputrc`.

rl_variable_bind

Set inputrc variable `$name` to the string `$value` (e.g. `completion-ignore-case`, `On`).

rl_variable_value

Return the current value of inputrc variable `$variable` as a string, or `undef` if it is unknown.

rl_variable_dumper

Print every inputrc variable and its current value to `rl_outstream`. Pass truthy `$readable` for inputrc syntax.

rl_function_dumper

Print every named readline command and the keys bound to it. Pass truthy `$readable` for inputrc-style output.

rl_macro_dumper

Print every key macro bound in the current keymap. Pass truthy `$readable` for inputrc-style output.

rl_get_termcap

Look up the termcap capability named `$cap` and return its string value, or `undef` if not defined.

Input

rl_clear_timeout

Cancel any timeout previously set with `rl_set_timeout`, so reads wait indefinitely again.

rl_timeout_remaining

Report how much of the timeout set by `rl_set_timeout` is still left.

rl_read_key

Block until one key is available on `rl_instream` and return its byte value.

rl_getc

Read one byte from `$stream` (defaulting to `rl_instream`), bypassing readline's macro and timeout logic.

rl_stuff_char

Inject one byte onto readline's pending-input queue, as if the user had just pressed it.

rl_execute_next

Arrange for the next call to `rl_read_key` to return `$c` without doing any terminal I/O.

rl_clear_pending_input

Discard any input queued with `rl_stuff_char` or `rl_execute_next`.

rl_set_keyboard_input_timeout

Set the multi-byte escape-sequence timeout in microseconds. Returns the previous value.

Terminal

tgetstr

Return the terminal's escape sequence for termcap capability `$id`, or `undef` if it has none.

rl_prep_terminal

Put the terminal into raw mode for line editing. Pass a truthy `$meta_flag` to enable 8-bit input.

rl_deprep_terminal

Undo `rl_prep_terminal` and restore the terminal to cooked mode.

rl_reset_terminal

Re-query termcap for `$terminal_name` (default `$ENV{TERM}`), refreshing readline's idea of the terminal.

rl_set_screen_size

Override readline's idea of the window size with `$rows` and `$cols`.

rl_get_screen_size

Return the terminal size readline is currently assuming as `($rows, $cols)`.

rl_resize_terminal

Re-read the terminal size from the OS - typically called from a `SIGWINCH` handler.

rl_reset_screen_size

Force readline to re-query `TIOCGWINSZ` and reset `screenheight`/`screenwidth` from scratch.

Line editing

rl_completion_matches

Build the list of completions for `$text` by calling a generator repeatedly. Returns the matches.

rl_insert_text

Insert `$text` into the line buffer at point, advancing the cursor past the inserted characters.

rl_delete_text

Delete the characters of the current line in the half-open range `[$start, $end)`. Returns the number removed.

rl_copy_text

Return a copy of the line buffer in the half-open range `[$start, $end)` without modifying the buffer.

rl_kill_text

Delete text in `[$start, $end)` and push it onto the kill ring so it becomes yankable with `C-y`.

rl_replace_line

Replace the entire line buffer with `$text`. Passing a truthy `$clear_undo` also drops the undo list.

rl_push_macro_input

Queue `$macro` as pending input so readline re-reads it character-by-character on the next key request.

Display

rl_display_match_list

Display an array reference of completion matches in columns, the way readline lists them itself.

rl_redisplay

Repaint the current line from readline's in-memory buffer. Call this after you edit the buffer programmatically.

rl_forced_update_display

Force a full repaint even if readline thinks nothing changed. Useful after background output corrupts the screen.

rl_on_new_line

Tell readline that the cursor is now on a fresh line, so the next redisplay reprints the prompt.

rl_on_new_line_with_prompt

Like `rl_on_new_line`, but asserts the prompt is already physically on screen - skip the reprint.

rl_reset_line_state

Reset the display state so the next `rl_redisplay` assumes a clean line (no partial output pending).

rl_clear_message

Remove the status message installed by `rl_message` and restore the original prompt.

`rl_crlf`

Emit a newline (CRLF) on the output stream, moving readline's cursor tracking with it.

`rl_ding`

Ring the terminal bell (or flash, depending on the `bell-style` inputrc setting).

`rl_show_char`

Display character `$c` using readline's usual control-character rendering (e.g. `^A` for byte 1).

`rl_save_prompt`

Save the current prompt and line state so it can be restored after a temporary message or sub-prompt.

`rl_restore_prompt`

Restore the prompt and line state previously stashed by `rl_save_prompt`.

`rl_set_prompt`

Change the prompt to `$prompt` for the next redisplay. Accepts `\[...\]` escapes to mark invisible regions.

`rl_expand_prompt`

Parse a prompt string for invisible-character escapes and return its visible length.

Basic readline

`rl_readline`

Display a prompt, read one line with full editing, and return the entered string (or `undef` on EOF).

`rl_initialize`

Initialise (or re-initialise) readline's internal state and re-read the inputrc. Returns `0` on success.

History

`add_history`

Append `$line` to the end of the history list so it shows up in later Up-arrow recalls and searches.

clear_history

Delete every entry from the in-memory history list.

using_history

Reset the history offset pointer back to the end of the list so `previous_history` starts from the newest entry.

where_history

Return the current offset into the history list (the index a subsequent `current_history` would read).

history_set_pos

Move the history cursor to `$pos`. Returns `1` on success, `0` if the position is out of range.

current_history

Return the line at the current history cursor, or `undef` if the cursor is past the end.

history_get

Return the history line at absolute `$offset`, or `undef` if no such entry exists.

previous_history

Move the history cursor one step back and return that entry, or `undef` if already at the oldest line.

next_history

Move the history cursor one step forward and return that entry, or `undef` if already past the newest line.

remove_history

Delete the history entry at `$which` and return its text.

replace_history_entry

Replace the history entry at `$which` with `$line` and return the previous text, or `undef` if out of range.

stifle_history

Cap the history list at `$max` entries, or lift the cap when `$max` is `undef`. Returns the effective cap.

unstifle_history

Remove any stifling cap on the history list and return the previous cap (or a negative value if none was set).

history_is_stifled

Return non-zero if the history list is currently stifled (length-capped), zero otherwise.

history_total_bytes

Return the total number of bytes occupied by all history lines in memory.

read_history_range

Load lines from a history file into memory. Reads the whole file by default, or lines `$from..$to` when given.

write_history

Write every entry in the current history list to `$filename`, overwriting the file. Returns `0` on success.

append_history

Append the last `$nelements` entries of the in-memory history to `$filename`, preserving existing contents.

history_truncate_file

Truncate `$filename` to at most `$nlines` of history, keeping the newest lines. Returns `0` on success.

history_expand

Apply csh-style history expansion (`!!`, `!$`, `^old^new^`) to `$line`. Returns `(code, expanded)`.

Synopsis

```
my ($code, $expanded) = $term->history_expand($input);
# $code: 0 no change, 1 expanded, -1 error, 2 print-only
```

history_search

Search the history for an entry that contains `$string`, moving the cursor to the match. Returns the match offset or `-1`.

history_search_prefix

Same as `history_search`, but matches only when the history entry *starts* with `$string`.

history_search_pos

Search the history starting at absolute `$pos` (default: current cursor). Returns match offset or `-1`.

history_tokenize

Split `$text` into tokens the way Bash history expansion does (word-splitting, quote awareness). Returns a list.

Character classification

rl_alphabetic

Return non-zero if byte `$c` is a word-constituent character under readline's locale-aware rules.

Undo

rl_begin_undo_group

Open a new undo group so a sequence of edits collapses into one user-visible undo step.

rl_end_undo_group

Close the undo group opened by `rl_begin_undo_group`.

rl_do_undo

Perform one undo step. Returns `0` when no further undo is available.

rl_free_undo_list

Drop the entire undo history for the current line.

rl_modifying

Record the range `[$start, $end)` of the line buffer on the undo stack before you modify it by hand.

Signal handling

`rl_cleanup_after_signal`

Restore the terminal to cooked mode and reset signal handlers - call this from your own signal handler.

`rl_free_line_state`

Free the partially-edited line buffer so a signal-handling exit leaves readline in a clean state.

`rl_reset_after_signal`

Re-prepare the terminal and reinstall readline's signal handlers after a `SIGCONT` or similar resumption.

`rl_check_signals`

Give readline a chance to run any deferred signal handlers.

`rl_echo_signal_char`

Echo the tty-configured character for signal `$sig` (e.g. `^C` for `SIGINT`).

`rl_pending_signal`

Return the number of a signal readline has observed but not yet dispatched, or `0` if none is pending.

`rl_set_signals`

Install readline's default signal handlers (for `SIGINT`, `SIGTERM`, `SIGQUIT`, `SIGWINCH`, `SIGHUP`, ...).

`rl_clear_signals`

Remove the signal handlers installed by `rl_set_signals`.

State save and restore

`rl_save_state`

Snapshot readline's internal state and return an opaque handle you can later pass to `rl_restore_state`.

rl_restore_state

Reinstall the state previously captured by `rl_save_state`.

rl_completion_matches

Build the list of completions for `$text` by calling a generator repeatedly. Returns the matches.

The generator is called as `$fn->($text, $state)`, with `$state` zero on the first call and non-zero on every call after it, and returns one match each time or `undef` once it is exhausted. Omitting `$fn` uses whatever `$attribs{completion_entry_function}` currently holds.

A generator passed here is in force for the duration of this call only: whatever completion function `$attribs` held is put back afterwards, so a one-off generator cannot disturb it.

rl_readline

Display a prompt, read one line with full editing, and return the entered string (or `undef` on EOF).

Synopsis

```
while (defined (my $line = $term->readline('prompt> '))) {
    $term->addhistory($line) if $line =~ /\S/;
    last if $line eq 'quit';
}
```

The call blocks until the user presses Return. A bare Ctrl-D on an empty line returns `undef`. The returned string has its trailing newline already stripped, matching the underlying `readline(3)` C function. The prompt argument is optional; pass `undef` for none.

add_history

Append `$line` to the end of the history list so it shows up in later Up-arrow recalls and searches.

Synopsis

```
$term->add_history($line) if $line =~ /\S/;
```

Readline does not deduplicate for you. Wrap the call in an `if` if you want to skip blank or repeated lines. Use `write_history` to persist the collected lines to disk.

read_history_range

Load lines from a history file into memory. Reads the whole file by default, or lines `$from..$to` when given.

Synopsis

```
$term->ReadHistory;                          # default file, full
↪range
$term->ReadHistory('~/.myhist', 100, -1);  # last hundred-odd lines
```

Passing `undef` for `$filename` reads from `~/.history`. `$to = -1` means "to the end". Returns `0` on success, a non-zero errno on failure.

rl_parse_and_bind

Apply a single line of inputrc syntax (`"\\C-x\\C-r": re-read-init-file`, `set editing-mode vi`, ...).

Synopsis

```
$term->parse_and_bind('set editing-mode vi');
$term->parse_and_bind('"\\C-l": clear-screen');
```

This is the programmatic door into the same grammar that `~/.inputrc` uses. Returns `0` on a successful parse.

Peta::Shell

ipps, the interactive mode of pperl

SYNOPSIS

```
$ pperl --shell
ipps> my @list = (1, 2, 3)
(1, 2, 3)
ipps> push @list, 4; scalar @list
4
ipps> for my $n (@list) {
 ...>     print $n * 2, " ";
 ...> }
2 4 6 8
ipps> ^D
```

From Perl:

```
use Peta::Shell;
exit Peta::Shell->run(@ARGV);
```

DESCRIPTION

ipps reads Perl, runs it, and prints the value of the last statement. Everything a line declares is still there for the next line: `my` variables, subs, loaded modules, pragmas, the current package. See *Peta::Shell::Session* for exactly what carries over and how.

At the prompt

- A line whose last statement ends in `;` prints nothing. Without the `;` its value is printed as *Peta::Shell::Printer* shows it: a number bare, a string in quotes, a list in parentheses, a reference as its structure, nothing for an empty list. A loop or a conditional typed by itself prints nothing either, and neither does a last statement that is a `print`, `printf` or `say`: its value is only whether the output succeeded.
- When what your code printed does not end in a newline, the prompt starts on the next line and a `%` in reverse video marks where the output stopped.
- The value last printed is in `$ans`, and all of them, when there were several, in `@ans`:

```
ipps> 6 * 7
42
ipps> $ans + 1
43
```

- Input that is not finished (an open bracket, an open quote, a here-document without its end) gets a second prompt, `...>`, until it is. Blank lines inside it are kept, so pasted code works.
- Tab completes the word in front of the cursor: a variable of the session, a key of one of its hashes, a method of an object or class, a sub, a module after `use`, a file name inside a string, a command. With several candidates it completes as far as they agree, and a second Tab lists them. *Peta::Shell::Complete* says exactly what is offered where. When none of that fits, Tab completes a file name.
- Ctrl-C at a prompt drops the input typed so far. Ctrl-C while a line runs stops that line; the session goes on.
- Ctrl-D on an empty prompt ends the session, and so does `exit`.
- An error is printed and the session goes on with everything it had.

Commands

A line that starts with a colon and a word is a command of ipps, not Perl. A command may be shortened while it stays unique: `:q`, `:v`.

- :help

  The list of commands, yours and the macros included.

- :vars

  The variables of the session: name, kind and value, shortened.

```
ipps> my @list = (1, 2, 3); my $name = "ada";
ipps> :vars
```

(continues on next page)

(continued from previous page)

```
@ans   array of 0  ()
$ans   undef       undef
@list  array of 3  (1, 2, 3)
$name  string      'ada'
```

- :scalar EXPR

  The value of EXPR in scalar context; a line is otherwise evaluated in list context. `:scalar @list` is the number of elements. The word `scalar` is put in front and nothing more, so `:scalar 1, 2` is the list `scalar(1), 2`, as in Perl.

- :dump EXPR

  The value whole: every element, every level, every character, whatever the settings say and whether or not the line ends in `;`.

- :time LINE

  Runs the line and then shows the wall-clock and the processor time it took, compiling included.

- :jit LINE

  Runs the line and then shows what the JIT did while it ran: how many loops it compiled to machine code, how many loops the line brought that it had not seen before, and compiled of seen for everything since ipps started, its own code included.

```
ipps> :jit my $t = 0; for my $i (1 .. 300_000) { $t += $i } $t
45000150000
jit: 1 loop compiled, 1 new loop seen (1 of 68 since the start)
```

  A loop is compiled once it has run often enough, and that may be in a later line than the one that wrote it: the loop of a sub is seen where the sub is defined and compiled where a call makes it hot. The counts say that a loop was compiled, not why another was not.

- :source SUB

  The code of a sub, by name or by an expression that gives a code reference, as *B::Deparse* writes it.

- :which MODULE

  The file pperl loads the module from, or `built into pperl`, and whether it is loaded. A file name works too: `:which ExtUtils/typemap`.

- :install MODULE ...

  Installs modules from CPAN with pperl's own client, as `pperl --cpan` does. They can be used at once.

- **programs,**
  programs NAME ...

  Makes each NAME a sub of `main` that runs the program of that name, as the `autorun` setting does for every unknown sub; see "Programs as functions". A sub so declared needs no parentheses. Without names it lists them.

- :load FILE

  Runs a file in the session: what it declares is there afterwards. The value of its last statement is not shown.

- :save FILE

  Writes every line that ran without an error, in order, as a script. Commands are not written, the lines they ran are.

- **set,**
  set NAME, :set NAME VALUE

  Shows all settings, shows one, changes one:

```
max_items   elements shown of an array or hash (24); 0: all
max_depth   levels shown of a nested structure (6); 0: all
max_string  characters shown of a string (240); 0: all
width       columns for a printed value; 0: the terminal's
time        on: show the times of every line
prompt      the prompt; quotes keep a space: :set prompt "> "
prompt2     the prompt for unfinished input
shell       what "! command" runs the command with
autorun     on: an unknown sub of main runs the program of its name
```

  To keep a setting, put it in `rc.pl`: see “set”.

- **history,**
  history N

  The last 20, or N, lines entered. On a terminal these include earlier sessions.

- :reset

  Starts the session afresh: no `my` variables, the first pragmas, the package `main`, no macros, and `rc.pl` run again. Subs and package variables are not the session's and stay.

- :quit

  Leaves, as Ctrl-D does.

`::name(...)` is Perl, a call into `main`, and stays Perl.

Commands of your shell

A line that starts with `!` and a space is handed to your shell as it stands and runs on the terminal:

```
ipps> ! ls -l *.pl
ipps> ! git status
```

Nothing of such a line is Perl, and its output is not kept. An exit status other than 0 is reported. To have a command's output in a variable, use Perl's backticks, which is Perl and works as always:

```
ipps> my @files = `ls *.pl`
```

The space matters: `!$done` and `!defined $x` are Perl's negation.

The command runs under `$SHELL -c`, or `/bin/sh -c` without `$SHELL`. Such a shell does not read the start file of an interactive one, so its aliases are unknown to it. The `shell` setting changes that; for bash's aliases:

```
ipps> :set shell bash -ic
```

Programs as functions

With `:set autorun on`, a call of a sub that `main` does not have runs the program of that name instead. The arguments are joined with spaces into a command line for `/bin/sh`; the result is the program's output, a list of lines in list context and one string otherwise. Its errors reach the terminal and its exit status is in `$?`:

```
ipps> :set autorun on
ipps> whoami()
"rj\n"
ipps> my @lines = uname('-a');
```

Perl's rules for calling a sub still decide what parses. Without parentheses a name must be declared, which `:programs` does: `:programs ls` and then `ls '-l'`. Each argument is Perl, a quoted string: `ls -l` is `ls(-l $_)`, a file test. The shell still expands `*` in them. Real shell syntax is what `! command` is for.

A program that cannot be run is an error, and so a typing error in the name of a sub is reported differently: `pritn("x")` says `Can't exec "pritn": No such file or directory`.

autorun is `main`'s `AUTOLOAD`, so it cannot be switched on while `main` has an `AUTOLOAD` of your own, and switching it off leaves yours alone. It works in package `main` only.

Macros and commands of your own

`rc.pl` is the place for them:

```
Peta::Shell->macro(p => 'print');
Peta::Shell->command(pwd => sub { require Cwd; print Cwd::getcwd(),
 ↪"\n" });
```

After that `p "hello\n"` at the prompt runs `print "hello\n"`, and `:pwd` prints the directory. See "macro" and "command".

Without a terminal

When standard input is not a terminal, ipps reads it line by line without prompt or editing and prints values and errors the same way:

```
$ echo 'my $x = 6; $x * 7' | pperl --shell
42
```

Files

Both live under `$PPERL_HOME/ipps`, by default `~/.pperl/ipps`.

- rc.pl

  Perl run in the session before the first prompt. What it declares is there at the prompt. Not read with `--norc`.

- history

  The lines typed at the prompt, read at start and written at the end.

OPTIONS

- –norc

  Do not run `rc.pl`.

METHODS

run

```
my $status = Peta::Shell->run(@arguments);
```

Runs a session to its end and returns the exit status: 0, or 2 for arguments it does not understand.

new

```
my $shell = Peta::Shell->new(%options);
```

Options: `rc` (run `rc.pl`, default true), `home` (the directory of `rc.pl` and `history`), `in`, `out`, `err` (file handles, default the standard three), `prompt` and `prompt2`.

loop

Reads and evaluates until the input ends. Returns the exit status.

session

The *Peta::Shell::Session* of this shell.

current

```
my $shell = Peta::Shell->current;
```

The shell that is running, or undef. This is how rc.pl and the lines typed at the prompt reach it.

macro

```
Peta::Shell->macro(p  => 'print');
Peta::Shell->macro(ls => sub { my ($rest) = @_; "print map { \"\$_\\
↪n\" } glob('$rest*')" });
```

Defines a macro: a word that, typed as the first word of a line and followed by a space or the end of the line, is replaced before the line is compiled. A string replaces the word and the rest of the line stays; a code reference gets the rest of the line and returns the Perl to run instead. undef removes the macro. Called on the class, it means the running shell.

command

```
Peta::Shell->command(pwd => sub { my ($shell, $rest) = @_; require␣
↪Cwd; print Cwd::getcwd(), "\n" });
```

Defines a command of your own, typed as :pwd. It gets the shell and the rest of the line. undef removes it. The names of the built-in commands are taken.

set

```
Peta::Shell->set(max_items => 100);
```

Changes a setting, as :set does at the prompt.

Modules

- Peta::Shell::Complete - what Tab offers in ipps
- Peta::Shell::Printer - show a Perl value as bounded, readable text
- Peta::Shell::Session - evaluate Perl line by line and keep the state

Peta::Shell::Complete

what Tab offers in ipps

SYNOPSIS

```
use Peta::Shell::Complete;
```

(continues on next page)

(continued from previous page)

```
my $complete = Peta::Shell::Complete->new(session => $session);

my ($from, @words) = $complete->complete('my $n = $cou', 12);
# $from is 8, @words is ('$count', '$country'):
# each word replaces the line's text from offset 8 to the cursor
```

DESCRIPTION

Given a line and the position of the cursor in it, this module says what the word in front of the cursor can become. It looks only at what exists: the variables of a *Peta::Shell::Session*, the packages and subs of the interpreter, the modules on the search path, the files on disk. It runs no code of the line, so it cannot complete through the result of a call.

What is offered depends on what stands in front of the cursor:

- `$na`, `@na`, `%na`

  The variables of the session and of the current package. After `$` an array is offered as `$name[` and a hash as `$name{`; after `@` a hash is offered as `@name{`, the slice. `$Some::Package::na` offers the variables of that package.

- `$hash{ke`, `$ref->{ke`, `$data{list}[0]{ke`

  The keys of that hash, with the closing brace. Only keys that can be written without quotes are offered.

- `$object->me`, `Class->me`

  The methods of the object's or the named class, inherited ones included. Those that start with an underscore are offered only after an underscore.

- `use Mo`, `no Mo`, `require Mo`

  The modules that are built into pperl or on the search path, one level at a time: `File::Te` offers `File::Temp`, and a namespace that has modules below it as `File::Spec::`.

- `"dir/fi` (inside a string)

  The files of that directory. A directory gets its `/`.

- `! command fi`

  The files of that directory, for the last word of a line that is handed to the shell.

- `na` (a plain word)

  The subs of the current package, Perl's functions and keywords, the loaded packages, and at the start of a line the macros. `Some::Package::na` offers the subs and packages below it.

- `:co`

  The commands of ipps, and after some of them what they take: a setting after `:set`, a file after `:load` and `:save`, a module after `:which` and `:install`, a sub after `:source`.

METHODS

new

```
my $complete = Peta::Shell::Complete->new(%options);
```

Options:

- session

  The *Peta::Shell::Session* whose variables and current package are offered. Without one, only what does not need it is offered, in the package `main`.

- commands

  A code reference that returns the names of the commands, without the colon.

- command

  A code reference that gets a command as typed, maybe shortened, and returns its full name or nothing.

- settings

  A code reference that returns the names of the settings.

- macros

  A code reference that returns the names of the macros.

complete

```
my ($from, @words) = $complete->complete($line, $cursor);
```

Returns the offset in `$line` from which the text up to `$cursor` is replaced, and the words it can be replaced with, sorted, each once. Returns the empty list when there is nothing to offer. `$cursor` defaults to the end of the line.

Peta::Shell::Printer

show a Perl value as bounded, readable text

SYNOPSIS

```
use Peta::Shell::Printer;

my $printer = Peta::Shell::Printer->new(width => 72);

print $printer->format(42), "\n";                  # 42
print $printer->format('$5 each'), "\n";           # '$5 each'
print $printer->format("a\tb"), "\n";              # "a\tb"
print $printer->format(1, 2, 3), "\n";             # (1, 2, 3)
print $printer->format({ b => [1, 2], a => undef }), "\n";
                                                   # { a => undef, b
 ↪=> [1, 2] }
```

DESCRIPTION

`format` turns values into text.

- A number is shown bare, a string quoted, `undef` as `undef`, a boolean as `true` or `false`. Whether a scalar is a number or a string is what it was created as: `42` and `'42'` look different. A float is shown with as many digits as it takes to read it back as the same number: `0.1 + 0.2` is `0.30000000000000004`.
- A string is quoted so that it needs as few escapes as it allows; what is shown is Perl that evaluates to the string.

```
text                 'text'
$x and @y            '$x and @y'
say "hi"             'say "hi"'
it's "quoted"        q{it's "quoted"}
C:\dir\file          'C:\dir\file'
a tab inside         "a\tb"
a tab and a "        qq{a\t"b"}
```

  When nothing in it has to be written as an escape sequence it is in a literal form, where `$`, `@` and `"` are characters like any other: `'...'`, or `q{...}` when it has a single quote, with the first of `{}`, `()`, `[]`, `<>` that does not occur in it. When it has a character that cannot be shown as itself it is in an interpolating form, `"..."` or `qq{...}` by the same choice. Shown as a number, never as themselves, are control characters, the bytes above 127 of a byte string, and in a character string whatever has no shape of its own or changes how its neighbours are laid out: a zero-width space, a direction override, a no-break space.
- An array is `[1, 2, 3]`, a hash `{ key => value }` with its keys sorted. Both stay on one line when they fit the width and are laid out one element per line when they do not.
- An object is its class in front of its contents, `Point { x => 1 }`, or its class and its string when it overloads stringification: `URI("http://example.com/")`.
- A code reference is `\&main::name` when the sub has a name and `sub { ... }` when it has none. A compiled pattern is `qr/.../`, a glob `\*main::STDOUT`, a reference to a scalar `\"text"`.
- Several values are a list in parentheses. No value at all is the empty string.

Everything is bounded: a container shows its first `max_items` elements and says how many more there are, nesting stops at `max_depth`, a string is cut at `max_string` characters with its full length noted. A structure that contains itself is shown once and marked where it repeats.

METHODS

new

```
my $printer = Peta::Shell::Printer->new(%options);
```

- width

  Columns available; a container that fits is printed on one line. Default 80.

- max_items

  Elements of one array or hash to show. Default 24. 0 means all.

- max_depth

  Levels of nesting to show. Default 6. 0 means all.

- max_string

  Characters of one string to show. Default 240. 0 means all.

- indent

  Spaces per level in the multi-line layout. Default 2.

format

```
my $text = $printer->format(@values);
```

The values as text, without a trailing newline.

Peta::Shell::Session

evaluate Perl line by line and keep the state

SYNOPSIS

```
use Peta::Shell::Session;

my $session = Peta::Shell::Session->new;

my $r = $session->eval('my @list = (1, 2, 3); scalar @list');
print "$r->{status}: @{ $r->{values} }\n";      # ok: 3

$r = $session->eval('push @list, 4; "@list"');  # @list is still
↪there
print "@{ $r->{values} }\n";                    # 1 2 3 4

$r = $session->eval('for my $i (@list) {');
print $r->{status}, "\n";                       # incomplete
```

DESCRIPTION

A session evaluates pieces of Perl one after another as if they were consecutive lines of one program. What a line leaves behind is there for every later line:

- its `my`, `state` and `our` variables, when declared at the line's top level;
- its pragmas: `use strict`, `use warnings`, `use feature`, `use v5.36` and any other lexical pragma;
- the current package;
- subs, packages and loaded modules, as in any program.

A variable declared inside a block of the line (`for my $i (...)`, `if (...) { my $t }`) belongs to that block and is gone with it, as in a program.

Each line is compiled once and run at once. A line that does not compile changes nothing. A line that dies keeps what it had declared and assigned before it died.

How a `my` variable survives

The variable itself is kept, not a copy: after the line that declares it, later lines reach it under its name as a package variable of a hidden package, bound with `our` when the line is compiled. So a closure and a named sub see every later assignment, `\$x` is the same reference in every line, and a second `my $x` in a later line makes a new variable while subs compiled before keep the old one.

Two things follow from "package variable". `local $x` works on it in a later line, which a true lexical refuses. And the name is resolved when a line is compiled, so text compiled later at run time (a string `eval`) sees the variable, but a symbolic reference by its plain name does not.

METHODS

new

```
my $session = Peta::Shell::Session->new(%options);
```

Options:

- name

  The name lines are reported under. Line 12 of a session named `ipps` is the file `(ipps:12)` in messages. Default `ipps`.

- package

  The package the first line is compiled in. Default `main`.

- preamble

  Perl evaluated before the first line; it sets the pragmas the session starts with and is not counted as a line. Default `use strict qw(refs subs); use warnings;`. Pass an empty string for none.

Dies if the preamble does not compile or dies.

eval

```
my $result = $session->eval($text);
```

Passes `$text` through the preprocessors, compiles it and runs it in list context. `$text` may hold several lines. Returns a hash reference:

- status

  ok, `error`, `incomplete` or `empty`.

`incomplete` means the text ends inside a construct: an open bracket, a string or here-document without its end, an expression that expects more. Nothing was run. The caller reads more input and calls `eval` again with the whole text.

`empty` means there was nothing but blank space, before or after the preprocessors.

- values

  Array reference: what the last statement returned, in list context. Empty unless the status is `ok`.

- error

  For `error`: the exception, as it was thrown (a string or an object).

- phase

  For `error`: `preprocess`, `compile` or `run`.

- warnings

  Array reference of the warnings the line raised, compiling or running, unless a line has installed its own `$SIG{__WARN__}`, which then gets them.

- line

  The line number this text got. Only `ok` and `error` texts are numbered.

- text

  The text after the preprocessors.

- wall, cpu

  Seconds of wall-clock and of processor time, compiling and running.

variables

```
my @names = $session->variables;
```

The names of the session's variables with their sigils, sorted.

variable

```
my $ref = $session->variable('@list');
```

A reference to the variable itself, or `undef` when the session has none of that name.

bind

```
$session->bind('$ans', \$value);
```

Makes the referenced variable the session's variable of that name, as if a line had declared it. The name is a sigil ($, @ or %) and an identifier; the reference must be of the matching kind.

package

The package the next line is compiled in.

line

The number of the last numbered line; 0 before the first.

name

The session's name.

add_preprocessor

```
$session->add_preprocessor(sub { my ($text, $session) = @_; ...;
↪$text });
```

Appends a preprocessor. Before a text is compiled it passes through the preprocessors in the order they were added; each gets the text and the session and returns the text to go on with. Returning an empty string ends the line there (status `empty`). An exception makes the line an `error` of phase `preprocess`.

This is where macros, aliases and commands of the shell live: they turn what was typed into Perl.

remove_preprocessor

```
$session->remove_preprocessor($code);
```

Removes that preprocessor. Returns true if it was there.

preprocessors

The list of preprocessors, in order.

LIMITS

- Text that turns out to be incomplete has still been compiled as far as it went, so a `use` or `BEGIN` block in it has already run and runs again when the completed text is compiled.
- `local` at a line's top level lasts to the end of that line.
- The match variables (`$1`, `$&`) are scoped to the line.
- A lexical sub (`my sub name`) is not kept.

Peta::WebGPU

`Peta::WebGPU` - compute and draw on the graphics card.

WebGPU is the modern successor to OpenGL: a portable API that reaches Vulkan, Metal and Direct3D 12 underneath, so the same program runs on whichever of those the machine has. `Peta::WebGPU` is pperl's binding to it, built on the wgpu implementation.

```
use Peta::WebGPU;

my $gpu = Peta::WebGPU::Device->new;
printf "%s on %s\n", $gpu->info->{name}, $gpu->info->{backend};

my $shader = $gpu->shader(<<'WGSL');
@group(0) @binding(0) var<storage, read_write> data: array<u32>;
@compute @workgroup_size(1)
fn main(@builtin(global_invocation_id) id: vec3<u32>) {
    data[id.x] = data[id.x] * 2u;
}
WGSL

my $buf = $gpu->storage_buffer(pack 'L*', 1 .. 8);
$gpu->compute($shader, 'main', [$buf], 8);
print join(',', unpack 'L*', $buf->read), "\n";   # 2,4,6,8,10,12,
 ↪14,16
```

This build may not have it

The GPU backend is large, so it is a build option that is off by default. `Peta::WebGPU::have_gpu()` answers whether this pperl has it AND found a usable adapter, without dying, so a program can fall back to doing the work on the CPU:

```
if (Peta::WebGPU::have_gpu()) { ... } else { ... }
```

A pperl built without the option tells you so when you load the module, and names the flag to rebuild with.

Shaders are WGSL

The programs that run on the card are written in WGSL, WebGPU's own shading language - not GLSL. `$gpu->shader($source)` compiles one, and a syntax error in it raises a Perl exception with the compiler's message rather than killing the program.

Errors

Everything here reports failure as an exception, carrying the GPU driver's own text, so `eval` is how you trap one. A call that succeeds and has nothing else to give back returns the object it was called on, so `or die` behaves and calls chain:

```
$gpu->compute($shader, 'main', [$buf], 8)->poll;
```

fork

A device, and every buffer, shader, pipeline, surface and texture made from it, belongs to the process that created it. `fork` copies the objects but not the graphics driver's threads, so in the child they cannot be used: any method on one dies with

```
this Peta::WebGPU::Device was created before fork; open a
↪Peta::WebGPU::Device in the child process
```

The child may leave them alone and exit, or open a device of its own and work on that; the parent's objects are unaffected either way. One limit: where the parent has opened the `gl` backend (the default `all` includes it), the child's device comes from the other backends only, and asking for `backend => 'gl'` there fails - OpenGL's display connection is shared with the parent and cannot be opened twice. A worker pool is best served by opening its device inside each worker:

```
my $pid = fork // die "fork: $!";
if (!$pid) {
    my $gpu = Peta::WebGPU::Device->new;    # this process's own
    ...
    exit 0;
}
```

Once a process has asked for an adapter or opened a device, a child forked from it ends, after Perl's own `END` blocks and destructors, without running the exit handlers of the C libraries loaded before the graphics drivers: the drivers' own handlers wait for threads the child does not have. Buffered output is flushed first.

Peta::WebGPU::Device

Opening a device is the first thing any GPU program does. It picks an adapter, opens it, and gives you the one object everything else hangs off:

```
my $gpu = Peta::WebGPU::Device->new;                          # let it
↪choose
my $gpu = Peta::WebGPU::Device->new(power => 'high-performance');
my $gpu = Peta::WebGPU::Device->new(backend => 'vulkan');
```

`power` is `high-performance` (prefer the discrete card) or `low-power` (prefer the integrated one, which saves battery and is often quicker to start). `backend` restricts the choice to one of `vulkan`, `metal`, `dx12`, `gl` or `all`. Opening fails with an exception if the machine has no usable adapter.

On `gl`, what the device can do depends on the OpenGL context the driver hands out. Drawing always works. Compute shaders need OpenGL 4.3 or OpenGL ES 3.1; on an older context (the NVIDIA driver gives 3.3) `compute` and `pipeline` die saying so, and `vulkan` is the backend to use.

- `$gpu->info` - a hash reference describing the adapter actually opened: `name`, `backend`, `device_type`, `driver`, `vendor`, `device`.
- `$gpu->limits` - what this device can do: `max_buffer_size`, `max_compute_workgroups_per_dimension` and the other ceilings you have to stay under.

These are the real hardware limits, not a portable minimum, so a program that runs on one machine can still ask for more than another machine offers.

- `$gpu->shader($wgsl)` - compile a shader.
- `$gpu->storage_buffer($data_or_size)` - a buffer a shader can read and write, and that you can read back.
- `$gpu->uniform_buffer($data_or_size)` - a small read-only buffer of parameters.
- `$gpu->pipeline($shader, $entry)` - prepare a shader entry point to run. Worth keeping when you dispatch the same shader repeatedly.
- `$gpu->compute($shader, $entry, \@buffers, $x, $y, $z)` - build the pipeline, bind and dispatch in one call.
- `$gpu->run($pipeline, \@buffers, $x, $y, $z)` - dispatch a pipeline you already have.
- `$gpu->run($binding, $x, $y, $z)` - dispatch a pipeline whose buffers are already bound (`$pipeline->bind`, below).
- `$gpu->begin`, `$gpu->commit` and `$gpu->batch(sub { ... })` - send several dispatches to the card at once; see "Many dispatches".
- `$gpu->features` - the optional features this device has, as a list of names: `shader-f16` (a shader may say `enable f16;`) and `timestamp-query`. A device has each one its adapter offers. `$gpu->has_feature($name)` is the yes or no for one of them.
- `$gpu->poll` - wait for everything submitted so far to finish.
- `$gpu->memory` - `{ allocated => ..., reserved => ... }`: the bytes this device's buffers and textures occupy, and the bytes it holds from the driver to put them in. `undef` on a backend that keeps no such account (Vulkan and DirectX 12 do).

`Device->new(memory_limit => 90)` makes the device refuse a buffer that would take it past 90 percent of the memory the driver allows this process. The refusal is an `Out of Memory` exception from `storage_buffer`, which a program can catch; without the option the driver decides, and may take the device with it.

The buffers are bound in the order given, as `@binding(0)`, `@binding(1)` and so on within `@group(0)`.

Many dispatches

Each `run` on its own is a round trip to the driver, and binding a list of buffers is work of its own. A program that dispatches the same few shaders over and over - a simulation step, a training step - saves both:

```
my $step = $pipeline->bind([$state, $params]);   # once

$gpu->batch(sub {                                # every step
    $gpu->run($step, $groups);
    $params->write(pack 'f', $rate);
```

(continues on next page)

(continued from previous page)

```
    $gpu->run($step, $groups);
});
```

`$pipeline->bind(\@buffers)` returns a `Peta::WebGPU::Binding`: the pipeline with those buffers bound, which keeps the buffers alive and can be run any number of times.

`$gpu->batch($code)` records every `run` the sub makes and sends them to the card together when it returns. `$gpu->begin` and `$gpu->commit` are the same without a sub. Inside a batch:

- a mistake in a `run` is reported when the batch is sent, by `commit` or `batch`, not by the `run` itself; for a wrong buffer list the message says which run of the batch it was;
- `$buf->write` takes effect in order: dispatches before it see the old contents, dispatches after it the new;
- `$buf->read` sends what is recorded so far, reads, and the batch goes on;
- a sub that dies leaves nothing sent, and its exception reaches the caller of `batch`.

Batches do not nest.

`$gpu->timed($code)` is `batch` with a stopwatch on the card: it returns the nanoseconds the card spent on the batch's dispatches, which is what a shader costs, without the time the program spent getting it there. It waits for the card to finish, and dies on a device without `timestamp-query`.

```
my $ns = $gpu->timed(sub { $gpu->run($step, $groups) });
printf "%.2f ms on the card\n", $ns / 1e6;
```

`Peta::WebGPU::Buffer`

- `$buf->size` - in bytes.
- `$buf->read` - the contents, as a byte string. Unpack it exactly as the shader's type says: `L*` for `array<u32>`, `l*` for `array<i32>`, `f*` for `array<f32>`.
- `$buf->read($offset, $length)` - a part of the contents, both in bytes and both multiples of four; without `$length`, from `$offset` to the end. Reading the one number you need costs the card far less than reading the buffer it is in.
- `$buf->write($bytes, $offset)` - replace part of it. A PDL ndarray of type `float`, `long` or `ulong` in place of `$bytes` is written as it is, so a large buffer can be filled piece by piece.
- `$buf->destroy` - give the buffer's memory back to the card now. A buffer is freed when its last reference goes, but a binding made with it is such a reference; `destroy` does not wait for those. The buffer cannot be used afterwards, and a binding that has it dies when run.

A buffer's size is always rounded up to a multiple of four bytes, because that is the granularity the GPU copies at. Pack `L`, `l` or `f` and it never comes up; pack an odd number of bytes and `size` will be a little larger than what you passed, with the extra bytes zero. A `write` must be aligned the same way.

Reading is not free: it waits for the card to finish and copies the data back across the bus, so read once at the end rather than inside a loop.

Workgroups, not elements

`compute` and `run` take the number of WORKGROUPS to launch, not the number of data elements. A shader declaring `@workgroup_size(64)` processes 64 elements per workgroup, so 1000 elements needs `int(($n + 63) / 64)` workgroups - and the shader must check its own index against the array length, because the last workgroup runs past the end:

```
my $wgsl = <<'WGSL';
@group(0) @binding(0) var<storage, read_write> v: array<f32>;
@compute @workgroup_size(64)
fn scale(@builtin(global_invocation_id) id: vec3<u32>) {
    if (id.x >= arrayLength(&v)) { return; }
    v[id.x] = v[id.x] * 2.0;
}
WGSL
my $buf = $gpu->storage_buffer(pack 'f*', @data);
$gpu->compute($gpu->shader($wgsl), 'scale', [$buf], int((@data + 63)
 ↪/ 64));
```

Drawing

The other half of a graphics card. A program that draws does the same three things every frame: get somewhere to draw, record what to draw there, and hand it over.

```
use SDL2 qw(:all);
use Peta::WebGPU;

my $win  = SDL2::Window->new(title => 'triangle', w => 800, h =>
 ↪600);
my $gpu  = Peta::WebGPU::Device->new(window => $win);
my $surf = $gpu->surface;

my $pipe = $gpu->render_pipeline($gpu->shader($WGSL),
    vertex => 'vs', fragment => 'fs', format => $surf->format,
    vertex_buffers => [ { attributes => ['float32x2', 'float32x3'] }
 ↪],
);
my $verts = $gpu->vertex_buffer(pack 'f*', @triangle);

while (running()) {
    my $frame = $surf->next_frame or next;
    $gpu->render_pass($frame, clear => [0.1, 0.2, 0.3, 1])
        ->pipeline($pipe)->vertices($verts)->draw(3)->end;
    $frame->present;
}
```

Every recording call returns the pass, so a frame reads as one sentence. `end` is what

hands the work to the card; nothing is drawn before it.

Drawing without a window

A render pass does not care whether its target is on screen. Give it a texture instead of a frame and the picture comes back as bytes:

```
my $gpu = Peta::WebGPU::Device->new;           # no window at all
my $img = $gpu->texture(width => 800, height => 600);
$gpu->render_pass($img, clear => [0, 0, 0, 1])
    ->pipeline($pipe)->vertices($verts)->draw(3)->end;
my $rgba = $img->read;                          # 800*600*4 bytes
```

This needs no display, no window manager and no compositor, so it is how a program renders on a server - and how this module's own tests check what was drawn, pixel by pixel.

What a window has to be

`Peta::WebGPU` opens no windows. It asks a window object two questions, and anything that answers them can be drawn on:

- `$win->wm_info` - a hash reference with `driver` (`x11` or `wayland`), `display`, and `window` or `surface`.
- `$win->size` - the list `($width, $height)`.

`SDL2::Window` answers both, which is where a window normally comes from here. A window system this build cannot make a surface on is refused by name rather than failing later.

The window belongs at `Device->new`, not at `surface`, because the adapter has to be one that can actually present to that window - a machine with two cards can otherwise open the one that does not drive the screen.

`Peta::WebGPU::Surface`

`$gpu->surface` - the window's swap chain, configured and ready. `$gpu->surface($other_window)` gives a second window its own.

- `$surf->format` - the pixel format the compositor wants, as a WebGPU name such as `bgra8unorm-srgb`. Pass it to `render_pipeline` so the two agree; they must.
- `$surf->size` - the list `($width, $height)`.
- `$surf->configure($width, $height)` - after the window was resized.
- `$surf->next_frame` - the image to draw this frame into, or **undef** when the window is hidden or the compositor is busy. That is not an error: skip the frame and come back.

`present_mode` chooses how frames reach the screen: `fifo` waits for the display and never tears (the safe default), `immediate` does not wait, `mailbox` keeps only the newest. A mode the window does not offer is refused, with the ones it does offer in the message.

A swap chain lives on the window system's connection. `SDL_Quit`, and `SDL_QuitSubSystem` for video, close that connection, so they take the swap chains of every surface first, with the frames not yet presented. The objects remain: `size` and `format` still answer, while `configure`, `next_frame`, `present` and a render pass on such a frame die saying that the window system was shut down. The device is not affected and goes on working off screen. A window object that is not SDL2's has to outlive its surfaces by itself.

`Peta::WebGPU::Frame`

One image out of the swap chain. It is a render target exactly as a texture is, and `$frame->present` puts it on the screen. A frame that goes out of scope unpresented is discarded, which is the right thing for a frame abandoned half-drawn. Presenting twice is refused.

End the pass before presenting: end is what submits the drawing.

`Peta::WebGPU::Texture`

`$gpu->texture(width => .., height => .., format => ..)` - somewhere to draw that is not the screen. The format defaults to `rgba8unorm`, the same default `render_pipeline` has, so the two agree unless you say otherwise.

- `$tex->size` - the list `($width, $height)`.
- `$tex->format` - its format, by name.
- `$tex->read` - the pixels, tightly packed, top row first. Four bytes per pixel for the 8-bit-per-channel formats, so `unpack 'C*'` gives R, G, B, A in that order.

Reading waits for the card and copies the image back across the bus, exactly as reading a buffer does. It belongs at the end, not in a loop.

`$gpu->depth_buffer($width, $height)` makes the other kind of target: a depth texture, which is what stops a far triangle painting over a near one. Pass it to `render_pass` as `depth => $d` and build the pipeline with `depth => 1`; both halves are needed, and wgpu says so if only one is there.

Render pipelines

`$gpu->render_pipeline($shader, %options)` compiles one shader's vertex and fragment entry points into everything the card needs to draw. It is worth keeping: build it once, use it every frame.

- `vertex` / `fragment` - the entry point names in the WGSL. With one of each in the shader they can be left out.
- `format` - the format of what is being drawn into. Default `rgba8unorm`; for a window it must be `$surf->format`.
- `vertex_buffers` - the shape of the vertex data; see below.
- `topology` - `triangle-list` (the default), `triangle-strip`, `line-list`, `line-strip` or `point-list`.

- `cull` - none (the default), `back` or `front`, and `front_face` (`ccw`, the default, or `cw`), which decide which side of a triangle is its front and whether the other one is drawn at all.
- `depth => 1` - take part in depth testing.
- `blend => 1` - blend with what is already there, by alpha.

Describing the vertices

Each entry of `vertex_buffers` describes one buffer, as a list of `attributes`. The short form is a list of format names, which are laid end to end at `@location(0)`, `@location(1)` and so on:

```
vertex_buffers => [ { attributes => ['float32x3', 'float32x3'] } ],
```

That is the layout of a WGSL function taking consecutive locations, which is what almost every shader has. Anything left out is computed: offsets follow each other, locations count up, and the stride is where the last attribute ends. A hash gives the long form for the cases that need it:

```
vertex_buffers => [ {
    stride     => 32,
    attributes => [ { format => 'float32x3', location => 0, offset
↪=> 0 },
                    { format => 'float32x2', location => 5, offset
↪=> 20 } ],
} ],
```

Formats are WebGPU's own names - `float32`, `float32x2`, `float32x3`, `float32x4`, and the `uint32`, `sint32`, `unorm8x4` and `float16x2` families. A name that is not one of them is refused with the whole list, because a typo here otherwise shows up as geometry that is nearly right.

`Peta::WebGPU::Pass`

`$gpu->render_pass($target, %options)` starts recording. The target is a `Frame` or a `Texture`.

- `clear => [$r, $g, $b, $a]` - the colour to start from, each component 0 to 1. `clear => 0` keeps what is already there, which is how a second pass draws on top of the first. Left out altogether, it clears to black.
- `depth => $depth_texture` - test against depth while drawing.

Then, in order:

- `$pass->pipeline($pipeline)` - what to draw with. First, always: the bind groups take their layout from it.
- `$pass->bind(\@buffers, [$group])` - the uniform and storage buffers of `@group(0)`, in binding order, exactly as `compute` takes them.
- `$pass->vertices($buffer, [$slot])` - the vertex data.

- `$pass->indices($buffer, ['uint16'|'uint32'])` - the index data. 32-bit by default, because a mesh with more than 65536 vertices is ordinary.
- `$pass->draw($vertex_count, [$instances])` or `$pass->draw_indexed($index_count, [$instances])`.
- `$pass->end` - stop recording and hand the work to the card.

Every one of them returns the pass, so they chain. A pass that is dropped without end throws its recording away rather than submitting half a frame.

`$gpu->vertex_buffer($data)` and `$gpu->index_buffer($data)` make the buffers, and take the same arguments `storage_buffer` does - a byte string, a size, or a PDL ndarray of a 32-bit type.

Functions

Availability

have_gpu

True when this pperl can actually reach a graphics card: the build has the GPU option AND an adapter was found.

Adapters

adapters

Every graphics adapter this machine offers, as a list of hash references.

have_gpu

True when this pperl can actually reach a graphics card: the build has the GPU option AND an adapter was found.

It answers rather than dying, so a program can choose a CPU fallback:

```
my $sum = Peta::WebGPU::have_gpu() ? gpu_sum(@data) : cpu_
↪sum(@data);
```

A machine with no GPU driver at all, or a headless container, will say false here even on a build that has the option compiled in.

adapters

Every graphics adapter this machine offers, as a list of hash references.

Each carries `name`, `backend` (`"Vulkan"`, `"Metal"`, `"Dx12"`, `"Gl"`), `device_type` (`"DiscreteGpu"`, `"IntegratedGpu"`, `"Cpu"`, `"Other"`) and `driver`. A machine commonly has more than one - an integrated chip and a discrete card, or the same card through two different backends:

```
for my $a (Peta::WebGPU::adapters()) {
    printf "%-14s %-14s %s\n", $a->{backend}, $a->{device_type}, $a-
↪>{name};
}
```

Which one `Peta::WebGPU::Device->new` picks is decided by its `power` option, not by this list; the list is for reporting and for deciding whether the machine has the card you need.

Peta::XS

Peta::XS - the interpreter's capability layer for pure Perl.

Unlike pperl's other native modules, this is not a CPAN module reimplemented: it is pperl's own versioned, semantic API that lets third-party pure Perl do what internals-XS modules do on perl5.

```
use Peta::XS qw(:facts :magic);

my $x = 1;
print "refcount ", refcount(\$x), "\n";          # 2: $x, and the␣
↪reference passed
my $reads = 0;
my $w = wizard(
    get  => sub { $reads++ },
    set  => sub { my ($ref, $data) = @_; $data->{sets}++ },
    data => sub { return { sets => 0 } },
);
cast(\$x, $w);                                   # attach the hooks␣
↪to $x
my $y = $x + $x;                                 # two reads
$x = 5;                                          # one write
print "reads $reads, sets ", getdata(\$x, $w)->{sets}, "\n";   #␣
↪reads 2, sets 1
print join(",", map { $_->{type} } magic(\$x)), "\n";          # ~
dispell(\$x, $w);                                # detach
```

Capabilities are imported by tag: `:facts` (`refcount`, `value_slots`, `magic`) and `:magic` (`wizard`, `cast`, `dispell`, `getdata`). `use Peta::XS;` alone imports nothing.

CONTRACT (v1.0)

Capabilities: `facts`, `magic`. Evolution is additive within a major version; a capability is never silently redefined.

1. Facts and verbs, never layout. Every exposed datum is defined by observable semantics (never flag bits, offsets, addresses or struct names) and survives any internal refactor.
2. Opaque handles. Anything referring to an interpreter entity (a wizard, a magic attachment) is a runtime-owned opaque value - no arithmetic, no raw pointers as IVs.

3. The boundary speaks Perl. Callbacks are coderefs; data crosses as ordinary Perl values. Consumers are pure Perl and cannot be binary-broken by any pperl release. The loudest possible failure is a compile-time version/capability refusal in `import`.

A variable carrying Peta::XS magic is never JIT-compiled, so its callbacks fire on every access, exactly as the interpreter would fire them. Callbacks run with exactly the timing of perl's tied variables.

Functions

Other Functions

`refcount`

Reference count of the thing `$ref` points at.

`value_slots`

Which value representations the runtime currently considers valid for the scalar `$ref` points at.

`magic`

Inventory of the magic attached to the variable `$ref` points at.

`wizard`

Create a wizard: an opaque bundle of magic callbacks.

`cast`

Attach a wizard to the variable `$ref` points at.

`dispell`

Detach a wizard from the variable `$ref` points at.

`getdata`

Retrieve the private data this wizard's attachment carries.

`refcount`

Reference count of the thing `$ref` points at.

Synopsis

```
use Peta::XS ':facts';
my $x;
my $n = refcount(\$x);   # 2 - see the observer effect below
```

What you get back

An integer: how many references the runtime currently holds to the referent, at call time. Works on any referent - scalars, arrays, hashes, subs, aggregate elements. Croaks if the argument is not a reference; never dies otherwise.

Observer effect

The reference you pass in contributes one count of its own for the duration of the call, exactly as with `Devel::Refcount::refcount`. It is deliberately NOT subtracted: a lexical with no other references reports 2 (its pad slot plus your temporary reference).

Examples

```
my $x = 42;
say refcount(\$x);        # 2: pad slot + the \$x you just made

my $r = \$x;
say refcount($r);         # 2: pad slot + $r (no extra temporary)
my $r2 = \$x;
say refcount($r);         # 3: pad slot + $r + $r2

my @queue = ($r);
say refcount($r);         # 4: the array element counts too

# leak hunting: a cache entry still pinning an object
my $obj = SomeClass->new;
$cache{key} = $obj;
say refcount(\$obj);      # 3 - the cache holds one; forget it and
delete $cache{key};       # the count drops back to 2
```

See also

- `value_slots` - which representations the value holds.
- `magic` - what is attached to the variable.
- `Scalar::Util::refaddr` / `builtin::refaddr` - identity, not count.

value_slots

Which value representations the runtime currently considers valid for the scalar `$ref` points at.

Synopsis

```
use Peta::XS ':facts';
my @slots = value_slots(\42);      # ('int')
my @slots = value_slots(\3.14);    # ('num')
my @slots = value_slots(\'hi');    # ('str')
```

(continues on next page)

(continued from previous page)

```
my @slots = value_slots(\\1);      # ('ref')
my @slots = value_slots(\undef);   # ()
```

What you get back

In list context, a subset of `qw(int num str ref)` - the representations the runtime currently considers valid for this value. In scalar context, how many of them there are. A scalar can hold several at once: after a numeric string is used in numeric context it carries both its string and its numeric form, and a `Scalar::Util::dualvar` carries `int` and `str` by construction.

Contract

This is an observation of optimization state, not of meaning: the same program may legitimately report different slot sets across runtimes and versions. Branch on it for diagnostics and teaching, never for program semantics.

Examples

```
# watch a scalar accumulate representations
my $port = '8080';
say "@{[ value_slots(\$port) ]}";   # str
my $next = $port + 1;               # numeric use caches the number
say "@{[ value_slots(\$port) ]}";   # int str

# dualvars hold two truths at once ($! is the classic case)
use Scalar::Util qw(dualvar);
my $dv = dualvar(5, 'five');
say "@{[ value_slots(\$dv) ]}";     # int str

# a classroom one-liner: show why "10" == 10 costs a conversion
# only the first time
```

See also

- `refcount` - how many references point here.
- `magic` - what is attached to the variable.

magic

Inventory of the magic attached to the variable `$ref` points at.

Synopsis

```
use Peta::XS ':facts';
tie my %h, 'Some::Tie';
```

(continues on next page)

(continued from previous page)

```
for my $m (magic(\%h)) {
    printf "type=%s get=%d set=%d\n", $m->{type}, $m->{get}, $m->
↪{set};
}
```

What you get back

One hashref per attachment, in chain order (most recently attached first, mirroring perl's prepend discipline): `type` is perl's public one-character magic type (the `perlguts` vocabulary - 'P' tied array/hash, 'q' tied scalar, '~' extension magic as used by `cast`), `get`/`set` are booleans saying whether the attachment hooks reads/writes. In scalar context: the number of attachments. An unmagical variable yields an empty list / 0.

Examples

```
my $plain = 1;
say scalar magic(\$plain);          # 0

# is this hash really tied?
tie my %cfg, 'Config::Tied';
my ($m) = magic(\%cfg);
say "tied!" if $m && $m->{type} eq 'P';

# see your own wizards (and their hook shape) from the outside
use Peta::XS ':magic';
my $w = wizard(get => sub { });
my $x = 1;
cast(\$x, $w);
my @mg = magic(\$x);                # ({ type => '~', get => 1, set␣
↪=> 1 })

# debugging aid: why is every read of $slow slow?
printf "%s(get=%d,set=%d)\n", @{$_}{qw(type get set)}
    for magic(\$slow);
```

See also

- `cast` / `dispell` - attach and detach '~' magic.
- `refcount`, `value_slots` - the other facts.
- `perldoc perlguts` - the magic type vocabulary.

wizard

Create a wizard: an opaque bundle of magic callbacks.

Synopsis

```
use Peta::XS ':magic';
my $w = wizard(
    get  => sub { my ($ref, $data) = @_; ... },
    set  => sub { my ($ref, $data) = @_; ... },
    free => sub { my ($ref, $data) = @_; ... },
    data => sub { my ($ref, @args) = @_; return { count => 0 } },
);
```

What you get back

An opaque handle. Hold it, pass it to `cast`, `dispell` and `getdata` - nothing else. One wizard can be cast on any number of variables; each attachment gets its own private data.

Contract

All keys are optional but at least one of `get`/`set`/`free` is required; every value must be a code reference (anything else croaks, as does an unknown key). `get` is called before the variable's value is used, `set` after it changed, `free` while it is being destroyed; each receives `(\$var, $data)` and its return value is ignored (no value substitution in v1). `data` is called once per `cast` to build that attachment's private data. A callback that dies propagates exactly like a dying tied-variable FETCH/STORE. A variable carrying wizard magic is never JIT-compiled, so callbacks fire on every single access, compiled neighbors notwithstanding.

Examples

```
# a watchpoint: who is touching $config?
my $watch = wizard(
    get => sub { warn "read at @{[ join ':', caller ]}\n" },
    set => sub { warn "now: ${$_[0]}\n" },
);

# lazy initialization: get may write the variable before the
# pending read uses it (magic is suspended inside callbacks, so
# this does not recurse)
my $lazy = wizard(get => sub { ${$_[0]} //= expensive() });
my $val;
cast(\$val, $lazy);
say $val;                # expensive() runs here, exactly once
say $val;                # cached now

# per-attachment counters via a data callback
my $counted = wizard(
    data => sub { { reads => 0, writes => 0 } },
    get  => sub { $_[1]{reads}++ },
    set  => sub { $_[1]{writes}++ },
);
```

See also

- `cast` - attach the wizard to a variable.
- `getdata` - read an attachment's private data back.
- `magic` - the facts-side view of an attachment.

cast

Attach a wizard to the variable `$ref` points at.

Synopsis

```
use Peta::XS ':magic';
my $x;
cast(\$x, $w);              # true
cast(\$x, $w);              # false - same wizard twice is a no-op
cast(\$x, $other, 1, 2);    # @args reach $other's data callback
```

What you get back

True when the wizard was attached, false when this wizard was already on the variable (casting twice is a no-op and calls no callbacks). The cast itself fires nothing - the first `get`/`set` comes from the first read/write after it.

Contract

If the wizard has a `data` callback it is called once, here, as `$data_cb->(\$referent, @args)`, and its return value becomes the attachment's private data (see `getdata`). Multiple distinct wizards may be cast on one variable; they fire most recently cast first. Read-only variables can be cast (attaching magic does not write the value - writes still croak before `set` would fire). The attachment lives exactly as long as the variable or until `dispell`ed.

Examples

```
# any scalar works, including aggregate elements
my @row = (1, 2, 3);
cast(\$row[1], $watch);     # watch one array element
my %cfg = (mode => 'dev');
cast(\$cfg{mode}, $watch);  # watch one hash value

# cast args parameterize the attachment via the data callback
my $tagged = wizard(
    data => sub { my ($ref, $tag) = @_; $tag },
    set  => sub { warn "[$_[1]] changed\n" },
);
cast(\my $a, $tagged, 'alpha');
cast(\my $b, $tagged, 'beta');
```

(continues on next page)

(continued from previous page)

```
$a = 1;                     # warns "[alpha] changed"
$b = 2;                     # warns "[beta] changed"
```

See also

- `wizard` - build the handle and its callbacks.
- `dispell` - the inverse operation.
- `getdata` - fetch what the data callback returned.

dispell

Detach a wizard from the variable `$ref` points at.

Synopsis

```
use Peta::XS ':magic';
dispell(\$x, $w);   # true if $w was attached, false otherwise
```

What you get back

True if this wizard's attachment was found and removed, false if it was not attached. Idempotent: a second dispell of the same wizard returns false and changes nothing.

Contract

Removes exactly this wizard's attachment - other wizards on the same variable stay, and so does unrelated magic (a tie, for example). The attachment's private data is released. The `free` callback does NOT run: dispelling is detachment, not destruction; `free` fires only when the variable itself dies.

Examples

```
# scoped watching: attach, observe, detach
cast(\$config, $watch);
run_suspicious_code();
dispell(\$config, $watch);   # $config is an ordinary scalar again

# wizards detach independently
cast(\$x, $w1);
cast(\$x, $w2);
dispell(\$x, $w1);           # $w2 keeps firing
say scalar magic(\$x);       # 1
```

See also

- `cast` - the inverse operation.
- `magic` - check what is (still) attached.

getdata

Retrieve the private data this wizard's attachment carries.

Synopsis

```
use Peta::XS ':magic';
my $data = getdata(\$x, $w);   # whatever $w's data callback
 ↪returned
```

What you get back

The attachment's private data - the value the wizard's `data` callback returned at `cast` time - or undef when this wizard has no attachment on the variable or the wizard has no `data` callback. It is the same value the `get`/`set`/`free` callbacks receive as their second argument: if it is a reference, mutations made inside callbacks are visible here.

Examples

```
# per-variable access statistics, read back after the fact
my $counted = wizard(
    data => sub { { reads => 0, writes => 0 } },
    get  => sub { $_[1]{reads}++ },
    set  => sub { $_[1]{writes}++ },
);
my $x = 1;
cast(\$x, $counted);
my $v = $x; $v = $x;          # two reads
$x = 2;                       # one write
my $stats = getdata(\$x, $counted);
say "$stats->{reads}/$stats->{writes}";   # 2/1

# distinguishing "no attachment" from "no data"
say defined getdata(\$x, $counted) ? 'attached' : 'not here';
```

See also

- `cast` - where the data callback runs.
- `wizard` - declaring the data callback.

5.2.84 Pod

Namespace containing the following modules:

- `Pod::Checker` - check pod documents for syntax errors
- `Pod::Escapes` - for resolving Pod E<...> sequences
- `Pod::Functions` - Group Perl's functions a la perlfunc.pod
- `Pod::Html` - module to convert pod files to HTML
- `Pod::Man` - Convert POD data to formatted *roff input
- `Pod::ParseLink` - Parse an L<> formatting code in POD text
- `Pod::Perldoc` - Look up Perl documentation in Pod format.
- `Pod::Simple` - framework for parsing Pod
- `Pod::Text` - Convert POD data to formatted text
- `Pod::Usage` - extracts POD documentation and shows usage information

Pod::Checker

check pod documents for syntax errors

SYNOPSIS

```
use Pod::Checker;

$syntax_okay = podchecker($filepath, $outputpath, %options);

my $checker = Pod::Checker->new(%options);
$checker->parse_from_file($filepath, \*STDERR);
```

OPTIONS/ARGUMENTS

`$filepath` is the input POD to read and `$outputpath` is where to write POD syntax error messages. Either argument may be a scalar indicating a file-path, or else a reference to an open filehandle. If unspecified, the input-file it defaults to `\*STDIN`, and the output-file defaults to `\*STDERR`.

podchecker()

This function can take a hash of options:

- **-warnings** => *val*

  Turn warnings on/off. *val* is usually 1 for on, but higher values trigger additional warnings. See "Warnings".

- **-quiet** => *val*

  If `val` is true, do not print any errors/warnings.

DESCRIPTION

podchecker will perform syntax checking of Perl5 POD format documentation.

Curious/ambitious users are welcome to propose additional features they wish to see in **Pod::Checker** and **podchecker** and verify that the checks are consistent with perlpod.

The following checks are currently performed:

- Unknown '=xxxx' commands, unknown 'X<...>' interior-sequences, and unterminated interior sequences.
- Check for proper balancing of `=begin` and `=end`. The contents of such a block are generally ignored, i.e. no syntax checks are performed.
- Check for proper nesting and balancing of `=over`, `=item` and `=back`.
- Check for same nested interior-sequences (e.g. `L<...L<...>...>`).
- Check for malformed or non-existing entities `E<...>`.
- Check for correct syntax of hyperlinks `L<...>`. See perlpod for details.
- Check for unresolved document-internal links. This check may also reveal misspelled links that seem to be internal links but should be links to something else.

DIAGNOSTICS

Errors

- empty =headn

  A heading (`=head1` or `=head2`) without any text? That ain't no heading!

- =over on line *N* without closing =back
- You forgot a '=back' before '=head_N_'
- =over is the last thing in the document?!

  The `=over` command does not have a corresponding `=back` before the next heading (`=head1` or `=head2`) or the end of the file.

- '=item' outside of any '=over'
- =back without =over

  An `=item` or `=back` command has been found outside a `=over`/`=back` block.

- Can't have a 0 in =over *N*

  You need to indent a strictly positive number of spaces, not 0.

- =over should be: '=over' or '=over positive_number'

  Either have an argumentless =over, or have its argument a strictly positive number.

- =begin *TARGET* without matching =end *TARGET*

  A `=begin` command was found that has no matching =end command.

- =begin without a target?

  A `=begin` command was found that is not followed by the formatter specification.

- =end *TARGET* without matching =begin.

  A standalone =end command was found.

- '=end' without a target?

  '=end' directives need to have a target, just like =begin directives.

- '=end *TARGET*' is invalid.

  TARGET needs to be one word

- =end *CONTENT* doesn't match =begin *TARGET*

  CONTENT needs to match =begin's *TARGET*.

- =for without a target?

  There is no specification of the formatter after the `=for` command.

- unresolved internal link *NAME*

  The given link to *NAME* does not have a matching node in the current POD. This also happened when a single word node name is not enclosed in `""`.

- Unknown directive: *CMD*

  An invalid POD command has been found. Valid are `=head1`, `=head2`, `=head3`, `=head4`, `=over`, `=item`, `=back`, `=begin`, `=end`, `=for`, `=pod`, `=cut`

- Deleting unknown formatting code *SEQ*

  An invalid markup command has been encountered. Valid are: `B<>`, `C<>`, `E<>`, `F<>`, `I<>`, `L<>`, `S<>`, `X<>`, `Z<>`

- Unterminated *SEQ*<> sequence

  An unclosed formatting code

- An E<...> surrounding strange content

  The *STRING* found cannot be interpreted as a character entity.

- An empty E<>

- An empty `L<>`

- An empty X<>

  There needs to be content inside E, L, and X formatting codes.

- Spurious text after =pod / =cut

  The commands =pod and `=cut` do not take any arguments.

- =back doesn't take any parameters, but you said =back *ARGUMENT*

  The `=back` command does not take any arguments.

- =pod directives shouldn't be over one line long! Ignoring all *N* lines of content

  Self explanatory

- =cut found outside a pod block.

  A '=cut' directive found in the middle of non-POD

- Invalid =encoding syntax: *CONTENT*

  Syntax error in =encoding directive

Warnings

These may not necessarily cause trouble, but indicate mediocre style.

- nested commands *CMD*<...*CMD*<...>...>

  Two nested identical markup commands have been found. Generally this does not make sense.

- multiple occurrences (*N*) of link target *name*

  The POD file has some `=item` and/or `=head` commands that have the same text. Potential hyperlinks to such a text cannot be unique then. This warning is printed only with warning level greater than one.

- line containing nothing but whitespace in paragraph

  There is some whitespace on a seemingly empty line. POD is very sensitive to such things, so this is flagged. **vi** users switch on the **list** option to avoid this problem.

- =item has no contents

  There is a list `=item` that has no text contents. You probably want to delete empty items.

- You can't have =items (as at line *N*) unless the first thing after the =over is an =item

  A list introduced by `=over` starts with a text or verbatim paragraph, but continues with `=items`. Move the non-item paragraph out of the `=over`/`=back` block.

- Expected '=item *EXPECTED VALUE*'

- Expected '=item *'

- Possible =item type mismatch: '*x*' found leading a supposed definition =item

  A list started with e.g. a bullet-like `=item` and continued with a numbered one. This is obviously inconsistent. For most translators the type of the *first* `=item` determines the type of the list.

- You have '=item x' instead of the expected '=item *N*'

  Erroneous numbering of =item numbers; they need to ascend consecutively.

- Unknown E content in E<*CONTENT*>

  A character entity was found that does not belong to the standard ISO set or the POD specials `verbar` and `sol`. *Currently, this warning only appears if a character entity was found that does not have a Unicode character. This should be fixed to adhere to the original warning.*

- empty =over/=back block

  The list opened with `=over` does not contain anything.

- empty section in previous paragraph

The previous section (introduced by a =head command) does not contain any valid content. This usually indicates that something is missing. Note: A =head1 followed immediately by =head2 does not trigger this warning.

- Verbatim paragraph in NAME section

  The NAME section (=head1 NAME) should consist of a single paragraph with the script/module name, followed by a dash `-' and a very short description of what the thing is good for.

- =head_n_ without preceding higher level

  For example if there is a =head2 in the POD file prior to a =head1.

- A non-empty Z<>

  The Z<> sequence is supposed to be empty. Caveat: this issue is detected in *Pod::Simple* and will be flagged as an *ERROR* by any client code; any contents of Z<...> will be disregarded, anyway.

Hyperlinks

There are some warnings with respect to malformed hyperlinks:

- ignoring leading/trailing whitespace in link

  There is whitespace at the beginning or the end of the contents of L<...>.

- alternative text/node '%s' contains non-escaped | or /

  The characters | and / are special in the L<...> context. Although the hyperlink parser does its best to determine which "/" is text and which is a delimiter in case of doubt, one ought to escape these literal characters like this:

```
/     E<sol>
|     E<verbar>
```

Note that the line number of the error/warning may refer to the line number of the start of the paragraph in which the error/warning exists, not the line number that the error/warning is on. This bug is present in errors/warnings related to formatting codes. *This should be fixed.*

RETURN VALUE

podchecker returns the number of POD syntax errors found or -1 if there were no POD commands at all found in the file.

EXAMPLES

See "SYNOPSIS"

SCRIPTS

The **podchecker** script that comes with this distribution is a lean wrapper around this module. See the online manual with

```
podchecker -help
podchecker -man
```

INTERFACE

While checking, this module collects document properties, e.g. the nodes for hyperlinks (`=headX`, `=item`) and index entries (`X<>`). POD translators can use this feature to syntax-check and get the nodes in a first pass before actually starting to convert. This is expensive in terms of execution time, but allows for very robust conversions.

Since v1.24 the **Pod::Checker** module uses only the **poderror** method to print errors and warnings. The summary output (e.g. "Pod syntax OK") has been dropped from the module and has been included in **podchecker** (the script). This allows users of **Pod::Checker** to control completely the output behavior. Users of **podchecker** (the script) get the well-known behavior.

v1.45 inherits from *Pod::Simple* as opposed to all previous versions inheriting from Pod::Parser. Do **not** use Pod::Simple's interface when using Pod::Checker unless it is documented somewhere on this page. I repeat, DO **NOT** USE POD::SIMPLE'S INTERFACE.

The following list documents the overrides to Pod::Simple, primarily to make Pod::Coverage happy:

- end_B
- end_C
- end_Document
- end_F
- end_I
- end_L
- end_Para
- end_S
- end_X
- end_fcode
- end_for
- end_head
- end_head1
- end_head2
- end_head3
- end_head4
- end_item

- end_item_bullet
- end_item_number
- end_item_text
- handle_pod_and_cut
- handle_text
- handle_whiteline
- hyperlink
- scream
- start_B
- start_C
- start_Data
- start_F
- start_I
- start_L
- start_Para
- start_S
- start_Verbatim
- start_X
- start_fcode
- start_for
- start_head
- start_head1
- start_head2
- start_head3
- start_head4
- start_item_bullet
- start_item_number
- start_item_text
- start_over
- start_over_block
- start_over_bullet
- start_over_empty
- start_over_number
- start_over_text

- whine
- `Pod::Checker->new( %options )`

  Return a reference to a new Pod::Checker object that inherits from Pod::Simple and is used for calling the required methods later. The following options are recognized:

  `-warnings => num` Print warnings if `num` is true. The higher the value of `num`, the more warnings are printed. Currently there are only levels 1 and 2.

  `-quiet => num` If `num` is true, do not print any errors/warnings. This is useful when Pod::Checker is used to munge POD code into plain text from within POD formatters.
- `$checker->poderror( @args )`
- `$checker->poderror( {%opts}, @args )`

  Internal method for printing errors and warnings. If no options are given, simply prints "@_". The following options are recognized and used to form the output:

  ```
  -msg
  ```

  A message to print prior to `@args`.

  ```
  -line
  ```

  The line number the error occurred in.

  ```
  -file
  ```

  The file (name) the error occurred in. Defaults to the name of the current file being processed.

  ```
  -severity
  ```

  The error level, should be 'WARNING' or 'ERROR'.
- `$checker->num_errors()`

  Set (if argument specified) and retrieve the number of errors found.
- `$checker->num_warnings()`

  Set (if argument specified) and retrieve the number of warnings found.
- `$checker->name()`

  Set (if argument specified) and retrieve the canonical name of POD as found in the `=head1 NAME` section.
- `$checker->node()`

  Add (if argument specified) and retrieve the nodes (as defined by `=headX` and `=item`) of the current POD. The nodes are returned in the order of their occurrence. They consist of plain text, each piece of whitespace is collapsed to a single blank.

- `$checker->idx()`

  Add (if argument specified) and retrieve the index entries (as defined by `X<>`) of the current POD. They consist of plain text, each piece of whitespace is collapsed to a single blank.

- `$checker->hyperlinks()`

  Retrieve an array containing the hyperlinks to things outside the current POD (as defined by `L<>`).

  Each is an instance of a class with the following methods:

- line()

  Returns the approximate line number in which the link was encountered

- type()

  Returns the type of the link; one of: `"url"` for things like `http://www.foo`, `"man"` for man pages, or `"pod"`.

- page()

  Returns the linked-to page or url.

- node()

  Returns the anchor or node within the linked-to page, or an empty string (`""`) if none appears in the link.

AUTHOR

Please report bugs using http://rt.cpan.org.

Brad Appleton bradapp@enteract.com (initial version), Marek Rouchal marekr@cpan.org, Marc Green marcgreen@cpan.org (port to Pod::Simple) Ricardo Signes rjbs@cpan.org (more porting to Pod::Simple) Karl Williamson khw@cpan.org (more porting to Pod::Simple)

Based on code for **Pod::Text::pod2text()** written by Tom Christiansen tchrist@mox.perl.com

Pod::Escapes

for resolving Pod E<...> sequences

SYNOPSIS

```
use Pod::Escapes qw(e2char);
...la la la, parsing POD, la la la...
$text = e2char($e_node->label);
unless(defined $text) {
  print "Unknown E sequence \"", $e_node->label, "\"!";
}
...else print/interpolate $text...
```

DESCRIPTION

This module provides things that are useful in decoding Pod E<...> sequences. Presumably, it should be used only by Pod parsers and/or formatters.

By default, Pod::Escapes exports none of its symbols. But you can request any of them to be exported. Either request them individually, as with `use Pod::Escapes qw(symbolname symbolname2...);`, or you can do `use Pod::Escapes qw(:ALL);` to get all exportable symbols.

GOODIES

- e2char($e_content)

  Given a name or number that could appear in a `E<name_or_num>` sequence, this returns the string that it stands for. For example, `e2char('sol')`, `e2char('47')`, `e2char('0x2F')`, and `e2char('057')` all return "/", because `E<sol>`, `E<47>`, `E<0x2f>`, and `E<057>`, all mean "/". If the name has no known value (as with a name of "qacute") or is syntactically invalid (as with a name of "1/4"), this returns undef.

- e2charnum($e_content)

  Given a name or number that could appear in a `E<name_or_num>` sequence, this returns the number of the Unicode character that this stands for. For example, `e2char('sol')`, `e2char('47')`, `e2char('0x2F')`, and `e2char('057')` all return 47, because `E<sol>`, `E<47>`, `E<0x2f>`, and `E<057>`, all mean "/", whose Unicode number is 47. If the name has no known value (as with a name of "qacute") or is syntactically invalid (as with a name of "1/4"), this returns undef.

- $Name2character{*name*}

  Maps from names (as in `E<_name_>`) like "eacute" or "sol" to the string that each stands for. Note that this does not include numerics (like "64" or "x981c"). Under old Perl versions (before 5.7) you get a "?" in place of characters whose Unicode value is over 255.

- $Name2character_number{*name*}

  Maps from names (as in `E<_name_>`) like "eacute" or "sol" to the Unicode value that each stands for. For example, `$Name2character_number{'eacute'}` is 201, and `$Name2character_number{'eacute'}` is 8364. You get the correct Unicode value, regardless of the version of Perl you're using – which differs from `%Name2character`'s behavior under pre-5.7 Perls.

  Note that this hash does not include numerics (like "64" or "x981c").

- $Latin1Code_to_fallback{*integer*}

  For numbers in the range 160 (0x00A0) to 255 (0x00FF), this maps from the character code for a Latin-1 character (like 233 for lowercase e-acute) to the US-ASCII character that best aproximates it (like "e"). You may find this useful if you are rendering POD in a format that you think deals well only with US-ASCII characters.

- $Latin1Char_to_fallback{*character*}

  Just as above, but maps from characters (like "\xE9", lowercase e-acute) to characters (like "e").

- $Code2USASCII{*integer*}

  This maps from US-ASCII codes (like 32) to the corresponding character (like space, for 32). Only characters 32 to 126 are defined. This is meant for use by `e2char($x)` when it senses that it's running on a non-ASCII platform (where chr(32) doesn't get you a space – but $Code2USASCII{32} will). It's documented here just in case you might find it useful.

CAVEATS

On Perl versions before 5.7, Unicode characters with a value over 255 (like lambda or emdash) can't be conveyed. This module does work under such early Perl versions, but in the place of each such character, you get a "?". Latin-1 characters (characters 160-255) are unaffected.

Under EBCDIC platforms, `e2char($n)` may not always be the same as `chr(e2charnum($n))`, and ditto for `$Name2character{$name}` and `chr($Name2character_number{$name})`, because the strings are returned as native, and the numbers are returned as Unicode. However, for Perls starting with v5.8, `e2char($n)` is the same as `chr(utf8::unicode_to_native(e2charnum($n)))`, and ditto for `$Name2character{$name}` and `chr(utf8::unicode_to_native($Name2character_number{$name}))`.

SEE ALSO

Pod::Browser - a pod web server based on Catalyst.

Pod::Checker - check pod documents for syntax errors.

Pod::Coverage - check if the documentation for a module is comprehensive.

perlpod - description of pod format (for people documenting with pod).

perlpodspec - specification of pod format (for people processing it).

Text::Unidecode - ASCII transliteration of Unicode text.

REPOSITORY

https://github.com/neilbowers/Pod-Escapes

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2001-2004 Sean M. Burke. All rights reserved.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

Portions of the data tables in this module are derived from the entity declarations in the W3C XHTML specification.

Currently (October 2001), that's these three:

```
http://www.w3.org/TR/xhtml1/DTD/xhtml-lat1.ent
http://www.w3.org/TR/xhtml1/DTD/xhtml-special.ent
http://www.w3.org/TR/xhtml1/DTD/xhtml-symbol.ent
```

AUTHOR

Sean M. Burke sburke@cpan.org

Now being maintained by Neil Bowers neilb@cpan.org

Pod::Functions

Group Perl's functions a la perlfunc.pod

SYNOPSIS

```
use Pod::Functions;

my @misc_ops = @{ $Kinds{ 'Misc' } };
my $misc_dsc = $Type_Description{ 'Misc' };
```

or

```
perl /path/to/lib/Pod/Functions.pm
```

This will print a grouped list of Perl's functions, like the *"Perl Functions by Category" in perlfunc* section.

DESCRIPTION

It exports the following variables:

- This holds a hash-of-lists. Each list contains the functions in the category the key denotes.
- In this hash each key represents a function and the value is the category. The category can be a comma separated list.
- In this hash each key represents a function and the value is a short description of that function.
- In this hash each key represents a category of functions and the value is a short description of that category.
- @Type_Order

  This list of categories is used to produce the same order as the *"Perl Functions by Category" in perlfunc* section.

Pod::Html

module to convert pod files to HTML

SYNOPSIS

```
use Pod::Html;
pod2html([options]);
```

DESCRIPTION

Converts files from pod format (see perlpod) to HTML format. It can automatically generate indexes and cross-references, and it keeps a cache of things it knows how to cross-reference.

FUNCTIONS

pod2html

```
pod2html("pod2html",
         "--podpath=lib:ext:pod:vms",
         "--podroot=/usr/src/perl",
         "--htmlroot=/perl/nmanual",
         "--recurse",
         "--infile=foo.pod",
         "--outfile=/perl/nmanual/foo.html");
```

pod2html takes the following arguments:

- backlink

  ```
  --backlink
  ```

  Turns every head1 heading into a link back to the top of the page. By default, no backlinks are generated.

- cachedir

  ```
  --cachedir=name
  ```

  Creates the directory cache in the given directory.

- css

  ```
  --css=stylesheet
  ```

  Specify the URL of a cascading style sheet. Also disables all HTML/CSS `style` attributes that are output by default (to avoid conflicts).

- flush

  ```
  --flush
  ```

Flushes the directory cache.

- header

```
--header
--noheader
```

Creates header and footer blocks containing the text of the NAME section. By default, no headers are generated.

- help

```
--help
```

Displays the usage message.

- htmldir

```
--htmldir=name
```

Sets the directory to which all cross references in the resulting html file will be relative. Not passing this causes all links to be absolute since this is the value that tells Pod::Html the root of the documentation tree.

Do not use this and –htmlroot in the same call to pod2html; they are mutually exclusive.

- htmlroot

```
--htmlroot=name
```

Sets the base URL for the HTML files. When cross-references are made, the HTML root is prepended to the URL.

Do not use this if relative links are desired: use –htmldir instead.

Do not pass both this and –htmldir to pod2html; they are mutually exclusive.

- index

```
--index
--noindex
```

Generate an index at the top of the HTML file. This is the default behaviour.

- infile

```
--infile=name
```

Specify the pod file to convert. Input is taken from STDIN if no infile is specified.

- outfile

```
--outfile=name
```

Specify the HTML file to create. Output goes to STDOUT if no outfile is specified.

- poderrors

```
--poderrors
--nopoderrors
```

Include a “POD ERRORS” section in the outfile if there were any POD errors in the infile. This section is included by default.

- podpath

```
--podpath=name:...:name
```

Specify which subdirectories of the podroot contain pod files whose HTML converted forms can be linked to in cross references.

- podroot

```
--podroot=name
```

Specify the base directory for finding library pods. Default is the current working directory.

- quiet

```
--quiet
--noquiet
```

Don’t display *mostly harmless* warning messages. These messages will be displayed by default. But this is not the same as `verbose` mode.

- recurse

```
--recurse
--norecurse
```

Recurse into subdirectories specified in podpath (default behaviour).

- title

```
--title=title
```

Specify the title of the resulting HTML file.

- verbose

```
--verbose
--noverbose
```

Display progress messages. By default, they won’t be displayed.

Formerly Exported Auxiliary Functions

Prior to perl-5.36, the following three functions were exported by `Pod::Html`, either by default or on request:

- `htmlify()` (by default)
- `anchorify()` (upon request)

- `relativize_url()` (upon request)

The definition and documentation of these functions have been moved to `Pod::Html::Util`, viewable via `perldoc Pod::Html::Util`.

Beginning with perl-5.38 these functions must be explicitly imported from `Pod::Html::Util`. Please modify your code as needed.

ENVIRONMENT

Uses `$Config{pod2html}` to setup default options.

AUTHOR

Marc Green, marcgreen@cpan.org.

Original version by Tom Christiansen, tchrist@perl.com.

SEE ALSO

perlpod

COPYRIGHT

This program is distributed under the Artistic License.

Modules

- `Pod::Html::Util` - helper functions for Pod-Html

Pod::Html::Util

helper functions for Pod-Html

SUBROUTINES

Note: While these functions are importable on request from `Pod::Html::Util`, they are specifically intended for use within (a) the `Pod-Html` distribution (modules and test programs) shipped as part of the Perl 5 core and (b) other parts of the core such as the `installhtml` program. These functions may be modified or relocated within the core distribution – or removed entirely therefrom – as the core's needs evolve. Hence, you should not rely on these functions in situations other than those just described.

`process_command_line()`

Process command-line switches (options). Returns a reference to a hash. Will provide usage message if `--help` switch is present or if parameters are invalid.

Calling this subroutine may modify `@ARGV`.

usage()

Display customary Pod::Html usage information on STDERR.

unixify()

Ensure that Pod::Html's internals and tests handle paths consistently across Unix, Windows and VMS.

relativize_url()

Convert an absolute URL to one relative to a base URL. Assumes both end in a filename.

html_escape()

Make text safe for HTML.

htmlify()

```
htmlify($heading);
```

Converts a pod section specification to a suitable section specification for HTML. Note that we keep spaces and special characters except " , ? (Netscape problem) and the hyphen (writer's problem...).

anchorify()

```
anchorify(@heading);
```

Similar to `htmlify()`, but turns non-alphanumerics into underscores. Note that `anchorify()` is not exported by default.

trim_leading_whitespace()

Remove any level of indentation (spaces or tabs) from each code block consistently. Adapted from: https://metacpan.org/source/HAARG/MetaCPAN-Pod-XHTML-0.002001/lib/Pod/Simple/Role/StripVerbatimIndent.pm

Pod::Man

Convert POD data to formatted *roff input

SYNOPSIS

```
use Pod::Man;
my $parser = Pod::Man->new (release => $VERSION, section => 8);

# Read POD from STDIN and write to STDOUT.
```

(continues on next page)

(continued from previous page)

```
$parser->parse_file (\*STDIN);

# Read POD from file.pod and write to file.1.
$parser->parse_from_file ('file.pod', 'file.1');
```

DESCRIPTION

Pod::Man is a module to convert documentation in the POD format (the preferred language for documenting Perl) into *roff input using the man macro set. The resulting *roff code is suitable for display on a terminal using nroff(1), normally via man(1), or printing using troff(1). It is conventionally invoked using the driver script **pod2man**, but it can also be used directly.

By default (on non-EBCDIC systems), Pod::Man outputs UTF-8. Its output should work with the **man** program on systems that use **groff** (most Linux distributions) or **mandoc** (most BSD variants), but may result in mangled output on older UNIX systems. To choose a different, possibly more backward-compatible output mangling on such systems, set the `encoding` option to `roff` (the default in earlier Pod::Man versions). See the `encoding` option and "ENCODING" for more details.

See "COMPATIBILITY" for the versions of Pod::Man with significant backward-incompatible changes (other than constructor options, whose versions are documented below), and the versions of Perl that included them.

CLASS METHODS

- new(ARGS)

  Create a new Pod::Man object. ARGS should be a list of key/value pairs, where the keys are chosen from the following. Each option is annotated with the version of Pod::Man in which that option was added with its current meaning.

  - center

    [1.00] Sets the centered page header for the `.TH` macro. The default, if this option is not specified, is `User Contributed Perl Documentation`.

  - date

    [4.00] Sets the centered page footer for the `.TH` macro. If this option is not set, the contents of the environment variable POD_MAN_DATE, if set, will be used. Failing that, the value of SOURCE_DATE_EPOCH, the modification date of the input file, or the current time if stat() can't find that file (which will be the case if the input is from `STDIN`) will be used. If taken from any source other than POD_MAN_DATE (which is used verbatim), the date will be formatted as `YYYY-MM-DD` and will be based on UTC (so that the output will be reproducible regardless of local time zone).

    Note that some system an macro sets assume that the centered footer will be a modification date and will prepend something like `Last modified: `.

  - encoding

[5.00] Specifies the encoding of the output. The value must be an encoding recognized by the *Encode* module (see Encode::Supported), or the special values `roff` or `groff`. The default on non-EBCDIC systems is UTF-8.

If the output contains characters that cannot be represented in this encoding, that is an error that will be reported as configured by the `errors` option. If error handling is other than `die`, the unrepresentable character will be replaced with the Encode substitution character (normally `?`).

If the `encoding` option is set to the special value `groff` (the default on EBCDIC systems), or if the Encode module is not available and the encoding is set to anything other than `roff`, Pod::Man will translate all non-ASCII characters to `\[uNNNN]` Unicode escapes. These are not traditionally part of the *roff language, but are supported by **groff** and **mandoc** and thus by the majority of manual page processors in use today.

If the `encoding` option is set to the special value `roff`, Pod::Man will do its historic transformation of (some) ISO 8859-1 characters into *roff escapes that may be adequate in troff and may be readable (if ugly) in nroff. This was the default behavior of versions of Pod::Man before 5.00. With this encoding, all other non-ASCII characters will be replaced with `X`. It may be required for very old troff and nroff implementations that do not support UTF-8, but its representation of any non-ASCII character is very poor and often specific to European languages.

If the output file handle has a PerlIO encoding layer set, setting `encoding` to anything other than `groff` or `roff` will be ignored and no encoding will be done by Pod::Man. It will instead rely on the encoding layer to make whatever output encoding transformations are desired.

WARNING: The input encoding of the POD source is independent from the output encoding, and setting this option does not affect the interpretation of the POD input. Unless your POD source is US-ASCII, its encoding should be declared with the `=encoding` command in the source. If this is not done, Pod::Simple will will attempt to guess the encoding and may be successful if it's Latin-1 or UTF-8, but it will produce warnings. See perlpod(1) for more information.

– errors

[2.27] How to report errors. `die` says to throw an exception on any POD formatting error. `stderr` says to report errors on standard error, but not to throw an exception. pod says to include a POD ERRORS section in the resulting documentation summarizing the errors. none ignores POD errors entirely, as much as possible.

The default is pod.

– fixed

[6.1.0] The fixed-width font to use for verbatim text and code. Defaults to `CR`. Solaris systems historically preferred `CW` instead. Only matters for **troff** output.

– fixedbold

[1.00] Bold version of the fixed-width font. Defaults to CB. Only matters for **troff** output.

– fixeditalic

[1.00] Italic version of the fixed-width font (something of a misnomer, since most fixed-width fonts only have an oblique version, not an italic version). Defaults to CI. Only matters for **troff** output.

– fixedbolditalic

[1.00] Bold italic (in theory, probably oblique in practice) version of the fixed-width font. Pod::Man doesn't assume you have this, and defaults to CB. Some systems (such as Solaris) have this font available as CX. Only matters for **troff** output.

– guesswork

[5.00] By default, Pod::Man applies some default formatting rules based on guesswork and regular expressions that are intended to make writing Perl documentation easier and require less explicit markup. These rules may not always be appropriate, particularly for documentation that isn't about Perl. This option allows turning all or some of it off.

The special value `all` enables all guesswork. This is also the default for backward compatibility reasons. The special value none disables all guesswork. Otherwise, the value of this option should be a comma-separated list of one or more of the following keywords:

* functions

  Convert function references like `foo()` to bold even if they have no markup. The function name accepts valid Perl characters for function names (including :), and the trailing parentheses must be present and empty.

* manref

  Make the first part (before the parentheses) of manual page references like `foo(1)` bold even if they have no markup. The section must be a single number optionally followed by lowercase letters.

* quoting

  If no guesswork is enabled, any text enclosed in C<> is surrounded by double quotes in nroff (terminal) output unless the contents are already quoted. When this guesswork is enabled, quote marks will also be suppressed for Perl variables, function names, function calls, numbers, and hex constants.

* variables

  Convert Perl variable names to a fixed-width font even if they have no markup. This transformation will only be apparent in troff output, or some other output format (unlike nroff terminal output) that supports fixed-width fonts.

Any unknown guesswork name is silently ignored (for potential future compatibility), so be careful about spelling.

– language

[5.00] Add commands telling **groff** that the input file is in the given language. The value of this setting must be a language abbreviation for which **groff** provides supplemental configuration, such as `ja` (for Japanese) or `zh` (for Chinese).

Specifically, this adds:

```
.mso <language>.tmac
.hla <language>
```

to the start of the file, which configure correct line breaking for the specified language. Without these commands, groff may not know how to add proper line breaks for Chinese and Japanese text if the manual page is installed into the normal manual page directory, such as `/usr/share/man`.

On many systems, this will be done automatically if the manual page is installed into a language-specific manual page directory, such as `/usr/share/man/zh_CN`. In that case, this option is not required.

Unfortunately, the commands added with this option are specific to **groff** and will not work with other **troff** and **nroff** implementations.

– lquote

– rquote

[4.08] Sets the quote marks used to surround C<> text. `lquote` sets the left quote mark and `rquote` sets the right quote mark. Either may also be set to the special value none, in which case no quote mark is added on that side of C<> text (but the font is still changed for troff output).

Also see the `quotes` option, which can be used to set both quotes at once. If both `quotes` and one of the other options is set, `lquote` or `rquote` overrides `quotes`.

– name

[4.08] Set the name of the manual page for the `.TH` macro. Without this option, the manual name is set to the uppercased base name of the file being converted unless the manual section is 3, in which case the path is parsed to see if it is a Perl module path. If it is, a path like `.../lib/Pod/Man.pm` is converted into a name like `Pod::Man`. This option, if given, overrides any automatic determination of the name.

If generating a manual page from standard input, the name will be set to `STDIN` if this option is not provided. In this case, providing this option is strongly recommended to set a meaningful manual page name.

– nourls

[2.27] Normally, L<> formatting codes with a URL but anchor text are formatted to show both the anchor text and the URL. In other words:

```
L<foo|http://example.com/>
```

is formatted as:

```
foo <http://example.com/>
```

This option, if set to a true value, suppresses the URL when anchor text is given, so this example would be formatted as just `foo`. This can produce less cluttered output in cases where the URLs are not particularly important.

– quotes

[4.00] Sets the quote marks used to surround C<> text. If the value is a single character, it is used as both the left and right quote. Otherwise, it is split in half, and the first half of the string is used as the left quote and the second is used as the right quote.

This may also be set to the special value none, in which case no quote marks are added around C<> text (but the font is still changed for troff output).

Also see the `lquote` and `rquote` options, which can be used to set the left and right quotes independently. If both `quotes` and one of the other options is set, `lquote` or `rquote` overrides `quotes`.

– release

[1.12] Set the left-hand page footer for the `.TH` macro. By default, this is set to the version of Perl you run Pod::Man under. Setting this to the empty string will cause some *roff implementations to use the system default value.

– section

[1.00] Set the section for the `.TH` macro. The standard section numbering convention is to use 1 for user commands, 2 for system calls, 3 for functions, 4 for devices, 5 for file formats, 6 for games, 7 for miscellaneous information, and 8 for administrator commands. There is a lot of variation here, however; some systems (like Solaris) use 4 for file formats, 5 for miscellaneous information, and 7 for devices. Still others use 1m instead of 8, or some mix of both. About the only section numbers that are reliably consistent are 1, 2, and 3.

By default, section 1 will be used unless the file ends in `.pm` in which case section 3 will be selected.

– stderr

[2.19] If set to a true value, send error messages about invalid POD to standard error instead of appending a POD ERRORS section to the generated *roff output. This is equivalent to setting `errors` to `stderr` if `errors` is not already set.

This option is for backward compatibility with Pod::Man versions that did not support `errors`. Normally, the `errors` option should be used instead.

– utf8

[2.21] This option used to set the output encoding to UTF-8. Since this is now the default, it is ignored and does nothing.

INSTANCE METHODS

As a derived class from Pod::Simple, Pod::Man supports the same methods and interfaces. See *Pod::Simple* for all the details. This section summarizes the most-frequently-used methods and the ones added by Pod::Man.

- output_fh(FH)

  Direct the output from parse_file(), parse_lines(), or parse_string_document() to the file handle FH instead of STDOUT.

- output_string(REF)

  Direct the output from parse_file(), parse_lines(), or parse_string_document() to the scalar variable pointed to by REF, rather than STDOUT. For example:

```
my $man = Pod::Man->new();
my $output;
$man->output_string(\$output);
$man->parse_file('/some/input/file');
```

  Be aware that the output in that variable will already be encoded in UTF-8.

- parse_file(PATH)

  Read the POD source from PATH and format it. By default, the output is sent to STDOUT, but this can be changed with the output_fh() or output_string() methods.

- parse_from_file(INPUT, OUTPUT)

- parse_from_filehandle(FH, OUTPUT)

  Read the POD source from INPUT, format it, and output the results to OUTPUT.

  parse_from_filehandle() is provided for backward compatibility with older versions of Pod::Man. parse_from_file() should be used instead.

- parse_lines(LINES[, ...[, undef]])

  Parse the provided lines as POD source, writing the output to either STDOUT or the file handle set with the output_fh() or output_string() methods. This method can be called repeatedly to provide more input lines. An explicit `undef` should be passed to indicate the end of input.

  This method expects raw bytes, not decoded characters.

- parse_string_document(INPUT)

  Parse the provided scalar variable as POD source, writing the output to either STDOUT or the file handle set with the output_fh() or output_string() methods.

  This method expects raw bytes, not decoded characters.

ENCODING

As of Pod::Man 5.00, the default output encoding for Pod::Man is UTF-8. This should work correctly on any modern system that uses either **groff** (most Linux distributions) or **mandoc** (Alpine Linux and most BSD variants, including macOS).

The user will probably have to use a UTF-8 locale to see correct output. This may be done by default; if not, set the LANG or LC_CTYPE environment variables to an appropriate local. The locale `C.UTF-8` is available on most systems if one wants correct output without changing the other things locales affect, such as collation.

The backward-compatible output format used in Pod::Man versions before 5.00 is available by setting the `encoding` option to `roff`. This may produce marginally nicer results on older UNIX versions that do not use **groff** or **mandoc**, but none of the available options will correctly render Unicode characters on those systems.

Below are some additional details about how this choice was made and some discussion of alternatives.

History

The default output encoding for Pod::Man has been a long-standing problem. **troff** and **nroff** predate Unicode by a significant margin, and their implementations for many UNIX systems reflect that legacy. It's common for Unicode to not be supported in any form.

Because of this, versions of Pod::Man prior to 5.00 maintained the highly conservative output of the original pod2man, which output pure ASCII with complex macros to simulate common western European accented characters when processed with troff. The nroff output was awkward and sometimes incorrect, and characters not used in western European scripts were replaced with X. This choice maximized backwards compatibility with **man** and **nroff**/**troff** implementations at the cost of incorrect rendering of many POD documents, particularly those containing people's names.

The modern implementations, **groff** (used in most Linux distributions) and **mandoc** (used by most BSD variants), do now support Unicode. Other UNIX systems often do not, but they're now a tiny minority of the systems people use on a daily basis. It's increasingly common (for very good reasons) to use Unicode characters for POD documents rather than using ASCII conversions of people's names or avoiding non-English text, making the limitations in the old output format more apparent.

Four options have been proposed to fix this:

- Optionally support UTF-8 output but don't change the default. This is the approach taken since Pod::Man 2.1.0, which added the `utf8` option. Some Pod::Man users use this option for better output on platforms known to support Unicode, but since the defaults have not changed, people continued to encounter (and file bug reports about) the poor default rendering.
- Convert characters to troff `\(xx` escapes. This requires maintaining a large translation table and addresses only a tiny part of the problem, since many Unicode characters have no standard troff name. **groff** has the largest list, but if one is willing to assume **groff** is the formatter, the next option is better.
- Convert characters to groff `\[uNNNN]` escapes. This is implemented as the `groff` encoding for those who want to use it, and is supported by both **groff** and **mandoc**. However, it is no better than UTF-8 output for portability to other implementations. See "Testing results" for more details.
- Change the default output format to UTF-8 and ask those who want maximum backward compatibility to explicitly select the old encoding. This fixes the issue for most users at the cost of backwards compatibility. While the rendering of non-ASCII

characters is different on older systems that don't support UTF-8, it's not always worse than the old output.

Pod::Man 5.00 and later makes the last choice. This arguably produces worse output when manual pages are formatted with **troff** into PostScript or PDF, but doing this is rare and normally manual, so the encoding can be changed in those cases. The older output encoding is available by setting `encoding` to `roff`.

Testing results

Here is the results of testing `encoding` values of `utf-8` and `groff` on various operating systems. The testing methodology was to create `man/man1` in the current directory, copy `encoding.utf8` or `encoding.groff` from the podlators 5.00 distribution to `man/man1/encoding.1`, and then run:

```
LANG=C.UTF-8 MANPATH=$(pwd)/man man 1 encoding
```

If the locale is not explicitly set to one that includes UTF-8, the Unicode characters were usually converted to ASCII (by, for example, dropping an accent) or deleted or replaced with <?> if there was no conversion.

Tested on 2022-09-25. Many thanks to the GCC Compile Farm project for access to testing hosts.

```
OS                  UTF-8      groff
------------------  -------    -------
AIX 7.1             no [1]     no [2]
Alpine 3.15.0       yes        yes
CentOS 7.9          yes        yes
Debian 7            yes        yes
FreeBSD 13.0        yes        yes
NetBSD 9.2          yes        yes
OpenBSD 7.1         yes        yes
openSUSE Leap 15.4  yes        yes
Solaris 10          yes        no [2]
Solaris 11          no [3]     no [3]
```

I did not have access to a macOS system for testing, but since it uses **mandoc**, it's behavior is probably the same as the BSD hosts.

Notes:

- [1]

  Unicode characters were converted to one or two random ASCII characters unrelated to the original character.

- [2]

  Unicode characters were shown as the body of the groff escape rather than the indicated character (in other words, text like `[u00EF]`).

- [3]

  Unicode characters were deleted entirely, as if they weren't there. Using `nroff -man` instead of **man** to format the page showed the same results as Solaris 10.

Using `groff -k -man -Tutf8` to format the page produced the correct output.

PostScript and PDF output using groff on a Debian 12 system do not support combining accent marks or SMP characters due to a lack of support in the default output font.

Testing on additional platforms is welcome. Please let the author know if you have additional results.

DIAGNOSTICS

- roff font should be 1 or 2 chars, not "%s"

  (F) You specified a *roff font (using `fixed`, `fixedbold`, etc.) that wasn't either one or two characters. Pod::Man doesn't support *roff fonts longer than two characters, although some *roff extensions do (the canonical versions of **nroff** and **troff** don't either).

- Invalid errors setting "%s"

  (F) The `errors` parameter to the constructor was set to an unknown value.

- Invalid quote specification "%s"

  (F) The quote specification given (the `quotes` option to the constructor) was invalid. A quote specification must be either one character long or an even number (greater than one) characters long.

- POD document had syntax errors

  (F) The POD document being formatted had syntax errors and the `errors` option was set to `die`.

ENVIRONMENT

- PERL_CORE

  If set and Encode is not available, silently fall back to an encoding of `groff` without complaining to standard error. This environment variable is set during Perl core builds, which build Encode after podlators. Encode is expected to not (yet) be available in that case.

- POD_MAN_DATE

  If set, this will be used as the value of the left-hand footer unless the `date` option is explicitly set, overriding the timestamp of the input file or the current time. This is primarily useful to ensure reproducible builds of the same output file given the same source and Pod::Man version, even when file timestamps may not be consistent.

- SOURCE_DATE_EPOCH

  If set, and POD_MAN_DATE and the `date` options are not set, this will be used as the modification time of the source file, overriding the timestamp of the input file or the current time. It should be set to the desired time in seconds since UNIX epoch. This is primarily useful to ensure reproducible builds of the same output file given the same source and Pod::Man version, even when file timestamps may not be consistent. See https://reproducible-builds.org/specs/source-date-epoch/ for the full specification.

(Arguably, according to the specification, this variable should be used only if the timestamp of the input file is not available and Pod::Man uses the current time. However, for reproducible builds in Debian, results were more reliable if this variable overrode the timestamp of the input file.)

COMPATIBILITY

Pod::Man 1.02 (based on Pod::Parser) was the first version included with Perl, in Perl 5.6.0.

The current API based on *Pod::Simple* was added in Pod::Man 2.00. Pod::Man 2.04 was included in Perl 5.9.3, the first version of Perl to incorporate those changes. This is the first version that correctly supports all modern POD syntax. The parse_from_filehandle() method was re-added for backward compatibility in Pod::Man 2.09, included in Perl 5.9.4.

Support for anchor text in L<> links of type URL was added in Pod::Man 2.23, included in Perl 5.11.5.

parse_lines(), parse_string_document(), and parse_file() set a default output file handle of STDOUT if one was not already set as of Pod::Man 2.28, included in Perl 5.19.5.

Support for SOURCE_DATE_EPOCH and POD_MAN_DATE was added in Pod::Man 4.00, included in Perl 5.23.7, and generated dates were changed to use UTC instead of the local time zone. This is also the first release that aligned the module version and the version of the podlators distribution. All modules included in podlators, and the podlators distribution itself, share the same version number from this point forward.

Pod::Man 4.10, included in Perl 5.27.8, changed the formatting for manual page references and function names to bold instead of italic, following the current Linux manual page standard.

Pod::Man 5.00, included in Perl 5.37.7, changed the default output encoding to UTF-8, overridable with the new `encoding` option. It also fixed problems with bold or italic extending too far when used with C<> escapes, and began converting Unicode zero-width spaces (U+200B) to the `\:` *roff escape. It also dropped attempts to add subtle formatting corrections in the output that would only be visible when typeset with **troff**, which had previously been a significant source of bugs.

Pod::Man v6.0.0 and later, included in Perl 5.41.2, unconditionally convert `-` to the `\-` *roff escape, representing an ASCII hyphen-minus. Earlier versions attempted to use heuristics to decide when a given `-` character should translate to a hyphen-minus or a true hyphen, but these heuristics were buggy and fragile. v6.0.0 and later also unconditionally convert `` ` `` and `'` to ASCII grave accent and apostrophe marks instead of the default *roff behavior of interpreting them as paired quotes.

Pod::Man v6.1.0 and later use `CR` as the default fixed-width font instead of `CW`, as was used in previous versions. This should only matter for **troff** output.

BUGS

There are numerous bugs and language-specific assumptions in the nroff fallbacks for accented characters in the `roff` encoding. Since the point of this encoding is backward compatibility with the output from earlier versions of Pod::Man, and it is deprecated except when necessary to support old systems, those bugs are unlikely to ever be fixed.

Pod::Man doesn't handle font names longer than two characters. Neither do most **troff** implementations, but groff does as an extension. It would be nice to support as an option for those who want to use it.

CAVEATS

Sentence spacing

It is difficult to write POD documents such that the resulting *roff output has consistent intersentence spacing. For most documents, writing the POD using whatever conventions you prefer and letting the *roff output be inconsistent is probably the best trade-off to make. But if you want to dive into the details, read on.

Pod::Man copies the input spacing verbatim to the output *roff document. This means your output will be affected by how *roff handles sentence spacing.

*roff dates from an era in which it was standard to use two spaces after sentences, and will always add two spaces after a line-ending period (or similar punctuation) when reflowing text, with one exception mentioned below. For example, the following input:

```
=pod

One sentence.
Another sentence.
```

will result in two spaces after the period in **nroff** output when the text is reflowed.

One significant exception, however, is that different versions of groff vary in their interpretation of `."` (a sentence ending with an ASCII double quote). Versions of groff prior to 1.24.0 treat this as the end of a sentence if there are two spaces after the closing quote, or if it comes at the end of a line. Later versions will never treat this as the end of a sentence and will collapse any following spaces down to a single space. For consistent behavior for documents written in POD, you must use Unicode double quotes (“”, U+201C and U+201D) and declare UTF-8 encoding (or some other Unicode encoding). The Unicode closing double-quote, when preceded by a period and followed by two spaces or the end of the line, will always be interpreted as the end of a sentence in modern versions of groff.

Therefore, if you want two spaces after sentences in the **nroff** output, you can achieve this by using Unicode double quotes and using two spaces after periods that end a sentence. You will have to be careful to not end a line with an abbreviation such as `e.g.` or `Ms.`. Alternately, you can use the *roff style guide (and XKCD 1285) recommendation of putting a line break after each sentence, but you will still need to use Unicode double quotes.

If you prefer one space after sentences (which is the more modern style), your best option is to configure your local *roff installation to always format man pages that way, rather than attempt to force this result via your POD source. For example, for groff, groff_man_style(7) documents putting:

```
.\" Put only one space after the end of a sentence.
.ss 12 0 \" See groff(7).
```

in `/usr/share/groff/site-tmac/man.local` (on Debian-derived systems, `/etc/`

`groff/man.local` is used instead). This will affect all man pages formatted with groff on that system, regardless of how their source is formatted.

Hyphens and quotes

The *roff language distinguishes between two types of hyphens: `-`, which is a true typesetting hyphen (roughly equivalent to the Unicode U+2010 code point), and `\-`, which is the ASCII hyphen-minus (U+002D) that is used for UNIX command options and most filenames. Hyphens, where appropriate, produce better typesetting, but incorrectly using them for command names and options can cause problems with searching and cut-and-paste.

POD does not draw this distinction. Before podlators v6.0.0, Pod::Man attempted to translate `-` in the input into either a hyphen or a hyphen-minus, depending on context. However, this distinction proved impossible to do correctly with heuristics. Pod::Man therefore translates all `-` characters in the input to `\-` in the output, ensuring that command names and options are correct at the cost of somewhat inferior typesetting and line breaking issues with long hyphenated phrases.

To use true hyphens in the Pod::Man output, declare an input character set of UTF-8 (or some other Unicode encoding) and use Unicode hyphens. Pod::Man and *roff should handle those correctly with the default output format and most modern *roff implementations.

Similarly, Pod::Man disables the default *roff behavior of turning `` ` `` and `'` characters into matched quotes, and pairs of those characters into matched double quotes, because there is no good way to tell from the POD input whether this interpretation is desired or whether the intent is to use a literal grave accent or neutral apostrophe. If you want paired quotes in the output, use Unicode and its paired quote characters.

AUTHOR

Written by Russ Allbery rra@cpan.org, based on the original **pod2man** by Tom Christiansen tchrist@mox.perl.com.

The modifications to work with Pod::Simple instead of Pod::Parser were contributed by Sean Burke sburke@cpan.org, but I've since hacked them beyond recognition and all bugs are mine.

COPYRIGHT AND LICENSE

Copyright 1999-2020, 2022-2026 Russ Allbery rra@cpan.org

Substantial contributions by Sean Burke sburke@cpan.org.

This program is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

SEE ALSO

Encode::Supported, *Pod::Simple*, perlpod(1), pod2man(1), nroff(1), troff(1), man(1), man(7)

Ossanna, Joseph F., and Brian W. Kernighan. "Troff User's Manual," Computing Science Technical Report No. 54, AT&T Bell Laboratories. This is the best documentation of standard **nroff** and **troff**. At the time of this writing, it's available at http://www.troff.org/54.pdf.

The manual page documenting the man macro set may be man(5) instead of man(7) on your system.

See perlpodstyle(1) for documentation on writing manual pages in POD if you've not done it before and aren't familiar with the conventions.

The current version of this module is always available from its web site at https://www.eyrie.org/~eagle/software/podlators/. It is also part of the Perl core distribution as of 5.6.0.

Pod::ParseLink

Parse an L<> formatting code in POD text

SYNOPSIS

```
use Pod::ParseLink;
my $link = get_link();
my ($text, $inferred, $name, $section, $type) = parselink($link);
```

DESCRIPTION

This module only provides a single function, parselink(), which takes the text of an L<> formatting code and parses it. It returns the anchor text for the link (if any was given), the anchor text possibly inferred from the name and section, the name or URL, the section if any, and the type of link. The type will be one of `url`, pod, or `man`, indicating a URL, a link to a POD page, or a link to a Unix manual page.

Parsing is implemented per perlpodspec. For backward compatibility, links where there is no section and name contains spaces, or links where the entirety of the link (except for the anchor text if given) is enclosed in double-quotes are interpreted as links to a section (L</section>).

The inferred anchor text is implemented per perlpodspec:

```
L<name>         =>  L<name|name>
L</section>     =>  L<"section"|/section>
L<name/section> =>  L<"section" in name|name/section>
```

The name may contain embedded E<> and Z<> formatting codes, and the section, anchor text, and inferred anchor text may contain any formatting codes. Any double quotes around the section are removed as part of the parsing, as is any leading or trailing whitespace.

If the text of the L<> escape is entirely enclosed in double quotes, it's interpreted as a link to a section for backward compatibility.

No attempt is made to resolve formatting codes. This must be done after calling parselink() (since E<> formatting codes can be used to escape characters that would oth-

erwise be significant to the parser and resolving them before parsing would result in an incorrect parse of a formatting code like:

```
L<verticalE<verbar>barE<sol>slash>
```

which should be interpreted as a link to the `vertical|bar/slash` POD page and not as a link to the `slash` section of the `bar` POD page with an anchor text of `vertical`. Note that not only the anchor text will need to have formatting codes expanded, but so will the target of the link (to deal with E<> and Z<> formatting codes), and special handling of the section may be necessary depending on whether the translator wants to consider markup in sections to be significant when resolving links. See perlpodspec for more information.

AUTHOR

Russ Allbery rra@cpan.org

COPYRIGHT AND LICENSE

Copyright 2001, 2008, 2009, 2014, 2018-2019, 2022, 2024 Russ Allbery rra@cpan.org

This program is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

SEE ALSO

Pod::Parser

The current version of this module is always available from its web site at https://www.eyrie.org/~eagle/software/podlators/.

Pod::Perldoc

Look up Perl documentation in Pod format.

SYNOPSIS

```
use Pod::Perldoc ();

Pod::Perldoc->run();
```

DESCRIPTION

The guts of perldoc utility.

SEE ALSO

perldoc

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2002-2007 Sean M. Burke.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Current maintainer: Mark Allen <mallen@cpan.org>

Past contributions from: brian d foy <bdfoy@cpan.org> Adriano R. Ferreira <ferreira@cpan.org>, Sean M. Burke <sburke@cpan.org>

Modules

- Pod::Perldoc::BaseTo - Base for Pod::Perldoc formatters
- Pod::Perldoc::GetOptsOO - Customized option parser for Pod::Perldoc
- Pod::Perldoc::ToANSI - render Pod with ANSI color escapes
- Pod::Perldoc::ToChecker - let Perldoc check Pod for errors
- Pod::Perldoc::ToMan - let Perldoc render Pod as man pages
- Pod::Perldoc::ToNroff - let Perldoc convert Pod to nroff
- Pod::Perldoc::ToPod - let Perldoc render Pod as ...
- Pod::Perldoc::ToRtf - let Perldoc render Pod as RTF
- Pod::Perldoc::ToTerm - render Pod with terminal escapes
- Pod::Perldoc::ToText - let Perldoc render Pod as plaintext
- Pod::Perldoc::ToTk - let Perldoc use Tk::Pod to render Pod
- Pod::Perldoc::ToXml - let Perldoc render Pod as XML

Pod::Perldoc::BaseTo

Base for Pod::Perldoc formatters

SYNOPSIS

```
package Pod::Perldoc::ToMyFormat;

use parent qw( Pod::Perldoc::BaseTo );
...
```

DESCRIPTION

This package is meant as a base of Pod::Perldoc formatters, like *Pod::Perldoc::ToText*, *Pod::Perldoc::ToMan*, etc.

It provides default implementations for the methods

```
is_pageable
write_with_binmode
output_extension
_perldoc_elem
```

The concrete formatter must implement

```
new
parse_from_file
```

SEE ALSO

perldoc

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2002-2007 Sean M. Burke.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Current maintainer: Mark Allen <mallen@cpan.org>

Past contributions from: brian d foy <bdfoy@cpan.org> Adriano R. Ferreira <ferreira@cpan.org>, Sean M. Burke <sburke@cpan.org>

Pod::Perldoc::GetOptsOO

Customized option parser for Pod::Perldoc

SYNOPSIS

```
use Pod::Perldoc::GetOptsOO ();

Pod::Perldoc::GetOptsOO::getopts( $obj, \@args, $truth )
   or die "wrong usage";
```

DESCRIPTION

Implements a customized option parser used for *Pod::Perldoc*.

Rather like Getopt::Std's getopts:

- Call Pod::Perldoc::GetOptsOO::getopts($object, \@ARGV, $truth)
- Given -n, if there's a opt_n_with, it'll call $object->opt_n_with(ARGUMENT) (e.g., "-n foo" => $object->opt_n_with('foo'). Ditto "-nfoo")
- Otherwise (given -n) if there's an opt_n, we'll call it $object->opt_n($truth) (Truth defaults to 1)
- Otherwise we try calling $object->handle_unknown_option('n') (and we increment the error count by the return value of it)
- If there's no handle_unknown_option, then we just warn, and then increment the error counter

The return value of Pod::Perldoc::GetOptsOO::getopts is true if no errors, otherwise it's false.

SEE ALSO

Pod::Perldoc

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2002-2007 Sean M. Burke.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Current maintainer: Mark Allen <mallen@cpan.org>

Past contributions from: brian d foy <bdfoy@cpan.org> Adriano R. Ferreira <ferreira@cpan.org>, Sean M. Burke <sburke@cpan.org>

Pod::Perldoc::ToANSI

render Pod with ANSI color escapes

SYNOPSIS

```
perldoc -o ansi Some::Modulename
```

DESCRIPTION

This is a “plug-in” class that allows Perldoc to use Pod::Text as a formatter class.

It supports the following options, which are explained in *Pod::Text*: alt, indent, loose, quotes, sentence, width

For example:

```
perldoc -o term -w indent:5 Some::Modulename
```

CAVEAT

This module may change to use a different text formatter class in the future, and this may change what options are supported.

SEE ALSO

Pod::Text, *Pod::Text::Color*, *Pod::Perldoc*

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2011 Mark Allen. All rights reserved.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Current maintainer: Mark Allen <mallen@cpan.org>

Past contributions from: brian d foy <bdfoy@cpan.org> Adriano R. Ferreira <ferreira@cpan.org>, Sean M. Burke <sburke@cpan.org>

Pod::Perldoc::ToChecker

let Perldoc check Pod for errors

SYNOPSIS

```
% perldoc -o checker SomeFile.pod
No Pod errors in SomeFile.pod
(or an error report)
```

DESCRIPTION

This is a "plug-in" class that allows Perldoc to use Pod::Simple::Checker as a "formatter" class (or if that is not available, then Pod::Checker), to check for errors in a given Pod file.

This is actually a Pod::Simple::Checker (or Pod::Checker) subclass, and inherits all its options.

SEE ALSO

Pod::Simple::Checker, *Pod::Simple*, *Pod::Checker*, *Pod::Perldoc*

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2002 Sean M. Burke. All rights reserved.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Current maintainer: Mark Allen <mallen@cpan.org>

Past contributions from: brian d foy <bdfoy@cpan.org> Adriano R. Ferreira <ferreira@cpan.org>, Sean M. Burke <sburke@cpan.org>

Pod::Perldoc::ToMan

let Perldoc render Pod as man pages

SYNOPSIS

```
perldoc -o man Some::Modulename
```

DESCRIPTION

This is a "plug-in" class that allows Perldoc to use Pod::Man and `groff` for reading Pod pages.

The following options are supported: center, date, fixed, fixedbold, fixeditalic, fixedbolditalic, quotes, release, section

(Those options are explained in *Pod::Man*.)

For example:

```
perldoc -o man -w center:Pod Some::Modulename
```

CAVEAT

This module may change to use a different pod-to-nroff formatter class in the future, and this may change what options are supported.

SEE ALSO

Pod::Man, *Pod::Perldoc*, *Pod::Perldoc::ToNroff*

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2011 brian d foy. All rights reserved.

Copyright (c) 2002,3,4 Sean M. Burke. All rights reserved.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Current maintainer: Mark Allen <mallen@cpan.org>

Past contributions from: brian d foy <bdfoy@cpan.org> Adriano R. Ferreira <ferreira@cpan.org>, Sean M. Burke <sburke@cpan.org>

Pod::Perldoc::ToNroff

let Perldoc convert Pod to nroff

SYNOPSIS

```
perldoc -o nroff -d something.3 Some::Modulename
```

DESCRIPTION

This is a “plug-in” class that allows Perldoc to use Pod::Man as a formatter class.

The following options are supported: center, date, fixed, fixedbold, fixeditalic, fixedbolditalic, quotes, release, section

Those options are explained in *Pod::Man*.

For example:

```
perldoc -o nroff -w center:Pod -d something.3 Some::Modulename
```

CAVEAT

This module may change to use a different pod-to-nroff formatter class in the future, and this may change what options are supported.

SEE ALSO

Pod::Man, *Pod::Perldoc*, *Pod::Perldoc::ToMan*

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2002 Sean M. Burke. All rights reserved.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Current maintainer: Mark Allen <mallen@cpan.org>

Past contributions from: brian d foy <bdfoy@cpan.org> Adriano R. Ferreira <ferreira@cpan.org>, Sean M. Burke <sburke@cpan.org>

Pod::Perldoc::ToPod

let Perldoc render Pod as ... Pod!

SYNOPSIS

```
perldoc -opod Some::Modulename
```

(That's currently the same as the following:)

```
perldoc -u Some::Modulename
```

DESCRIPTION

This is a "plug-in" class that allows Perldoc to display Pod source as itself! Pretty Zen, huh?

Currently this class works by just filtering out the non-Pod stuff from a given input file.

SEE ALSO

Pod::Perldoc

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2002 Sean M. Burke. All rights reserved.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Current maintainer: Mark Allen <mallencpan.org>

Past contributions from: brian d foy <bdfoy@cpan.org> Adriano R. Ferreira <ferreira@cpan.org>, Sean M. Burke <sburke@cpan.org>

Pod::Perldoc::ToRtf

let Perldoc render Pod as RTF

SYNOPSIS

```
perldoc -o rtf Some::Modulename
```

DESCRIPTION

This is a “plug-in” class that allows Perldoc to use Pod::Simple::RTF as a formatter class.

This is actually a Pod::Simple::RTF subclass, and inherits all its options.

You have to have Pod::Simple::RTF installed (from the Pod::Simple dist), or this module won’t work.

If Perldoc is running under MSWin and uses this class as a formatter, the output will be opened with `write.exe` or whatever program is specified in the environment variable `RTFREADER`. For example, to specify that RTF files should be opened the same as they are when you double-click them, you would do `set RTFREADER=start.exe` in your `autoexec.bat`.

Handy tip: put `set PERLDOC=-ortf` in your `autoexec.bat` and that will set this class as the default formatter to run when you do `perldoc whatever`.

SEE ALSO

Pod::Simple::RTF, *Pod::Simple*, *Pod::Perldoc*

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2002 Sean M. Burke. All rights reserved.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Current maintainer: Mark Allen <mallen@cpan.org>

Past contributions from: brian d foy <bdfoy@cpan.org> Adriano R. Ferreira <ferreira@cpan.org>, Sean M. Burke <sburke@cpan.org>

Pod::Perldoc::ToTerm

render Pod with terminal escapes

SYNOPSIS

```
perldoc -o term Some::Modulename
```

DESCRIPTION

This is a "plug-in" class that allows Perldoc to use Pod::Text as a formatter class.

It supports the following options, which are explained in *Pod::Text*: alt, indent, loose, quotes, sentence, width

For example:

```
perldoc -o term -w indent:5 Some::Modulename
```

PAGER FORMATTING

Depending on the platform, and because this class emits terminal escapes it will attempt to set the `-R` flag on your pager by injecting the flag into your environment variable for `less` or `more`.

On Windows and DOS, this class will not modify any environment variables.

CAVEAT

This module may change to use a different text formatter class in the future, and this may change what options are supported.

SEE ALSO

Pod::Text, *Pod::Text::Termcap*, *Pod::Perldoc*

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2017 Mark Allen.

This program is free software; you can redistribute it and/or modify it under the terms of either: the GNU General Public License as published by the Free Software Foundation; or the Artistic License.

See http://dev.perl.org/licenses/ for more information.

AUTHOR

Mark Allen <mallen@cpan.org>

Pod::Perldoc::ToText

let Perldoc render Pod as plaintext

SYNOPSIS

```
perldoc -o text Some::Modulename
```

DESCRIPTION

This is a “plug-in” class that allows Perldoc to use Pod::Text as a formatter class.

It supports the following options, which are explained in *Pod::Text*: alt, indent, loose, quotes, sentence, width

For example:

```
perldoc -o text -w indent:5 Some::Modulename
```

CAVEAT

This module may change to use a different text formatter class in the future, and this may change what options are supported.

SEE ALSO

Pod::Text, *Pod::Perldoc*

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2002 Sean M. Burke. All rights reserved.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Current maintainer: Mark Allen <mallen@cpan.org>

Past contributions from: brian d foy <bdfoy@cpan.org> Adriano R. Ferreira <ferreira@cpan.org>, Sean M. Burke <sburke@cpan.org>

Pod::Perldoc::ToTk

let Perldoc use Tk::Pod to render Pod

SYNOPSIS

```
perldoc -o tk Some::Modulename &
```

DESCRIPTION

This is a "plug-in" class that allows Perldoc to use Tk::Pod as a formatter class.

You have to have installed Tk::Pod first, or this class won't load.

SEE ALSO

Tk::Pod, *Pod::Perldoc*

AUTHOR

Current maintainer: Mark Allen <mallen@cpan.org>

Past contributions from: brian d foy <bdfoy@cpan.org> Adriano R. Ferreira <ferreira@cpan.org>; Sean M. Burke <sburke@cpan.org>; significant portions copied from tkpod in the Tk::Pod dist, by Nick Ing-Simmons, Slaven Rezic, et al.

Pod::Perldoc::ToXml

let Perldoc render Pod as XML

SYNOPSIS

```
perldoc -o xml -d out.xml Some::Modulename
```

DESCRIPTION

This is a "plug-in" class that allows Perldoc to use Pod::Simple::XMLOutStream as a formatter class.

This is actually a Pod::Simple::XMLOutStream subclass, and inherits all its options.

You have to have installed Pod::Simple::XMLOutStream (from the Pod::Simple dist), or this class won't work.

SEE ALSO

Pod::Simple::XMLOutStream, *Pod::Simple*, *Pod::Perldoc*

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2002 Sean M. Burke. All rights reserved.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Current maintainer: Mark Allen <mallen@cpan.org>

Past contributions from: brian d foy <bdfoy@cpan.org> Adriano R. Ferreira <ferreira@cpan.org>, Sean M. Burke <sburke@cpan.org>

Pod::Simple

framework for parsing Pod

SYNOPSIS

```
# using an existing formatter
use Pod::Simple::XHTML;
my $parser = Pod::Simple::XHTML->new;
$parser->parse_file('path/to/file.pod');

# creating a new formatter
package Pod::Simple::SomeFormatter;
use parent qw(Pod::Simple);
sub _handle_element_start {
    my ($parser, $element_name, $attr_hash_r) = @_;
    ...
}
...
```

DESCRIPTION

Pod::Simple is a Perl library for parsing text in the Pod ("plain old documentation") markup language that is typically used for writing documentation for Perl and for Perl modules. The Pod format is explained in perlpod; the most common formatter is called `perldoc`.

Be sure to read "ENCODING" if your Pod contains non-ASCII characters.

Pod formatters can use Pod::Simple to parse Pod documents and render them into plain text, HTML, or any number of other formats. Typically, such formatters will be subclasses of Pod::Simple, and so they will inherit its methods, like `parse_file`. But note that Pod::Simple doesn't understand and properly parse Perl itself, so if you have a file which contains a Perl program that has a multi-line quoted string which has lines that look like pod, Pod::Simple will treat them as pod. This can be avoided if the file makes these into indented here documents instead.

If you're reading this document just because you have a Pod-processing subclass that you want to use, this document (plus the documentation for the subclass) is probably all you need to read.

If you're reading this document because you want to write a formatter subclass, continue reading it and then read Pod::Simple::Subclassing, and then possibly even read perlpodspec (some of which is for parser-writers, but much of which is notes to formatter-writers).

MAIN METHODS

- `$parser = _SomeClass_->new();`

  This returns a new parser object, where `SomeClass` is a subclass of Pod::Simple.

- `$parser->output_fh( *OUT );`

  This sets the filehandle that `$parser`'s output will be written to. You can pass `*STDOUT` or `*STDERR`, otherwise you should probably do something like this:

```
my $outfile = "output.txt";
open TXTOUT, ">$outfile" or die "Can't write to $outfile: $!";
$parser->output_fh(*TXTOUT);
```

  ...before you call one of the `$parser->parse__whatever_` methods.

- `$parser->output_string( \$somestring );`

  This sets the string that `$parser`'s output will be sent to, instead of any filehandle.

- `$parser->parse_file( _$some_filename_ );`

- `$parser->parse_file( *INPUT_FH );`

  This reads the Pod content of the file (or filehandle) that you specify, and processes it with that `$parser` object, according to however `$parser`'s class works, and according to whatever parser options you have set up for this `$parser` object.

- `$parser->parse_string_document( _$all_content_ );`

  This works just like `parse_file` except that it reads the Pod content not from a file, but from a string that you have already in memory.

- `$parser->parse_lines( _...@lines..._, undef );`

  This processes the lines in `@lines` (where each list item must be a defined value, and must contain exactly one line of content – so no items like `"foo\nbar"` are allowed). The final `undef` is used to indicate the end of document being parsed.

  The other `parser__whatever_` methods are meant to be called only once per `$parser` object; but `parse_lines` can be called as many times per `$parser` ob-

ject as you want, as long as the last call (and only the last call) ends with an undef value.

- `$parser->content_seen`

  This returns true only if there has been any real content seen for this document. Returns false in cases where the document contains content, but does not make use of any Pod markup.

- `_SomeClass_->filter( _$filename_ );`
- `_SomeClass_->filter( _*INPUT_FH_ );`
- `_SomeClass_->filter( _\$document_content_ );`

  This is a shortcut method for creating a new parser object, setting the output handle to STDOUT, and then processing the specified file (or filehandle, or in-memory document). This is handy for one-liners like this:

```
perl -MPod::Simple::Text -e "Pod::Simple::Text->filter('thingy.
↪pod')"
```

SECONDARY METHODS

Some of these methods might be of interest to general users, as well as of interest to formatter-writers.

Note that the general pattern here is that the accessor-methods read the attribute's value with `$value = $parser->_attribute_` and set the attribute's value with `$parser->_attribute_(_newvalue_)`. For each accessor, I typically only mention one syntax or another, based on which I think you are actually most likely to use.

- `$parser->parse_characters( _SOMEVALUE_ )`

  The Pod parser normally expects to read octets and to convert those octets to characters based on the `=encoding` declaration in the Pod source. Set this option to a true value to indicate that the Pod source is already a Perl character stream. This tells the parser to ignore any `=encoding` command and to skip all the code paths involving decoding octets.

- `$parser->no_whining( _SOMEVALUE_ )`

  If you set this attribute to a true value, you will suppress the parser's complaints about irregularities in the Pod coding. By default, this attribute's value is false, meaning that irregularities will be reported.

  Note that turning this attribute to true won't suppress one or two kinds of complaints about rarely occurring unrecoverable errors.

- `$parser->no_errata_section( _SOMEVALUE_ )`

  If you set this attribute to a true value, you will stop the parser from generating a "POD ERRORS" section at the end of the document. By default, this attribute's value is false, meaning that an errata section will be generated, as necessary.

- `$parser->complain_stderr( _SOMEVALUE_ )`

If you set this attribute to a true value, it will send reports of parsing errors to STDERR. By default, this attribute's value is false, meaning that no output is sent to STDERR.

Setting `complain_stderr` also sets `no_errata_section`.

- `$parser->source_filename`

  This returns the filename that this parser object was set to read from.

- `$parser->doc_has_started`

  This returns true if `$parser` has read from a source, and has seen Pod content in it.

- `$parser->source_dead`

  This returns true if `$parser` has read from a source, and come to the end of that source.

- `$parser->strip_verbatim_indent( _SOMEVALUE_ )`

  The perlpod spec for a Verbatim paragraph is "It should be reproduced exactly...", which means that the whitespace you've used to indent your verbatim blocks will be preserved in the output. This can be annoying for outputs such as HTML, where that whitespace will remain in front of every line. It's an unfortunate case where syntax is turned into semantics.

  If the POD you're parsing adheres to a consistent indentation policy, you can have such indentation stripped from the beginning of every line of your verbatim blocks. This method tells Pod::Simple what to strip. For two-space indents, you'd use:

```
$parser->strip_verbatim_indent('  ');
```

  For tab indents, you'd use a tab character:

```
$parser->strip_verbatim_indent("\t");
```

  If the POD is inconsistent about the indentation of verbatim blocks, but you have figured out a heuristic to determine how much a particular verbatim block is indented, you can pass a code reference instead. The code reference will be executed with one argument, an array reference of all the lines in the verbatim block, and should return the value to be stripped from each line. For example, if you decide that you're fine to use the first line of the verbatim block to set the standard for indentation of the rest of the block, you can look at the first line and return the appropriate value, like so:

```
$new->strip_verbatim_indent(sub {
    my $lines = shift;
    (my $indent = $lines->[0]) =~ s/\S.*//;
    return $indent;
});
```

  If you'd rather treat each line individually, you can do that, too, by just transforming them in-place in the code reference and returning `undef`. Say that you don't want *any* lines indented. You can do something like this:

```
$new->strip_verbatim_indent(sub {
    my $lines = shift;
    sub { s/^\s+// for @{ $lines },
    return undef;
});
```

- `$parser->expand_verbatim_tabs( _n_ )`

  Default: 8

  If after any stripping of indentation in verbatim blocks, there remain tabs, this method call indicates what to do with them. `0` means leave them as tabs, any other number indicates that each tab is to be translated so as to have tab stops every n columns.

  This is independent of other methods (except that it operates after any verbatim input stripping is done).

  Like the other methods, the input parameter is not checked for validity. `undef` or containing non-digits has the same effect as 8.

TERTIARY METHODS

- `$parser->abandon_output_fh()`

  Cancel output to the file handle. Any POD read by the `$parser` is not effected.

- `$parser->abandon_output_string()`

  Cancel output to the output string. Any POD read by the `$parser` is not effected.

- `$parser->accept_code( @codes )`

  Alias for accept_codes.

- `$parser->accept_codes( @codes )`

  Allows `$parser` to accept a list of "Formatting Codes" in perlpod. This can be used to implement user-defined codes.

- `$parser->accept_directive_as_data( @directives )`

  Allows `$parser` to accept a list of directives for data paragraphs. A directive is the label of a "Command Paragraph" in perlpod. A data paragraph is one delimited by `=begin/=for/=end` directives. This can be used to implement user-defined directives.

- `$parser->accept_directive_as_processed( @directives )`

  Allows `$parser` to accept a list of directives for processed paragraphs. A directive is the label of a "Command Paragraph" in perlpod. A processed paragraph is also known as "Ordinary Paragraph" in perlpod. This can be used to implement user-defined directives.

- `$parser->accept_directive_as_verbatim( @directives )`

  Allows `$parser` to accept a list of directives for "Verbatim Paragraph" in perlpod. A directive is the label of a "Command Paragraph" in perlpod. This can be used to implement user-defined directives.

- `$parser->accept_target( @targets )`

  Alias for accept_targets.

- `$parser->accept_target_as_text( @targets )`

  Alias for accept_targets_as_text.

- `$parser->accept_targets( @targets )`

  Accepts targets for `=begin/=for/=end` sections of the POD.

- `$parser->accept_targets_as_text( @targets )`

  Accepts targets for `=begin/=for/=end` sections that should be parsed as POD. For details, see "About Data Paragraphs" in perlpodspec.

- `$parser->any_errata_seen()`

  Used to check if any errata was seen.

  Example:

```
die "too many errors\n" if $parser->any_errata_seen();
```

- `$parser->errata_seen()`

  Returns a hash reference of all errata seen, both whines and screams. The hash reference's keys are the line number and the value is an array reference of the errors for that line.

  Example:

```
if ( $parser->any_errata_seen() ) {
   $logger->log( $parser->errata_seen() );
}
```

- `$parser->detected_encoding()`

  Return the encoding corresponding to `=encoding`, but only if the encoding was recognized and handled.

- `$parser->encoding()`

  Return encoding of the document, even if the encoding is not correctly handled.

- `$parser->parse_from_file( $source, $to )`

  Parses from `$source` file to `$to` file. Similar to "parse_from_file" in Pod::Parser.

- `$parser->scream( @error_messages )`

  Log an error that can't be ignored.

- `$parser->unaccept_code( @codes )`

  Alias for unaccept_codes.

- `$parser->unaccept_codes( @codes )`

  Removes `@codes` as valid codes for the parse.

- `$parser->unaccept_directive( @directives )`

  Alias for unaccept_directives.

- `$parser->unaccept_directives( @directives )`

  Removes `@directives` as valid directives for the parse.

- `$parser->unaccept_target( @targets )`

  Alias for unaccept_targets.

- `$parser->unaccept_targets( @targets )`

  Removes `@targets` as valid targets for the parse.

- `$parser->version_report()`

  Returns a string describing the version.

- `$parser->whine( @error_messages )`

  Log an error unless `$parser->no_whining( TRUE );`.

ENCODING

The Pod::Simple parser expects to read **octets**. The parser will decode the octets into Perl's internal character string representation using the value of the `=encoding` declaration in the POD source.

If the POD source does not include an `=encoding` declaration, the parser will attempt to guess the encoding (selecting one of UTF-8 or CP 1252) by examining the first non-ASCII bytes and applying the heuristic described in perlpodspec. (If the POD source contains only ASCII bytes, the encoding is assumed to be ASCII.)

If you set the `parse_characters` option to a true value the parser will expect characters rather than octets; will ignore any `=encoding`; and will make no attempt to decode the input.

SEE ALSO

Pod::Simple::Subclassing

perlpod

perlpodspec

Pod::Escapes

perldoc

SUPPORT

Questions or discussion about POD and Pod::Simple should be sent to the pod-people@perl.org mail list. Send an empty email to pod-people-subscribe@perl.org to subscribe.

This module is managed in an open GitHub repository, https://github.com/perl-pod/pod-simple/. Feel free to fork and contribute, or to clone https://github.com/perl-pod/pod-simple.git and send patches!

Please use https://github.com/perl-pod/pod-simple/issues/new to file a bug report.

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2002 Sean M. Burke.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Pod::Simple was created by Sean M. Burke sburke@cpan.org. But don't bother him, he's retired.

Pod::Simple is maintained by:

- Allison Randal `allison@perl.org`
- Hans Dieter Pearcey `hdp@cpan.org`
- David E. Wheeler `dwheeler@cpan.org`
- Karl Williamson `khw@cpan.org`

Documentation has been contributed by:

- Gabor Szabo `szabgab@gmail.com`
- Shawn H Corey `SHCOREY at cpan.org`

Modules

- `Pod::Simple::Checker` - check the Pod syntax of a document
- `Pod::Simple::Debug` - put Pod::Simple into trace/debug mode
- `Pod::Simple::DumpAsText` - dump Pod-parsing events as text
- `Pod::Simple::DumpAsXML` - turn Pod into XML
- `Pod::Simple::HTML` - convert Pod to HTML
- `Pod::Simple::HTMLBatch` - convert several Pod files to several HTML files
- `Pod::Simple::JustPod` - just the Pod, the whole Pod, and nothing but the Pod
- `Pod::Simple::LinkSection` - represent "section" attributes of L codes
- `Pod::Simple::Methody` - turn Pod::Simple events into method calls
- `Pod::Simple::PullParser` - a pull-parser interface to parsing Pod
- `Pod::Simple::PullParserEndToken` - end-tokens from Pod::Simple::PullParser
- `Pod::Simple::PullParserStartToken` - start-tokens from Pod::Simple::PullParser

- `Pod::Simple::PullParserTextToken` - text-tokens from Pod::Simple::PullParser
- `Pod::Simple::PullParserToken` - tokens from Pod::Simple::PullParser
- `Pod::Simple::RTF` - format Pod as RTF
- `Pod::Simple::Search` - find POD documents in directory trees
- `Pod::Simple::SimpleTree` - parse Pod into a simple parse tree
- `Pod::Simple::Text` - format Pod as plaintext
- `Pod::Simple::TextContent` - get the text content of Pod
- `Pod::Simple::XHTML` - format Pod as validating XHTML
- `Pod::Simple::XMLOutStream` - turn Pod into XML

Pod::Simple::Checker

check the Pod syntax of a document

SYNOPSIS

```
perl -MPod::Simple::Checker -e \
 "exit Pod::Simple::Checker->filter(shift)->any_errata_seen" \
 thingy.pod
```

DESCRIPTION

This class is for checking the syntactic validity of Pod. It works by basically acting like a simple-minded version of *Pod::Simple::Text* that formats only the "Pod Errors" section (if Pod::Simple even generates one for the given document).

This is a subclass of *Pod::Simple* and inherits all its methods.

SEE ALSO

Pod::Simple, *Pod::Simple::Text*, *Pod::Checker*

SUPPORT

Questions or discussion about POD and Pod::Simple should be sent to the pod-people@perl.org mail list. Send an empty email to pod-people-subscribe@perl.org to subscribe.

This module is managed in an open GitHub repository, https://github.com/perl-pod/pod-simple/. Feel free to fork and contribute, or to clone https://github.com/perl-pod/pod-simple.git and send patches!

Patches against Pod::Simple are welcome. Please send bug reports to bug-pod-simple@rt.cpan.org.

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2002 Sean M. Burke.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Pod::Simple was created by Sean M. Burke sburke@cpan.org. But don't bother him, he's retired.

Pod::Simple is maintained by:

- Allison Randal `allison@perl.org`
- Hans Dieter Pearcey `hdp@cpan.org`
- David E. Wheeler `dwheeler@cpan.org`

Pod::Simple::Debug

put Pod::Simple into trace/debug mode

SYNOPSIS

```
use Pod::Simple::Debug (5);  # or some integer
```

Or:

```
my $debuglevel;
use Pod::Simple::Debug (\$debuglevel, 0);
...some stuff that uses Pod::Simple to do stuff, but which
 you don't want debug output from...

$debug_level = 4;
...some stuff that uses Pod::Simple to do stuff, but which
 you DO want debug output from...

$debug_level = 0;
```

DESCRIPTION

This is an internal module for controlling the debug level (a.k.a. trace level) of Pod::Simple. This is of interest only to Pod::Simple developers.

CAVEATS

Note that you should load this module *before* loading Pod::Simple (or any Pod::Simple-based class). If you try loading Pod::Simple::Debug after &Pod::Simple::DEBUG is already defined, Pod::Simple::Debug will throw a fatal error to the effect that "It's too late to call Pod::Simple::Debug".

Note that the `use Pod::Simple::Debug (\$x, _somenum_)` mode will make Pod::Simple (et al) run rather slower, since &Pod::Simple::DEBUG won't be a constant sub anymore, and so Pod::Simple (et al) won't compile with constant-folding.

GUTS

Doing this:

```
use Pod::Simple::Debug (5);  # or some integer
```

is basically equivalent to:

```
BEGIN { sub Pod::Simple::DEBUG () {5} }  # or some integer
use Pod::Simple ();
```

And this:

```
use Pod::Simple::Debug (\$debug_level,0);  # or some integer
```

is basically equivalent to this:

```
my $debug_level;
BEGIN { $debug_level = 0 }
BEGIN { sub Pod::Simple::DEBUG () { $debug_level }
use Pod::Simple ();
```

SEE ALSO

Pod::Simple

The article "Constants in Perl", in *The Perl Journal* issue 21. See http://interglacial.com/tpj/21/

SUPPORT

Questions or discussion about POD and Pod::Simple should be sent to the pod-people@perl.org mail list. Send an empty email to pod-people-subscribe@perl.org to subscribe.

This module is managed in an open GitHub repository, https://github.com/perl-pod/pod-simple/. Feel free to fork and contribute, or to clone https://github.com/perl-pod/pod-simple.git and send patches!

Patches against Pod::Simple are welcome. Please send bug reports to bug-pod-simple@rt.cpan.org.

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2002 Sean M. Burke.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Pod::Simple was created by Sean M. Burke sburke@cpan.org. But don't bother him, he's retired.

Pod::Simple is maintained by:

- Allison Randal `allison@perl.org`
- Hans Dieter Pearcey `hdp@cpan.org`
- David E. Wheeler `dwheeler@cpan.org`

Pod::Simple::DumpAsText

dump Pod-parsing events as text

SYNOPSIS

```
perl -MPod::Simple::DumpAsText -e \
 "exit Pod::Simple::DumpAsText->filter(shift)->any_errata_seen" \
 thingy.pod
```

DESCRIPTION

This class is for dumping, as text, the events gotten from parsing a Pod document. This class is of interest to people writing Pod formatters based on Pod::Simple. It is useful for seeing exactly what events you get out of some Pod that you feed in.

This is a subclass of *Pod::Simple* and inherits all its methods.

SEE ALSO

Pod::Simple::DumpAsXML

Pod::Simple

SUPPORT

Questions or discussion about POD and Pod::Simple should be sent to the pod-people@perl.org mail list. Send an empty email to pod-people-subscribe@perl.org to subscribe.

This module is managed in an open GitHub repository, https://github.com/perl-pod/pod-simple/. Feel free to fork and contribute, or to clone https://github.com/perl-pod/pod-simple.git and send patches!

Patches against Pod::Simple are welcome. Please send bug reports to bug-pod-simple@rt.cpan.org.

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2002 Sean M. Burke.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Pod::Simple was created by Sean M. Burke sburke@cpan.org. But don't bother him, he's retired.

Pod::Simple is maintained by:

- Allison Randal `allison@perl.org`
- Hans Dieter Pearcey `hdp@cpan.org`
- David E. Wheeler `dwheeler@cpan.org`

Pod::Simple::DumpAsXML

turn Pod into XML

SYNOPSIS

```
perl -MPod::Simple::DumpAsXML -e \
 "exit Pod::Simple::DumpAsXML->filter(shift)->any_errata_seen" \
 thingy.pod
```

DESCRIPTION

Pod::Simple::DumpAsXML is a subclass of *Pod::Simple* that parses Pod and turns it into indented and wrapped XML. This class is of interest to people writing Pod formatters based on Pod::Simple.

Pod::Simple::DumpAsXML inherits methods from *Pod::Simple*.

SEE ALSO

Pod::Simple::XMLOutStream is rather like this class. Pod::Simple::XMLOutStream's output is space-padded in a way that's better for sending to an XML processor (that is, it has no ignorable whitespace). But Pod::Simple::DumpAsXML's output is much more human-readable, being (more-or-less) one token per line, with line-wrapping.

Pod::Simple::DumpAsText is rather like this class, except that it doesn't dump with XML syntax. Try them and see which one you like best!

Pod::Simple, *Pod::Simple::DumpAsXML*

The older libraries Pod::PXML, Pod::XML, Pod::SAX

SUPPORT

Questions or discussion about POD and Pod::Simple should be sent to the pod-people@perl.org mail list. Send an empty email to pod-people-subscribe@perl.org to subscribe.

This module is managed in an open GitHub repository, https://github.com/perl-pod/pod-simple/. Feel free to fork and contribute, or to clone https://github.com/perl-pod/pod-simple.git and send patches!

Patches against Pod::Simple are welcome. Please send bug reports to bug-pod-simple@rt.cpan.org.

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2002 Sean M. Burke.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Pod::Simple was created by Sean M. Burke sburke@cpan.org. But don't bother him, he's retired.

Pod::Simple is maintained by:

- Allison Randal `allison@perl.org`
- Hans Dieter Pearcey `hdp@cpan.org`
- David E. Wheeler `dwheeler@cpan.org`

Pod::Simple::HTML

convert Pod to HTML

SYNOPSIS

```
perl -MPod::Simple::HTML -e Pod::Simple::HTML::go thingy.pod
```

DESCRIPTION

This class is for making an HTML rendering of a Pod document.

This is a subclass of *Pod::Simple::PullParser* and inherits all its methods (and options).

Note that if you want to do a batch conversion of a lot of Pod documents to HTML, you should see the module *Pod::Simple::HTMLBatch*.

CALLING FROM THE COMMAND LINE

TODO

```
perl -MPod::Simple::HTML -e Pod::Simple::HTML::go Thing.pod Thing.
 ↪html
```

CALLING FROM PERL

Minimal code

```
use Pod::Simple::HTML;
my $p = Pod::Simple::HTML->new;
$p->output_string(\my $html);
$p->parse_file('path/to/Module/Name.pm');
open my $out, '>', 'out.html' or die "Cannot open 'out.html': $!\n";
print $out $html;
```

More detailed example

```
use Pod::Simple::HTML;
```

Set the content type:

```
$Pod::Simple::HTML::Content_decl =  q{<meta http-equiv="Content-Type
 ↪" content="text/html; charset=UTF-8" >};

my $p = Pod::Simple::HTML->new;
```

Include a single javascript source:

```
$p->html_javascript('http://abc.com/a.js');
```

Or insert multiple javascript source in the header (or for that matter include anything, thought this is not recommended)

```
$p->html_javascript('
    <script type="text/javascript" src="http://abc.com/b.js"></
↪script>
    <script type="text/javascript" src="http://abc.com/c.js"></
↪script>');
```

Include a single css source in the header:

```
$p->html_css('/style.css');
```

or insert multiple css sources:

```
$p->html_css('
    <link rel="stylesheet" type="text/css" title="pod_stylesheet"␣
↪href="http://remote.server.com/jquery.css">
    <link rel="stylesheet" type="text/css" title="pod_stylesheet"␣
↪href="/style.css">');
```

Tell the parser where should the output go. In this case it will be placed in the $html variable:

```
my $html;
$p->output_string(\$html);
```

Parse and process a file with pod in it:

```
$p->parse_file('path/to/Module/Name.pm');
```

METHODS

TODO all (most?) accessorized methods

The following variables need to be set **before** the call to the ->new constructor.

Set the string that is included before the opening <html> tag:

```
$Pod::Simple::HTML::Doctype_decl = qq{<!DOCTYPE HTML PUBLIC "-//W3C/
↪/DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">\n};
```

Set the content-type in the HTML head: (defaults to ISO-8859-1)

```
$Pod::Simple::HTML::Content_decl =  q{<meta http-equiv="Content-Type
↪" content="text/html; charset=UTF-8" >};
```

Set the value that will be embedded in the opening tags of F, C tags and verbatim text. F maps to , C maps to <code>, Verbatim text maps to <pre> (Computerese defaults to “”)

```
$Pod::Simple::HTML::Computerese =  ' class="some_class_name'';
```

html_css

html_javascript

title_prefix

title_postfix

html_header_before_title

This includes everything before the <title> opening tag including the Document type and including the opening <title> tag. The following call will set it to be a simple HTML file:

```
$p->html_header_before_title('<html><head><title>');
```

top_anchor

By default Pod::Simple::HTML adds a dummy anchor at the top of the HTML. You can change it by calling

```
$p->top_anchor('<a name="zz" >');
```

html_h_level

Normally =head1 will become <h1>, =head2 will become <h2> etc. Using the html_h_level method will change these levels setting the h level of =head1 tags:

```
$p->html_h_level(3);
```

Will make sure that =head1 will become <h3> and =head2 will become <h4> etc...

index

Set it to some true value if you want to have an index (in reality a table of contents) to be added at the top of the generated HTML.

```
$p->index(1);
```

html_header_after_title

Includes the closing tag of </title> and through the rest of the head till the opening of the body

```
$p->html_header_after_title('</title>...</head><body id="my_id">');
```

html_footer

The very end of the document:

```
$p->html_footer( qq[\n<!-- end doc -->\n\n</body></html>\n] );
```

SUBCLASSING

Can use any of the methods described above but for further customization one needs to override some of the methods:

```
package My::Pod;
use strict;
use warnings;

use base 'Pod::Simple::HTML';

# needs to return a URL string such
# http://some.other.com/page.html
# #anchor_in_the_same_file
# /internal/ref.html
sub do_pod_link {
  # My::Pod object and Pod::Simple::PullParserStartToken object
  my ($self, $link) = @_;

  say $link->tagname;          # will be L for links
  say $link->attr('to');       #
  say $link->attr('type');     # will be 'pod' always
  say $link->attr('section');

  # Links local to our web site
  if ($link->tagname eq 'L' and $link->attr('type') eq 'pod') {
    my $to = $link->attr('to');
    if ($to =~ /^Padre::/) {
        $to =~ s{::}{/}g;
        return "/docs/Padre/$to.html";
    }
  }

  # all other links are generated by the parent class
  my $ret = $self->SUPER::do_pod_link($link);
  return $ret;
}

1;
```

Meanwhile in script.pl:

```
use My::Pod;

my $p = My::Pod->new;

my $html;
$p->output_string(\$html);
```

(continues on next page)

(continued from previous page)

```
$p->parse_file('path/to/Module/Name.pm');
open my $out, '>', 'out.html' or die;
print $out $html;
```

TODO

maybe override do_beginning do_end

SEE ALSO

Pod::Simple, *Pod::Simple::HTMLBatch*

TODO: a corpus of sample Pod input and HTML output? Or common idioms?

SUPPORT

Questions or discussion about POD and Pod::Simple should be sent to the pod-people@perl.org mail list. Send an empty email to pod-people-subscribe@perl.org to subscribe.

This module is managed in an open GitHub repository, https://github.com/perl-pod/pod-simple/. Feel free to fork and contribute, or to clone https://github.com/perl-pod/pod-simple.git and send patches!

Patches against Pod::Simple are welcome. Please send bug reports to bug-pod-simple@rt.cpan.org.

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2002-2004 Sean M. Burke.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Pod::Simple was created by Sean M. Burke sburke@cpan.org. But don't bother him, he's retired.

Pod::Simple is maintained by:

- Allison Randal `allison@perl.org`
- Hans Dieter Pearcey `hdp@cpan.org`
- David E. Wheeler `dwheeler@cpan.org`

Pod::Simple::HTMLBatch

convert several Pod files to several HTML files

SYNOPSIS

```
perl -MPod::Simple::HTMLBatch -e 'Pod::Simple::HTMLBatch::go' in out
```

DESCRIPTION

This module is used for running batch-conversions of a lot of HTML documents

This class is NOT a subclass of Pod::Simple::HTML (nor of bad old Pod::Html) – although it uses Pod::Simple::HTML for doing the conversion of each document.

The normal use of this class is like so:

```
use Pod::Simple::HTMLBatch;
my $batchconv = Pod::Simple::HTMLBatch->new;
$batchconv->some_option( some_value );
$batchconv->some_other_option( some_other_value );
$batchconv->batch_convert( \@search_dirs, $output_dir );
```

FROM THE COMMAND LINE

Note that this class also provides (but does not export) the function Pod::Simple::HTMLBatch::go. This is basically just a shortcut for `Pod::Simple::HTMLBatch->batch_convert(@ARGV)`. It's meant to be handy for calling from the command line.

However, the shortcut requires that you specify exactly two command-line arguments, `indirs` and `outdir`.

Example:

```
% mkdir out_html
% perl -MPod::Simple::HTMLBatch -e Pod::Simple::HTMLBatch::go @INC␣
↪out_html
    (to convert the pod from Perl's @INC
     files under the directory ./out_html)
```

(Note that the command line there contains a literal atsign-I-N-C. This is handled as a special case by batch_convert, in order to save you having to enter the odd-looking "" as the first command-line parameter when you mean "just use whatever's in @INC".)

Example:

```
% mkdir ../seekrut
% chmod og-rx ../seekrut
% perl -MPod::Simple::HTMLBatch -e Pod::Simple::HTMLBatch::go . ../
↪seekrut
    (to convert the pod under the current dir into HTML
     files under the directory ./seekrut)
```

Example:

```
% perl -MPod::Simple::HTMLBatch -e Pod::Simple::HTMLBatch::go␣
↪happydocs .
    (to convert all pod from happydocs into the current directory)
```

MAIN METHODS

- $batchconv = Pod::Simple::HTMLBatch->new;

  This creates a new batch converter. The method doesn't take parameters. To change the converter's attributes, use the "ACCESSOR METHODS" below.

- $batchconv->batch_convert(*indirs*, *outdir*);

  This searches the directories given in *indirs* and writes HTML files for each of these to a corresponding directory in *outdir*. The directory *outdir* must exist.

- $batchconv->batch_convert(undef , ...);

- $batchconv->batch_convert(q{@INC}, ...);

  These two values for *indirs* specify that the normal Perl @INC

- $batchconv->batch_convert(\@dirs , ...);

  This specifies that the input directories are the items in the arrayref `\@dirs`.

- $batchconv->batch_convert("somedir" , ...);

  This specifies that the director "somedir" is the input. (This can be an absolute or relative path, it doesn't matter.)

  A common value you might want would be just "." for the current directory:

  ```
  $batchconv->batch_convert( "." , ...);
  ```

- $batchconv->batch_convert('somedir:someother:also' , ...);

  This specifies that you want the dirs "somedir", "someother", and "also" scanned, just as if you'd passed the arrayref `[qw( somedir someother also)]`. Note that a ":"-separator is normal under Unix, but Under MSWin, you'll need `'somedir;someother;also'` instead, since the pathsep on MSWin is ";" instead of ":". (And *that* is because ":" often comes up in paths, like `"c:/perl/lib"`.)

  (Exactly what separator character should be used, is gotten from `$Config::Config{'path_sep'}`, via the *Config* module.)

- $batchconv->batch_convert(... , undef);

  This specifies that you want the HTML output to go into the current directory.

  (Note that a missing or undefined value means a different thing in the first slot than in the second. That's so that `batch_convert()` with no arguments (or undef arguments) means "go from @INC, into the current directory.)

- $batchconv->batch_convert(... , 'somedir');

  This specifies that you want the HTML output to go into the directory 'somedir'. (This can be an absolute or relative path, it doesn't matter.)

Note that you can also call `batch_convert` as a class method, like so:

```
Pod::Simple::HTMLBatch->batch_convert( ... );
```

That is just short for this:

```
Pod::Simple::HTMLBatch-> new-> batch_convert(...);
```

That is, it runs a conversion with default options, for whatever inputdirs and output dir you specify.

ACCESSOR METHODS

The following are all accessor methods – that is, they don't do anything on their own, but just alter the contents of the conversion object, which comprises the options for this particular batch conversion.

We show the "put" form of the accessors below (i.e., the syntax you use for setting the accessor to a specific value). But you can also call each method with no parameters to get its current value. For example, `$self->contents_file()` returns the current value of the contents_file attribute.

- $batchconv->verbose(*nonnegative_integer*);

  This controls how verbose to be during batch conversion, as far as notes to STDOUT (or whatever is `select`'d) about how the conversion is going. If 0, no progress information is printed. If 1 (the default value), some progress information is printed. Higher values print more information.

- $batchconv->index(*true-or-false*);

  This controls whether or not each HTML page is liable to have a little table of contents at the top (which we call an "index" for historical reasons). This is true by default.

- $batchconv->contents_file(*filename*);

  If set, should be the name of a file (in the output directory) to write the HTML index to. The default value is "index.html". If you set this to a false value, no contents file will be written.

- $batchconv->contents_page_start(*HTML_string*);

  This specifies what string should be put at the beginning of the contents page. The default is a string more or less like this:

  ```
  <html>
  <head><title>Perl Documentation</title></head>
  <body class='contentspage'>
  <h1>Perl Documentation</h1>
  ```

- $batchconv->contents_page_end(*HTML_string*);

  This specifies what string should be put at the end of the contents page. The default is a string more or less like this:

```
<p class='contentsfooty'>Generated by
Pod::Simple::HTMLBatch v3.01 under Perl v5.008
<br >At Fri May 14 22:26:42 2004 GMT,
which is Fri May 14 14:26:42 2004 local time.</p>
```

- $batchconv->add_css($url);

  TODO

- $batchconv->add_javascript($url);

  TODO

- $batchconv->css_flurry(*true-or-false*);

  If true (the default value), we autogenerate some CSS files in the output directory, and set our HTML files to use those. TODO: continue

- $batchconv->javascript_flurry(*true-or-false*);

  If true (the default value), we autogenerate a JavaScript in the output directory, and set our HTML files to use it. Currently, the JavaScript is used only to get the browser to remember what stylesheet it prefers. TODO: continue

- $batchconv->no_contents_links(*true-or-false*);

  TODO

- $batchconv->html_render_class(*classname*);

  This sets what class is used for rendering the files. The default is "Pod::Simple::HTML". If you set it to something else, it should probably be a subclass of Pod::Simple::HTML, and you should `require` or `use` that class so that's it's loaded before Pod::Simple::HTMLBatch tries loading it.

- $batchconv->search_class(*classname*);

  This sets what class is used for searching for the files. The default is "Pod::Simple::Search". If you set it to something else, it should probably be a subclass of Pod::Simple::Search, and you should `require` or `use` that class so that's it's loaded before Pod::Simple::HTMLBatch tries loading it.

NOTES ON CUSTOMIZATION

TODO

```
call add_css($someurl) to add stylesheet as alternate
call add_css($someurl,1) to add as primary stylesheet

call add_javascript

subclass Pod::Simple::HTML and set $batchconv->html_render_class to
  that classname
and maybe override
  $page->batch_mode_page_object_init($self, $module, $infile,
↪$outfile, $depth)
```

(continues on next page)

(continued from previous page)

```
or maybe override
  $batchconv->batch_mode_page_object_init($page, $module, $infile,
↪$outfile, $depth)
subclass Pod::Simple::Search and set $batchconv->search_class to
  that classname
```

SEE ALSO

Pod::Simple, *Pod::Simple::HTMLBatch*, perlpod, perlpodspec

SUPPORT

Questions or discussion about POD and Pod::Simple should be sent to the pod-people@perl.org mail list. Send an empty email to pod-people-subscribe@perl.org to subscribe.

This module is managed in an open GitHub repository, https://github.com/perl-pod/pod-simple/. Feel free to fork and contribute, or to clone https://github.com/perl-pod/pod-simple.git and send patches!

Patches against Pod::Simple are welcome. Please send bug reports to bug-pod-simple@rt.cpan.org.

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2002 Sean M. Burke.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Pod::Simple was created by Sean M. Burke sburke@cpan.org. But don't bother him, he's retired.

Pod::Simple is maintained by:

- Allison Randal `allison@perl.org`
- Hans Dieter Pearcey `hdp@cpan.org`
- David E. Wheeler `dwheeler@cpan.org`

Pod::Simple::JustPod

just the Pod, the whole Pod, and nothing but the Pod

SYNOPSIS

```
my $infile  = "mixed_code_and_pod.pm";
my $outfile = "just_the_pod.pod";
open my $fh, ">$outfile" or die "Can't write to $outfile: $!";

my $parser = Pod::Simple::JustPod->new();
$parser->output_fh($fh);
$parser->parse_file($infile);
close $fh or die "Can't close $outfile: $!";
```

DESCRIPTION

This class returns a copy of its input, translated into Perl's internal encoding (UTF-8), and with all the non-Pod lines removed.

This is a subclass of *Pod::Simple::Methody* and inherits all its methods. And since, that in turn is a subclass of *Pod::Simple*, you can use any of its methods. This means you can output to a string instead of a file, or you can parse from an array.

This class strives to return the Pod lines of the input completely unchanged, except for any necessary translation into Perl's internal encoding, and it makes no effort to return trailing spaces on lines; these likely will be stripped. If the input pod is well-formed with no warnings nor errors generated, the extracted pod should generate the same documentation when formatted by a Pod formatter as the original file does.

By default, warnings are output to STDERR

SEE ALSO

Pod::Simple, *Pod::Simple::Methody*

SUPPORT

Questions or discussion about POD and Pod::Simple should be sent to the mailto:pod-people@perl.org mail list. Send an empty email to mailto:pod-people-subscribe@perl.org to subscribe.

This module is managed in an open GitHub repository, https://github.com/theory/pod-simple/. Feel free to fork and contribute, or to clone https://github.com/theory/pod-simple.git and send patches!

Patches against Pod::Simple are welcome. Please send bug reports to mailto:<bug-pod-simple@rt.cpan.org.

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2002 Sean M. Burke.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Pod::Simple was created by Sean M. Burke sburke@cpan.org. But don't bother him, he's retired.

Pod::Simple is maintained by:

- Allison Randal `allison@perl.org`
- Hans Dieter Pearcey `hdp@cpan.org`
- David E. Wheeler `dwheeler@cpan.org`

Pod::Simple::JustPod was developed by John SJ Anderson `genehack@genehack.org`, with contributions from Karl Williamson `khw@cpan.org`.

Pod::Simple::LinkSection

represent "section" attributes of L codes

SYNOPSIS

```
# a long story
```

DESCRIPTION

This class is not of interest to general users.

Pod::Simple uses this class for representing the value of the "section" attribute of "L" start-element events. Most applications can just use the normal stringification of objects of this class; they stringify to just the text content of the section, such as "foo" for `L<Stuff/foo>`, and "bar" for `L<Stuff/bI<ar>>`.

However, anyone particularly interested in getting the full value of the treelet, can just traverse the content of the treeleet @$treelet_object. To wit:

```
  % perl -MData::Dumper -e
    "use base qw(Pod::Simple::Methody);
     sub start_L { print Dumper($_[1]{'section'} ) }
     __PACKAGE__->new->parse_string_document('=head1 L<Foo/bI<ar>
↪baz>>')
    "
Output:
  $VAR1 = bless( [
                   '',
                   {},
                   'b',
                   bless( [
                            'I',
```

(continues on next page)

(continued from previous page)

```
                                {},
                                'ar'
                              ], 'Pod::Simple::LinkSection' ),
                    'baz'
                  ], 'Pod::Simple::LinkSection' );
```

But stringify it and you get just the text content:

```
  % perl -MData::Dumper -e
    "use base qw(Pod::Simple::Methody);
     sub start_L { print Dumper( '' . $_[1]{'section'} ) }
     __PACKAGE__->new->parse_string_document('=head1 L<Foo/bI<ar>
↪baz>>')
    "
Output:
  $VAR1 = 'barbaz';
```

SEE ALSO

Pod::Simple

SUPPORT

Questions or discussion about POD and Pod::Simple should be sent to the pod-people@perl.org mail list. Send an empty email to pod-people-subscribe@perl.org to subscribe.

This module is managed in an open GitHub repository, https://github.com/perl-pod/pod-simple/. Feel free to fork and contribute, or to clone https://github.com/perl-pod/pod-simple.git and send patches!

Patches against Pod::Simple are welcome. Please send bug reports to bug-pod-simple@rt.cpan.org.

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2004 Sean M. Burke.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Pod::Simple was created by Sean M. Burke sburke@cpan.org. But don't bother him, he's retired.

Pod::Simple is maintained by:

- Allison Randal allison@perl.org
- Hans Dieter Pearcey hdp@cpan.org
- David E. Wheeler dwheeler@cpan.org

Pod::Simple::Methody

turn Pod::Simple events into method calls

SYNOPSIS

```
require 5;
use strict;
package SomePodFormatter;
use base qw(Pod::Simple::Methody);

sub handle_text {
  my($self, $text) = @_;
  ...
}

sub start_head1 {
  my($self, $attrs) = @_;
  ...
}
sub end_head1 {
  my($self) = @_;
  ...
}
```

...and start_/end_ methods for whatever other events you want to catch.

DESCRIPTION

This class is of interest to people writing Pod formatters based on Pod::Simple.

This class (which is very small – read the source) overrides Pod::Simple's _handle_element_start, _handle_text, and _handle_element_end methods so that parser events are turned into method calls. (Otherwise, this is a subclass of *Pod::Simple* and inherits all its methods.)

You can use this class as the base class for a Pod formatter/processor.

METHOD CALLING

When Pod::Simple sees a "=head1 Hi there", for example, it basically does this:

```
$parser->_handle_element_start( "head1", \%attributes );
$parser->_handle_text( "Hi there" );
$parser->_handle_element_end( "head1" );
```

But if you subclass Pod::Simple::Methody, it will instead do this when it sees a "=head1 Hi there":

```
$parser->start_head1( \%attributes ) if $parser->can('start_head1');
$parser->handle_text( "Hi there" )   if $parser->can('handle_text');
$parser->end_head1()                 if $parser->can('end_head1');
```

If Pod::Simple sends an event where the element name has a dash, period, or colon, the corresponding method name will have a underscore in its place. For example, "foo.bar:baz" becomes start_foo_bar_baz and end_foo_bar_baz.

See the source for Pod::Simple::Text for an example of using this class.

SEE ALSO

Pod::Simple, Pod::Simple::Subclassing

SUPPORT

Questions or discussion about POD and Pod::Simple should be sent to the pod-people@perl.org mail list. Send an empty email to pod-people-subscribe@perl.org to subscribe.

This module is managed in an open GitHub repository, https://github.com/perl-pod/pod-simple/. Feel free to fork and contribute, or to clone https://github.com/perl-pod/pod-simple.git and send patches!

Patches against Pod::Simple are welcome. Please send bug reports to bug-pod-simple@rt.cpan.org.

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2002 Sean M. Burke.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Pod::Simple was created by Sean M. Burke sburke@cpan.org. But don't bother him, he's retired.

Pod::Simple is maintained by:

- Allison Randal `allison@perl.org`
- Hans Dieter Pearcey `hdp@cpan.org`
- David E. Wheeler `dwheeler@cpan.org`

Pod::Simple::PullParser

a pull-parser interface to parsing Pod

SYNOPSIS

```
my $parser = SomePodProcessor->new;
$parser->set_source( "whatever.pod" );
$parser->run;
```

Or:

```
my $parser = SomePodProcessor->new;
$parser->set_source( $some_filehandle_object );
$parser->run;
```

Or:

```
my $parser = SomePodProcessor->new;
$parser->set_source( \$document_source );
$parser->run;
```

Or:

```
my $parser = SomePodProcessor->new;
$parser->set_source( \@document_lines );
$parser->run;
```

And elsewhere:

```
require 5;
package SomePodProcessor;
use strict;
use base qw(Pod::Simple::PullParser);

sub run {
  my $self = shift;
 Token:
  while(my $token = $self->get_token) {
    ...process each token...
  }
}
```

DESCRIPTION

This class is for using Pod::Simple to build a Pod processor – but one that uses an interface based on a stream of token objects, instead of based on events.

This is a subclass of *Pod::Simple* and inherits all its methods.

A subclass of Pod::Simple::PullParser should define a `run` method that calls `$token = $parser->get_token` to pull tokens.

See the source for Pod::Simple::RTF for an example of a formatter that uses Pod::Simple::PullParser.

METHODS

- my $token = $parser->get_token

  This returns the next token object (which will be of a subclass of *Pod::Simple::PullParserToken*), or undef if the parser-stream has hit the end of the document.

- $parser->unget_token($token)
- $parser->unget_token($token1, $token2, ...)

  This restores the token object(s) to the front of the parser stream.

The source has to be set before you can parse anything. The lowest-level way is to call `set_source`:

- $parser->set_source($filename)
- $parser->set_source($filehandle_object)
- $parser->set_source(\$document_source)
- $parser->set_source(\@document_lines)

Or you can call these methods, which Pod::Simple::PullParser has defined to work just like Pod::Simple's same-named methods:

- $parser->parse_file(...)
- $parser->parse_string_document(...)
- $parser->filter(...)
- $parser->parse_from_file(...)

For those to work, the Pod-processing subclass of Pod::Simple::PullParser has to have defined a $parser->run method – so it is advised that all Pod::Simple::PullParser subclasses do so. See the Synopsis above, or the source for Pod::Simple::RTF.

Authors of formatter subclasses might find these methods useful to call on a parser object that you haven't started pulling tokens from yet:

- my $title_string = $parser->get_title

  This tries to get the title string out of $parser, by getting some tokens, and scanning them for the title, and then ungetting them so that you can process the token-stream from the beginning.

  For example, suppose you have a document that starts out:

```
=head1 NAME

Hoo::Boy::Wowza -- Stuff B<wow> yeah!
```

  $parser->get_title on that document will return "Hoo::Boy::Wowza – Stuff wow yeah!". If the document starts with:

```
=head1 Name

Hoo::Boy::W00t -- Stuff B<w00t> yeah!
```

Then you'll need to pass the nocase option in order to recognize "Name":

```
$parser->get_title(nocase => 1);
```

In cases where get_title can't find the title, it will return empty-string ("").

- my $title_string = $parser->get_short_title

  This is just like get_title, except that it returns just the modulename, if the title seems to be of the form "SomeModuleName – description".

  For example, suppose you have a document that starts out:

```
=head1 NAME

Hoo::Boy::Wowza -- Stuff B<wow> yeah!
```

  then $parser->get_short_title on that document will return "Hoo::Boy::Wowza".

  But if the document starts out:

```
=head1 NAME

Hooboy, stuff B<wow> yeah!
```

  then $parser->get_short_title on that document will return "Hooboy, stuff wow yeah!". If the document starts with:

```
=head1 Name

Hoo::Boy::W00t -- Stuff B<w00t> yeah!
```

  Then you'll need to pass the nocase option in order to recognize "Name":

```
$parser->get_short_title(nocase => 1);
```

  If the title can't be found, then get_short_title returns empty-string ("").

- $author_name = $parser->get_author

  This works like get_title except that it returns the contents of the "=head1 AUTHOR\n\nParagraph...\n" section, assuming that that section isn't terribly long. To recognize a "=head1 Author\n\nParagraph\n" section, pass the nocase option:

```
$parser->get_author(nocase => 1);
```

  (This method tolerates "AUTHORS" instead of "AUTHOR" too.)

- $description_name = $parser->get_description

  This works like get_title except that it returns the contents of the "=head1 DESCRIPTION\n\nParagraph...\n" section, assuming that that section isn't terribly long. To

recognize a "=head1 Description\n\nParagraph\n" section, pass the `nocase` option:

```
$parser->get_description(nocase => 1);
```

- $version_block = $parser->get_version

  This works like get_title except that it returns the contents of the "=head1 VERSION\n\n[BIG BLOCK]\n" block. Note that this does NOT return the module's `$VERSION`!! To recognize a "=head1 Version\n\n[BIG BLOCK]\n" section, pass the `nocase` option:

```
$parser->get_version(nocase => 1);
```

NOTE

You don't actually *have* to define a `run` method. If you're writing a Pod-formatter class, you should define a `run` just so that users can call `parse_file` etc, but you don't *have* to.

And if you're not writing a formatter class, but are instead just writing a program that does something simple with a Pod::PullParser object (and not an object of a subclass), then there's no reason to bother subclassing to add a `run` method.

SEE ALSO

Pod::Simple

Pod::Simple::PullParserToken – and its subclasses *Pod::Simple::PullParserStartToken*, *Pod::Simple::PullParserTextToken*, and *Pod::Simple::PullParserEndToken*.

HTML::TokeParser, which inspired this.

SUPPORT

Questions or discussion about POD and Pod::Simple should be sent to the pod-people@perl.org mail list. Send an empty email to pod-people-subscribe@perl.org to subscribe.

This module is managed in an open GitHub repository, https://github.com/perl-pod/pod-simple/. Feel free to fork and contribute, or to clone https://github.com/perl-pod/pod-simple.git and send patches!

Patches against Pod::Simple are welcome. Please send bug reports to bug-pod-simple@rt.cpan.org.

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2002 Sean M. Burke.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Pod::Simple was created by Sean M. Burke sburke@cpan.org. But don't bother him, he's retired.

Pod::Simple is maintained by:

- Allison Randal `allison@perl.org`
- Hans Dieter Pearcey `hdp@cpan.org`
- David E. Wheeler `dwheeler@cpan.org`

Pod::Simple::PullParserEndToken

end-tokens from Pod::Simple::PullParser

SYNOPSIS

(See *Pod::Simple::PullParser*)

DESCRIPTION

When you do $parser->get_token on a *Pod::Simple::PullParser*, you might get an object of this class.

This is a subclass of *Pod::Simple::PullParserToken* and inherits all its methods, and adds these methods:

- $token->tagname

  This returns the tagname for this end-token object. For example, parsing a "=head1 ..." line will give you a start-token with the tagname of "head1", token(s) for its content, and then an end-token with the tagname of "head1".

- $token->tagname(*somestring*)

  This changes the tagname for this end-token object. You probably won't need to do this.

- $token->tag(...)

  A shortcut for $token->tagname(...)

- $token->is_tag(*somestring*) or $token->is_tagname(*somestring*)

  These are shortcuts for `$token->tag() eq _somestring_`

You're unlikely to ever need to construct an object of this class for yourself, but if you want to, call `Pod::Simple::PullParserEndToken->new( _tagname_ )`

SEE ALSO

Pod::Simple::PullParserToken, *Pod::Simple*, Pod::Simple::Subclassing

SUPPORT

Questions or discussion about POD and Pod::Simple should be sent to the pod-people@perl.org mail list. Send an empty email to pod-people-subscribe@perl.org to subscribe.

This module is managed in an open GitHub repository, https://github.com/perl-pod/pod-simple/. Feel free to fork and contribute, or to clone https://github.com/perl-pod/pod-simple.git and send patches!

Patches against Pod::Simple are welcome. Please send bug reports to bug-pod-simple@rt.cpan.org.

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2002 Sean M. Burke.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Pod::Simple was created by Sean M. Burke sburke@cpan.org. But don't bother him, he's retired.

Pod::Simple is maintained by:

- Allison Randal `allison@perl.org`
- Hans Dieter Pearcey `hdp@cpan.org`
- David E. Wheeler `dwheeler@cpan.org`

Pod::Simple::PullParserStartToken

start-tokens from Pod::Simple::PullParser

SYNOPSIS

(See *Pod::Simple::PullParser*)

DESCRIPTION

When you do $parser->get_token on a *Pod::Simple::PullParser* object, you might get an object of this class.

This is a subclass of *Pod::Simple::PullParserToken* and inherits all its methods, and adds these methods:

- $token->tagname

This returns the tagname for this start-token object. For example, parsing a "=head1 ..." line will give you a start-token with the tagname of "head1", token(s) for its content, and then an end-token with the tagname of "head1".

- $token->tagname(*somestring*)

  This changes the tagname for this start-token object. You probably won't need to do this.

- $token->tag(...)

  A shortcut for $token->tagname(...)

- $token->is_tag(*somestring*) or $token->is_tagname(*somestring*)

  These are shortcuts for `$token->tag() eq _somestring_`

- $token->attr(*attrname*)

  This returns the value of the *attrname* attribute for this start-token object, or undef.

  For example, parsing a L<Foo/"Bar"> link will produce a start-token with a "to" attribute with the value "Foo", a "type" attribute with the value "pod", and a "section" attribute with the value "Bar".

- $token->attr(*attrname*, *newvalue*)

  This sets the *attrname* attribute for this start-token object to *newvalue*. You probably won't need to do this.

- $token->attr_hash

  This returns the hashref that is the attribute set for this start-token. This is useful if (for example) you want to ask what all the attributes are – you can just do `keys %{$token->attr_hash}`

You're unlikely to ever need to construct an object of this class for yourself, but if you want to, call `Pod::Simple::PullParserStartToken->new( _tagname_, _attrhash_ )`

SEE ALSO

Pod::Simple::PullParserToken, *Pod::Simple*, Pod::Simple::Subclassing

SEE ALSO

Pod::Simple::PullParserToken, *Pod::Simple*, Pod::Simple::Subclassing

SUPPORT

Questions or discussion about POD and Pod::Simple should be sent to the pod-people@perl.org mail list. Send an empty email to pod-people-subscribe@perl.org to subscribe.

This module is managed in an open GitHub repository, https://github.com/perl-pod/pod-simple/. Feel free to fork and contribute, or to clone https://github.com/perl-pod/pod-simple.git and send patches!

Patches against Pod::Simple are welcome. Please send bug reports to bug-pod-simple@rt.cpan.org.

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2002 Sean M. Burke.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Pod::Simple was created by Sean M. Burke sburke@cpan.org. But don't bother him, he's retired.

Pod::Simple is maintained by:

- Allison Randal `allison@perl.org`
- Hans Dieter Pearcey `hdp@cpan.org`
- David E. Wheeler `dwheeler@cpan.org`

Pod::Simple::PullParserTextToken

text-tokens from Pod::Simple::PullParser

SYNOPSIS

(See *Pod::Simple::PullParser*)

DESCRIPTION

When you do $parser->get_token on a *Pod::Simple::PullParser*, you might get an object of this class.

This is a subclass of *Pod::Simple::PullParserToken* and inherits all its methods, and adds these methods:

- $token->text

  This returns the text that this token holds. For example, parsing C<foo> will return a C start-token, a text-token, and a C end-token. And if you want to get the "foo" out of the text-token, call `$token->text`

- $token->text(*somestring*)

  This changes the string that this token holds. You probably won't need to do this.

- $token->text_r()

This returns a scalar reference to the string that this token holds. This can be useful if you don't want to memory-copy the potentially large text value (well, as large as a paragraph or a verbatim block) as calling $token->text would do.

Or, if you want to alter the value, you can even do things like this:

```
for ( ${  $token->text_r  } ) {  # Aliases it with $_ !!

  s/ The / the /g; # just for example

  if( 'A' eq chr(65) ) {  # (if in an ASCII world)
    tr/\xA0/ /;
    tr/\xAD//d;
  }

  ...or however you want to alter the value...
  (Note that starting with Perl v5.8, you can use, e.g.,

      my $nbsp = chr utf8::unicode_to_native(0xA0);
      s/$nbsp/ /g;

  to handle the above regardless if it's an ASCII world or not)
}
```

You're unlikely to ever need to construct an object of this class for yourself, but if you want to, call `Pod::Simple::PullParserTextToken->new( _text_ )`

SEE ALSO

Pod::Simple::PullParserToken, *Pod::Simple*, Pod::Simple::Subclassing

SUPPORT

Questions or discussion about POD and Pod::Simple should be sent to the pod-people@perl.org mail list. Send an empty email to pod-people-subscribe@perl.org to subscribe.

This module is managed in an open GitHub repository, https://github.com/perl-pod/pod-simple/. Feel free to fork and contribute, or to clone https://github.com/perl-pod/pod-simple.git and send patches!

Patches against Pod::Simple are welcome. Please send bug reports to bug-pod-simple@rt.cpan.org.

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2002 Sean M. Burke.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Pod::Simple was created by Sean M. Burke sburke@cpan.org. But don't bother him, he's retired.

Pod::Simple is maintained by:

- Allison Randal `allison@perl.org`
- Hans Dieter Pearcey `hdp@cpan.org`
- David E. Wheeler `dwheeler@cpan.org`

Pod::Simple::PullParserToken

tokens from Pod::Simple::PullParser

SYNOPSIS

Given a $parser that's an object of class Pod::Simple::PullParser (or a subclass)...

```
while(my $token = $parser->get_token) {
  $DEBUG and print STDERR "Token: ", $token->dump, "\n";
  if($token->is_start) {
    ...access $token->tagname, $token->attr, etc...

  } elsif($token->is_text) {
    ...access $token->text, $token->text_r, etc...

  } elsif($token->is_end) {
    ...access $token->tagname...

  }
}
```

(Also see *Pod::Simple::PullParser*)

DESCRIPTION

When you do $parser->get_token on a *Pod::Simple::PullParser*, you should get an object of a subclass of Pod::Simple::PullParserToken.

Subclasses will add methods, and will also inherit these methods:

- $token->type

  This returns the type of the token. This will be either the string "start", the string "text", or the string "end".

  Once you know what the type of an object is, you then know what subclass it belongs to, and therefore what methods it supports.

  Yes, you could probably do the same thing with code like $token->isa('Pod::Simple::PullParserEndToken'), but that's not so pretty as using just $token->type, or even the following shortcuts:

- $token->is_start

  This is a shortcut for `$token->type() eq "start"`

- $token->is_text

  This is a shortcut for `$token->type() eq "text"`

- $token->is_end

  This is a shortcut for `$token->type() eq "end"`

- $token->dump

  This returns a handy stringified value of this object. This is useful for debugging, as in:

```
while(my $token = $parser->get_token) {
  $DEBUG and print STDERR "Token: ", $token->dump, "\n";
  ...
}
```

SEE ALSO

My subclasses: *Pod::Simple::PullParserStartToken*, *Pod::Simple::PullParserTextToken*, and *Pod::Simple::PullParserEndToken*.

Pod::Simple::PullParser and *Pod::Simple*

SUPPORT

Questions or discussion about POD and Pod::Simple should be sent to the pod-people@perl.org mail list. Send an empty email to pod-people-subscribe@perl.org to subscribe.

This module is managed in an open GitHub repository, https://github.com/perl-pod/pod-simple/. Feel free to fork and contribute, or to clone https://github.com/perl-pod/pod-simple.git and send patches!

Patches against Pod::Simple are welcome. Please send bug reports to bug-pod-simple@rt.cpan.org.

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2002 Sean M. Burke.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Pod::Simple was created by Sean M. Burke sburke@cpan.org. But don't bother him, he's retired.

Pod::Simple is maintained by:

- Allison Randal `allison@perl.org`
- Hans Dieter Pearcey `hdp@cpan.org`
- David E. Wheeler `dwheeler@cpan.org`

Pod::Simple::RTF

format Pod as RTF

SYNOPSIS

```
perl -MPod::Simple::RTF -e \
 "exit Pod::Simple::RTF->filter(shift)->any_errata_seen" \
 thingy.pod > thingy.rtf
```

DESCRIPTION

This class is a formatter that takes Pod and renders it as RTF, good for viewing/printing in MSWord, WordPad/write.exe, TextEdit, etc.

This is a subclass of *Pod::Simple* and inherits all its methods.

FORMAT CONTROL ATTRIBUTES

You can set these attributes on the parser object before you call `parse_file` (or a similar method) on it:

- $parser->head1_halfpoint_size(*halfpoint_integer*);
- $parser->head2_halfpoint_size(*halfpoint_integer*);
- $parser->head3_halfpoint_size(*halfpoint_integer*);
- $parser->head4_halfpoint_size(*halfpoint_integer*);

  These methods set the size (in half-points, like 52 for 26-point) that these heading levels will appear as.

- $parser->codeblock_halfpoint_size(*halfpoint_integer*);

  This method sets the size (in half-points, like 21 for 10.5-point) that codeblocks ("verbatim sections") will appear as.

- $parser->header_halfpoint_size(*halfpoint_integer*);

  This method sets the size (in half-points, like 15 for 7.5-point) that the header on each page will appear in. The header is usually just "*modulename* p. *pagenumber*".

- $parser->normal_halfpoint_size(*halfpoint_integer*);

  This method sets the size (in half-points, like 26 for 13-point) that normal paragraphic text will appear in.

- $parser->no_proofing_exemptions(*true_or_false*);

  Set this value to true if you don't want the formatter to try putting a hidden code on all Perl symbols (as best as it can notice them) that labels them as being not in English, and so not worth spellchecking.

- $parser->doc_lang(*microsoft_decimal_language_code*)

  This sets the language code to tag this document as being in. By default, it is currently the value of the environment variable `RTFDEFLANG`, or if that's not set, then the value 1033 (for US English).

  Setting this appropriately is useful if you want to use the RTF to spellcheck, and/or if you want it to hyphenate right.

  Here are some notable values:

```
1033  US English
2057  UK English
3081  Australia English
4105  Canada English
1034  Spain Spanish
2058  Mexico Spanish
1031  Germany German
1036  France French
3084  Canada French
1035  Finnish
1044  Norwegian (Bokmal)
2068  Norwegian (Nynorsk)
```

If you are particularly interested in customizing this module's output even more, see the source and/or write to me.

SEE ALSO

Pod::Simple, RTF::Writer, RTF::Cookbook, RTF::Document, RTF::Generator

SUPPORT

Questions or discussion about POD and Pod::Simple should be sent to the pod-people@perl.org mail list. Send an empty email to pod-people-subscribe@perl.org to subscribe.

This module is managed in an open GitHub repository, https://github.com/perl-pod/pod-simple/. Feel free to fork and contribute, or to clone https://github.com/perl-pod/pod-simple.git and send patches!

Patches against Pod::Simple are welcome. Please send bug reports to bug-pod-simple@rt.cpan.org.

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2002 Sean M. Burke.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Pod::Simple was created by Sean M. Burke sburke@cpan.org. But don't bother him, he's retired.

Pod::Simple is maintained by:

- Allison Randal `allison@perl.org`
- Hans Dieter Pearcey `hdp@cpan.org`
- David E. Wheeler `dwheeler@cpan.org`

Pod::Simple::Search

find POD documents in directory trees

SYNOPSIS

```
use Pod::Simple::Search;
my $name2path = Pod::Simple::Search->new->limit_glob('LWP::*')->
 ↪survey;
print "Looky see what I found: ",
  join(' ', sort keys %$name2path), "\n";

print "LWPUA docs = ",
  Pod::Simple::Search->new->find('LWP::UserAgent') || "?",
  "\n";
```

DESCRIPTION

Pod::Simple::Search is a class that you use for running searches for Pod files. An object of this class has several attributes (mostly options for controlling search options), and some methods for searching based on those attributes.

The way to use this class is to make a new object of this class, set any options, and then call one of the search options (probably `survey` or `find`). The sections below discuss the syntaxes for doing all that.

CONSTRUCTOR

This class provides the one constructor, called new. It takes no parameters:

```
use Pod::Simple::Search;
my $search = Pod::Simple::Search->new;
```

ACCESSORS

This class defines several methods for setting (and, occasionally, reading) the contents of an object. With two exceptions (discussed at the end of this section), these attributes are just for controlling the way searches are carried out.

Note that each of these return `$self` when you call them as `$self->_whatever(value)_`. That's so that you can chain together set-attribute calls like this:

```
my $name2path =
  Pod::Simple::Search->new
  -> inc(0) -> verbose(1) -> callback(\&blab)
  ->survey(@there);
```

...which works exactly as if you'd done this:

```
my $search = Pod::Simple::Search->new;
$search->inc(0);
$search->verbose(1);
$search->callback(\&blab);
my $name2path = $search->survey(@there);
```

- $search->inc(*true-or-false*);

  This attribute, if set to a true value, means that searches should implicitly add perl's *@INC* paths. This automatically considers paths specified in the `PERL5LIB` environment as this is prepended to *@INC* by the Perl interpreter itself. This attribute's default value is **TRUE**. If you want to search only specific directories, set $self->inc(0) before calling $inc->survey or $inc->find.

- $search->verbose(*nonnegative-number*);

  This attribute, if set to a nonzero positive value, will make searches output (via `warn`) notes about what they're doing as they do it. This option may be useful for debugging a pod-related module. This attribute's default value is zero, meaning that no `warn` messages are produced. (Setting verbose to 1 turns on some messages, and setting it to 2 turns on even more messages, i.e., makes the following search(es) even more verbose than 1 would make them.)

- $search->limit_glob(*some-glob-string*);

  This option means that you want to limit the results just to items whose podnames match the given glob/wildcard expression. For example, you might limit your search to just "LWP::*", to search only for modules starting with "LWP::*" (but not including the module "LWP" itself); or you might limit your search to "LW*" to see only modules whose (full) names begin with "LW"; or you might search for "*Find*"

to search for all modules with "Find" somewhere in their full name. (You can also use "?" in a glob expression; so "DB?" will match "DBI" and "DBD".)

- $search->callback(\&*some_routine*);

  This attribute means that every time this search sees a matching Pod file, it should call this callback routine. The routine is called with two parameters: the current file's filespec, and its pod name. (For example: `("/etc/perljunk/File/Crunk.pm", "File::Crunk")` would be in `@_`.)

  The callback routine's return value is not used for anything.

  This attribute's default value is false, meaning that no callback is called.

- $search->laborious(*true-or-false*);

  Unless you set this attribute to a true value, Pod::Search will apply Perl-specific heuristics to find the correct module PODs quickly. This attribute's default value is false. You won't normally need to set this to true.

  Specifically: Turning on this option will disable the heuristics for seeing only files with Perl-like extensions, omitting subdirectories that are numeric but do *not* match the current Perl interpreter's version ID, suppressing `site_perl` as a module hierarchy name, etc.

- $search->recurse(*true-or-false*);

  Unless you set this attribute to a false value, Pod::Search will recurse into subdirectories of the search directories.

- $search->shadows(*true-or-false*);

  Unless you set this attribute to a true value, Pod::Simple::Search will consider only the first file of a given modulename as it looks thru the specified directories; that is, with this option off, if Pod::Simple::Search has seen a `somepathdir/Foo/Bar.pm` already in this search, then it won't bother looking at a `somelaterpathdir/Foo/Bar.pm` later on in that search, because that file is merely a "shadow". But if you turn on `$self->shadows(1)`, then these "shadow" files are inspected too, and are noted in the pathname2podname return hash.

  This attribute's default value is false; and normally you won't need to turn it on.

- $search->is_case_insensitive(*true-or-false*);

  Pod::Simple::Search will by default internally make an assumption based on the underlying filesystem where the class file is found whether it is case insensitive or not.

  If it is determined to be case insensitive, during survey() it may skip pod files/modules that happen to be equal to names it's already seen, ignoring case.

  However, it's possible to have distinct files in different directories that intentionally has the same name, just differing in case, that should be reported. Hence, you may force the behavior by setting this to true or false.

- $search->limit_re(*some-regxp*);

  Setting this attribute (to a value that's a regexp) means that you want to limit the results just to items whose podnames match the given regexp. Normally this option is not needed, and the more efficient `limit_glob` attribute is used instead.

- $search->dir_prefix(*some-string-value*);

  Setting this attribute to a string value means that the searches should begin in the specified subdirectory name (like "Pod" or "File::Find", also expressible as "File/Find"). For example, the search option `$search->limit_glob("File::Find::R*")` is the same as the combination of the search options `$search->limit_re("^File::Find::R") -> dir_prefix("File::Find")`.

  Normally you don't need to know about the `dir_prefix` option, but I include it in case it might prove useful for someone somewhere.

  (Implementationally, searching with limit_glob ends up setting limit_re and usually dir_prefix.)

- $search->progress(*some-progress-object*);

  If you set a value for this attribute, the value is expected to be an object (probably of a class that you define) that has a `reach` method and a done method. This is meant for reporting progress during the search, if you don't want to use a simple callback.

  Normally you don't need to know about the `progress` option, but I include it in case it might prove useful for someone somewhere.

  While a search is in progress, the progress object's `reach` and done methods are called like this:

```
# Every time a file is being scanned for pod:
$progress->reach($count, "Scanning $file");   ++$count;

# And then at the end of the search:
$progress->done("Noted $count Pod files total");
```

  Internally, we often set this to an object of class Pod::Simple::Progress. That class is probably undocumented, but you may wish to look at its source.

- $name2path = $self->name2path;

  This attribute is not a search parameter, but is used to report the result of `survey` method, as discussed in the next section.

- $path2name = $self->path2name;

  This attribute is not a search parameter, but is used to report the result of `survey` method, as discussed in the next section.

MAIN SEARCH METHODS

Once you've actually set any options you want (if any), you can go ahead and use the following methods to search for Pod files in particular ways.

`$search->survey( @directories )`

The method `survey` searches for POD documents in a given set of files and/or directories. This runs the search according to the various options set by the accessors above. (For example, if the `inc` attribute is on, as it is by default, then the perl @INC directories are implicitly added to the list of directories (if any) that you specify.)

The return value of `survey` is two hashes:

- `name2path`

  A hash that maps from each pod-name to the filespec (like "Stuff::Thing" => "/whatever/plib/Stuff/Thing.pm")

- `path2name`

  A hash that maps from each Pod filespec to its pod-name (like "/whatever/plib/Stuff/Thing.pm" => "Stuff::Thing")

Besides saving these hashes as the hashref attributes `name2path` and `path2name`, calling this function also returns these hashrefs. In list context, the return value of `$search->survey` is the list `(\%name2path, \%path2name)`. In scalar context, the return value is `\%name2path`. Or you can just call this in void context.

Regardless of calling context, calling `survey` saves its results in its `name2path` and `path2name` attributes.

E.g., when searching in `$HOME/perl5lib`, the file `$HOME/perl5lib/MyModule.pm` would get the POD name *MyModule*, whereas `$HOME/perl5lib/Myclass/Subclass.pm` would be *Myclass::Subclass*. The name information can be used for POD translators.

Only text files containing at least one valid POD command are found.

In verbose mode, a warning is printed if shadows are found (i.e., more than one POD file with the same POD name is found, e.g. `CPAN.pm` in different directories). This usually indicates duplicate occurrences of modules in the *@INC* search path, which is occasionally inadvertent (but is often simply a case of a user's path dir having a more recent version than the system's general path dirs in general.)

The options to this argument is a list of either directories that are searched recursively, or files. (Usually you wouldn't specify files, but just dirs.) Or you can just specify an empty-list, as in $name2path; with the `inc` option on, as it is by default.

The POD names of files are the plain basenames with any Perl-like extension (.pm, .pl, .pod) stripped, and path separators replaced by `::`'s.

Calling Pod::Simple::Search->search(...) is short for Pod::Simple::Search->new->search(...). That is, a throwaway object with default attribute values is used.

`$search->simplify_name( $str )`

The method **simplify_name** is equivalent to **basename**, but also strips Perl-like extensions (.pm, .pl, .pod) and extensions like `.bat`, `.cmd` on Win32 and OS/2, or `.com` on VMS, respectively.

`$search->find( $pod )`

`$search->find( $pod, @search_dirs )`

Returns the location of a Pod file, given a Pod/module/script name (like "Foo::Bar" or "perlvar" or "perldoc"), and an idea of what files/directories to look in. It searches according to the various options set by the accessors above. (For example, if the `inc` attribute is on, as it is by default, then the perl @INC directories are implicitly added to the list of directories (if any) that you specify.)

This returns the full path of the first occurrence to the file. Package names (eg 'A::B') are automatically converted to directory names in the selected directory. Additionally, '.pm', '.pl' and '.pod' are automatically appended to the search as required. (So, for example, under Unix, "A::B" is converted to "somedir/A/B.pm", "somedir/A/B.pod", or "somedir/A/B.pl", as appropriate.)

If no such Pod file is found, this method returns undef.

If any of the given search directories contains a `pod/` subdirectory, then it is searched. (That's how we manage to find `perlfunc`, for example, which is usually in `pod/perlfunc` in most Perl dists.)

The `verbose` and `inc` attributes influence the behavior of this search; notably, `inc`, if true, adds @INC *and also $Config::Config{'scriptdir'}* to the list of directories to search.

It is common to simply say `$filename = Pod::Simple::Search-> new \ ->find("perlvar")` so that just the @INC (well, and scriptdir) directories are searched. (This happens because the `inc` attribute is true by default.)

Calling Pod::Simple::Search->find(...) is short for Pod::Simple::Search->new->find(...). That is, a throwaway object with default attribute values is used.

`$self->contains_pod( $file )`

Returns true if the supplied filename (not POD module) contains some Pod documentation.

SUPPORT

Questions or discussion about POD and Pod::Simple should be sent to the pod-people@perl.org mail list. Send an empty email to pod-people-subscribe@perl.org to subscribe.

This module is managed in an open GitHub repository, https://github.com/perl-pod/pod-simple/. Feel free to fork and contribute, or to clone https://github.com/perl-pod/pod-simple.git and send patches!

Patches against Pod::Simple are welcome. Please send bug reports to bug-pod-simple@rt.cpan.org.

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2002 Sean M. Burke.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Pod::Simple was created by Sean M. Burke sburke@cpan.org with code borrowed from Marek Rouchal's Pod::Find, which in turn heavily borrowed code from Nick Ing-Simmons' `PodToHtml`.

But don't bother him, he's retired.

Pod::Simple is maintained by:

- Allison Randal `allison@perl.org`
- Hans Dieter Pearcey `hdp@cpan.org`
- David E. Wheeler `dwheeler@cpan.org`

Pod::Simple::SimpleTree

parse Pod into a simple parse tree

SYNOPSIS

```
% cat ptest.pod

=head1 PIE

I like B<pie>!

% perl -MPod::Simple::SimpleTree -MData::Dumper -e \
   "print Dumper(Pod::Simple::SimpleTree->new->parse_file(shift)->
↪root)" \
   ptest.pod

$VAR1 = [
          'Document',
          { 'start_line' => 1 },
          [
            'head1',
            { 'start_line' => 1 },
            'PIE'
          ],
          [
```

(continues on next page)

(continued from previous page)

```
            'Para',
            { 'start_line' => 3 },
            'I like ',
            [
              'B',
              {},
              'pie'
            ],
            '!'
          ]
        ];
```

DESCRIPTION

This class is of interest to people writing a Pod processor/formatter.

This class takes Pod and parses it, returning a parse tree made just of arrayrefs, and hashrefs, and strings.

This is a subclass of *Pod::Simple* and inherits all its methods.

This class is inspired by XML::Parser's "Tree" parsing-style, although it doesn't use exactly the same LoL format.

METHODS

At the end of the parse, call `$parser->root` to get the tree's top node.

Tree Contents

Every element node in the parse tree is represented by an arrayref of the form: `[ _elementname_, \%attributes, _...subnodes..._ ]`. See the example tree dump in the Synopsis, above.

Every text node in the tree is represented by a simple (non-ref) string scalar. So you can test `ref($node)` to see whether you have an element node or just a text node.

The top node in the tree is `[ 'Document', \%attributes, _...subnodes..._ ]`

SEE ALSO

Pod::Simple

perllol

The "Tree" subsubsection in XML::Parser

SUPPORT

Questions or discussion about POD and Pod::Simple should be sent to the pod-people@perl.org mail list. Send an empty email to pod-people-subscribe@perl.org to subscribe.

This module is managed in an open GitHub repository, https://github.com/perl-pod/pod-simple/. Feel free to fork and contribute, or to clone https://github.com/perl-pod/pod-simple.git and send patches!

Patches against Pod::Simple are welcome. Please send bug reports to bug-pod-simple@rt.cpan.org.

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2002 Sean M. Burke.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Pod::Simple was created by Sean M. Burke sburke@cpan.org. But don't bother him, he's retired.

Pod::Simple is maintained by:

- Allison Randal `allison@perl.org`
- Hans Dieter Pearcey `hdp@cpan.org`
- David E. Wheeler `dwheeler@cpan.org`

Pod::Simple::Text

format Pod as plaintext

SYNOPSIS

```
perl -MPod::Simple::Text -e \
 "exit Pod::Simple::Text->filter(shift)->any_errata_seen" \
 thingy.pod
```

DESCRIPTION

This class is a formatter that takes Pod and renders it as wrapped plaintext.

Its wrapping is done by *Text::Wrap*, so you can change `$Text::Wrap::columns` as you like.

This is a subclass of *Pod::Simple* and inherits all its methods.

SEE ALSO

Pod::Simple, *Pod::Simple::TextContent*, *Pod::Text*

SUPPORT

Questions or discussion about POD and Pod::Simple should be sent to the pod-people@perl.org mail list. Send an empty email to pod-people-subscribe@perl.org to subscribe.

This module is managed in an open GitHub repository, https://github.com/perl-pod/pod-simple/. Feel free to fork and contribute, or to clone https://github.com/perl-pod/pod-simple.git and send patches!

Patches against Pod::Simple are welcome. Please send bug reports to bug-pod-simple@rt.cpan.org.

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2002 Sean M. Burke.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Pod::Simple was created by Sean M. Burke sburke@cpan.org. But don't bother him, he's retired.

Pod::Simple is maintained by:

- Allison Randal `allison@perl.org`
- Hans Dieter Pearcey `hdp@cpan.org`
- David E. Wheeler `dwheeler@cpan.org`

Pod::Simple::TextContent

get the text content of Pod

SYNOPSIS

```
TODO

 perl -MPod::Simple::TextContent -e \
  "exit Pod::Simple::TextContent->filter(shift)->any_errata_seen" \
  thingy.pod
```

DESCRIPTION

This class is that parses Pod and dumps just the text content. It is mainly meant for use by the Pod::Simple test suite, but you may find some other use for it.

This is a subclass of *Pod::Simple* and inherits all its methods.

SEE ALSO

Pod::Simple, *Pod::Simple::Text*, Pod::Spell

SUPPORT

Questions or discussion about POD and Pod::Simple should be sent to the pod-people@perl.org mail list. Send an empty email to pod-people-subscribe@perl.org to subscribe.

This module is managed in an open GitHub repository, https://github.com/perl-pod/pod-simple/. Feel free to fork and contribute, or to clone https://github.com/perl-pod/pod-simple.git and send patches!

Patches against Pod::Simple are welcome. Please send bug reports to bug-pod-simple@rt.cpan.org.

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2002 Sean M. Burke.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Pod::Simple was created by Sean M. Burke sburke@cpan.org. But don't bother him, he's retired.

Pod::Simple is maintained by:

- Allison Randal `allison@perl.org`
- Hans Dieter Pearcey `hdp@cpan.org`
- David E. Wheeler `dwheeler@cpan.org`

Pod::Simple::XHTML

format Pod as validating XHTML

SYNOPSIS

```
use Pod::Simple::XHTML;

my $parser = Pod::Simple::XHTML->new();

...

$parser->parse_file('path/to/file.pod');
```

DESCRIPTION

This class is a formatter that takes Pod and renders it as XHTML validating HTML.

This is a subclass of *Pod::Simple::Methody* and inherits all its methods. The implementation is entirely different than *Pod::Simple::HTML*, but it largely preserves the same interface.

Minimal code

```
use Pod::Simple::XHTML;
my $psx = Pod::Simple::XHTML->new;
$psx->output_string(\my $html);
$psx->parse_file('path/to/Module/Name.pm');
open my $out, '>', 'out.html' or die "Cannot open 'out.html': $!\n";
print $out $html;
```

You can also control the character encoding and entities. For example, if you're sure that the POD is properly encoded (using the `=encoding` command), you can prevent high-bit characters from being encoded as HTML entities and declare the output character set as UTF-8 before parsing, like so:

```
  $psx->html_charset('UTF-8');
use warnings;
  $psx->html_encode_chars(q{&<>'"});
```

METHODS

Pod::Simple::XHTML offers a number of methods that modify the format of the HTML output. Call these after creating the parser object, but before the call to `parse_file`:

```
my $parser = Pod::PseudoPod::HTML->new();
$parser->set_optional_param("value");
$parser->parse_file($file);
```

perldoc_url_prefix

In turning LFoo::Bar into http://whatever/Foo%3a%3aBar, what to put before the "Foo%3a%3aBar". The default value is "https://metacpan.org/pod/".

perldoc_url_postfix

What to put after "Foo%3a%3aBar" in the URL. This option is not set by default.

man_url_prefix

In turning `L<crontab(5)>` into http://whatever/man/1/crontab.1, what to put before the "1/crontab.1". The default value is "https://man7.org/linux/man-pages/man".

man_url_postfix

What to put after "1/crontab.1" in the URL. This option is set to ".html" by default.

title_prefix, title_postfix

What to put before and after the title in the head. The values should already be &-escaped.

html_css

```
$parser->html_css('path/to/style.css');
```

The URL or relative path of a CSS file to include. This option is not set by default.

html_javascript

The URL or relative path of a JavaScript file to pull in. This option is not set by default.

html_doctype

A document type tag for the file. This option is not set by default.

html_charset

The character set to declare in the Content-Type meta tag created by default for `html_header_tags`. Note that this option will be ignored if the value of `html_header_tags` is changed. Defaults to "ISO-8859-1".

html_header_tags

Additional arbitrary HTML tags for the header of the document. The default value is just a content type header tag:

```
<meta http-equiv="Content-Type" content="text/html; charset=ISO-
↪8859-1">
```

Add additional meta tags here, or blocks of inline CSS or JavaScript (wrapped in the appropriate tags).

html_encode_chars

A string containing all characters that should be encoded as HTML entities, specified using the regular expression character class syntax (what you find within brackets in regular expressions). This value will be passed as the second argument to the `encode_entities` function of *HTML::Entities*. If *HTML::Entities* is not installed, then any characters other than &<"> will be encoded numerically.

html_h_level

This is the level of HTML "Hn" element to which a Pod "head1" corresponds. For example, if `html_h_level` is set to 2, a head1 will produce an H2, a head2 will produce an H3, and so on.

default_title

Set a default title for the page if no title can be determined from the content. The value of this string should already be &-escaped.

force_title

Force a title for the page (don't try to determine it from the content). The value of this string should already be &-escaped.

html_header, html_footer

Set the HTML output at the beginning and end of each file. The default header includes a title, a doctype tag (if `html_doctype` is set), a content tag (customized by `html_header_tags`), a tag for a CSS file (if `html_css` is set), and a tag for a Javascript file (if `html_javascript` is set). The default footer simply closes the `html` and body tags.

The options listed above customize parts of the default header, but setting `html_header` or `html_footer` completely overrides the built-in header or footer. These may be useful if you want to use template tags instead of literal HTML headers and footers or are integrating converted POD pages in a larger website.

If you want no headers or footers output in the HTML, set these options to the empty string.

index

Whether to add a table-of-contents at the top of each page (called an index for the sake of tradition).

anchor_items

Whether to anchor every definition `=item` directive. This needs to be enabled if you want to be able to link to specific `=item` directives, which are output as `<dt>` elements. Disabled by default.

backlink

Whether to turn every =head1 directive into a link pointing to the top of the page (specifically, the opening body tag).

SUBCLASSING

If the standard options aren't enough, you may want to subclass Pod::Simple::XHMTL. These are the most likely candidates for methods you'll want to override when subclassing.

handle_text

This method handles the body of text within any element: it's the body of a paragraph, or everything between a "=begin" tag and the corresponding "=end" tag, or the text within an L entity, etc. You would want to override this if you are adding a custom element type that does more than just display formatted text. Perhaps adding a way to generate HTML tables from an extended version of POD.

So, let's say you want to add a custom element called 'foo'. In your subclass's `new` method, after calling `SUPER::new` you'd call:

```
$new->accept_targets_as_text( 'foo' );
```

Then override the `start_for` method in the subclass to check for when "$flags->{'target'}" is equal to 'foo' and set a flag that marks that you're in a foo block (maybe "$self->{'in_foo'} = 1"). Then override the `handle_text` method to check for the flag, and pass $text to your custom subroutine to construct the HTML output for 'foo' elements, something like:

```
sub handle_text {
    my ($self, $text) = @_;
    if ($self->{'in_foo'}) {
        $self->{'scratch'} .= build_foo_html($text);
        return;
    }
    $self->SUPER::handle_text($text);
}
```

handle_code

This method handles the body of text that is marked up to be code. You might for instance override this to plug in a syntax highlighter. The base implementation just escapes the text.

The callback methods `start_code` and `end_code` emits the `code` tags before and after `handle_code` is invoked, so you might want to override these together with `handle_code` if this wrapping isn't suitable.

Note that the code might be broken into multiple segments if there are nested formatting codes inside a `C<...>` sequence. In between the calls to `handle_code` other markup tags might have been emitted in that case. The same is true for verbatim sections if the `codes_in_verbatim` option is turned on.

accept_targets_as_html

This method behaves like `accept_targets_as_text`, but also marks the region as one whose content should be emitted literally, without HTML entity escaping or wrapping in a `div` element.

resolve_pod_page_link

```
my $url = $pod->resolve_pod_page_link('Net::Ping', 'INSTALL');
my $url = $pod->resolve_pod_page_link('perlpodspec');
my $url = $pod->resolve_pod_page_link(undef, 'SYNOPSIS');
```

Resolves a POD link target (typically a module or POD file name) and section name to a URL. The resulting link will be returned for the above examples as:

```
https://metacpan.org/pod/Net::Ping#INSTALL
https://metacpan.org/pod/perlpodspec
#SYNOPSIS
```

Note that when there is only a section argument the URL will simply be a link to a section in the current document.

resolve_man_page_link

```
my $url = $pod->resolve_man_page_link('crontab(5)', 'EXAMPLE CRON␣
↪FILE');
my $url = $pod->resolve_man_page_link('crontab');
```

Resolves a man page link target and numeric section to a URL. The resulting link will be returned for the above examples as:

```
https://man7.org/linux/man-pages/man5/crontab.5.html
https://man7.org/linux/man-pages/man1/crontab.1.html
```

Note that the first argument is required. The section number will be parsed from it, and if it's missing will default to 1. The second argument is currently ignored. Subclass to link to a different man page HTTP server.

idify

```
my $id   = $pod->idify($text);
my $hash = $pod->idify($text, 1);
```

This method turns an arbitrary string into a valid XHTML ID attribute value. The rules enforced, following http://webdesign.about.com/od/htmltags/a/aa031707.htm, are:

- The id must start with a letter (a-z or A-Z)
- All subsequent characters can be letters, numbers (0-9), hyphens (-), underscores (_), colons (:), and periods (.).
- The final character can't be a hyphen, colon, or period. URLs ending with these characters, while allowed by XHTML, can be awkward to extract from plain text.
- Each id must be unique within the document.

In addition, the returned value will be unique within the context of the Pod::Simple::XHTML object unless a second argument is passed a true value. ID attributes should always be unique within a single XHTML document, but pass the true value if you are creating not an ID but a URL hash to point to an ID (i.e., if you need to put the "#foo" in `<a href="#foo">foo</a>`.

batch_mode_page_object_init

```
$pod->batch_mode_page_object_init($batchconvobj, $module, $infile,
 ↪$outfile, $depth);
```

Called by *Pod::Simple::HTMLBatch* so that the class has a chance to initialize the converter. Internally it sets the `batch_mode` property to true and sets `batch_mode_current_level()`, but Pod::Simple::XHTML does not currently use those features. Subclasses might, though.

SEE ALSO

Pod::Simple, *Pod::Simple::Text*, Pod::Spell

SUPPORT

Questions or discussion about POD and Pod::Simple should be sent to the pod-people@perl.org mail list. Send an empty email to pod-people-subscribe@perl.org to subscribe.

This module is managed in an open GitHub repository, https://github.com/perl-pod/pod-simple/. Feel free to fork and contribute, or to clone https://github.com/perl-pod/pod-simple.git and send patches!

Patches against Pod::Simple are welcome. Please send bug reports to bug-pod-simple@rt.cpan.org.

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2003-2005 Allison Randal.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Pod::Simple::XHTML was created by Allison Randal allison@perl.org.

Pod::Simple was created by Sean M. Burke sburke@cpan.org. But don't bother him, he's retired.

Pod::Simple is maintained by:

- Allison Randal `allison@perl.org`
- Hans Dieter Pearcey `hdp@cpan.org`
- David E. Wheeler `dwheeler@cpan.org`

Pod::Simple::XMLOutStream

turn Pod into XML

SYNOPSIS

```
perl -MPod::Simple::XMLOutStream -e \
 "exit Pod::Simple::XMLOutStream->filter(shift)->any_errata_seen" \
 thingy.pod
```

DESCRIPTION

Pod::Simple::XMLOutStream is a subclass of *Pod::Simple* that parses Pod and turns it into XML.

Pod::Simple::XMLOutStream inherits methods from *Pod::Simple*.

SEE ALSO

Pod::Simple::DumpAsXML is rather like this class; see its documentation for a discussion of the differences.

Pod::Simple, *Pod::Simple::DumpAsXML*, Pod::SAX

Pod::Simple::Subclassing

The older (and possibly obsolete) libraries Pod::PXML, Pod::XML

ABOUT EXTENDING POD

TODO: An example or two of =extend, then point to Pod::Simple::Subclassing

SEE ALSO

Pod::Simple, *Pod::Simple::Text*, Pod::Spell

SUPPORT

Questions or discussion about POD and Pod::Simple should be sent to the pod-people@perl.org mail list. Send an empty email to pod-people-subscribe@perl.org to subscribe.

This module is managed in an open GitHub repository, https://github.com/perl-pod/pod-simple/. Feel free to fork and contribute, or to clone https://github.com/perl-pod/pod-simple.git and send patches!

Patches against Pod::Simple are welcome. Please send bug reports to bug-pod-simple@rt.cpan.org.

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2002-2004 Sean M. Burke.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

Pod::Simple was created by Sean M. Burke sburke@cpan.org. But don't bother him, he's retired.

Pod::Simple is maintained by:

- Allison Randal `allison@perl.org`
- Hans Dieter Pearcey `hdp@cpan.org`
- David E. Wheeler `dwheeler@cpan.org`

Pod::Text

Convert POD data to formatted text

SYNOPSIS

```
use Pod::Text;
my $parser = Pod::Text->new (sentence => 1, width => 78);

# Read POD from STDIN and write to STDOUT.
$parser->parse_from_filehandle;

# Read POD from file.pod and write to file.txt.
$parser->parse_from_file ('file.pod', 'file.txt');
```

DESCRIPTION

Pod::Text is a module that can convert documentation in the POD format (the preferred language for documenting Perl) into formatted text. It uses no special formatting controls or codes, and its output is therefore suitable for nearly any device.

Encoding

Pod::Text uses the following logic to choose an output encoding, in order:

1. If a PerlIO encoding layer is set on the output file handle, do not do any output encoding and will instead rely on the PerlIO encoding layer.
2. If the `encoding` or `utf8` options are set, use the output encoding specified by those options.
3. If the input encoding of the POD source file was explicitly specified (using `=encoding`) or automatically detected by Pod::Simple, use that as the output encoding as well.
4. Otherwise, if running on a non-EBCDIC system, use UTF-8 as the output encoding. Since this is a superset of ASCII, this will result in ASCII output unless the POD input contains non-ASCII characters without declaring or autodetecting an encoding (usually via E<> escapes).
5. Otherwise, for EBCDIC systems, output without doing any encoding and hope this works.

One caveat: Pod::Text has to commit to an output encoding the first time it outputs a non-ASCII character, and then has to stick with it for consistency. However, `=encoding` commands don't have to be at the beginning of a POD document. If someone uses a non-ASCII character early in a document with an escape, such as E<0xEF>, and then puts `=encoding iso-8859-1` later, ideally Pod::Text would follow rule 3 and output the entire document as ISO 8859-1. Instead, it will commit to UTF-8 following rule 4 as soon as it sees that escape, and then stick with that encoding for the rest of the document.

Unfortunately, there's no universally good choice for an output encoding. Each choice will be incorrect in some circumstances. This approach was chosen primarily for backwards compatibility. Callers should consider forcing the output encoding via `encoding` if they have any knowledge about what encoding the user may expect.

In particular, consider importing the Encode::Locale module, if available, and setting `encoding` to `locale` to use an output encoding appropriate to the user's locale. But be aware that if the user is not using locales or is using a locale of C, Encode::Locale will set the output encoding to US-ASCII. This will cause all non-ASCII characters will be replaced

with ? and produce a flurry of warnings about unsupported characters, which may or may not be what you want.

CLASS METHODS

- new(ARGS)

  Create a new Pod::Text object. ARGS should be a list of key/value pairs, where the keys are chosen from the following. Each option is annotated with the version of Pod::Text in which that option was added with its current meaning.

  - alt

    [2.00] If set to a true value, selects an alternate output format that, among other things, uses a different heading style and marks `=item` entries with a colon in the left margin. Defaults to false.

  - code

    [2.13] If set to a true value, the non-POD parts of the input file will be included in the output. Useful for viewing code documented with POD blocks with the POD rendered and the code left intact.

  - encoding

    [6.1.0] Specifies the encoding of the output. The value must be an encoding recognized by the *Encode* module (see Encode::Supported) or the special value none. If the output contains characters that cannot be represented in this encoding, that is an error that will be reported as configured by the `errors` option. If error handling is other than `die`, the unrepresentable character will be replaced with the Encode substitution character (normally ?).

    If the output file handle has a PerlIO encoding layer set, this parameter will be ignored and no encoding will be done by Pod::Man. It will instead rely on the encoding layer to make whatever output encoding transformations are desired.

    As a special case, if the encoding is set to none, no encoding will be done and characters will be output in Perl's internal representation. This option only makes sense in combination with output_string(). It is intended for special cases when the results of formatting are kept in memory and will be encoded for output at some later step.

    WARNING: The input encoding of the POD source is independent from the output encoding, and setting this option does not affect the interpretation of the POD input. Unless your POD source is US-ASCII, its encoding should be declared with the `=encoding` command in the source, as near to the top of the file as possible. If this is not done, Pod::Simple will will attempt to guess the encoding and may be successful if it's Latin-1 or UTF-8, but it will produce warnings. See perlpod(1) for more information.

  - errors

    [3.17] How to report errors. `die` says to throw an exception on any POD formatting error. `stderr` says to report errors on standard error, but not to throw an exception. pod says to include a POD ERRORS section in the resulting docu-

mentation summarizing the errors. none ignores POD errors entirely, as much as possible.

The default is pod.

– guesswork

[5.01] By default, Pod::Text applies some default formatting rules based on guesswork and regular expressions that are intended to make writing Perl documentation easier and require less explicit markup. These rules may not always be appropriate, particularly for documentation that isn't about Perl. This option allows turning all or some of it off.

The special value `all` enables all guesswork. This is also the default for backward compatibility reasons. The special value none disables all guesswork. Otherwise, the value of this option should be a comma-separated list of one or more of the following keywords:

* quoting

  If no guesswork is enabled, any text enclosed in C<> is surrounded by double quotes in nroff (terminal) output unless the contents are already quoted. When this guesswork is enabled, quote marks will also be suppressed for Perl variables, function names, function calls, numbers, and hex constants.

Any unknown guesswork name is silently ignored (for potential future compatibility), so be careful about spelling.

– indent

[2.00] The number of spaces to indent regular text, and the default indentation for `=over` blocks. Defaults to 4.

– loose

[2.00] If set to a true value, a blank line is printed after a `=head1` heading. If set to false (the default), no blank line is printed after `=head1`, although one is still printed after `=head2`. This is the default because it's the expected formatting for manual pages; if you're formatting arbitrary text documents, setting this to true may result in more pleasing output.

– margin

[2.21] The width of the left margin in spaces. Defaults to 0. This is the margin for all text, including headings, not the amount by which regular text is indented; for the latter, see the *indent* option. To set the right margin, see the *width* option.

– nourls

[3.17] Normally, L<> formatting codes with a URL but anchor text are formatted to show both the anchor text and the URL. In other words:

```
L<foo|http://example.com/>
```

is formatted as:

```
foo <http://example.com/>
```

This option, if set to a true value, suppresses the URL when anchor text is given, so this example would be formatted as just `foo`. This can produce less cluttered output in cases where the URLs are not particularly important.

– quotes

[4.00] Sets the quote marks used to surround C<> text. If the value is a single character, it is used as both the left and right quote. Otherwise, it is split in half, and the first half of the string is used as the left quote and the second is used as the right quote.

This may also be set to the special value none, in which case no quote marks are added around C<> text.

– sentence

[3.00] If set to a true value, Pod::Text will assume that each sentence ends in two spaces, and will try to preserve that spacing. If set to false, all consecutive whitespace in non-verbatim paragraphs is compressed into a single space. Defaults to false.

– stderr

[3.10] Send error messages about invalid POD to standard error instead of appending a POD ERRORS section to the generated output. This is equivalent to setting `errors` to `stderr` if `errors` is not already set. It is supported for backward compatibility.

– utf8

[3.12] If this option is set to a true value, the output encoding is set to UTF-8. This is equivalent to setting `encoding` to `UTF-8` if `encoding` is not already set. It is supported for backward compatibility.

– width

[2.00] The column at which to wrap text on the right-hand side. Defaults to 76.

INSTANCE METHODS

As a derived class from Pod::Simple, Pod::Text supports the same methods and interfaces. See *Pod::Simple* for all the details. This section summarizes the most-frequently-used methods and the ones added by Pod::Text.

- output_fh(FH)

  Direct the output from parse_file(), parse_lines(), or parse_string_document() to the file handle FH instead of `STDOUT`.

- output_string(REF)

  Direct the output from parse_file(), parse_lines(), or parse_string_document() to the scalar variable pointed to by REF, rather than `STDOUT`. For example:

```
my $man = Pod::Man->new();
my $output;
$man->output_string(\$output);
$man->parse_file('/some/input/file');
```

Be aware that the output in that variable will already be encoded (see "Encoding") unless the `encoding` option to Pod::Text is set to none.

- parse_file(PATH)

  Read the POD source from PATH and format it. By default, the output is sent to `STDOUT`, but this can be changed with the output_fh() or output_string() methods.

- parse_from_file(INPUT, OUTPUT)

- parse_from_filehandle(FH, OUTPUT)

  Read the POD source from INPUT, format it, and output the results to OUTPUT.

  parse_from_filehandle() is provided for backward compatibility with older versions of Pod::Man. parse_from_file() should be used instead.

- parse_lines(LINES[, ...[, undef]])

  Parse the provided lines as POD source, writing the output to either `STDOUT` or the file handle set with the output_fh() or output_string() methods. This method can be called repeatedly to provide more input lines. An explicit `undef` should be passed to indicate the end of input.

  This method expects raw bytes, not decoded characters.

- parse_string_document(INPUT)

  Parse the provided scalar variable as POD source, writing the output to either `STDOUT` or the file handle set with the output_fh() or output_string() methods.

  This method expects raw bytes, not decoded characters.

FUNCTIONS

Pod::Text exports one function for backward compatibility with older versions. This function is deprecated; instead, use the object-oriented interface described above.

- pod2text([[-a,] [-NNN,]] INPUT[, OUTPUT])

  Convert the POD source from INPUT to text and write it to OUTPUT. If OUTPUT is not given, defaults to `STDOUT`. INPUT can be any expression supported as the second argument to two-argument open().

  If `-a` is given as an initial argument, pass the `alt` option to the Pod::Text constructor. This enables alternative formatting.

  If `-NNN` is given as an initial argument, pass the `width` option to the Pod::Text constructor with the number `NNN` as its argument. This sets the wrap line width to NNN.

DIAGNOSTICS

- Bizarre space in item
- Item called without tag

  (W) Something has gone wrong in internal `=item` processing. These messages indicate a bug in Pod::Text; you should never see them.

- Can't open %s for reading: %s

  (F) Pod::Text was invoked via the compatibility mode pod2text() interface and the input file it was given could not be opened.

- Invalid errors setting "%s"

  (F) The `errors` parameter to the constructor was set to an unknown value.

- Invalid quote specification "%s"

  (F) The quote specification given (the `quotes` option to the constructor) was invalid. A quote specification must be either one character long or an even number (greater than one) characters long.

- Margin is wider than the output width

  (W) The requested margin width, determined from the sum of indentations requested by `=over` and the baseline indentation for text, is equal to or wider than the desired output width specified with `width`. The text will be formatted as if the margin was set to zero.

- POD document had syntax errors

  (F) The POD document being formatted had syntax errors and the `errors` option was set to `die`.

COMPATIBILITY

Pod::Text 2.03 (based on Pod::Parser) was the first version of this module included with Perl, in Perl 5.6.0. Earlier versions of Perl had a different Pod::Text module, with a different API.

The current API based on *Pod::Simple* was added in Pod::Text 3.00. Pod::Text 3.01 was included in Perl 5.9.3, the first version of Perl to incorporate those changes. This is the first version that correctly supports all modern POD syntax. The parse_from_filehandle() method was re-added for backward compatibility in Pod::Text 3.07, included in Perl 5.9.4.

Pod::Text 3.12, included in Perl 5.10.1, first implemented the current practice of attempting to match the default output encoding with the input encoding of the POD source, unless overridden by the `utf8` option or (added later) the `encoding` option.

Support for anchor text in L<> links of type URL was added in Pod::Text 3.14, included in Perl 5.11.5.

parse_lines(), parse_string_document(), and parse_file() set a default output file handle of STDOUT if one was not already set as of Pod::Text 3.18, included in Perl 5.19.5.

Pod::Text 4.00, included in Perl 5.23.7, aligned the module version and the version of the podlators distribution. All modules included in podlators, and the podlators distribution itself, share the same version number from this point forward.

Pod::Text 4.09, included in Perl 5.25.7, fixed a serious bug on EBCDIC systems, present in all versions back to 3.00, that would cause opening brackets to disappear.

Pod::Text 5.00 and later, included in Perl 5.37.7, default, on non-EBCDIC systems, to UTF-8 encoding if it sees a non-ASCII character in the input and the input encoding is not specified. They also commit to an encoding with the first non-ASCII character and does not change the output encoding if the input encoding changes. The *Encode* module is now used for all output encoding rather than PerlIO layers, which fixes earlier problems with output to scalars.

Pod::Text 6.1.0 added support for the none option to `encoding`.

CAVEATS

Line wrapping is done only at ASCII spaces and tabs, rather than using a correct Unicode-aware line wrapping algorithm.

AUTHOR

Russ Allbery rra@cpan.org, based *very* heavily on the original Pod::Text by Tom Christiansen tchrist@mox.perl.com and its conversion to Pod::Parser by Brad Appleton bradapp@enteract.com. Sean Burke's initial conversion of Pod::Man to use Pod::Simple provided much-needed guidance on how to use Pod::Simple.

COPYRIGHT AND LICENSE

Copyright 1999-2002, 2004, 2006, 2008-2009, 2012-2016, 2018-2019, 2022, 2024-2026 Russ Allbery rra@cpan.org

This program is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

SEE ALSO

Encode::Locale, Encode::Supported, *Pod::Simple*, *Pod::Text::Termcap*, perlpod(1), pod2text(1)

The current version of this module is always available from its web site at https://www.eyrie.org/~eagle/software/podlators/. It is also part of the Perl core distribution as of 5.6.0.

Modules

- `Pod::Text::Color` - Convert POD data to formatted color ASCII text
- `Pod::Text::Overstrike` - Convert POD data to formatted overstrike text
- `Pod::Text::Termcap` - Convert POD data to ASCII text with format escapes

Pod::Text::Color

Convert POD data to formatted color ASCII text

SYNOPSIS

```
use Pod::Text::Color;
my $parser = Pod::Text::Color->new (sentence => 0, width => 78);

# Read POD from STDIN and write to STDOUT.
$parser->parse_from_filehandle;

# Read POD from file.pod and write to file.txt.
$parser->parse_from_file ('file.pod', 'file.txt');
```

DESCRIPTION

Pod::Text::Color is a simple subclass of Pod::Text that highlights output text using ANSI color escape sequences. Apart from the color, it in all ways functions like Pod::Text. See *Pod::Text* for details and available options.

Term::ANSIColor is used to get colors and therefore must be installed to use this module.

COMPATIBILITY

Pod::Text::Color 0.05 (based on Pod::Parser) was the first version of this module included with Perl, in Perl 5.6.0.

The current API based on *Pod::Simple* was added in Pod::Text::Color 2.00. Pod::Text::Color 2.01 was included in Perl 5.9.3, the first version of Perl to incorporate those changes.

Several problems with wrapping and line length were fixed as recently as Pod::Text::Color 6.0.0.

This module inherits its API and most behavior from Pod::Text, so the details in *"COMPATIBILITY" in Pod::Text* also apply. Pod::Text and Pod::Text::Color have had the same module version since 4.00, included in Perl 5.23.7. (They unfortunately diverge in confusing ways prior to that.)

CAVEATS

Line wrapping is done only at ASCII spaces and tabs, rather than using a correct Unicode-aware line wrapping algorithm.

AUTHOR

Russ Allbery rra@cpan.org.

COPYRIGHT AND LICENSE

Copyright 1999, 2001, 2004, 2006, 2008, 2009, 2018-2019, 2022, 2024, 2026 Russ Allbery rra@cpan.org

This program is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

SEE ALSO

Pod::Text, *Pod::Simple*

The current version of this module is always available from its web site at https://www.eyrie.org/~eagle/software/podlators/. It is also part of the Perl core distribution as of 5.6.0.

Pod::Text::Overstrike

Convert POD data to formatted overstrike text

SYNOPSIS

```
use Pod::Text::Overstrike;
my $parser = Pod::Text::Overstrike->new (sentence => 0, width =>␣
↪78);

# Read POD from STDIN and write to STDOUT.
$parser->parse_from_filehandle;

# Read POD from file.pod and write to file.txt.
$parser->parse_from_file ('file.pod', 'file.txt');
```

DESCRIPTION

Pod::Text::Overstrike is a simple subclass of Pod::Text that highlights output text using overstrike sequences, in a manner similar to nroff. Characters in bold text are overstruck (character, backspace, character) and characters in underlined text are converted to overstruck underscores (underscore, backspace, character). This format was originally designed for hard-copy terminals and/or line printers, yet is readable on soft-copy (CRT) terminals.

Overstruck text is best viewed by page-at-a-time programs that take advantage of the terminal's **stand-out** and *underline* capabilities, such as the less program on Unix.

Apart from the overstrike, it in all ways functions like Pod::Text. See *Pod::Text* for details and available options.

BUGS

Currently, the outermost formatting instruction wins, so for example underlined text inside a region of bold text is displayed as simply bold. There may be some better approach possible.

COMPATIBILITY

Pod::Text::Overstrike 1.01 (based on Pod::Parser) was the first version of this module included with Perl, in Perl 5.6.1.

The current API based on *Pod::Simple* was added in Pod::Text::Overstrike 2.00, included in Perl 5.9.3.

Several problems with wrapping and line length were fixed as recently as Pod::Text::Overstrike 6.0.0.

This module inherits its API and most behavior from Pod::Text, so the details in *"COMPATIBILITY" in Pod::Text* also apply. Pod::Text and Pod::Text::Overstrike have had the same module version since 4.00, included in Perl 5.23.7. (They unfortunately diverge in confusing ways prior to that.)

CAVEATS

Line wrapping is done only at ASCII spaces and tabs, rather than using a correct Unicode-aware line wrapping algorithm.

AUTHOR

Originally written by Joe Smith Joe.Smith@inwap.com, using the framework created by Russ Allbery rra@cpan.org. Subsequently updated by Russ Allbery.

COPYRIGHT AND LICENSE

Copyright 2000 Joe Smith Joe.Smith@inwap.com

Copyright 2001, 2004, 2008, 2014, 2018-2019, 2022, 2024, 2026 Russ Allbery rra@cpan.org

This program is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

SEE ALSO

Pod::Text, *Pod::Simple*

The current version of this module is always available from its web site at https://www.eyrie.org/~eagle/software/podlators/. It is also part of the Perl core distribution as of 5.6.0.

Pod::Text::Termcap

Convert POD data to ASCII text with format escapes

SYNOPSIS

```
use Pod::Text::Termcap;
my $parser = Pod::Text::Termcap->new (sentence => 0, width => 78);

# Read POD from STDIN and write to STDOUT.
$parser->parse_from_filehandle;

# Read POD from file.pod and write to file.txt.
$parser->parse_from_file ('file.pod', 'file.txt');
```

DESCRIPTION

Pod::Text::Termcap is a simple subclass of Pod::Text that highlights output text using the correct termcap escape sequences for the current terminal. Apart from the format codes, it in all ways functions like Pod::Text. See *Pod::Text* for details and available options.

This module uses *Term::Cap* to find the correct terminal settings. See the documentation of that module for how it finds terminal database information and how to override that behavior if necessary. If unable to find control strings for bold and underscore formatting, that formatting is skipped, resulting in the same output as Pod::Text.

COMPATIBILITY

Pod::Text::Termcap 0.04 (based on Pod::Parser) was the first version of this module included with Perl, in Perl 5.6.0.

The current API based on *Pod::Simple* was added in Pod::Text::Termcap 2.00. Pod::Text::Termcap 2.01 was included in Perl 5.9.3, the first version of Perl to incorporate those changes.

Pod::Text::Termcap 4.13 stopped setting the TERMPATH environment variable during module load. It also stopped falling back on VT100 escape sequences if Term::Cap was not able to find usable escape sequences, instead producing unformatted output for better results on dumb terminals. The next version to be incorporated into Perl, 4.14, was included in Perl 5.31.8.

Several problems with wrapping and line length were fixed as recently as Pod::Text::Termcap 6.0.0.

This module inherits its API and most behavior from Pod::Text, so the details in *"COMPATIBILITY" in Pod::Text* also apply. Pod::Text and Pod::Text::Termcap have had the same module version since 4.00, included in Perl 5.23.7. (They unfortunately diverge in confusing ways prior to that.)

CAVEATS

Line wrapping is done only at ASCII spaces and tabs, rather than using a correct Unicode-aware line wrapping algorithm.

AUTHOR

Russ Allbery rra@cpan.org

COPYRIGHT AND LICENSE

Copyright 1999, 2001-2002, 2004, 2006, 2008-2009, 2014-2015, 2018-2019, 2022, 2024, 2026 Russ Allbery rra@cpan.org

This program is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

SEE ALSO

Pod::Text, *Pod::Simple*, *Term::Cap*

The current version of this module is always available from its web site at https://www.eyrie.org/~eagle/software/podlators/. It is also part of the Perl core distribution as of 5.6.0.

Pod::Usage

extracts POD documentation and shows usage information

SYNOPSIS

```
use Pod::Usage;

my $message_text  = "This text precedes the usage message.";
my $exit_status   = 2;          ## The exit status to use
my $verbose_level = 0;          ## The verbose level to use
my $filehandle    = \*STDERR;   ## The filehandle to write to

pod2usage($message_text);

pod2usage($exit_status);

pod2usage( { -message => $message_text ,
             -exitval => $exit_status  ,
             -verbose => $verbose_level,
             -output  => $filehandle } );

pod2usage(   -msg     => $message_text ,
             -exitval => $exit_status  ,
             -verbose => $verbose_level,
             -output  => $filehandle );
```

(continues on next page)

(continued from previous page)

```
pod2usage(    -verbose => 2,
              -noperldoc => 1  );

pod2usage(    -verbose => 2,
              -perlcmd => $path_to_perl,
              -perldoc => $path_to_perldoc,
              -perldocopt => $perldoc_options );
```

ARGUMENTS

pod2usage should be given either a single argument, or a list of arguments corresponding to an associative array (a "hash"). When a single argument is given, it should correspond to exactly one of the following:

- A string containing the text of a message to print *before* printing the usage message
- A numeric value corresponding to the desired exit status
- A reference to a hash

If more than one argument is given then the entire argument list is assumed to be a hash. If a hash is supplied (either as a reference or as a list) it should contain one or more elements with the following keys:

- `-message` *string*
- `-msg` *string*

  The text of a message to print immediately prior to printing the program's usage message.

- `-exitval` *value*

  The desired exit status to pass to the **exit()** function. This should be an integer, or else the string `NOEXIT` to indicate that control should simply be returned without terminating the invoking process.

- `-verbose` *value*

  The desired level of "verboseness" to use when printing the usage message. If the value is 0, then only the "SYNOPSIS" and/or "USAGE" sections of the pod documentation are printed. If the value is 1, then the "SYNOPSIS" and/or "USAGE" sections, along with any section entitled "OPTIONS", "ARGUMENTS", or "OPTIONS AND ARGUMENTS" is printed. If the corresponding value is 2 or more then the entire manpage is printed, using perldoc if available; otherwise *Pod::Text* is used for the formatting. For better readability, the all-capital headings are downcased, e.g. `SYNOPSIS` => `Synopsis`.

  The special verbosity level 99 requires to also specify the -sections parameter; then these sections are extracted and printed.

- `-sections` *spec*

  There are two ways to specify the selection. Either a string (scalar) representing a selection regexp for sections to be printed when -verbose is set to 99, e.g.

```
"NAME|SYNOPSIS|DESCRIPTION|VERSION"
```

With the above regexp all content following (and including) any of the given =head1 headings will be shown. It is possible to restrict the output to particular subsections only, e.g.:

```
"DESCRIPTION/Algorithm"
```

This will output only the =head2 Algorithm heading and content within the =head1 DESCRIPTION section. The regexp binding is stronger than the section separator, such that e.g.:

```
"DESCRIPTION|OPTIONS|ENVIRONMENT/Caveats"
```

will print any =head2 Caveats section (only) within any of the three =head1 sections.

Alternatively, an array reference of section specifications can be used:

```
pod2usage(-verbose => 99, -sections => [
  qw(DESCRIPTION DESCRIPTION/Introduction) ] );
```

This will print only the content of =head1 DESCRIPTION and the =head2 Introduction sections, but no other =head2, and no other =head1 either.

- `-output` *handle*

  A reference to a filehandle, or the pathname of a file to which the usage message should be written. The default is *STDERR unless the exit value is less than 2 (in which case the default is *STDOUT).

- `-input` *handle*

  A reference to a filehandle, or the pathname of a file from which the invoking script's pod documentation should be read. It defaults to the file indicated by $0 ($PROGRAM_NAME for users of English.pm).

  If you are calling **pod2usage()** from a module and want to display that module's POD, you can use this:

```
use Pod::Find qw(pod_where);
pod2usage( -input => pod_where({-inc => 1}, __PACKAGE__) );
```

- `-pathlist` *string*

  A list of directory paths. If the input file does not exist, then it will be searched for in the given directory list (in the order the directories appear in the list). It defaults to the list of directories implied by $ENV{PATH}. The list may be specified either by a reference to an array, or by a string of directory paths which use the same path separator as $ENV{PATH} on your system (e.g., : for Unix, ; for MSWin32 and DOS).

- `-noperldoc`

  By default, Pod::Usage will call perldoc when -verbose >= 2 is specified. This does not work well e.g. if the script was packed with PAR. This option suppresses the

external call to perldoc and uses the simple text formatter (*Pod::Text*) to output the POD.

- `-perlcmd`

  By default, Pod::Usage will call perldoc when -verbose >= 2 is specified. In case of special or unusual Perl installations, this option may be used to supply the path to a perl executable which should run perldoc.

- `-perldoc` *path-to-perldoc*

  By default, Pod::Usage will call perldoc when -verbose >= 2 is specified. In case perldoc is not installed where the perl interpreter thinks it is (see *Config*), the -perldoc option may be used to supply the correct path to perldoc.

- `-perldocopt` *string*

  By default, Pod::Usage will call perldoc when -verbose >= 2 is specified. This option may be used to supply options to perldoc. The string may contain several, space-separated options.

Formatting base class

The default text formatter is *Pod::Text*. The base class for Pod::Usage can be defined by pre-setting `$Pod::Usage::Formatter` *before* loading Pod::Usage, e.g.:

```
BEGIN { $Pod::Usage::Formatter = 'Pod::Text::Termcap'; }
use Pod::Usage qw(pod2usage);
```

Pod::Usage uses *Pod::Simple*'s _handle_element_end() method to implement the section selection, and in case of verbosity < 2 it down-cases the all-caps headings to first capital letter and rest lowercase, and adds a colon/newline at the end of the headings, for better readability. Same for verbosity = 99.

Pass-through options

The following options are passed through to the underlying text formatter. See the manual pages of these modules for more information.

```
alt code indent loose margin quotes sentence stderr utf8 width
```

DESCRIPTION

pod2usage will print a usage message for the invoking script (using its embedded pod documentation) and then exit the script with the desired exit status. The usage message printed may have any one of three levels of "verboseness": If the verbose level is 0, then only a synopsis is printed. If the verbose level is 1, then the synopsis is printed along with a description (if present) of the command line options and arguments. If the verbose level is 2, then the entire manual page is printed.

Unless they are explicitly specified, the default values for the exit status, verbose level, and output stream to use are determined as follows:

- If neither the exit status nor the verbose level is specified, then the default is to use an exit status of 2 with a verbose level of 0.

- If an exit status *is* specified but the verbose level is *not*, then the verbose level will default to 1 if the exit status is less than 2 and will default to 0 otherwise.
- If an exit status is *not* specified but verbose level *is* given, then the exit status will default to 2 if the verbose level is 0 and will default to 1 otherwise.
- If the exit status used is less than 2, then output is printed on STDOUT. Otherwise output is printed on STDERR.

Although the above may seem a bit confusing at first, it generally does "the right thing" in most situations. This determination of the default values to use is based upon the following typical Unix conventions:

- An exit status of 0 implies "success". For example, **diff(1)** exits with a status of 0 if the two files have the same contents.
- An exit status of 1 implies possibly abnormal, but non-defective, program termination. For example, **grep(1)** exits with a status of 1 if it did *not* find a matching line for the given regular expression.
- An exit status of 2 or more implies a fatal error. For example, **ls(1)** exits with a status of 2 if you specify an illegal (unknown) option on the command line.
- Usage messages issued as a result of bad command-line syntax should go to STDERR. However, usage messages issued due to an explicit request to print usage (like specifying **-help** on the command line) should go to STDOUT, just in case the user wants to pipe the output to a pager (such as **more(1)**).
- If program usage has been explicitly requested by the user, it is often desirable to exit with a status of 1 (as opposed to 0) after issuing the user-requested usage message. It is also desirable to give a more verbose description of program usage in this case.

pod2usage does not force the above conventions upon you, but it will use them by default if you don't expressly tell it to do otherwise. The ability of **pod2usage()** to accept a single number or a string makes it convenient to use as an innocent looking error message handling function:

```
use strict;
use Pod::Usage;
use Getopt::Long;

## Parse options
my %opt;
GetOptions(\%opt, "help|?", "man", "flag1")  ||  pod2usage(2);
pod2usage(1)  if ($opt{help});
pod2usage(-exitval => 0, -verbose => 2)  if ($opt{man});

## Check for too many filenames
pod2usage("$0: Too many files given.\n")  if (@ARGV > 1);
```

Some user's however may feel that the above "economy of expression" is not particularly readable nor consistent and may instead choose to do something more like the following:

```
use strict;
use Pod::Usage qw(pod2usage);
use Getopt::Long qw(GetOptions);

## Parse options
my %opt;
GetOptions(\%opt, "help|?", "man", "flag1")  ||
  pod2usage(-verbose => 0);

pod2usage(-verbose => 1)  if ($opt{help});
pod2usage(-verbose => 2)  if ($opt{man});

## Check for too many filenames
pod2usage(-verbose => 2, -message => "$0: Too many files given.\n")
  if (@ARGV > 1);
```

As with all things in Perl, *there's more than one way to do it*, and **pod2usage()** adheres to this philosophy. If you are interested in seeing a number of different ways to invoke **pod2usage** (although by no means exhaustive), please refer to "EXAMPLES".

Scripts

The Pod::Usage distribution comes with a script pod2usage which offers a command line interface to the functionality of Pod::Usage. See pod2usage.

EXAMPLES

Each of the following invocations of `pod2usage()` will print just the "SYNOPSIS" section to STDERR and will exit with a status of 2:

```
pod2usage();

pod2usage(2);

pod2usage(-verbose => 0);

pod2usage(-exitval => 2);

pod2usage({-exitval => 2, -output => \*STDERR});

pod2usage({-verbose => 0, -output  => \*STDERR});

pod2usage(-exitval => 2, -verbose => 0);

pod2usage(-exitval => 2, -verbose => 0, -output => \*STDERR);
```

Each of the following invocations of `pod2usage()` will print a message of "Syntax error." (followed by a newline) to STDERR, immediately followed by just the "SYNOPSIS" section (also printed to STDERR) and will exit with a status of 2:

```
pod2usage("Syntax error.");

pod2usage(-message => "Syntax error.", -verbose => 0);

pod2usage(-msg  => "Syntax error.", -exitval => 2);

pod2usage({-msg => "Syntax error.", -exitval => 2, -output => \
 ↪*STDERR});

pod2usage({-msg => "Syntax error.", -verbose => 0, -output => \
 ↪*STDERR});

pod2usage(-msg  => "Syntax error.", -exitval => 2, -verbose => 0);

pod2usage(-message => "Syntax error.",
          -exitval => 2,
          -verbose => 0,
          -output  => \*STDERR);
```

Each of the following invocations of pod2usage() will print the "SYNOPSIS" section and any "OPTIONS" and/or "ARGUMENTS" sections to STDOUT and will exit with a status of 1:

```
pod2usage(1);

pod2usage(-verbose => 1);

pod2usage(-exitval => 1);

pod2usage({-exitval => 1, -output => \*STDOUT});

pod2usage({-verbose => 1, -output => \*STDOUT});

pod2usage(-exitval => 1, -verbose => 1);

pod2usage(-exitval => 1, -verbose => 1, -output => \*STDOUT});
```

Each of the following invocations of pod2usage() will print the entire manual page to STDOUT and will exit with a status of 1:

```
pod2usage(-verbose  => 2);

pod2usage({-verbose => 2, -output => \*STDOUT});

pod2usage(-exitval  => 1, -verbose => 2);

pod2usage({-exitval => 1, -verbose => 2, -output => \*STDOUT});
```

Recommended Use

Most scripts should print some type of usage message to STDERR when a command line syntax error is detected. They should also provide an option (usually -H or -help) to print a (possibly more verbose) usage message to STDOUT. Some scripts may even wish to go so far as to provide a means of printing their complete documentation to STDOUT (perhaps by allowing a -man option). The following complete example uses **Pod::Usage** in combination with **Getopt::Long** to do all of these things:

```
use strict;
use Getopt::Long qw(GetOptions);
use Pod::Usage qw(pod2usage);

my $man = 0;
my $help = 0;
## Parse options and print usage if there is a syntax error,
## or if usage was explicitly requested.
GetOptions('help|?' => \$help, man => \$man) or pod2usage(2);
pod2usage(1) if $help;
pod2usage(-verbose => 2) if $man;

## If no arguments were given, then allow STDIN to be used only
## if it's not connected to a terminal (otherwise print usage)
pod2usage("$0: No files given.")  if ((@ARGV == 0) && (-t STDIN));

__END__

=head1 NAME

sample - Using GetOpt::Long and Pod::Usage

=head1 SYNOPSIS

sample [options] [file ...]

 Options:
   -help            brief help message
   -man             full documentation

=head1 OPTIONS

=over 4

=item B<-help>

Prints a brief help message and exits.

=item B<-man>

Prints the manual page and exits.
```

(continues on next page)

(continued from previous page)

```
=back

=head1 DESCRIPTION

B<This program> will read the given input file(s) and do something
useful with the contents thereof.

=cut
```

CAVEATS

By default, **pod2usage()** will use `$0` as the path to the pod input file. Unfortunately, not all systems on which Perl runs will set `$0` properly (although if `$0` is not found, **pod2usage()** will search `$ENV{PATH}` or else the list specified by the `-pathlist` option). If this is the case for your system, you may need to explicitly specify the path to the pod docs for the invoking script using something similar to the following:

```
pod2usage(-exitval => 2, -input => "/path/to/your/pod/docs");
```

In the pathological case that a script is called via a relative path *and* the script itself changes the current working directory (see *"chdir" in perlfunc*) *before* calling pod2usage, Pod::Usage will fail even on robust platforms. Don't do that. Or use *FindBin* to locate the script:

```
use FindBin;
pod2usage(-input => $FindBin::Bin . "/" . $FindBin::Script);
```

SUPPORT

This module is managed in a GitHub repository, https://github.com/Dual-Life/Pod-Usage Feel free to fork and contribute, or to clone and send patches!

Please use https://github.com/Dual-Life/Pod-Usage/issues/new to file a bug report. The previous ticketing system, https://rt.cpan.org/Dist/Display.html?Queue=Pod-Usage, is deprecated for this package.

More general questions or discussion about POD should be sent to the `pod-people@perl.org` mail list. Send an empty email to `pod-people-subscribe@perl.org` to subscribe.

AUTHOR

Marek Rouchal marekr@cpan.org

Nicolas R nicolas@atoomic.org

Brad Appleton bradapp@enteract.com

Based on code for **Pod::Text::pod2text()** written by Tom Christiansen tchrist@mox.perl.com

LICENSE

Pod::Usage (the distribution) is licensed under the same terms as Perl.

ACKNOWLEDGMENTS

Nicolas R (ATOOMIC) for setting up the Github repo and modernizing this package.

rjbs for refactoring Pod::Usage to not use Pod::Parser any more.

Steven McDougall swmcd@world.std.com for his help and patience with re-writing this manpage.

SEE ALSO

Pod::Usage is now a standalone distribution, depending on *Pod::Text* which in turn depends on *Pod::Simple*.

Pod::Perldoc, *Getopt::Long*, Pod::Find, *FindBin*, *Pod::Text*, *Pod::Text::Termcap*, *Pod::Simple*

5.2.85 SDBM_File

Tie a Perl hash to an on-disk SDBM database so keys and values persist between runs.

SDBM_File is a **tied-hash interface**: you never call `FETCH`, `STORE`, `DELETE`, or `EXISTS` directly. You bind a hash to a database file with `tie`, and from then on any ordinary hash syntax - `$h{key}`, `delete $h{key}`, `exists $h{key}`, `keys %h`, `each %h` - is routed by Perl's tie machinery to the uppercase hook methods defined here.

The database itself lives in two files alongside each other, `NAME.dir` and `NAME.pag`, that persist between program runs. Neither `untie` nor program exit removes them - clean-up is the caller's responsibility.

Synopsis

```
use Fcntl;
use SDBM_File;

tie my %h, 'SDBM_File', 'mydb', O_RDWR | O_CREAT, 0666
    or die "tie failed: $!";
$h{greeting} = 'hello';
print $h{greeting};        # hello
untie %h;
```

Persistence

Tying creates (or opens) two files named after the `$basename` argument: `$basename.dir` and `$basename.pag`. Values written through the tied hash land in these files immediately and survive the process. `untie` closes the handle but leaves the files in place for the next run. Use `unlink` if you want them gone.

The six-argument form of `tie` lets you name the two files independently when the `.dir` / `.pag` naming convention does not fit - typically with `File::Temp` handles.

Size limits

SDBM is a fixed-page format. The combined length of a key and its value must not exceed **1008 bytes** (the `PAIRMAX` constant). A `STORE` for a too-large pair croaks with `sdbm store returned -1, errno 22, key "..."` and the pair is not written. The two filename-extension constants `PAGFEXT` and `DIRFEXT` are exported for code that needs to locate the backing files.

Security

Do not open SDBM files from untrusted sources. The on-disk layout was designed for speed, not integrity, and a maliciously crafted file can drive the reader into undefined behavior.

This is a native implementation of SDBM_File 1.17, its sdbm library included; `SDBM_File.pm` is used unchanged. The files it writes are byte for byte those perl5's SDBM_File writes, and each reads the other's.

Differences from upstream

None.

Functions

Tie hooks

`error`

Report whether the database has a sticky I/O-error flag set. Called as a method on the tied object: `(tied %h)->error`.

`sdbm_clearerr`

Clear the sticky I/O-error flag on the database. Called as a method on the tied object: `(tied %h)->sdbm_clearerr`.

Filters

`filter_fetch_key`

Install a callback that rewrites keys on the way out of the database. Called as a method on the tied object: `(tied %h)->filter_fetch_key(sub { ... })`.

`filter_store_key`

Install a callback that rewrites keys on the way into the database. Called as a method on the tied object: `(tied %h)->filter_store_key(sub { ... })`.

`filter_fetch_value`

Install a callback that rewrites values on the way out of the database. Called as a method on the tied object: `(tied %h)->filter_fetch_value(sub { ... })`.

`filter_store_value`

Install a callback that rewrites values on the way into the database. Called as a method on the tied object: `(tied %h)->filter_store_value(sub { ... })`.

5.2.86 SDL2

Perl bindings for SDL2 - windows, input, timers and hardware-accelerated 2D drawing.

SDL2 is the library behind a large share of the cross-platform games and media applications written in the last decade. This module makes it available to Perl with no compiled binding to build: drawing and input go to the `libSDL2-2.0.so.0` installed on the machine, which is located the first time the module is used. If that library is absent, the first SDL call dies with a message naming the distribution package to install.

Two layers

There are two ways to use this module, and they interoperate freely.

The **direct layer** mirrors SDL2's C API one-for-one - same function names, same argument order, same return values. Every prototype in SDL2's own documentation applies unchanged, which is what you want when you are following an SDL tutorial or porting C:

```
use SDL2 qw(:all);

SDL_Init(SDL_INIT_VIDEO);
my $win = SDL_CreateWindow('hello', 100, 100, 640, 480, SDL_WINDOW_
 ↪SHOWN);
```

The **object layer** wraps the same calls as Perl objects that free themselves, which is what you want when writing Perl rather than transcribing C:

```
use SDL2;

my $win = SDL2::Window->new(title => 'hello', w => 640, h => 480);
my $ren = $win->renderer;
```

Handles pass between the layers in both directions: a direct-layer function accepts an object wherever it expects a handle, and `SDL2::Window->from_handle($h)` adopts a raw handle.

Lifetime

An object frees its SDL resource when the last Perl reference to it goes away, so nothing has to be destroyed by hand. Raw handles from the direct layer are yours to destroy, exactly as in C. Destroying a resource through the direct layer while an object also holds it is safe: the object is emptied at the same time, so it will not free it a second time.

Errors

The two layers report failure differently, each following its own convention. A direct-layer function returns what C returns - a negative integer or an undefined handle - and leaves the reason in `SDL_GetError()`. An object-layer method croaks with that text instead, so an error is an exception you can trap with `eval`.

Because a failed object-layer call is an exception, a successful one returns the object itself. That means calls chain:

```
$ren->set_draw_color(30, 30, 40, 255)->clear;
$ren->fill_rect([10, 10, 40, 20]);
$ren->present;
```

and it means **`or die` is for the object layer only**. On a direct-layer call it is always wrong, because C reports success with zero:

```
$ren->clear or die;              # correct: clear croaks, so this
                                 # never fires
SDL_RenderClear($ren) or die;    # WRONG: 0 IS success
SDL_RenderClear($ren) == 0 or die "clear: " . SDL_GetError();   #␣
↪right
```

The one object-layer method that returns nothing is `destroy`. It is the explicit finalizer, so there is nothing left to chain from.

Exports and constants

`use SDL2` exports nothing. Every direct-layer `SDL_*` function and every constant can be asked for by name, or by tag: `:init` (`SDL_INIT_*`), `:window` (`SDL_WINDOW_*`, `SDL_WINDOWPOS_*`), `:gl` (`SDL_GL_*` attributes), `:render` (renderer, flip, texture-access and blend-mode flags), `:events` (event types such as `SDL_QUIT` and `SDL_KEYDOWN`, window events, mouse buttons), `:keycode` (`SDLK_*`, `KMOD_*`), `:scancode` (`SDL_SCANCODE_*`), `:pixels` (`SDL_PIXELFORMAT_*`), `:syswm` (`SDL_SYSWM_*`), `:misc` (`IMG_INIT_*`, for `SDL2::Image`), and `:all` for everything. The values are those of SDL 2's own headers. The object-layer methods are reached through their objects and are never exported.

Modules

- `SDL2::Events` - Keyboard, mouse and window events.
- `SDL2::Image` - loading and saving image files
- `SDL2::Render` - Hardware-accelerated 2D drawing: renderers and textures.
- `SDL2::Surface` - Surfaces - images in ordinary memory.
- `SDL2::Video` - Windows and OpenGL contexts.

Functions

Lifecycle

SDL_Init

Initialise the SDL subsystems named by the OR of `SDL_INIT_*` flags.

SDL_InitSubSystem

Initialise one further subsystem after `SDL_Init` has run.

SDL_Quit

Shut SDL down and release everything it holds.

SDL_QuitSubSystem

Shut down just the subsystems named by the flags.

SDL_WasInit

Which of the subsystems named by the flags are currently initialised, as the OR of their `SDL_INIT_*` bits. Pass 0 to ask about all of them.

have_sdl

True when libSDL2 was found on this machine.

Errors and version

SDL_GetError

The message describing the last SDL call that failed, or the empty string if none has. The direct layer leaves failures here rather than dying, so this is how you find out what went wrong.

SDL_ClearError

Forget the last error message.

SDL_GetVersion

The version of the SDL2 library actually loaded, as the list `($major, $minor, $patch)`.

version

The loaded SDL2 library's version as a string, `"2.30.1"`.

SDL_GetPlatform

The name SDL gives this platform, such as `"Linux"`.

Timing

SDL_GetTicks

Milliseconds since SDL was initialised.

SDL_Delay

Sleep for the given number of milliseconds.

Renderers

SDL_CreateRenderer

Creates a renderer for a window and returns its handle, or undef.

SDL_DestroyRenderer

Frees a renderer. Accepts a raw handle or an `SDL2::Renderer`, emptying the object so it will not free the renderer again.

SDL_RenderPresent

Shows everything drawn since the last present. The last call of every frame; nothing appears on screen until you make it.

SDL_RenderSetLogicalSize

Fixes the coordinate system the renderer draws in, letting SDL scale to whatever size the window actually is. Draw against 320x240 and it stays correct when the user resizes. Returns 0 on success.

SDL_GetRendererOutputSize

The renderer's drawable size in pixels, as the list `($w, $h)`.

SDL_RenderReadPixels

Reads pixels back from the rendering target as a byte string, four bytes per pixel in RGBA order, row by row with no padding.

SDL_SetRenderTarget

Directs subsequent drawing into a texture instead of the window. Pass undef to go back to the window. The texture must have been created with `SDL_TEXTUREACCESS_TARGET`. Returns 0 on success.

Drawing

SDL_RenderClear

Fills the whole rendering target with the current draw color. The first call of every frame. Returns 0 on success.

SDL_SetRenderDrawColor

Sets the color used by the drawing calls and by `SDL_RenderClear`. Components are 0 to 255; alpha defaults to 255 (opaque) and only has an effect when the blend mode is not `SDL_BLENDMODE_NONE`.

SDL_GetRenderDrawColor

The current draw color, as the list `($r, $g, $b, $a)`.

SDL_SetRenderDrawBlendMode

Chooses how drawing combines with what is already there. `SDL_BLENDMODE_NONE` overwrites; `SDL_BLENDMODE_BLEND` composites using the alpha channel, which is what you want for anything translucent. Returns 0 on success.

SDL_RenderDrawPoint

Draws one pixel in the current draw color. Returns 0 on success.

SDL_RenderDrawLine

Draws a line between two points in the current draw color. Returns 0 on success.

SDL_RenderDrawRect

Draws the outline of a rectangle. Pass undef for the whole target. Returns 0 on success.

SDL_RenderFillRect

Fills a rectangle with the current draw color. Pass undef for the whole target. Returns 0 on success.

SDL_RenderCopy

Draws a texture. `$src` selects part of the texture and `$dst` says where on the target it goes; undef means the whole of either. Returns 0 on success.

SDL_RenderCopyEx

Draws a texture rotated and/or mirrored. `$angle` is in degrees clockwise, `$center` is the pivot as `[x, y]` relative to `$dst` (undef for its middle), and `$flip` is the OR of `SDL_FLIP_*`. Returns 0 on success.

Textures

SDL_CreateTexture

Creates an empty texture. `$access` is one of `SDL_TEXTUREACCESS_STATIC`, `STREAMING` or `TARGET`. Returns the handle, or undef on failure.

SDL_CreateTextureFromSurface

Uploads a surface to the GPU as a texture. Returns the handle, or undef.

SDL_DestroyTexture

Frees a texture. Accepts a raw handle or an `SDL2::Texture`.

SDL_QueryTexture

A texture's properties as the list `($format, $access, $w, $h)`.

SDL_SetTextureAlphaMod

Scales the texture's alpha when it is drawn, 0 to 255. Returns 0 on success.

SDL_SetTextureColorMod

Tints the texture when it is drawn: each channel is multiplied by the value you give divided by 255. Returns 0 on success.

SDL_SetTextureBlendMode

Chooses how the texture combines with what is under it. Returns 0 on success.

Event queue

SDL_PollEvent

Takes the next event off the queue and returns it as a hash reference, or undef when there is nothing waiting.

SDL_PushEvent

Puts an event on the queue, so a later `SDL_PollEvent` returns it.

SDL_WaitEvent

Waits for the next event and returns it as a hash reference, blocking until one arrives. Returns undef if the wait fails.

SDL_WaitEventTimeout

Waits up to `$ms` milliseconds for an event. Returns the event as a hash reference, or undef if the timeout passed with nothing to report.

SDL_PumpEvents

Collects pending input from the operating system into SDL's queue.

SDL_FlushEvents

Discards queued events whose type falls between `$min` and `$max` inclusive. Called with no arguments it discards everything.

SDL_HasEvents

True when the queue holds an event whose type is between `$min` and `$max` inclusive.

Keyboard

SDL_GetKeyboardState

The whole keyboard as a string of bytes, one per scancode, each 1 when that key is down. Index it with a `SDL_SCANCODE_*` constant:

key_pressed

True when the physical key named by a `SDL_SCANCODE_*` constant is held down right now.

SDL_GetModState

The modifier keys held down right now, as the OR of the `KMOD_*` constants.

SDL_GetKeyName

The printable name of a key symbol, such as `"Left"` or `"A"`. The empty string if there is no such key.

SDL_GetScancodeName

The printable name of a physical key position, such as `"Left"`.

SDL_GetKeyFromName

The key symbol with the given name - the inverse of `SDL_GetKeyName`. Returns `SDLK_UNKNOWN` if the name is not one SDL knows.

SDL_GetKeyFromScancode

The key symbol a physical key currently produces, which depends on the user's keyboard layout.

SDL_GetScancodeFromKey

The physical key that currently produces a key symbol.

Mouse

SDL_GetMouseState

The mouse as the list `($buttons, $x, $y)`. `$buttons` is a bitmask - test it against `SDL_BUTTON_LMASK` and its siblings - and the position is relative to the focused window.

Surfaces

SDL_SaveBMP

Writes a surface as a BMP file. Returns 0 on success.

SDL_FreeSurface

Frees a surface. Accepts a raw handle or an `SDL2::Surface`.

Windows

SDL_CreateWindow

Creates a window and returns its handle, or undef on failure.

SDL_DestroyWindow

Closes a window and frees it. Safe to pass either a raw handle or an `SDL2::Window`; passing the object also empties it, so its own cleanup will not free the window a second time.

SDL_SetWindowTitle

Sets the window's caption.

SDL_GetWindowTitle

The window's caption.

SDL_SetWindowSize

Resizes the window's client area to `$w` by `$h` pixels.

SDL_GetWindowSize

The window's client area as the list `($w, $h)`.

SDL_SetWindowPosition

Moves the window so its top-left corner is at `($x, $y)`.

SDL_GetWindowPosition

The window's top-left corner as the list `($x, $y)`.

SDL_ShowWindow

Makes a hidden window visible.

SDL_HideWindow

Hides the window without destroying it.

SDL_RaiseWindow

Brings the window to the front and gives it the input focus.

SDL_MaximizeWindow

Makes the window as large as the window manager allows.

SDL_MinimizeWindow

Iconifies the window.

SDL_RestoreWindow

Returns the window to its size and position from before it was minimized or maximized.

SDL_SetWindowFullscreen

Switches the window between windowed and full screen. Pass 0 for windowed, `SDL_WINDOW_FULLSCREEN` for a real video-mode change, or `SDL_WINDOW_FULLSCREEN_DESKTOP` for a borderless window filling the display - which is what most programs want. Returns 0 on success.

SDL_SetWindowResizable

Allows or forbids the user resizing the window.

SDL_SetWindowBordered

Shows or hides the window's border and title bar.

SDL_GetWindowFlags

The OR of the `SDL_WINDOW_*` bits describing the window's current state.

SDL_GetWindowID

The window's numeric id - what an event's `window_id` field holds, so this is how you tell which of several windows an event came from.

SDL_GetWindowFromID

The window handle for a numeric window id, or undef if there is no such window. This is how an event's `window_id` becomes the window it happened in, when a program has more than one.

SDL_UpdateWindowSurface

Copies the window's surface to the screen. Only for windows drawn through `SDL_GetWindowSurface`; a window with a renderer uses `SDL_RenderPresent` instead. Returns 0 on success.

SDL_GetCurrentVideoDriver

The name of the video driver SDL initialised, or undef when the video subsystem is not up.

SDL_GetWindowWMInfo

The window manager's own handles for this window, as a hash reference, or undef when SDL will not say.

OpenGL

SDL_GL_SetAttribute

Requests an OpenGL context attribute BEFORE the window is created. Setting `SDL_GL_CONTEXT_MAJOR_VERSION`, `SDL_GL_CONTEXT_MINOR_VERSION` and `SDL_GL_CONTEXT_PROFILE_MASK` is how you ask for a specific OpenGL version. Returns 0 on success.

SDL_GL_GetAttribute

The value an OpenGL context attribute actually has - which can differ from what you asked for, since the driver may give you more than you requested. Returns the list `($result, $value)`, `$result` being 0 on success.

SDL_GL_CreateContext

Creates an OpenGL context for a window created with `SDL_WINDOW_OPENGL`, and makes it current. Returns the handle, or undef on failure. Free it with `SDL_GL_DeleteContext`.

SDL_GL_DeleteContext

Frees an OpenGL context.

SDL_GL_MakeCurrent

Makes a context current on a window, which is what you call when rendering to more than one window. Returns 0 on success.

SDL_GL_SwapWindow

Shows the frame OpenGL has just drawn - the OpenGL counterpart of `SDL_RenderPresent`.

SDL_GL_SetSwapInterval

Chooses how `SDL_GL_SwapWindow` waits: 0 for no wait, 1 to sync with the display's refresh, -1 for adaptive sync. Returns 0 on success, and -1 if the request is not supported.

SDL_GL_GetSwapInterval

The swap interval currently in effect.

SDL_GL_GetProcAddress

The address of an OpenGL function, as an integer, or undef if this driver does not have it.

SDL_GL_GetDrawableSize

The drawable size in pixels as the list `($w, $h)`. On a high-DPI display this is larger than the window size, which is in screen coordinates - use this one for `glViewport`.

SDL2::Events

Keyboard, mouse and window events.

`SDL_PollEvent` takes the next event off the queue and returns it as a plain hash reference, or `undef` when the queue is empty. Draining the queue each frame is the shape of every SDL program:

```
while (my $ev = SDL_PollEvent()) {
    last if $ev->{type} == SDL_QUIT;
    if ($ev->{type} == SDL_KEYDOWN) {
        $paddle_x -= 25 if $ev->{sym} == SDLK_LEFT;
        $paddle_x += 25 if $ev->{sym} == SDLK_RIGHT;
    }
}
```

Every event has `type` and `timestamp`. What else it carries depends on the type, and only the relevant keys are present:

Event	Keys
SDL_QUIT	-
SDL_KEYDOWN, SDL_KEYUP	window_id, state, repeat, scancode, sym, mod, key_name
SDL_TEXTINPUT	window_id, text
SDL_MOUSEMOTION	window_id, which, state, x, y, xrel, yrel
SDL_MOUSEBUTTONDOWN, SDL_MOUSEBUTTONUP	window_id, which, button, state, clicks, x, y
SDL_MOUSEWHEEL	window_id, which, x, y, direction, precise_x, precise_y
SDL_WINDOWEVENT	window_id, event, data1, data2

`sym` is the key as a layout-dependent symbol - compare it against the `SDLK_*` constants, which is what you want for "the user pressed left". `scancode` is the physical key position, layout-independent, and compares against `SDL_SCANCODE_*`; that is what you want for WASD, which should stay in the same place on an AZERTY keyboard.

Held keys

Events tell you when a key changed. For "is it held down right now", which is what smooth movement needs, ask the keyboard state instead:

```
SDL_PumpEvents();
$x -= 4 if SDL2::key_pressed(SDL_SCANCODE_LEFT);
$x += 4 if SDL2::key_pressed(SDL_SCANCODE_RIGHT);
```

`key_pressed` reads SDL's live key array directly and allocates nothing, so it is safe to call several times per frame.

SDL2::Image

`SDL2::Image` - loading and saving image files.

Core SDL2 reads and writes exactly one image format, BMP. This module adds the rest through `libSDL2_image`: PNG and JPEG both ways, and reading for everything else that library was built with - GIF, WEBP, TIFF, AVIF, QOI, XPM and more.

```
use SDL2;
use SDL2::Image;

my $tex = SDL2::Image::load_texture($ren, 'sprite.png');
$ren->copy($tex, undef, [10, 10, 64, 64]);

$ren->save_png('screenshot.png');     # the frame just drawn
```

Saving what you drew

`$ren->save_png($path)`, `$ren->save_jpg($path, $quality)` and `$ren->save_bmp($path)` write the renderer's current target to a file. They exist because doing it by hand has two traps that produce a wrong-looking image rather than an error:

- the bytes come back **RGBA**, and most image libraries want a different channel order, so they have to be swizzled;
- a row of pixels is not always `width * 4` bytes - libraries pad rows to an alignment, and the padding has to be skipped.

These methods do neither by hand: the pixels go straight into a surface that already agrees with them, and SDL2_image writes it.

What this build can do

`SDL2::Image::have_image()` says whether libSDL2_image was found at all, and `SDL2::Image::init($flags)` asks it to load the codecs you need, returning the ones it got. A format its build lacks fails on use rather than at load.

Functions

Loading

load

Reads an image file into an `SDL2::Surface`, guessing the format from the file's own contents rather than its name. Croaks if the file cannot be read or its format is not one this build has.

`load_texture`

Reads an image file straight into an `SDL2::Texture` for a renderer, which is what you want for a sprite: it skips the surface entirely.

Saving

`save_png`

Writes a surface as a PNG. Returns 0 on success.

`save_jpg`

Writes a surface as a JPEG at the given quality, 0 to 100. Returns 0 on success.

Library information

`have_image`

True when libSDL2_image was found on this machine.

`init`

Loads the codecs named by the OR of `IMG_INIT_*` flags, returning the OR of those it actually got - which may be fewer than you asked for.

`quit`

Releases the codecs `init` loaded.

SDL2::Render

Hardware-accelerated 2D drawing: renderers and textures.

A renderer draws into a window, using the GPU when one is available and falling back to software when it is not. It is the fast path for 2D work: clear the target, draw shapes and textures, then present the finished frame.

Rectangles

Anywhere a rectangle is wanted you may pass an array reference `[$x, $y, $w, $h]`, a hash reference `{x => .., y => .., w => .., h => ..}`, or `undef` to mean "the whole target". The array form is the short one and is what the examples use.

The `SDL2::Renderer` object

Usually obtained from a window rather than constructed directly:

```
my $ren = $win->renderer;                    # accelerated, vsync␣
↪off
my $ren = $win->renderer(vsync => 1);        # wait for the refresh
my $ren = $win->renderer(software => 1);     # force software␣
↪rendering
```

Its methods:

- `$ren->clear` - fill the whole target with the draw color.
- `$ren->present` - show the frame just drawn.
- `$ren->set_draw_color($r, $g, $b, $a)` - alpha defaults to 255.
- `$ren->draw_color` - the list `($r, $g, $b, $a)`.
- `$ren->set_blend_mode($mode)` - one of the `SDL_BLENDMODE_*` constants.
- `$ren->draw_point($x, $y)`, `$ren->draw_line($x1, $y1, $x2, $y2)`.
- `$ren->draw_rect($rect)` - outline; `$ren->fill_rect($rect)` - solid.
- `$ren->copy($texture, $src, $dst)` - blit a texture.
- `$ren->copy_ex($texture, $src, $dst, $angle, $center, $flip)` - blit rotated and/or flipped.
- `$ren->output_size` - the list `($w, $h)` of the drawable area.
- `$ren->set_logical_size($w, $h)` - draw in fixed coordinates and let SDL scale to the real window size.
- `$ren->read_pixels($rect)` - the raw RGBA bytes of the target, four bytes per pixel, row by row. Useful for saving a frame or checking in a test what was actually drawn.
- `$ren->handle`, `$ren->destroy`.

The SDL2::Texture object

A texture is an image living in the renderer's memory, which is what makes drawing it fast.

- `SDL2::Texture->new($renderer, %opts)` - `w`, `h`, `access`, `format`.
- `$tex->size` - the list `($w, $h)`.
- `$tex->set_alpha_mod($a)`, `$tex->set_color_mod($r, $g, $b)`, `$tex->set_blend_mode($mode)` - modulation applied when it is drawn.
- `$tex->handle`, `$tex->destroy`.

SDL2::Surface

Surfaces - images in ordinary memory.

A surface is a block of pixels the CPU can read and write, as opposed to a texture, which lives in the renderer's memory and is fast to draw but awkward to inspect. Loading an

image file gives you a surface; `$ren->read_pixels` gives you the bytes to make one from; and saving a frame goes through one.

The SDL2::Surface object

```
# wrap pixels you already have - four bytes per pixel, RGBA order
my $surf = SDL2::Surface->from_pixels($bytes, $w, $h);

$surf->save_bmp('frame.bmp');            # core SDL2, always␣
↪available
my $tex = $ren->texture_from_surface($surf);
```

Its methods:

- `SDL2::Surface->from_pixels($bytes, $w, $h, [$format])` - a surface over a copy of `$bytes`. The default format is RGBA, four bytes per pixel, which is what `read_pixels` returns.
- `$surf->size` - the list `($w, $h)`.
- `$surf->pitch` - the bytes per row, which is not always `$w * 4`.
- `$surf->pixels` - the raw bytes back again.
- `$surf->save_bmp($path)` - write it out. BMP needs no extra library, so it works everywhere; for PNG or JPEG see `SDL2::Image`.
- `$surf->handle`, `$surf->destroy`, and `SDL2::Surface->from_handle($h)` to adopt a raw handle from the direct layer.

SDL2::Video

Windows and OpenGL contexts.

This is the part of SDL2 that puts something on the screen: it opens a window, moves and resizes it, and - when you ask for one - gives it an OpenGL context to render into.

The SDL2::Window object

`SDL2::Window->new` takes named arguments, all optional except that a window with no size is not much use:

```
my $win = SDL2::Window->new(
    title => 'my program',   # window caption
    x     => undef,          # position; undef means "let the system␣
↪choose"
    y     => undef,
    w     => 640,            # size in pixels (default 640 x 480)
    h     => 480,
    flags => 0,              # OR of SDL_WINDOW_* constants
    shown => 1,              # false opens the window hidden
    opengl => 0,             # true adds SDL_WINDOW_OPENGL
    resizable => 0,          # true adds SDL_WINDOW_RESIZABLE
);
```

It calls `SDL_Init(SDL_INIT_VIDEO)` for you if the video subsystem is not up yet, and croaks if the window cannot be created. The window closes and is freed when the last reference to `$win` goes away.

Its methods:

- `$win->title` / `$win->set_title($text)` - the caption.
- `$win->size` - the list `($w, $h)`; `$win->set_size($w, $h)` to change it.
- `$win->position` - the list `($x, $y)`; `$win->set_position($x, $y)`.
- `$win->show` / `$win->hide` / `$win->raise` - visibility.
- `$win->minimize` / `$win->maximize` / `$win->restore`.
- `$win->set_fullscreen($flags)` - 0 for windowed, `SDL_WINDOW_FULLSCREEN_DESKTOP` for borderless full screen.
- `$win->id` - the numeric id events carry in their `window_id` field.
- `$win->flags` - the OR of the `SDL_WINDOW_*` bits currently set.
- `$win->wm_info` - the window manager's own handles for this window, as a hash reference. What a graphics API other than OpenGL needs in order to draw here; see below.
- `$win->renderer(%opts)` - a `SDL2::Renderer` drawing into this window.
- `$win->gl_context` - an `SDL2::GLContext`, made current immediately.
- `$win->gl_swap` - show the frame just drawn by OpenGL.
- `$win->destroy` - close it now rather than at scope exit.
- `$win->handle` - the raw pointer, for passing to the direct layer.

The `SDL2::GLContext` object

`$win->gl_context` returns one, already current. Ask for the OpenGL version you need BEFORE creating the window, with `SDL_GL_SetAttribute`:

```
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 3);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 3);
SDL_GL_SetAttribute(SDL_GL_CONTEXT_PROFILE_MASK, SDL_GL_CONTEXT_
 ↪PROFILE_CORE);

my $win = SDL2::Window->new(w => 800, h => 600, opengl => 1);
my $ctx = $win->gl_context;
```

Its methods are `$ctx->make_current($win)`, `$ctx->handle` and `$ctx->destroy`. `SDL_GL_GetProcAddress($name)` resolves an OpenGL entry point through the context, which is how a Perl OpenGL binding finds the modern functions without GLEW.

Drawing on the window with something other than OpenGL

A GPU API that is not OpenGL does not take a context from SDL; it takes the window manager's own handles and builds its own surface on them. `$win->wm_info` is where those come from:

```
my $wm = $win->wm_info;
# { subsystem => 2, driver => 'x11', display => ..., window => ... }
```

`driver` is the same lowercase name `SDL_GetCurrentVideoDriver` returns, and it decides which other keys are there: `display` and `window` under X11, `display` and `surface` under Wayland. They are plain integers - a C pointer and an X11 XID - meant to be handed straight back to whatever asked for them, not followed from Perl.

`Peta::WebGPU::Device->new(window => $win)` is the one consumer in the runtime today; it reads exactly these keys.

SDL_CreateRenderer

Creates a renderer for a window and returns its handle, or undef.

```
my $ren = SDL_CreateRenderer($win, -1, SDL_RENDERER_ACCELERATED);
```

`-1` asks SDL to choose the driver, which is what you almost always want. `$flags` is the OR of `SDL_RENDERER_*`: `ACCELERATED` for the GPU, `PRESENTVSYNC` to sync presentation with the display's refresh, `SOFTWARE` to force CPU rendering, `TARGETTEXTURE` to allow rendering into a texture. Free it with `SDL_DestroyRenderer`.

SDL_PushEvent

Puts an event on the queue, so a later `SDL_PollEvent` returns it.

Takes the same hash reference shape `SDL_PollEvent` hands back, so an event can be captured, altered and replayed. Give at least `type`; any field that type carries may be set and the rest default to zero.

```
SDL_PushEvent({ type => SDL_KEYDOWN, sym => SDLK_LEFT });
SDL_PushEvent({ type => SDL_QUIT });
```

Returns 1 if the event was queued, 0 if a filter dropped it, and a negative number on error. Synthesising input this way is how a test drives a program that normally waits for a person.

SDL_GetKeyboardState

The whole keyboard as a string of bytes, one per scancode, each 1 when that key is down. Index it with a `SDL_SCANCODE_*` constant:

```
my $keys = SDL_GetKeyboardState();
my $left = ord substr $keys, SDL_SCANCODE_LEFT, 1;
```

`SDL2::key_pressed($scancode)` is the same question without building the string, and is what a per-frame check should use.

SDL_CreateWindow

Creates a window and returns its handle, or undef on failure.

```
my $win = SDL_CreateWindow($title, $x, $y, $w, $h, $flags);
```

`$x` and `$y` are pixel positions, or `SDL_WINDOWPOS_CENTERED` / `SDL_WINDOWPOS_UNDEFINED`. `$flags` is the OR of `SDL_WINDOW_*` constants - `SDL_WINDOW_SHOWN` to make it visible immediately, `SDL_WINDOW_OPENGL` if it will carry an OpenGL context, `SDL_WINDOW_RESIZABLE` to let the user resize it.

The window must be freed with `SDL_DestroyWindow`. `SDL2::Window->new` is the same call with a Perl object that frees itself.

SDL_GetWindowWMInfo

The window manager's own handles for this window, as a hash reference, or undef when SDL will not say.

```
my $wm = SDL_GetWindowWMInfo($win) or die SDL_GetError();
printf "%s display=%#x window=%#x\n", $wm->{driver}, $wm->{display},
 ↪$wm->{window};
```

`subsystem` is the numeric `SDL_SYSWM_*` value and `driver` is the same lowercase name `SDL_GetCurrentVideoDriver` uses. The remaining keys are the ones that windowing system has: `display` and `window` under X11, `display` and `surface` under Wayland - as plain integers, because they are C pointers and XIDs, not anything Perl can follow.

This is what a graphics API that is not OpenGL needs in order to draw on an SDL window: `Peta::WebGPU` creates its surface from exactly these two handles.

5.2.87 Safe

Compile and execute code in restricted compartments

SYNOPSIS

```
use Safe;

$compartment = new Safe;

$compartment->permit(qw(time sort :browse));

$result = $compartment->reval($unsafe_code);
```

DESCRIPTION

The Safe extension module allows the creation of compartments in which perl code can be evaluated. Each compartment has

- a new namespace

The "root" of the namespace (i.e. "main::") is changed to a different package and code evaluated in the compartment cannot refer to variables outside this namespace, even with run-time glob lookups and other tricks.

Code which is compiled outside the compartment can choose to place variables into (or *share* variables with) the compartment's namespace and only that data will be visible to code evaluated in the compartment.

By default, the only variables shared with compartments are the "underscore" variables $_ and @_ (and, technically, the less frequently used %_, the _ filehandle and so on). This is because otherwise perl operators which default to $_ will not work and neither will the assignment of arguments to @_ on subroutine entry.

- an operator mask

  Each compartment has an associated "operator mask". Recall that perl code is compiled into an internal format before execution. Evaluating perl code (e.g. via "eval" or "do 'file'") causes the code to be compiled into an internal format and then, provided there was no error in the compilation, executed. Code evaluated in a compartment compiles subject to the compartment's operator mask. Attempting to evaluate code in a compartment which contains a masked operator will cause the compilation to fail with an error. The code will not be executed.

  The default operator mask for a newly created compartment is the ':default' optag.

  It is important that you read the *Opcode* module documentation for more information, especially for detailed definitions of opnames, optags and opsets.

  Since it is only at the compilation stage that the operator mask applies, controlled access to potentially unsafe operations can be achieved by having a handle to a wrapper subroutine (written outside the compartment) placed into the compartment. For example,

```
$cpt = new Safe;
sub wrapper {
  # vet arguments and perform potentially unsafe operations
}
$cpt->share('&wrapper');
```

WARNING

The Safe module does not implement an effective sandbox for evaluating untrusted code with the perl interpreter.

Bugs in the perl interpreter that could be abused to bypass Safe restrictions are not treated as vulnerabilities. See perlsecpolicy for additional information.

The authors make **no warranty**, implied or otherwise, about the suitability of this software for safety or security purposes.

The authors shall not in any case be liable for special, incidental, consequential, indirect or other similar damages arising from the use of this software.

Your mileage will vary. If in any doubt **do not use it**.

METHODS

To create a new compartment, use

```
$cpt = new Safe;
```

Optional argument is (NAMESPACE), where NAMESPACE is the root namespace to use for the compartment (defaults to "Safe::Root0", incremented for each new compartment).

Note that version 1.00 of the Safe module supported a second optional parameter, MASK. That functionality has been withdrawn pending deeper consideration. Use the permit and deny methods described below.

The following methods can then be used on the compartment object returned by the above constructor. The object argument is implicit in each case.

permit (OP, ...)

Permit the listed operators to be used when compiling code in the compartment (in *addition* to any operators already permitted).

You can list opcodes by names, or use a tag name; see *"Predefined Opcode Tags" in Opcode*.

permit_only (OP, ...)

Permit *only* the listed operators to be used when compiling code in the compartment (*no* other operators are permitted).

deny (OP, ...)

Deny the listed operators from being used when compiling code in the compartment (other operators may still be permitted).

deny_only (OP, ...)

Deny *only* the listed operators from being used when compiling code in the compartment (*all* other operators will be permitted, so you probably don't want to use this method).

trap (OP, ...), untrap (OP, ...)

The trap and untrap methods are synonyms for deny and permit respectfully.

share (NAME, ...)

This shares the variable(s) in the argument list with the compartment. This is almost identical to exporting variables using the *Exporter* module.

Each NAME must be the **name** of a non-lexical variable, typically with the leading type identifier included. A bareword is treated as a function name.

Examples of legal names are '$foo' for a scalar, '@foo' for an array, '%foo' for a hash, '&foo' or 'foo' for a subroutine and '*foo' for a glob (i.e. all symbol table entries associated with "foo", including scalar, array, hash, sub and filehandle).

Each NAME is assumed to be in the calling package. See share_from for an alternative method (which `share` uses).

share_from (PACKAGE, ARRAYREF)

This method is similar to share() but allows you to explicitly name the package that symbols should be shared from. The symbol names (including type characters) are supplied as an array reference.

```
$safe->share_from('main', [ '$foo', '%bar', 'func' ]);
```

Names can include package names, which are relative to the specified PACKAGE. So these two calls have the same effect:

```
$safe->share_from('Scalar::Util', [ 'reftype' ]);
$safe->share_from('main', [ 'Scalar::Util::reftype' ]);
```

varglob (VARNAME)

This returns a glob reference for the symbol table entry of VARNAME in the package of the compartment. VARNAME must be the **name** of a variable without any leading type marker. For example:

```
${$cpt->varglob('foo')} = "Hello world";
```

has the same effect as:

```
$cpt = new Safe 'Root';
$Root::foo = "Hello world";
```

but avoids the need to know $cpt's package name.

reval (STRING, STRICT)

This evaluates STRING as perl code inside the compartment.

The code can only see the compartment's namespace (as returned by the **root** method). The compartment's root package appears to be the `main::` package to the code inside the compartment.

Any attempt by the code in STRING to use an operator which is not permitted by the compartment will cause an error (at run-time of the main program but at compile-time for the code in STRING). The error is of the form "'%s' trapped by operation mask...".

If an operation is trapped in this way, then the code in STRING will not be executed. If such a trapped operation occurs or any other compile-time or return error, then $@ is set to the error message, just as with an eval().

If there is no error, then the method returns the value of the last expression evaluated, or a return statement may be used, just as with subroutines and **eval()**. The context (list or scalar) is determined by the caller as usual.

If the return value of reval() is (or contains) any code reference, those code references are wrapped to be themselves executed always in the compartment. See "wrap_code_refs_within".

The formerly undocumented STRICT argument sets strictness: if true 'use strict;' is used, otherwise it uses 'no strict;'. **Note**: if STRICT is omitted 'no strict;' is the default.

Some points to note:

If the entereval op is permitted then the code can use eval "..." to 'hide' code which might use denied ops. This is not a major problem since when the code tries to execute the eval it will fail because the opmask is still in effect. However this technique would allow clever, and possibly harmful, code to 'probe' the boundaries of what is possible.

Any string eval which is executed by code executing in a compartment, or by code called from code executing in a compartment, will be eval'd in the namespace of the compartment. This is potentially a serious problem.

Consider a function foo() in package pkg compiled outside a compartment but shared with it. Assume the compartment has a root package called 'Root'. If foo() contains an eval statement like eval '$foo = 1' then, normally, $pkg::foo will be set to 1. If foo() is called from the compartment (by whatever means) then instead of setting $pkg::foo, the eval will actually set $Root::pkg::foo.

This can easily be demonstrated by using a module, such as the Socket module, which uses eval "..." as part of an AUTOLOAD function. You can 'use' the module outside the compartment and share an (autoloaded) function with the compartment. If an autoload is triggered by code in the compartment, or by any code anywhere that is called by any means from the compartment, then the eval in the Socket module's AUTOLOAD function happens in the namespace of the compartment. Any variables created or used by the eval'd code are now under the control of the code in the compartment.

A similar effect applies to *all* runtime symbol lookups in code called from a compartment but not compiled within it.

rdo (FILENAME)

This evaluates the contents of file FILENAME inside the compartment. It uses the same rules as perl's built-in do to locate the file, poossibly using `@INC`.

See above documentation on the **reval** method for further details.

root (NAMESPACE)

This method returns the name of the package that is the root of the compartment's namespace.

Note that this behaviour differs from version 1.00 of the Safe module where the root module could be used to change the namespace. That functionality has been withdrawn pending deeper consideration.

mask (MASK)

This is a get-or-set method for the compartment's operator mask.

With no MASK argument present, it returns the current operator mask of the compartment.

With the MASK argument present, it sets the operator mask for the compartment (equivalent to calling the deny_only method).

wrap_code_ref (CODEREF)

Returns a reference to an anonymous subroutine that, when executed, will call CODEREF with the Safe compartment 'in effect'. In other words, with the package namespace adjusted and the opmask enabled.

Note that the opmask doesn't affect the already compiled code, it only affects any *further* compilation that the already compiled code may try to perform.

This is particularly useful when applied to code references returned from reval().

(It also provides a kind of workaround for RT#60374: "Safe.pm sort {} bug with -Dusethreads". See https://rt.perl.org/rt3//Public/Bug/Display.html?id=60374 for *much* more detail.)

wrap_code_refs_within (...)

Wraps any CODE references found within the arguments by replacing each with the result of calling "wrap_code_ref" on the CODE reference. Any ARRAY or HASH references in the arguments are inspected recursively.

Returns nothing.

RISKS

This section is just an outline of some of the things code in a compartment might do (intentionally or unintentionally) which can have an effect outside the compartment.

- Memory

  Consuming all (or nearly all) available memory.

- CPU

  Causing infinite loops etc.

- Snooping

  Copying private information out of your system. Even something as simple as your user name is of value to others. Much useful information could be gleaned from your environment variables for example.

- Signals

  Causing signals (especially SIGFPE and SIGALARM) to affect your process.

  Setting up a signal handler will need to be carefully considered and controlled. What mask is in effect when a signal handler gets called? If a user can get an imported function to get an exception and call the user's signal handler, does that

user's restricted mask get re-instated before the handler is called? Does an imported handler get called with its original mask or the user's one?

- State Changes

  Ops such as chdir obviously effect the process as a whole and not just the code in the compartment. Ops such as rand and srand have a similar but more subtle effect.

AUTHOR

Originally designed and implemented by Malcolm Beattie.

Reworked to use the Opcode module and other changes added by Tim Bunce.

Currently maintained by the Perl 5 Porters, perl5-porters@perl.org.

5.2.88 Scalar

Namespace containing the following modules:

- `Scalar::List`
- `Scalar::Util` - General-utility subroutines for inspecting and tagging scalar values.

Scalar::List

Namespace containing the following modules:

- `Scalar::List::Utils` - A distribution of general-utility subroutines

Scalar::List::Utils

A distribution of general-utility subroutines

SYNOPSIS

```
use Scalar::Util qw(blessed);
use List::Util qw(any);
```

DESCRIPTION

`Scalar::List::Utils` does nothing on its own. It is packaged with several useful modules.

MODULES

List::Util

List::Util contains a selection of useful subroutines for operating on lists of values.

Scalar::Util

Scalar::Util contains a selection of useful subroutines for interrogating or manipulating scalar values.

Sub::Util

Sub::Util contains a selection of useful subroutines for interrogating or manipulating subroutine references.

Scalar::Util

General-utility subroutines for inspecting and tagging scalar values.

What kind of thing a value is - an object of which class (`blessed`), which basic type underneath (`reftype`), which address (`refaddr`), a number in disguise (`looks_like_number`), a v-string (`isvstring`), read-only (`readonly`), tainted (`tainted`), an open filehandle (`openhandle`) - and two ways of making one: a value with separate number and string sides (`dualvar`), and a reference that does not keep its target alive (`weaken`). Nothing is exported by default; name what you need.

This is Scalar::Util from Scalar-List-Utils 1.70; `Scalar/Util.pm` is upstream's. On perl 5.40 and later it makes `blessed`, `refaddr`, `reftype`, `weaken`, `unweaken` and `isweak` the `builtin::` functions of the same names (`isweak` is `builtin::is_weak`), as upstream does; the others are ListUtil.xs's XS functions.

Synopsis

```
use Scalar::Util qw(blessed reftype refaddr looks_like_number
↪dualvar
                    isdual readonly weaken isweak openhandle);

my $obj = bless {}, 'My::Class';
print blessed($obj), ' ', reftype($obj), "\n";        # My::Class
↪HASH
print defined(blessed({})) ? "blessed\n" : "plain\n";  # plain

print join(',', map { looks_like_number($_) ? 1 : 0 }
                    42, '4.2e1', ' 42 ', 'abc', undef), "\n";  # 1,
↪1,1,0,0

my $v = dualvar(5, 'five');
print $v + 1, ' ', "$v", "\n";                         # 6 five

my $ref = { name => 'x' };
my %cache = (x => $ref);
weaken($cache{x});                  # the cache does not keep it
↪alive
undef $ref;
print defined $cache{x} ? "kept\n" : "gone\n";         # gone
```

(continues on next page)

(continued from previous page)

```
print openhandle(\*STDIN) ? "open\n" : "closed\n";     # open
```

The builtin functions

- `blessed($ref)` - the class name of an object, undef for anything else (an unblessed reference, a plain scalar).
- `reftype($ref)` - the type a reference points to - HASH, ARRAY, SCALAR, CODE, GLOB, REGEXP, ... - ignoring any blessing; undef for a non-reference.
- `refaddr($ref)` - the address a reference points to, as a number; the same for every reference to the same thing, whatever its class or overloading.
- `weaken($ref)` - make this copy of a reference weak: it does not count towards keeping the target alive, and becomes undef when the target is freed. Copying it gives a strong reference again.
- `unweaken($ref)` - make a weak reference strong again.
- `isweak($ref)` - true if the reference is weak.

Differences from upstream

None: every function is checked against perl 5.44 by t/81-xs-native/Scalar-Util, and upstream's own tests pass unchanged (t/81-xs-native/List-Util, the `upstream` files).

Modules

- `Scalar::Util::Numeric` - Numeric tests for scalars: is a value a number, and what kind.

Functions

Inspection

`looks_like_number`

True if perl would use the value as a number without a "not numeric" warning: a number, or a string of digits with an optional sign, decimal point and exponent, surrounding whitespace allowed (`"42"`, `"-.5"`, `" 4.2e1 "`), or `"Inf"`, `"Infinity"`, `"NaN"`. False for undef, the empty string, `"abc"`, `"1e"`, and for the notations only literals have (`"0x10"`, `"1_000"`). An object that overloads `0+` counts by the value it converts to.

`tainted`

`tainted($var)` - returns true if `$var` is tainted.

readonly

`readonly($var)` - returns true if `$var` is read-only.

openhandle

The argument itself if it is an open filehandle - a glob, a reference to one, or an IO object, opened (or tied) - and undef otherwise: for a handle never opened, one already closed, or anything that is no handle.

isdual

True if the value currently has both a string and a numeric side - a `dualvar`, `$!`, but also an ordinary string once it has been used as a number (`my $n = "10"; $n + 0;` makes `isdual($n)` true), as perl caches the conversion.

isvstring

True if the value is a v-string - written as `v1.2.3`, or as a bare number with two or more dots (`1.2.3`) - and false for the string `"1.2.3"`.

Construction

dualvar

`dualvar($num, $str)`: a value that is `$num` where a number is used and `$str` where a string is - as `$!` is both an error number and its message.

Other Functions

reftype

`reftype($ref)` - returns type string without blessing.

set_prototype

`set_prototype($code, $prototype)` - sets the prototype of the subroutine given by the `$code` reference, or deletes it if `$prototype` is `undef`.

Scalar::Util::Numeric

Numeric tests for scalars: is a value a number, and what kind.

Each function asks perl's own number parser (`grok_number`, the one `looks_like_number` uses) about the string form of its argument, so a string and a number answer alike: `isint("42")` and `isint(42)` are both true.

Synopsis

```
use Scalar::Util::Numeric qw(isnum isint isfloat isneg);

print isnum("3.5e2") ? "number\n" : "not\n";   # number
print isint(" 7 "), "\n";                      # 1
print isint("-7"), "\n";                       # -1
print isfloat("1e3"), "\n";                    # 1
print isnum("abc"), "\n";                      # 0
```

Functions

`isnum($v)` is `grok_number`'s answer: 0 for a non-number, otherwise a set of flags - 0x01 the value fits a UV, 0x02 it is larger, 0x04 it has a fraction or an exponent, 0x08 it is negative, 0x10 infinity, 0x20 NaN. `isint` is 1 for a non-negative integer, -1 for a negative one and 0 otherwise; `isuv`, `isbig`, `isfloat`, `isneg`, `isinf` and `isnan` each test one flag and return 1 or 0. `uvmax()` is the largest UV. `undef` is not a number. Nothing is exported by default; `:all` exports the eight tests.

This is a native implementation of Scalar::Util::Numeric 0.40's `Numeric.xs`; its `Scalar/Util/Numeric.pm` is used unchanged.

Differences from upstream

None.

Functions

Numeric tests

uvmax

The largest unsigned integer perl holds natively.

Synopsis

```
my $max = Scalar::Util::Numeric::uvmax();   # 18446744073709551615
```

isnum

Whether a value is a number, and which kind.

isint

Whether a value is an integer, and its sign.

isuv

Whether a value is a number that fits an unsigned integer.

isbig

Whether a value is a number larger than the unsigned range.

isfloat

Whether a value is a number with a fraction or an exponent.

isneg

Whether a value is a negative number.

Synopsis

```
use Scalar::Util::Numeric qw(isneg);
isneg("-3");   # 1
```

What you get back

1 when the number has a leading minus sign, else 0.

isinf

Whether a value is infinity.

isnan

Whether a value is NaN.

Synopsis

```
use Scalar::Util::Numeric qw(isnan);
isnan("NaN");   # 1
```

What you get back

1 for a NaN, in any spelling perl reads as NaN, else 0.

isnum

Whether a value is a number, and which kind.

Synopsis

```
use Scalar::Util::Numeric qw(isnum);
my $flags = isnum($v);
```

What you get back

0 when $v is not a number or is undef; otherwise the flags of perl's number parser: 0x01 fits an unsigned integer, 0x02 is larger, 0x04 has a fraction or exponent, 0x08 is negative, 0x10 is infinity, 0x20 is NaN. Leading and trailing whitespace is allowed, as perl allows it when it converts a string.

isint

Whether a value is an integer, and its sign.

Synopsis

```
use Scalar::Util::Numeric qw(isint);
isint(42);       # 1
isint("-42");    # -1
isint(4.2);      # 0
```

What you get back

1 for a non-negative integer within the unsigned range, -1 for a negative integer within it, 0 for anything else - fractions, exponents, numbers beyond the range, and non-numbers.

isuv

Whether a value is a number that fits an unsigned integer.

Synopsis

```
use Scalar::Util::Numeric qw(isuv);
isuv("1.5");     # 1 - the integer part fits
```

What you get back

1 when the number's integer part is within the unsigned range, whatever its sign or fraction, else 0.

isbig

Whether a value is a number larger than the unsigned range.

Synopsis

```
use Scalar::Util::Numeric qw(isbig);
isbig("18446744073709551616");   # 1
```

What you get back

1 when the number's integer part exceeds the largest unsigned integer, else 0.

isfloat

Whether a value is a number with a fraction or an exponent.

Synopsis

```
use Scalar::Util::Numeric qw(isfloat);
isfloat("1.0");   # 1
isfloat("1e3");   # 1
isfloat(10);      # 0
```

What you get back

1 when the number is written with a decimal point or an exponent, or is infinity or NaN, else 0.

isinf

Whether a value is infinity.

Synopsis

```
use Scalar::Util::Numeric qw(isinf);
isinf("Inf");     # 1
isinf(9**9**9);   # 1
```

What you get back

1 for positive or negative infinity, in any spelling perl reads as infinity, else 0.

tainted

`tainted($var)` - returns true if `$var` is tainted.

Under taint mode (`-T`) everything that comes from outside the program is tainted: `%ENV`, `@ARGV`, `$0`, data read from files, and any value derived from them. Constants and values computed only from constants are not. Outside taint mode nothing is tainted, so this always returns false. Takes exactly one argument.

```
$taint = tainted("constant");       # false
$taint = tainted($ENV{PWD});        # true if running under -T
```

readonly

`readonly($var)` - returns true if `$var` is read-only.

The true value is not 1. Exactly as in perl5, it is the raw internal flag word, so a read-only scalar yields a number such as 134283264. Test it for truth; never compare it against 1. Takes exactly one argument.

```
sub foo { readonly($_[0]) }
$ro = foo($bar);                    # false
$ro = foo(0);                       # true
```

set_prototype

`set_prototype($code, $prototype)` - sets the prototype of the subroutine given by the `$code` reference, or deletes it if `$prototype` is `undef`.

Returns the `$code` reference itself, so calls can be chained. The code reference comes FIRST and the prototype second; `Sub::Util` spells the same operation with the arguments the other way round.

```
set_prototype \&foo, '$$';
```

The `&$` prototype makes the first argument a `\&name` or `sub {...}` and a call with fewer than two arguments a compile-time error. Called with `&` to bypass the prototype, a missing prototype argument deletes the prototype, and a first argument that is not a reference to a subroutine croaks with `set_prototype: not a subroutine reference` (`set_prototype: not a reference` when it is no reference at all).

5.2.89 Search

Namespace containing the following modules:

- `Search::Dict` - look - search for key in dictionary file

Search::Dict

look - search for key in dictionary file

SYNOPSIS

```
use Search::Dict;
look *FILEHANDLE, $key, $dict, $fold;

use Search::Dict;
look *FILEHANDLE, $key, $params;
```

DESCRIPTION

Sets file position in FILEHANDLE to be first line greater than or equal (stringwise) to *$key*. Returns the new file position, or -1 if an error occurs.

The flags specify dictionary order and case folding:

If *$dict* is true, search by dictionary order (ignore anything but word characters and whitespace). The default is honour all characters.

If *$fold* is true, ignore case. The default is to honour case.

If there are only three arguments and the third argument is a hash reference, the keys of that hash can have values `dict`, `fold`, and `comp` or `xfrm` (see below), and their corresponding values will be used as the parameters.

If a comparison subroutine (comp) is defined, it must return less than zero, zero, or greater than zero, if the first comparand is less than, equal, or greater than the second comparand.

If a transformation subroutine (xfrm) is defined, its value is used to transform the lines read from the filehandle before their comparison.

5.2.90 SelectSaver

save and restore selected file handle

SYNOPSIS

```
use SelectSaver;

{
   my $saver = SelectSaver->new(FILEHANDLE);
   # FILEHANDLE is selected
}
# previous handle is selected

{
   my $saver = SelectSaver->new;
   # new handle may be selected, or not
}
# previous handle is selected
```

DESCRIPTION

A `SelectSaver` object contains a reference to the file handle that was selected when it was created. If its new method gets an extra parameter, then that parameter is selected; otherwise, the selected file handle remains unchanged.

When a `SelectSaver` is destroyed, it re-selects the file handle that was selected when it was created.

5.2.91 SelfLoader

load functions only on demand

SYNOPSIS

```
package FOOBAR;
use SelfLoader;

... (initializing code)

__DATA__
sub {....
```

DESCRIPTION

This module tells its users that functions in the FOOBAR package are to be autoloaded from after the __DATA__ token. See also *"Autoloading" in perlsub*.

The __DATA__ token

The __DATA__ token tells the perl compiler that the perl code for compilation is finished. Everything after the __DATA__ token is available for reading via the filehandle FOOBAR::DATA, where FOOBAR is the name of the current package when the __DATA__ token is reached. This works just the same as __END__ does in package 'main', but for other modules data after __END__ is not automatically retrievable, whereas data after __DATA__ is. The __DATA__ token is not recognized in versions of perl prior to 5.001m.

Note that it is possible to have __DATA__ tokens in the same package in multiple files, and that the last __DATA__ token in a given package that is encountered by the compiler is the one accessible by the filehandle. This also applies to __END__ and main, i.e. if the 'main' program has an __END__, but a module 'require'd (_not_ 'use'd) by that program has a 'package main;' declaration followed by an '__DATA__', then the DATA filehandle is set to access the data after the __DATA__ in the module, _not_ the data after the __END__ token in the 'main' program, since the compiler encounters the 'require'd file later.

SelfLoader autoloading

The **SelfLoader** works by the user placing the __DATA__ token *after* perl code which needs to be compiled and run at 'require' time, but *before* subroutine declarations that can be loaded in later - usually because they may never be called.

The **SelfLoader** will read from the FOOBAR::DATA filehandle to load in the data after __DATA__, and load in any subroutine when it is called. The costs are the one-time parsing of the data after __DATA__, and a load delay for the _first_ call of any autoloaded function. The benefits (hopefully) are a speeded up compilation phase, with no need to load functions which are never used.

The **SelfLoader** will stop reading from __DATA__ if it encounters the __END__ token - just as you would expect. If the __END__ token is present, and is followed by the token DATA, then the **SelfLoader** leaves the FOOBAR::DATA filehandle open on the line after that token.

The **SelfLoader** exports the AUTOLOAD subroutine to the package using the **SelfLoader**, and this loads the called subroutine when it is first called.

There is no advantage to putting subroutines which will _always_ be called after the __DATA__ token.

Autoloading and package lexicals

A 'my $pack_lexical' statement makes the variable $pack_lexical local _only_ to the file up to the __DATA__ token. Subroutines declared elsewhere _cannot_ see these types of variables, just as if you declared subroutines in the package but in another file, they cannot see these variables.

So specifically, autoloaded functions cannot see package lexicals (this applies to both the **SelfLoader** and the Autoloader). The `vars` pragma provides an alternative to defining package-level globals that will be visible to autoloaded routines. See the documentation on **vars** in the pragma section of perlmod.

SelfLoader and AutoLoader

The **SelfLoader** can replace the AutoLoader - just change 'use AutoLoader' to 'use SelfLoader' (though note that the **SelfLoader** exports the AUTOLOAD function - but if you have your own AUTOLOAD and are using the AutoLoader too, you probably know what you're doing), and the __END__ token to __DATA__. You will need perl version 5.001m or later to use this (version 5.001 with all patches up to patch m).

There is no need to inherit from the **SelfLoader**.

The **SelfLoader** works similarly to the AutoLoader, but picks up the subs from after the __DATA__ instead of in the 'lib/auto' directory. There is a maintenance gain in not needing to run AutoSplit on the module at installation, and a runtime gain in not needing to keep opening and closing files to load subs. There is a runtime loss in needing to parse the code after the __DATA__. Details of the **AutoLoader** and another view of these distinctions can be found in that module's documentation.

__DATA__, __END__, and the FOOBAR::DATA filehandle

This section is only relevant if you want to use the FOOBAR::DATA together with the **SelfLoader**.

Data after the __DATA__ token in a module is read using the FOOBAR::DATA filehandle. __END__ can still be used to denote the end of the __DATA__ section if followed by the token DATA - this is supported by the **SelfLoader**. The FOOBAR::DATA filehandle is left open if an __END__ followed by a DATA is found, with the filehandle positioned at the start of the line after the __END__ token. If no __END__ token is present, or an __END__ token with no DATA token on the same line, then the filehandle is closed.

The **SelfLoader** reads from wherever the current position of the FOOBAR::DATA filehandle is, until the EOF or __END__. This means that if you want to use that filehandle (and ONLY if you want to), you should either

1. Put all your subroutine declarations immediately after the __DATA__ token and put your own data after those declarations, using the __END__ token to mark the end of

subroutine declarations. You must also ensure that the **SelfLoader** reads first by calling 'SelfLoader->load_stubs();', or by using a function which is selfloaded;

or

2. You should read the `FOOBAR::DATA` filehandle first, leaving the handle open and positioned at the first line of subroutine declarations.

You could conceivably do both.

Classes and inherited methods

For modules which are not classes, this section is not relevant. This section is only relevant if you have methods which could be inherited.

A subroutine stub (or forward declaration) looks like

```
sub stub;
```

i.e. it is a subroutine declaration without the body of the subroutine. For modules which are not classes, there is no real need for stubs as far as autoloading is concerned.

For modules which ARE classes, and need to handle inherited methods, stubs are needed to ensure that the method inheritance mechanism works properly. You can load the stubs into the module at 'require' time, by adding the statement 'SelfLoader->load_stubs();' to the module to do this.

The alternative is to put the stubs in before the `__DATA__` token BEFORE releasing the module, and for this purpose the `Devel::SelfStubber` module is available. However this does require the extra step of ensuring that the stubs are in the module. If this is done I strongly recommend that this is done BEFORE releasing the module - it should NOT be done at install time in general.

Multiple packages and fully qualified subroutine names

Subroutines in multiple packages within the same file are supported - but you should note that this requires exporting the `SelfLoader::AUTOLOAD` to every package which requires it. This is done automatically by the **SelfLoader** when it first loads the subs into the cache, but you should really specify it in the initialization before the `__DATA__` by putting a 'use SelfLoader' statement in each package.

Fully qualified subroutine names are also supported. For example,

```
__DATA__
sub foo::bar {23}
package baz;
sub dob {32}
```

will all be loaded correctly by the **SelfLoader**, and the **SelfLoader** will ensure that the packages 'foo' and 'baz' correctly have the **SelfLoader** `AUTOLOAD` method when the data after `__DATA__` is first parsed.

AUTHOR

`SelfLoader` is maintained by the perl5-porters. Please direct any questions to the canonical mailing list. Anything that is applicable to the CPAN release can be sent to its maintainer, though.

Author and Maintainer: The Perl5-Porters perl5-porters@perl.org

Maintainer of the CPAN release: Steffen Mueller smueller@cpan.org

COPYRIGHT AND LICENSE

This package has been part of the perl core since the first release of perl5. It has been released separately to CPAN so older installations can benefit from bug fixes.

This package has the same copyright and license as the perl core:

Copyright (C) 1993, 1994, 1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006 by Larry Wall and others

All rights reserved.

This program is free software; you can redistribute it and/or modify it under the terms of either:

- a)

  the GNU General Public License as published by the Free Software Foundation; either version 1, or (at your option) any later version, or

- b)

  the “Artistic License” which comes with this Kit.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See either the GNU General Public License or the Artistic License for more details.

You should have received a copy of the Artistic License with this Kit, in the file named “Artistic”. If not, I’ll be glad to provide one.

You should also have received a copy of the GNU General Public License along with this program in the file named “Copying”. If not, see <https://www.gnu.org/licenses/>.

For those of you that choose to use the GNU General Public License, my interpretation of the GNU General Public License is that no Perl script falls under the terms of the GPL unless you explicitly put said script under the terms of the GPL yourself. Furthermore, any object code linked with perl does not automatically fall under the terms of the GPL, provided such object code only adds definitions of subroutines and variables, and does not otherwise impair the resulting interpreter from executing any standard Perl script. I consider linking in C subroutines in this manner to be the moral equivalent of defining subroutines in the Perl language itself. You may sell such an object file as proprietary provided that you provide or offer to provide the Perl source, as specified by the GNU General Public License. (This is merely an alternate way of specifying input to the program.) You may also sell a binary produced by the dumping of a running Perl script that belongs to you, provided that you provide or offer to provide the Perl source as specified by the GPL. (The fact that a Perl interpreter and your code are in the same binary file is, in this case, a form of mere aggregation.) This is my interpretation of the GPL. If you

still have concerns or difficulties understanding my intent, feel free to contact me. Of course, the Artistic License spells all this out for your protection, so you may prefer to use that.

5.2.92 Socket

BSD socket constants and address pack/unpack helpers that feed Perl's built-in `socket`, `bind`, `connect`, `accept`, `send`, `recv`, `setsockopt`, and `shutdown`.

Typical flow: import a constant, pack it into a sockaddr, hand the result to the built-in.

```
use Socket;
socket(my $sock, PF_INET, SOCK_STREAM, IPPROTO_TCP) or die $!;
my $addr = pack_sockaddr_in(80, inet_aton("127.0.0.1"));
connect($sock, $addr) or die $!;
```

What lives here

- **Address families**: `AF_INET`, `AF_INET6`, `AF_UNIX`, `AF_LOCAL`, `AF_UNSPEC`, plus the matching `PF_*` protocol families.
- **Socket types**: `SOCK_STREAM`, `SOCK_DGRAM`, `SOCK_RAW`, `SOCK_SEQPACKET`.
- **Well-known addresses**: `INADDR_ANY`, `INADDR_LOOPBACK`, `INADDR_BROADCAST`, `INADDR_NONE`, `IN6ADDR_ANY`, `IN6ADDR_LOOPBACK` (packed binary, ready for `pack_sockaddr_in` / `pack_sockaddr_in6`).
- **Protocol numbers**: `IPPROTO_IP`, `IPPROTO_TCP`, `IPPROTO_UDP`, `IPPROTO_IPV6`, `IPPROTO_ICMP`, `IPPROTO_RAW`.
- **Socket-option levels and names**: `SOL_SOCKET`, `SO_REUSEADDR`, `SO_KEEPALIVE`, `SO_BROADCAST`, `SO_LINGER`, `SO_SNDBUF`, `SO_RCVBUF`, `SO_ERROR`, `SO_TYPE`, and the rest of the `SO_*` family.
- **TCP options**: `TCP_NODELAY`.
- **IP options**: `IP_TOS`, `IP_TTL`, `IP_MULTICAST_TTL`, `IP_MULTICAST_LOOP`, `IP_ADD_MEMBERSHIP`, `IP_DROP_MEMBERSHIP`, plus the `IPV6_*` counterparts.
- **Message flags**: `MSG_OOB`, `MSG_PEEK`, `MSG_DONTWAIT`, `MSG_WAITALL`, `MSG_NOSIGNAL`.
- **Shutdown modes**: `SHUT_RD`, `SHUT_WR`, `SHUT_RDWR`.
- **Address packing**: `pack_sockaddr_in`, `unpack_sockaddr_in`, `pack_sockaddr_in6`, `unpack_sockaddr_in6`, `pack_sockaddr_un`, `unpack_sockaddr_un`, plus the dual-mode `sockaddr_in` / `sockaddr_in6` shortcuts and the `sockaddr_family` extractor.
- **Address parsing**: `inet_aton`, `inet_ntoa`, `inet_pton`, `inet_ntop`.
- **Name resolution**: `getaddrinfo`, `getnameinfo`, plus the `AI_*`, `NI_*`, and `EAI_*` flag and error constants.

The dual-mode shortcuts

`sockaddr_in`, `sockaddr_in6` and `sockaddr_un` pack in scalar context and unpack in list context:

```
my $sin           = sockaddr_in(80, inet_aton("127.0.0.1"));
my ($port, $addr) = sockaddr_in(getpeername($sock));
```

A wrong argument count for the chosen direction croaks with a usage message. The old six-argument `sockaddr_in($af, $port, @quad)` still packs, and warns "6-ARG sockaddr_in call is deprecated" under `use warnings 'Socket'`.

A constant this system does not have dies when called: a name Socket knows but the platform lacks (`AF_802`, `SO_STATE`) with "Your vendor has not defined Socket macro AF_802, used at FILE line N", any other name with "FOO is not a valid Socket macro at FILE line N".

Differences from upstream

One, where perl 5.44 with Socket.xs fails: `unpack_sockaddr_un` of an abstract address (a `sun_path` starting with a NUL byte) that is longer than `sun_path` returns the 108 bytes `sun_path` holds. Socket.xs trusts the given length and returns bytes read from past the end of its copy of the structure (t/81-xs-native/Socket/190-or-better.t).

Functions

Address packing

pack_ip_mreq

Pack an IPv4 multicast group membership request, the value the `IP_ADD_MEMBERSHIP` and `IP_DROP_MEMBERSHIP` socket options take.

unpack_ip_mreq

Split a packed IPv4 multicast group membership request back into its two addresses.

pack_ip_mreq_source

Pack a source-specific IPv4 multicast group membership request, the value the `IP_ADD_SOURCE_MEMBERSHIP` family of socket options takes.

unpack_ip_mreq_source

Split a packed source-specific IPv4 multicast group membership request back into its three addresses.

pack_ipv6_mreq

Pack an IPv6 multicast group membership request, the value the `IPV6_ADD_MEMBERSHIP` and `IPV6_DROP_MEMBERSHIP` socket options take.

unpack_ipv6_mreq

Split a packed IPv6 multicast group membership request back into its address and interface index.

pack_sockaddr_in

Build a packed `sockaddr_in` from a port number and a 4-byte IPv4 address, ready to hand to `bind` or `connect`.

unpack_sockaddr_in

Split a packed `sockaddr_in` back into its port and 4-byte address.

pack_sockaddr_in6

Build a packed `sockaddr_in6` from a port, a 16-byte IPv6 address, and optional scope and flow-info fields.

unpack_sockaddr_in6

Split a packed `sockaddr_in6` into port, 16-byte address, scope id, and flow-info.

pack_sockaddr_un

Build a packed `sockaddr_un` from a filesystem path, for use with Unix-domain sockets.

unpack_sockaddr_un

Extract the filesystem path from a packed `sockaddr_un`.

sockaddr_family

Read the address-family byte from any packed sockaddr without committing to one unpacking shape.

sockaddr_in

Dual-mode shortcut: in scalar context it behaves like `pack_sockaddr_in`; in list context it behaves like `unpack_sockaddr_in`.

sockaddr_in6

Dual-mode IPv6 shortcut: in scalar context it behaves like `pack_sockaddr_in6`; in list context it behaves like `unpack_sockaddr_in6`.

Address parsing

inet_aton

Turn a dotted-quad string or a hostname into a 4-byte packed IPv4 address, or undef if the name cannot be resolved.

inet_ntoa

Turn a 4-byte packed IPv4 address back into a dotted-quad string.

inet_pton

Parse an address string into its packed binary form for either IPv4 or IPv6, selected by the first argument.

inet_ntop

Turn a packed IPv4 or IPv6 address back into its printable string form, selected by the first argument.

Name resolution

getaddrinfo

Resolve a host name and/or service name into a list of ready-to-use address records, honouring IPv4/IPv6 and protocol preferences carried through a hints hash.

getnameinfo

Turn a packed sockaddr into a host name and service name, with optional flags to force numeric output or demand a successful DNS lookup.

inet_aton

Turn a dotted-quad string or a hostname into a 4-byte packed IPv4 address, or undef if the name cannot be resolved.

Synopsis

```
my $packed = inet_aton("127.0.0.1");
my $packed = inet_aton("localhost");
```

What you get back

A 4-byte string in network byte order suitable as the second argument to pack_sockaddr_in, or undef on failure. To see the bytes as a human-readable address, pass the result through inet_ntoa.

Examples

```
my $loop = inet_aton("127.0.0.1");        # 4-byte string
print join(".", unpack("C4", $loop));     # 127.0.0.1
```

```
my $sin = pack_sockaddr_in(80, inet_aton("example.com"));
# pass $sin to connect()
```

Edge cases

- A literal IPv4 address is parsed directly - no DNS lookup.
- A non-resolvable name returns `undef`.
- IPv6 addresses are rejected here; use `inet_pton(AF_INET6, ...)`.

Differences from upstream

Fully compatible with upstream Socket.

See also

- `inet_ntoa` - reverse direction, packed bytes to dotted-quad string.
- `inet_pton` - same idea but parameterised by address family.
- `pack_sockaddr_in` - the immediate consumer of the packed address.
- `getaddrinfo` - richer resolver when you also need a port or IPv6.

inet_ntoa

Turn a 4-byte packed IPv4 address back into a dotted-quad string.

Synopsis

```
my $str = inet_ntoa($packed_ipv4);
```

What you get back

A string like `"192.0.2.1"`.

Examples

```
print inet_ntoa(INADDR_LOOPBACK);    # 127.0.0.1
```

```
my ($port, $ip) = unpack_sockaddr_in($peer);
printf "%s:%d\n", inet_ntoa($ip), $port;
```

Edge cases

- Input of any length but 4 croaks “Bad arg length for Socket::inet_ntoa, length is N, should be 4”.
- A character above 255 croaks “Wide character in Socket::inet_ntoa”.

Differences from upstream

Fully compatible with upstream Socket.

See also

- `inet_aton` - reverse direction, string to packed bytes.
- `inet_ntop` - same idea for IPv4 or IPv6, parameterised by family.
- `unpack_sockaddr_in` - usually the producer of the packed address.

inet_pton

Parse an address string into its packed binary form for either IPv4 or IPv6, selected by the first argument.

Synopsis

```
my $v4 = inet_pton(AF_INET,  "192.0.2.1");
my $v6 = inet_pton(AF_INET6, "2001:db8::1");
```

What you get back

A 4-byte string for `AF_INET` or a 16-byte string for `AF_INET6`, in network byte order. Returns `undef` when the string is not a valid address for the requested family.

Examples

```
my $addr = inet_pton(AF_INET6, "::1");
my $sin6 = pack_sockaddr_in6(443, $addr);
```

```
defined(inet_pton(AF_INET, $input))
    or die "not a valid IPv4 address: $input\n";
```

Edge cases

- A family other than `AF_INET` or `AF_INET6` croaks “Bad address family for Socket::inet_pton”.
- Unlike `inet_aton`, there is no DNS fallback - hostnames are rejected.
- Scoped IPv6 literals like `"fe80::1%eth0"` are not accepted; use `getaddrinfo` to carry the zone index through.

Differences from upstream

Fully compatible with upstream Socket.

See also

- `inet_ntop` - reverse direction, packed bytes to string.
- `inet_aton` - IPv4-only convenience with DNS fallback.
- `pack_sockaddr_in6` - the usual consumer of an IPv6 packed address.

inet_ntop

Turn a packed IPv4 or IPv6 address back into its printable string form, selected by the first argument.

Synopsis

```
my $s = inet_ntop(AF_INET,  $packed4);
my $s = inet_ntop(AF_INET6, $packed6);
```

What you get back

A dotted-quad string for `AF_INET`, a canonical colon-hex string (with `::` compression) for `AF_INET6`.

Examples

```
print inet_ntop(AF_INET6, IN6ADDR_LOOPBACK);   # ::1
```

```
my ($port, $addr, $scope, $flow) = unpack_sockaddr_in6($peer);
printf "[%s]:%d\n", inet_ntop(AF_INET6, $addr), $port;
```

Edge cases

- Input that is not exactly the family's address length (4 or 16) croaks "Bad address length for Socket::inet_ntop".
- Any family other than `AF_INET` or `AF_INET6` croaks "Bad address family for Socket::inet_ntop".
- A character above 255 croaks "Wide character in Socket::inet_ntop".

Differences from upstream

Fully compatible with upstream Socket.

See also

- `inet_pton` - reverse direction, string to packed bytes.
- `inet_ntoa` - IPv4-only shortcut with no family argument.

pack_sockaddr_in

Build a packed `sockaddr_in` from a port number and a 4-byte IPv4 address, ready to hand to `bind` or `connect`.

Synopsis

```
my $sin = pack_sockaddr_in($port, $packed_ipv4);
```

What you get back

A fixed-size byte string whose layout matches the platform's `struct sockaddr_in`: address family (`AF_INET`), port in network byte order, the raw 4 address bytes, and trailing padding. The result is opaque from Perl's perspective - treat it as a cookie for the socket built-ins.

Examples

```
my $addr = pack_sockaddr_in(80, inet_aton("example.com"));
connect($sock, $addr) or die $!;
```

```
# Bind to every local interface on a chosen port.
bind($sock, pack_sockaddr_in(8080, INADDR_ANY)) or die $!;
```

Edge cases

- Port is truncated to 16 bits; a value above 65535 warns "Port number above 0xFFFF, will be truncated to N". An `undef` port is 0.
- An address of any length but 4 croaks "Bad arg length"; an `undef` address croaks "Undefined address".
- The address argument is a **packed** 4-byte string, not a dotted-quad literal. Pass `inet_aton(...)` or `INADDR_*`.

See also

- `unpack_sockaddr_in` - reverse direction.
- `sockaddr_in` - dual-mode shortcut: packs in scalar context, unpacks in list context (a Perl sub in Socket.pm).
- `pack_sockaddr_in6` - IPv6 counterpart.
- `inet_aton` - usual source of the address argument.

unpack_sockaddr_in

Split a packed `sockaddr_in` back into its port and 4-byte address.

Synopsis

```
my ($port, $packed_ipv4) = unpack_sockaddr_in($sockaddr);
```

What you get back

A two-element list: the port as an integer, the address as a 4-byte packed string. Feed the second through `inet_ntoa` to see it as dotted-quad. In scalar context only the address comes back.

Examples

```
my $peer = getpeername($sock);
my ($port, $ip) = unpack_sockaddr_in($peer);
printf "connected from %s:%d\n", inet_ntoa($ip), $port;
```

Edge cases

- Input that is not exactly a `sockaddr_in` croaks “Bad arg length”; `undef` croaks “Undefined address”.
- A family other than `AF_INET` croaks “Bad address family”.

See also

- `pack_sockaddr_in` - reverse direction.
- `sockaddr_family` - read just the family byte without unpacking.
- `unpack_sockaddr_in6` - IPv6 counterpart.
- `inet_ntoa` - turn the returned packed address into a string.

pack_sockaddr_in6

Build a packed `sockaddr_in6` from a port, a 16-byte IPv6 address, and optional scope and flow-info fields.

Synopsis

```
my $sin6 = pack_sockaddr_in6($port, $packed_ipv6);
my $sin6 = pack_sockaddr_in6($port, $packed_ipv6, $scope_id,
 ↪$flowinfo);
```

What you get back

A fixed-size byte string matching the platform's `struct sockaddr_in6`, ready for `bind` or `connect`.

Examples

```
my $addr = inet_pton(AF_INET6, "::1");
my $sin6 = pack_sockaddr_in6(443, $addr);
connect($sock, $sin6) or die $!;
```

```
# Link-local address with scope index for eth0.
my $scope = scalar getsockopt(...);   # or from if_nametoindex
my $sin6  = pack_sockaddr_in6(5353, inet_pton(AF_INET6, "fe80::1"),
 ↪$scope);
```

Edge cases

- `$scope_id` and `$flowinfo` default to 0 when omitted.
- An address of any length but 16 croaks "Bad arg length"; `undef` croaks "Undefined address".
- Port and flow-info are truncated to 16 and 32 bits respectively; a port above 65535 warns "Port number above 0xFFFF".

See also

- `unpack_sockaddr_in6` - reverse direction.
- `sockaddr_in6` - dual-mode shortcut: packs in scalar context, unpacks in list context (a Perl sub in Socket.pm).
- `pack_sockaddr_in` - IPv4 counterpart.
- `inet_pton` - usual source of the address argument.

unpack_sockaddr_in6

Split a packed `sockaddr_in6` into port, 16-byte address, scope id, and flow-info.

Synopsis

```
my ($port, $addr, $scope_id, $flowinfo) = unpack_sockaddr_in6(
 ↪$sockaddr);
```

What you get back

A four-element list in list context; the address alone in scalar context.

Examples

```
my ($port, $addr, $scope, $flow) = unpack_sockaddr_in6(getpeername(
 ↪$sock));
printf "[%s%%%d]:%d\n", inet_ntop(AF_INET6, $addr), $scope, $port;
```

Edge cases

- Scope id and flow-info are often 0 for non-link-local traffic.
- Input that is not exactly a `sockaddr_in6` croaks “Bad arg length”; `undef` croaks “Undefined address”.
- A family other than `AF_INET6` croaks “Bad address family”.

See also

- `pack_sockaddr_in6` - reverse direction.
- `unpack_sockaddr_in` - IPv4 counterpart.
- `inet_ntop` - turn the returned packed address into a string.

pack_sockaddr_un

Build a packed `sockaddr_un` from a filesystem path, for use with Unix-domain sockets.

Synopsis

```
my $sun = pack_sockaddr_un($path);
```

What you get back

A fixed-size byte string matching the platform’s `struct sockaddr_un`, ready for `bind` or `connect` against an `AF_UNIX` / `AF_LOCAL` socket.

Examples

```
socket(my $sock, AF_UNIX, SOCK_STREAM, 0) or die $!;
connect($sock, pack_sockaddr_un("/tmp/app.sock")) or die $!;
```

```
# Abstract Linux namespace: first byte is NUL.
my $sun = pack_sockaddr_un("\0app.mysvc");
```

Edge cases

- A path longer than `sun_path` (108 bytes) warns “Path length (N) is longer than maximum supported length (108) and will be truncated” and is cut to fit.
- An `undef` path croaks “Undefined path for Socket::pack_sockaddr_un”.

- An abstract name (leading NUL, more than one byte) gives a structure that ends where the name ends, since such a name may itself contain NULs; any other path gives the full structure.

See also

- `unpack_sockaddr_un` - reverse direction.
- `sockaddr_family` - read the family byte from an unknown sockaddr.

unpack_sockaddr_un

Extract the filesystem path from a packed `sockaddr_un`.

Synopsis

```
my ($path) = unpack_sockaddr_un($sockaddr);
```

What you get back

The `sun_path` field as a string, trimmed at the first NUL byte.

Examples

```
my ($path) = unpack_sockaddr_un(getsockname($sock));
print "listening on $path\n";
```

Edge cases

- An abstract Linux-namespace name (leading NUL) comes back whole, NULs included: its length is the structure's length less the family field.
- A structure shorter than `struct sockaddr_un` is accepted, as the kernel returns from `accept`, `recvfrom`, `getpeername` and `getsockname`.
- `undef` croaks "Undefined address for Socket::unpack_sockaddr_un"; a family other than `AF_UNIX` croaks "Bad address family".

See also

- `pack_sockaddr_un` - reverse direction.

sockaddr_family

Read the address-family byte from any packed sockaddr without committing to one unpacking shape.

Synopsis

```
my $family = sockaddr_family($sockaddr);
```

What you get back

An integer equal to one of `AF_INET`, `AF_INET6`, `AF_UNIX`, etc.

Examples

```
my $sa = getpeername($sock);
if (sockaddr_family($sa) == AF_INET6) {
    my ($port, $addr) = unpack_sockaddr_in6($sa);
    ...
} else {
    my ($port, $addr) = unpack_sockaddr_in($sa);
    ...
}
```

Edge cases

- The family byte is read in native byte order - this matches how every `pack_sockaddr_*` writes it.
- Input shorter than 2 bytes croaks “Bad arg length for Socket::sockaddr_family, length is N, should be at least 2”.

See also

- `unpack_sockaddr_in`, `unpack_sockaddr_in6`, `unpack_sockaddr_un`: the right unpacker depends on the family returned here.

getaddrinfo

Resolve a host name and/or service name into a list of ready-to-use address records, honouring IPv4/IPv6 and protocol preferences carried through a hints hash.

Synopsis

```
my ($err, @res) = getaddrinfo($host, $service);
my ($err, @res) = getaddrinfo($host, $service, { family => AF_INET6␣
↪});
```

What you get back

A list whose first element is the error, a dualvar: as a string it is empty on success, otherwise the message from `gai_strerror`; as a number it is 0 or the `EAI_*` code, so `$err == EAI_NONAME` works. Every subsequent element is a hash reference describing one resolved endpoint, with keys:

- `family` - integer address family (`AF_INET`, `AF_INET6`, ...).
- `socktype` - integer socket type (`SOCK_STREAM`, `SOCK_DGRAM`, ...).
- `protocol` - integer protocol number.
- `addr` - packed sockaddr ready for `bind` or `connect`.
- `canonname` - canonical hostname (only populated when the hints hash requests it with `flags => AI_CANONNAME`).

The hints hash accepts `family`, `socktype`, `protocol`, and `flags`, each an integer; unknown keys are ignored.

Examples

```
my ($err, @res) = getaddrinfo("example.com", "https",
    { socktype => SOCK_STREAM });
die "resolve: $err\n" if $err;
for my $r (@res) {
    socket(my $s, $r->{family}, $r->{socktype}, $r->{protocol}) or␣
 ↪next;
    connect($s, $r->{addr}) and last;
    close $s;
}
```

```
# Numeric-only, no DNS lookup, no service lookup.
my ($err, @res) = getaddrinfo("2001:db8::1", "443",
    { flags => AI_NUMERICHOST | AI_NUMERICSERV });
```

```
# Listening side: ask for a wildcard bind address.
my ($err, @res) = getaddrinfo(undef, "8080",
    { flags => AI_PASSIVE, family => AF_INET6 });
```

Edge cases

- On error the result list is exactly one element, the message, so `scalar @res` is 1 and the first hashref is absent.
- Pass `undef` for `$host` to resolve a service with no specific host (typically combined with `AI_PASSIVE`).
- Pass `undef` for `$service` to resolve a host with no port.
- Hints that are defined but not a hash reference croak “hints is not a HASH reference”.

See also

- `getnameinfo` - reverse direction, sockaddr to host and service.
- `inet_pton` - parse a single address string without a full resolver.
- `pack_sockaddr_in`, `pack_sockaddr_in6` - when you already have the address bytes and just need the struct.

getnameinfo

Turn a packed sockaddr into a host name and service name, with optional flags to force numeric output or demand a successful DNS lookup.

Synopsis

```
my ($err, $host, $service) = getnameinfo($sockaddr);
my ($err, $host, $service) = getnameinfo($sockaddr, $flags);
my ($err, $host, $service) = getnameinfo($sockaddr, $flags,
 ↪$xflags);
```

What you get back

`($err, $host, $service)`; on error, `$err` alone. `$err` is the same dualvar `getaddrinfo` returns. `$xflags` takes `NIx_NOHOST` and `NIx_NOSERV`, which skip that half of the lookup and return it as `undef`.

Useful flag bits:

- `NI_NUMERICHOST` - skip DNS, return a printable address.
- `NI_NUMERICSERV` - return the port number as a decimal string.
- `NI_NAMEREQD` - fail instead of falling back to numeric host.
- `NI_NOFQDN` - local hosts keep their short name.
- `NI_DGRAM` - look up the service as UDP rather than TCP.

Examples

```
my ($err, $host, $svc) = getnameinfo(getpeername($sock));
die "reverse lookup: $err\n" if $err;
print "peer: $host:$svc\n";
```

```
# Numeric formatting only - no DNS, no /etc/services.
my ($err, $host, $svc) = getnameinfo($sockaddr,
    NI_NUMERICHOST | NI_NUMERICSERV);
```

Edge cases

- A sockaddr that is not a string croaks “addr is not a string”.
- A malformed sockaddr is reported through `$err` (`EAI_FAMILY`).

See also

- `getaddrinfo` - reverse direction, name to sockaddr.
- `inet_ntop` - format a raw address without a resolver.

- `unpack_sockaddr_in`, `unpack_sockaddr_in6` - when you need the numeric port and address without any lookup.

sockaddr_in

Dual-mode shortcut: in scalar context it behaves like `pack_sockaddr_in`; in list context it behaves like `unpack_sockaddr_in`.

Synopsis

```
my $sin             = sockaddr_in($port, $packed_ipv4);
my ($port, $addr)   = sockaddr_in($sockaddr);
```

What you get back

The same values as the underlying `pack_sockaddr_in` or `unpack_sockaddr_in`, depending on calling context.

Examples

```
# Packing
connect($sock, sockaddr_in(80, inet_aton("127.0.0.1"))) or die $!;
```

```
# Unpacking
my ($port, $ip) = sockaddr_in(getpeername($sock));
```

Edge cases

- The dispatch is on calling context, not argument count. In list context exactly one argument is required, in scalar context exactly two (or the deprecated six-argument form); any other count croaks with a usage message, so `my ($p, $a) = sockaddr_in($port, $ip)` dies.
- Prefer the explicit `pack_sockaddr_in` / `unpack_sockaddr_in` in new code; this name exists for compatibility with older Perl idioms.

Differences from upstream

Fully compatible with upstream Socket.

See also

- `pack_sockaddr_in`, `unpack_sockaddr_in` - the explicit forms.
- `sockaddr_in6` - IPv6 counterpart with the same dual behaviour.

sockaddr_in6

Dual-mode IPv6 shortcut: in scalar context it behaves like `pack_sockaddr_in6`; in list context it behaves like `unpack_sockaddr_in6`.

Synopsis

```
my $sin6                              = sockaddr_in6($port, $addr);
my $sin6                              = sockaddr_in6($port, $addr,
 ↪$scope, $flow);
my ($port, $addr, $scope, $flow)  = sockaddr_in6($sockaddr);
```

What you get back

The same values as the underlying `pack_sockaddr_in6` or `unpack_sockaddr_in6`, depending on calling context.

Examples

```
my $sin6 = sockaddr_in6(443, inet_pton(AF_INET6, "::1"));
```

```
my ($port, $addr) = sockaddr_in6(getpeername($sock));
```

Edge cases

- The dispatch is on calling context - list context requires exactly one argument, scalar context two to four; any other count croaks with a usage message.
- Prefer the explicit pack/unpack forms in new code.

Differences from upstream

Fully compatible with upstream Socket.

See also

- `pack_sockaddr_in6`, `unpack_sockaddr_in6` - the explicit forms.
- `sockaddr_in` - IPv4 counterpart.

5.2.93 Storable

Serialise arbitrary Perl data structures to bytes, reconstruct them later, or deep-clone them in memory.

`Storable` turns a reference - to a scalar, array, hash, blessed object, or any tree built from them - into an opaque byte string that can be written to disk, sent over a socket, or fed back into the module to reproduce the original structure. The round trip preserves shared references, circular references, weak references, tied variables, regexp objects, and blessed classes (with `STORABLE_freeze` / `STORABLE_thaw` hooks honoured).

Two axes of choice cover every use:

- **Where the bytes go.** `freeze` returns them as a string, `store` writes them to a named file, `store_fd` writes them to an open filehandle. `thaw`, `retrieve`, and `fd_retrieve` are the matching readers.
- **Byte order on the wire.** `freeze` / `store` / `store_fd` use the host's native byte order - fastest, but only portable between machines of the same endianness and integer size. The n-prefixed variants `nfreeze`, `nstore`, `nstore_fd` use network (big-endian) order and are portable across architectures. The reader side auto-detects which was used.

`dclone` is the in-memory shortcut for "I want a structurally independent copy of this data." It serialises and immediately deserialises, so every element is freshly allocated and every `STORABLE_freeze` / `STORABLE_thaw` hook runs - which is the only safe way to clone objects that wrap non-SV resources (a PDL's engine pointer, a tied variable's backing state, a file handle).

`file_magic` and `read_magic` inspect a frozen payload without thawing it, returning a hash describing byte order, header size, and version metadata. Use them to validate untrusted input or to pick a reader before committing to a thaw.

This is Storable 3.41; `Storable.pm` is upstream's, its engine native. An image written by perl5's Storable reads here and the other way round.

Synopsis

```
use Storable qw(nstore retrieve nfreeze thaw dclone);

my %config = (name => 'demo', ports => [80, 443], nested => { on =>␣
↪1 });
nstore(\%config, 'config.sto');             # portable byte order
my $back = retrieve('config.sto');
print "$back->{name} @{ $back->{ports} }\n"; # demo 80 443

my $bytes = nfreeze([1, { two => 2 }]);      # a string, e.g. for a␣
↪socket
print thaw($bytes)->[1]{two}, "\n";          # 2

my $deep = dclone(\%config);                 # an independent copy
$deep->{ports}[0] = 8080;                    # %config keeps 80

package Counter;                             # what an object stores
sub STORABLE_freeze { my ($self, $cloning) = @_; return $self->{n} }
sub STORABLE_thaw   { my ($self, $cloning, $n) = @_; $self->{n} = $n␣
↪}
```

The functions, written in Perl

`store` and `retrieve` are exported by default, the rest on request.

- `store($ref, $file)`, `nstore($ref, $file)` - write to a file; `lock_store`, `lock_nstore` take an exclusive `flock` first.

- `retrieve($file)`, `lock_retrieve($file)` - read one back.
- `store_fd($ref, $fh)`, `nstore_fd($ref, $fh)`, `fd_retrieve($fh)` - the same over an open filehandle.
- `freeze($ref)`, `nfreeze($ref)`, `thaw($string)` - to and from a string.
- `file_magic($file)`, `read_magic($string)` - the header of an image as a hash, without reading it.

A writer dies on what it cannot store ("Can't store CODE items"). A reader returns undef for a truncated image and dies for one that is no Storable image at all ("Magic number checking on storable string failed", or a version it does not know). `$Storable::canonical` sorts hash keys, so equal data gives equal bytes; `$Storable::Deparse` and `$Storable::Eval` store and restore code references.

Differences from upstream

Two, each where perl 5.44 with Storable.xs fails (t/81-xs-native/Storable/980-or-better.t):

- An image whose length field asks for more memory than the process can get is a truncated image: the read returns undef. perl5 dies "Out of memory!", ending the process.
- `Storable::pretrieve` on a handle with nothing to read (opened, then closed) fails the image check; perl5 parses whatever an earlier `freeze` left in its buffer.

Functions

Deep cloning

dclone

A deep copy of what a reference points to: every array, hash and scalar in it newly made, shared and circular references kept as they were, objects re-blessed, and `STORABLE_freeze`/`STORABLE_thaw` hooks run - in one step, without a byte string in between. Dies "Not a reference" for anything else, and "Can't store CODE items" for a structure holding a code reference unless `$Storable::Deparse` and `$Storable::Eval` are set, as for `freeze`.

Inside a hook

last_op_in_netorder

True if the last store or retrieve used network byte order (`nstore`, `nfreeze`, or reading such an image), false after a native one. Not exportable: call it as `Storable::last_op_in_netorder()`.

is_storing

True while a store is running - inside a `STORABLE_freeze` hook, say - and false otherwise. Call it as `Storable::is_storing()`.

is_retrieving

True while a retrieve is running - inside a `STORABLE_thaw` hook, say - and false otherwise. Call it as `Storable::is_retrieving()`.

Recursion limits

stack_depth

How deeply nested arrays a store or retrieve may follow before it dies "Max. recursion depth with nested structures exceeded": `$Storable::recursion_limit`, 512 unless set (-1 for no limit).

stack_depth_hash

The same for nested hashes: `$Storable::recursion_limit_hash`, 256 unless set (-1 for no limit).

5.2.94 String

Namespace containing the following modules:

- `String::CRC32` - Compute the 32-bit CRC of a string or of what a filehandle reads, the CRC-32 of zip, gzip and PNG.

String::CRC32

Compute the 32-bit CRC of a string or of what a filehandle reads, the CRC-32 of zip, gzip and PNG.

`crc32` is exported by default. Its last argument, when it has more than one, is the CRC to continue from, so a large input can be taken in pieces. A filehandle is read from where it stands to its end.

The CRC is computed by the zlib engine of the built-in Compress::Raw::Zlib, so the two modules always agree.

This is a native implementation of String::CRC32 2.100's `CRC32.xs`; its `String/CRC32.pm` is used unchanged.

Synopsis

```
use String::CRC32;

printf "%08x\n", crc32("123456789");          # cbf43926
my $crc = crc32("12345");
printf "%08x\n", crc32("6789", $crc);         # cbf43926, continued

open my $fh, '<:raw', $path or die $!;
my $file_crc = crc32($fh);                    # the rest of the file
```

Differences from upstream

One, where perl 5.44 with CRC32.xs fails (t/81-xs-native/String-CRC32/150-or-better.t): a filehandle whose read fails, a closed one for instance, gives the CRC of no data. perl5 keeps the failed read's -1 as 65535 bytes read, from a buffer of 32768, and dumps core.

Functions

Checksums

`crc32`

`crc32($data)`, `crc32($data, $crc)`: the CRC-32 of a string (its bytes; a character string's UTF-8), or of what a filehandle - a glob, a reference to one, an IO object - reads from where it stands to its end. With more than one argument, the last is the CRC to continue from (0 when there is none), read as a number. The CRC is an unsigned integer.

5.2.95 Sub

Namespace containing the following modules:

- `Sub::Exporter`
- `Sub::Identify` - Ask a code reference for its name, its package and where it was defined.
- `Sub::Name` - Give a sub a name: what `caller`, stack traces and profilers report for it.
- `Sub::Util` - Names and prototypes of code references: read the name a sub was defined under, give an anonymous sub a name that stack traces, `caller` and profilers show, and set a sub's prototype at run time.

Sub::Exporter

Namespace containing the following modules:

- `Sub::Exporter::Progressive` - Only use Sub::Exporter if you need it

Sub::Exporter::Progressive

Only use Sub::Exporter if you need it

VERSION

version 0.001013

SYNOPSIS

```
package Syntax::Keyword::Gather;

use Sub::Exporter::Progressive -setup => {
```

(continues on next page)

(continued from previous page)

```
  exports => [qw( break gather gathered take )],
  groups => {
    default => [qw( break gather gathered take )],
  },
};

# elsewhere

# uses Exporter for speed
use Syntax::Keyword::Gather;

# somewhere else

# uses Sub::Exporter for features
use Syntax::Keyword::Gather 'gather', take => { -as => 'grab' };
```

DESCRIPTION

Sub::Exporter is an incredibly powerful module, but with that power comes great responsibility, er- as well as some runtime penalties. This module is a `Sub::Exporter` wrapper that will let your users just use *Exporter* if all they are doing is picking exports, but use `Sub::Exporter` if your users try to use `Sub::Exporter`'s more advanced features, like renaming exports, if they try to use them.

Note that this module will export `@EXPORT`, `@EXPORT_OK` and `%EXPORT_TAGS` package variables for `Exporter` to work. Additionally, if your package uses advanced `Sub::Exporter` features like currying, this module will only ever use `Sub::Exporter`, so you might as well use it directly.

CONTRIBUTORS

ilmari - Dagfinn Ilmari Mannsåker (cpan:ILMARI) ilmari@ilmari.org

mst - Matt S. Trout (cpan:MSTROUT) mst@shadowcat.co.uk

leont - Leon Timmermans (cpan:LEONT) leont@cpan.org

AUTHOR

Arthur Axel "fREW" Schmidt Sub-Exporter-Progressive@afoolishmanifesto.com

COPYRIGHT AND LICENSE

This software is copyright (c) 2016 by Arthur Axel "fREW" Schmidt.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

Sub::Identify

Ask a code reference for its name, its package and where it was defined.

Synopsis

```
use Sub::Identify qw(sub_name stash_name sub_fullname get_code_info
                     get_code_location is_sub_constant);

sub Foo::bar { 1 }
my $code = \&Foo::bar;
print sub_name($code), "\n";        # bar
print stash_name($code), "\n";      # Foo
print sub_fullname($code), "\n";    # Foo::bar
my ($file, $line) = get_code_location($code);
print is_sub_constant(sub () { 42 }) ? "constant\n" : "not\n";
```

Functions

`get_code_info($code)` returns the package and the name of the glob the sub was defined under - `('main', '__ANON__')` for an anonymous sub - or nothing for a sub still being compiled. `sub_name`, `stash_name` and `sub_fullname` are its parts. `get_code_location` returns the file and line of the sub's first statement, or nothing for an XSUB or a sub with no body. `is_sub_constant` is true for a sub perl inlines as a constant (`sub () { 42 }`, `use constant`). A non-reference gives nothing (0 for `is_sub_constant`). Nothing is exported by default; `:all` exports everything.

This is a native implementation of Sub::Identify 0.14's `Identify.xs`; its `Sub/Identify.pm` is used unchanged.

Differences from upstream

None.

Functions

Code introspection

get_code_info

The package and the name a sub was defined under.

get_code_location

The file and line where a sub's body begins.

is_sub_constant

Whether a sub is a constant perl inlines.

get_code_info

The package and the name a sub was defined under.

Synopsis

```
use Sub::Identify qw(get_code_info);
my ($package, $name) = get_code_info(\&Foo::bar);   # ('Foo', 'bar')
```

What you get back

The name of the sub's package and its own name, from the glob it was defined in: an anonymous sub gives `__ANON__`, a sub imported under another name keeps the name it was defined with. Nothing for a non-reference, or for a sub that is still being compiled.

get_code_location

The file and line where a sub's body begins.

Synopsis

```
use Sub::Identify qw(get_code_location);
my ($file, $line) = get_code_location(\&Foo::bar);
```

What you get back

The file the sub was compiled from and the line of its first statement. Nothing for a non-reference, an XSUB, or a sub declared but never defined.

is_sub_constant

Whether a sub is a constant perl inlines.

Synopsis

```
use Sub::Identify qw(is_sub_constant);
use constant PI => 3.14159;
is_sub_constant(\&PI);              # true
is_sub_constant(sub () { $x });     # false: it closes over $x
```

What you get back

True for a sub with an empty prototype that perl has made a constant - `use constant`, or `sub () { LITERAL }` - and false for everything else, including a non-reference.

Sub::Name

Give a sub a name: what `caller`, stack traces and profilers report for it.

Synopsis

```
use Sub::Name;

my $code = subname 'My::Pkg::handler' => sub { die "oops" };
$code->();           # oops at ... line N.  (in My::Pkg::handler)

subname 'helper', $code;   # an unqualified name goes to the caller
↪'s package
```

What you get back

`subname NAME, CODEREF` renames the sub in place and returns the code reference. A NAME with `::` (or the old `'` separator) puts the sub in that package, creating it; without one it goes to the package of the calling code. Only the sub's own name changes: no symbol table entry is created, so the sub is not callable under its new name. CODEREF can also be a glob or, without `strict refs`, the name of a sub. `subname` is exported by default.

This is a native implementation of Sub::Name 0.28's `Name.xs`; its `Sub/Name.pm` is used unchanged.

Differences from upstream

Renaming a sub under the debugger no longer leaks a reference to its `%DB::sub` entry, one per call: `Sub::Util::set_subname`, which perl ships and which was taken from this module, fixed that, and pperl runs that code for both modules (t/81-xs-native/Sub-Name/150-or-better.t).

Functions

Names

subname

Rename a sub in place.

subname

Rename a sub in place.

Synopsis

```
use Sub::Name;
my $named = subname 'Pkg::name', sub { ... };
```

What you get back

The code reference passed in, the sub now named NAME. The name is split at its last `::` or `'` into package and name; the package is created if it does not exist, and an unqualified name goes to the package of the calling code. Dies with `Not a subroutine reference`, `Undefined subroutine NAME`, or, under `strict refs`, `Can't use string ... as a subroutine ref` for a string that is not a code reference.

Sub::Util

Names and prototypes of code references: read the name a sub was defined under, give an anonymous sub a name that stack traces, `caller` and profilers show, and set a sub's prototype at run time.

This is Sub::Util from Scalar-List-Utils 1.70; `Sub/Util.pm` is upstream's, and its three functions are the XS ones of ListUtil.xs.

Synopsis

```
use Sub::Util qw(subname set_subname set_prototype);

my $anon = sub { (caller(0))[3] };
print subname($anon), "\n";                 # main::__ANON__

my $named = set_subname('My::helper', $anon);
print subname($named), "\n";                # My::helper
print $named->(), "\n";                     # caller() sees it:
 ↪My::helper

set_prototype('$$', $named);
print prototype($named), "\n";              # $$
```

Differences from upstream

None: upstream's own tests of the three functions pass unchanged (t/81-xs-native/List-Util/490-upstream-subname.t, 380-upstream-prototype.t).

Functions

Names

subname

Return the fully qualified name of a subroutine reference.

set_subname

Assign a name to a subroutine reference in place.

Prototypes

set_prototype

Set or clear the prototype on a subroutine reference.

subname

Return the fully qualified name of a subroutine reference.

Synopsis

```
use Sub::Util qw(subname);
my $name = subname \&some_sub;         # 'Pkg::some_sub'
my $name = subname sub { 1 };          # 'main::__ANON__'
```

What you get back

A string of the form `"Package::name"`. Anonymous subroutines produce `"Package::__ANON__"`. If the subroutine has been renamed with `set_subname`, the new name is returned. Passing something that is not a code reference croaks with `Not a subroutine reference`.

Examples

```
package My::Pkg;
sub hello { 1 }
print Sub::Util::subname(\&hello);    # 'My::Pkg::hello'
```

Differences from upstream

Fully compatible with upstream.

See also

- `set_subname` - rename a code reference (useful for stack traces).

set_subname

Assign a name to a subroutine reference in place.

Synopsis

```
use Sub::Util qw(set_subname);
my $code = set_subname 'My::Pkg::name', sub { ... };
```

What you get back

The same code reference, now reporting `$name` under `subname`, `caller`, and stack traces. If `$name` contains `::` or a quote-style separator (`'`), the left part becomes the package and the right part the short name. An unqualified name is attached to the current compilation package.

Examples

```
my $sub = set_subname 'My::Thing::handler', sub { process(@_) };
print Sub::Util::subname($sub);         # 'My::Thing::handler'
```

```
# Naming closures so stack traces are readable:
for my $op (qw( add sub mul )) {
    $dispatch{$op} = set_subname "op_$op", sub { ... };
}
```

Edge cases

- Croaks with `Not a subroutine reference` for non-CODE arguments.
- Croaks `Usage: Sub::Util::set_subname(name, sub)` unless given two args.

Differences from upstream

Fully compatible with upstream.

See also

- `subname` - read back the fully qualified name.
- `set_prototype` - adjust the prototype of a code reference.

set_prototype

Set or clear the prototype on a subroutine reference.

Synopsis

```
use Sub::Util qw(set_prototype);
my $code = set_prototype '$$', \&adder;
my $code = set_prototype undef, \&adder;   # clear the prototype
```

What you get back

The same code reference, with its prototype replaced by `$proto` (a string like `'$$'`, `'&@'`, `'\@'`) or cleared when `$proto` is `undef`.

Examples

```
set_prototype '$$', \&my_add;     # force two scalar args
set_prototype '&@',  \&wrapper;    # block followed by a list
set_prototype undef, \&free_form;  # no prototype
```

Edge cases

- Croaks with `set_prototype: not a reference` for scalars.
- Croaks with `set_prototype: not a subroutine reference` for non-CODE refs.
- Croaks `Usage: Sub::Util::set_prototype(proto, code)` unless given two args.

Differences from upstream

Fully compatible with upstream.

See also

- `set_subname` - rename a code reference.
- `prototype` - perl built-in that reads a prototype.

5.2.96 Symbol

manipulate Perl symbols and their names

SYNOPSIS

```
use Symbol;

$sym = gensym;
open($sym, '<', "filename");
$_ = <$sym>;
# etc.

ungensym $sym;      # no effect

# replace *FOO{IO} handle but not $FOO, %FOO, etc.
*FOO = geniosym;

print qualify("x"), "\n";             # "main::x"
print qualify("x", "FOO"), "\n";      # "FOO::x"
print qualify("BAR::x"), "\n";        # "BAR::x"
print qualify("BAR::x", "FOO"), "\n"; # "BAR::x"
print qualify("STDOUT", "FOO"), "\n"; # "main::STDOUT" (global)
print qualify(\*x), "\n";             # returns \*x
print qualify(\*x, "FOO"), "\n";      # returns \*x

use strict refs;
print { qualify_to_ref $fh } "foo!\n";
$ref = qualify_to_ref $name, $pkg;

use Symbol qw(delete_package);
delete_package('Foo::Bar');
print "deleted\n" unless exists $Foo::{'Bar::'};
```

DESCRIPTION

`Symbol::gensym` creates an anonymous glob and returns a reference to it. Such a glob reference can be used as a file or directory handle.

For backward compatibility with older implementations that didn't support anonymous globs, `Symbol::ungensym` is also provided. But it doesn't do anything.

`Symbol::geniosym` creates an anonymous IO handle. This can be assigned into an existing glob without affecting the non-IO portions of the glob.

`Symbol::qualify` turns unqualified symbol names into qualified variable names (e.g. "myvar" -> "MyPackage::myvar"). If it is given a second parameter, `qualify` uses it as the default package; otherwise, it uses the package of its caller. Regardless, global variable names (e.g. "STDOUT", "ENV", "SIG") are always qualified with "main::".

Qualification applies only to symbol names (strings). References are left unchanged under the assumption that they are glob references, which are qualified by their nature.

`Symbol::qualify_to_ref` is just like `Symbol::qualify` except that it returns a glob ref rather than a symbol name, so you can use the result even if `use strict 'refs'` is in effect.

`Symbol::delete_package` wipes out a whole package namespace. Note this routine is not exported by default–you may want to import it explicitly.

BUGS

`Symbol::delete_package` is a bit too powerful. It undefines every symbol that lives in the specified package. Since perl, for performance reasons, does not perform a symbol table lookup each time a function is called or a global variable is accessed, some code that has already been loaded and that makes use of symbols in package Foo may stop working after you delete Foo, even if you reload the Foo module afterwards.

5.2.97 Sys

Namespace containing the following modules:

- `Sys::Hostname` - The name of the machine the program runs on.
- `Sys::Syslog` - Messages to the system log: syslog(3) from Perl.

Sys::Hostname

The name of the machine the program runs on.

`hostname()`, exported by default, returns the system's host name (what gethostname(2) reports, the same as `uname -n`), and dies "Cannot get host name of local machine" if there is none. The first answer is kept in `$Sys::Hostname::host` and returned from then on; `ghname()` asks again every time.

This is Sys::Hostname 1.25; `Sys/Hostname.pm` is upstream's, and its XS half is native.

Synopsis

```
use Sys::Hostname;

my $host = hostname();                  # e.g. "build-01"
my $now  = Sys::Hostname::ghname();     # asked again, not cached
```

Differences from upstream

None: upstream's test passes unchanged (t/81-xs-native/Sys-Hostname/100-upstream-hostname.t), and the rest there checks against perl 5.44.

Functions

Other Functions

ghname

`ghname()` - asks the operating system for the hostname, uncached.

Sys::Syslog

Messages to the system log: syslog(3) from Perl.

This is Sys::Syslog 0.36; `Sys/Syslog.pm` is upstream's and is the interface. It sends a message by the first mechanism that works - the C library's syslog(3) ("native", these XSUBs), a UNIX socket, TCP, UDP, the console - and formats it, `printf`-style.

Synopsis

```
use Sys::Syslog qw(:standard :macros);

openlog('myprog', 'ndelay,pid', LOG_USER);   # ident, options, facility
syslog(LOG_INFO, 'started with %d workers', 4);
syslog('warning', 'disk %s at %d%%', '/var', 91);   # levels by name too
setlogmask(LOG_UPTO(LOG_NOTICE));             # drop info and debug
closelog();
```

The functions, written in Perl

`:standard` exports the first four:

- `openlog($ident, $options, $facility)` - name the program, choose options (`cons`, `ndelay`, `nofatal`, `perror`, `pid`), and the default facility (`LOG_USER`, `LOG_DAEMON`, `LOG_LOCAL0`..`LOG_LOCAL7`, ...). Optional: `syslog` opens with defaults.
- `syslog($priority, $format, @args)` - one message; `$priority` is a level (`LOG_ERR`, `"err"`, ...), optionally with a facility (`"local0|info"`, `LOG_LOCAL0 | LOG_INFO`).
- `setlogmask($mask)` - which levels get through; returns the old mask.
- `closelog()`.
- `setlogsock($type)` (`:extended`) - choose the mechanism: `native`, `unix`, `tcp`, `udp`, `stream`, `pipe`, `console`.

`:macros` exports the levels (`LOG_EMERG` .. `LOG_DEBUG`), the facilities, the option flags (`LOG_PID`, `LOG_NDELAY`, ...), `LOG_MASK` and `LOG_UPTO`.

Differences from upstream

None: upstream's tests pass unchanged (t/81-xs-native/Sys-Syslog, the `upstream` files), and 010-constants.t checks every constant against perl 5.44.

Functions

Masks

LOG_MASK

The `setlogmask` bit of one level: `setlogmask(LOG_MASK(LOG_ERR))` lets only errors through. Exported with `:macros`.

LOG_UPTO

The `setlogmask` bits of a level and every more urgent one: `setlogmask(LOG_UPTO(LOG_WARNING))` drops notices, informational and debug messages. Exported with `:macros`.

Priority values

LOG_FAC

The facility number of a combined priority value: `LOG_FAC(134)` is 16 for `LOG_LOCAL0` (whose value is 16 << 3). Not exported; call it as `Sys::Syslog::LOG_FAC`.

LOG_PRI

The level part of a combined priority value: `LOG_PRI(134)` is 6, `LOG_INFO`. Not exported; call it as `Sys::Syslog::LOG_PRI`.

LOG_MAKEPRI

A combined priority value from a facility and a level: `LOG_MAKEPRI(LOG_LOCAL0, LOG_INFO)` is 134. Not exported; call it as `Sys::Syslog::LOG_MAKEPRI`.

5.2.98 TAP

Namespace containing the following modules:

- `TAP::Base` - Base class that provides common functionality to *TAP::Parser* and *TAP::Harness*
- `TAP::Formatter`
- `TAP::Harness` - Run test scripts with statistics
- `TAP::Object` - Base class that provides common functionality to all `TAP::*` modules
- `TAP::Parser` - Parse TAP output

TAP::Base

Base class that provides common functionality to *TAP::Parser* and *TAP::Harness*

VERSION

Version 3.52

SYNOPSIS

```
package TAP::Whatever;

use base 'TAP::Base';
```

(continues on next page)

(continued from previous page)

```
# ... later ...

my $thing = TAP::Whatever->new();

$thing->callback( event => sub {
    # do something interesting
} );
```

DESCRIPTION

`TAP::Base` provides callback management.

METHODS

Class Methods

`callback`

Install a callback for a named event.

`get_time`

Return the current time using Time::HiRes if available.

`time_is_hires`

Return true if the time returned by get_time is high resolution (i.e. if Time::HiRes is available).

`get_times`

Return array reference of the four-element list of CPU seconds, as with *"times" in perlfunc*.

TAP::Formatter

Namespace containing the following modules:

- `TAP::Formatter::Base` - Base class for harness output delegates
- `TAP::Formatter::Color` - Run Perl test scripts with color
- `TAP::Formatter::Console` - Harness output delegate for default console output
- `TAP::Formatter::File` - Harness output delegate for file output
- `TAP::Formatter::Session` - Abstract base class for harness output delegate

TAP::Formatter::Base

Base class for harness output delegates

VERSION

Version 3.52

DESCRIPTION

This provides console orientated output formatting for TAP::Harness.

SYNOPSIS

```
use TAP::Formatter::Console;
my $harness = TAP::Formatter::Console->new( \%args );
```

METHODS

Class Methods

new

```
my %args = (
   verbose => 1,
)
my $harness = TAP::Formatter::Console->new( \%args );
```

The constructor returns a new `TAP::Formatter::Console` object. If a *TAP::Harness* is created with no `formatter` a `TAP::Formatter::Console` is automatically created. If any of the following options were given to TAP::Harness->new they well be passed to this constructor which accepts an optional hashref whose allowed keys are:

- `verbosity`

  Set the verbosity level.

- `verbose`

  Printing individual test results to STDOUT.

- `timer`

  Append run time for each test to output. Uses *Time::HiRes* if available.

- `failures`

  Show test failures (this is a no-op if `verbose` is selected).

- `comments`

  Show test comments (this is a no-op if `verbose` is selected).

- `quiet`

  Suppressing some test output (mostly failures while tests are running).

- `really_quiet`

  Suppressing everything but the tests summary.

- `silent`

  Suppressing all output.

- `errors`

  If parse errors are found in the TAP output, a note of this will be made in the summary report. To see all of the parse errors, set this argument to true:

  ```
  errors => 1
  ```

- `directives`

  If set to a true value, only test results with directives will be displayed. This overrides other settings such as `verbose`, `failures`, or `comments`.

- `stdout`

  A filehandle for catching standard output.

- `color`

  If defined specifies whether color output is desired. If `color` is not defined it will default to color output if color support is available on the current platform and output is not being redirected.

- `jobs`

  The number of concurrent jobs this formatter will handle.

- `show_count`

  Boolean value. If false, disables the `X/Y` test count which shows up while tests are running.

Any keys for which the value is `undef` will be ignored.

prepare

Called by Test::Harness before any test output is generated.

This is an advisory and may not be called in the case where tests are being supplied to Test::Harness by an iterator.

open_test

Called to create a new test session. A test session looks like this:

```
my $session = $formatter->open_test( $test, $parser );
while ( defined( my $result = $parser->next ) ) {
    $session->result($result);
    exit 1 if $result->is_bailout;
}
$session->close_test;
```

summary

```
$harness->summary( $aggregate );
```

`summary` prints the summary report after all tests are run. The first argument is an aggregate to summarise. An optional second argument may be set to a true value to indicate that the summary is being output as a result of an interrupted test run.

TAP::Formatter::Color

Run Perl test scripts with color

VERSION

Version 3.52

DESCRIPTION

Note that this harness is *experimental*. You may not like the colors I've chosen and I haven't yet provided an easy way to override them.

This test harness is the same as *TAP::Harness*, but test results are output in color. Passing tests are printed in green. Failing tests are in red. Skipped tests are blue on a white background and TODO tests are printed in white.

If *Term::ANSIColor* cannot be found (and Win32::Console::ANSI if running under Windows) tests will be run without color.

SYNOPSIS

```
use TAP::Formatter::Color;
my $harness = TAP::Formatter::Color->new( \%args );
$harness->runtests(@tests);
```

METHODS

Class Methods

new

The constructor returns a new `TAP::Formatter::Color` object. If *Term::ANSIColor* is not installed, returns undef.

can_color

```
Test::Formatter::Color->can_color()
```

Returns a boolean indicating whether or not this module can actually generate colored output. This will be false if it could not load the modules needed for the current platform.

set_color

Set the output color.

TAP::Formatter::Console

Harness output delegate for default console output

VERSION

Version 3.52

DESCRIPTION

This provides console orientated output formatting for TAP::Harness.

SYNOPSIS

```
use TAP::Formatter::Console;
my $harness = TAP::Formatter::Console->new( \%args );
```

open_test

See *TAP::Formatter::Base*

Modules

- `TAP::Formatter::Console::ParallelSession` - Harness output delegate for parallel console output
- `TAP::Formatter::Console::Session` - Harness output delegate for default console output

TAP::Formatter::Console::ParallelSession

Harness output delegate for parallel console output

VERSION

Version 3.52

DESCRIPTION

This provides console orientated output formatting for *TAP::Harness* when run with multiple *"jobs" in TAP::Harness*.

SYNOPSIS

METHODS

Class Methods

header

Output test preamble

result

```
Called by the harness for each line of TAP it receives .
```

clear_for_close

close_test

TAP::Formatter::Console::Session

Harness output delegate for default console output

VERSION

Version 3.52

DESCRIPTION

This provides console orientated output formatting for TAP::Harness.

clear_for_close

close_test

header

result

TAP::Formatter::File

Harness output delegate for file output

VERSION

Version 3.52

DESCRIPTION

This provides file orientated output formatting for TAP::Harness.

SYNOPSIS

```
use TAP::Formatter::File;
my $harness = TAP::Formatter::File->new( \%args );
```

open_test

See *TAP::Formatter::Base*

Modules

- `TAP::Formatter::File::Session` - Harness output delegate for file output

TAP::Formatter::File::Session

Harness output delegate for file output

VERSION

Version 3.52

DESCRIPTION

This provides file orientated output formatting for *TAP::Harness*. It is particularly important when running with parallel tests, as it ensures that test results are not interleaved, even when run verbosely.

METHODS

result

Stores results for later output, all together.

close_test

When the test file finishes, outputs the summary, together.

TAP::Formatter::Session

Abstract base class for harness output delegate

VERSION

Version 3.52

METHODS

Class Methods

new

```
my %args = (
   formatter => $self,
)
my $harness = TAP::Formatter::Console::Session->new( \%args );
```

The constructor returns a new `TAP::Formatter::Console::Session` object.

- `formatter`
- `parser`
- `name`
- `show_count`

header

Output test preamble

result

Called by the harness for each line of TAP it receives.

close_test

Called to close a test session.

clear_for_close

Called by `close_test` to clear the line showing test progress, or the parallel test ruler, prior to printing the final test result.

time_report

Return a formatted string about the elapsed (wall-clock) time and about the consumed CPU time.

TAP::Harness

Run test scripts with statistics

VERSION

Version 3.52

DESCRIPTION

This is a simple test harness which allows tests to be run and results automatically aggregated and output to STDOUT.

SYNOPSIS

```
use TAP::Harness;
my $harness = TAP::Harness->new( \%args );
$harness->runtests(@tests);
```

METHODS

Class Methods

new

```
my %args = (
   verbosity => 1,
   lib     => [ 'lib', 'blib/lib', 'blib/arch' ],
)
my $harness = TAP::Harness->new( \%args );
```

The constructor returns a new `TAP::Harness` object. It accepts an optional hashref whose allowed keys are:

- `verbosity`

  Set the verbosity level:

  ```
   1   verbose        Print individual test results to STDOUT.
   0   normal
  -1   quiet          Suppress some test output (mostly failures
                      while tests are running).
  -2   really quiet   Suppress everything but the tests summary.
  -3   silent         Suppress everything.
  ```

- `timer`

  Append run time for each test to output. Uses *Time::HiRes* if available.

- `failures`

  Show test failures (this is a no-op if `verbose` is selected).

- `comments`

  Show test comments (this is a no-op if `verbose` is selected).

- `show_count`

  Update the running test count during testing.

- `normalize`

  Set to a true value to normalize the TAP that is emitted in verbose modes.

- `lib`

  Accepts a scalar value or array ref of scalar values indicating which paths to allowed libraries should be included if Perl tests are executed. Naturally, this only makes sense in the context of tests written in Perl.

- `switches`

  Accepts a scalar value or array ref of scalar values indicating which switches should be included if Perl tests are executed. Naturally, this only makes sense in the context of tests written in Perl.

- `test_args`

  A reference to an `@INC` style array of arguments to be passed to each test program.

```
test_args => ['foo', 'bar'],
```

  if you want to pass different arguments to each test then you should pass a hash of arrays, keyed by the alias for each test:

```
test_args => {
  my_test    => ['foo', 'bar'],
  other_test => ['baz'],
}
```

- `color`

  Attempt to produce color output.

- `exec`

  Typically, Perl tests are run through this. However, anything which spits out TAP is fine. You can use this argument to specify the name of the program (and optional switches) to run your tests with:

```
exec => ['/usr/bin/ruby', '-w']
```

  You can also pass a subroutine reference in order to determine and return the proper program to run based on a given test script. The subroutine reference should expect the TAP::Harness object itself as the first argument, and the file name as the second argument. It should return an array reference containing the command to be run and including the test file name. It can also simply return `undef`, in which case TAP::Harness will fall back on executing the test script in Perl:

```
exec => sub {
    my ( $harness, $test_file ) = @_;

    # Let Perl tests run.
    return undef if $test_file =~ /[.]t$/;
    return [ qw( /usr/bin/ruby -w ), $test_file ]
      if $test_file =~ /[.]rb$/;
  }
```

  If the subroutine returns a scalar with a newline or a filehandle, it will be interpreted as raw TAP or as a TAP stream, respectively.

- `merge`

  If `merge` is true the harness will create parsers that merge STDOUT and STDERR together for any processes they start.

- `sources`

  NEW to 3.18.

  If set, `sources` must be a hashref containing the names of the *TAP::Parser::SourceHandler*s to load and/or configure. The values are a hash of configuration that will be accessible to the source handlers via *"config_for" in TAP::Parser::Source*.

  For example:

```
sources => {
  Perl => { exec => '/path/to/custom/perl' },
  File => { extensions => [ '.tap', '.txt' ] },
  MyCustom => { some => 'config' },
}
```

  The `sources` parameter affects how `source`, `tap` and `exec` parameters are handled.

  For more details, see the `sources` parameter in *"new" in TAP::Parser*, *TAP::Parser::Source*, and *TAP::Parser::IteratorFactory*.

- `aggregator_class`

  The name of the class to use to aggregate test results. The default is *TAP::Parser::Aggregator*.

- `version`

  NEW to 3.22.

  Assume this TAP version for *TAP::Parser* instead of default TAP version 12.

- `formatter_class`

  The name of the class to use to format output. The default is *TAP::Formatter::Console*, or *TAP::Formatter::File* if the output isn't a TTY.

- `multiplexer_class`

  The name of the class to use to multiplex tests during parallel testing. The default is *TAP::Parser::Multiplexer*.

- `parser_class`

  The name of the class to use to parse TAP. The default is *TAP::Parser*.

- `scheduler_class`

  The name of the class to use to schedule test execution. The default is *TAP::Parser::Scheduler*.

- `formatter`

  If set `formatter` must be an object that is capable of formatting the TAP output. See *TAP::Formatter::Console* for an example.

- `errors`

  If parse errors are found in the TAP output, a note of this will be made in the summary report. To see all of the parse errors, set this argument to true:

```
errors => 1
```

- `directives`

  If set to a true value, only test results with directives will be displayed. This overrides other settings such as `verbose` or `failures`.

- `ignore_exit`

  If set to a true value instruct `TAP::Parser` to ignore exit and wait status from test scripts.

- `jobs`

  The maximum number of parallel tests to run at any time. Which tests can be run in parallel is controlled by `rules`. The default is to run only one test at a time.

- `rules`

  A reference to a hash of rules that control which tests may be executed in parallel. If no rules are declared and *CPAN::Meta::YAML* is available, `TAP::Harness` attempts to load rules from a YAML file specified by the `rulesfile` parameter. If no rules file exists, the default is for all tests to be eligible to be run in parallel.

  Here some simple examples. For the full details of the data structure and the related glob-style pattern matching, see *"Rules data structure" in TAP::Parser::Scheduler*.

```
# Run all tests in sequence, except those starting with "p"
$harness->rules({
    par => 't/p*.t'
});

# Equivalent YAML file
---
par: t/p*.t

# Run all tests in parallel, except those starting with "p"
$harness->rules({
    seq => [
              { seq => 't/p*.t' },
              { par => '**'     },
          ],
});

# Equivalent YAML file
---
seq:
    - seq: t/p*.t
    - par: **
```

(continues on next page)

(continued from previous page)

```
# Run some  startup tests in sequence, then some parallel tests␣
↪than some
# teardown tests in sequence.
$harness->rules({
    seq => [
        { seq => 't/startup/*.t' },
        { par => ['t/a/*.t','t/b/*.t','t/c/*.t'], }
        { seq => 't/shutdown/*.t' },
    ],

});

# Equivalent YAML file
---
seq:
    - seq: t/startup/*.t
    - par:
        - t/a/*.t
        - t/b/*.t
        - t/c/*.t
    - seq: t/shutdown/*.t
```

This is an experimental feature and the interface may change.

- `rulesfiles`

  This specifies where to find a YAML file of test scheduling rules. If not provided, it looks for a default file to use. It first checks for a file given in the `HARNESS_RULESFILE` environment variable, then it checks for `testrules.yml` and then `t/testrules.yml`.

- `stdout`

  A filehandle for catching standard output.

- `trap`

  Attempt to print summary information if run is interrupted by SIGINT (Ctrl-C).

Any keys for which the value is `undef` will be ignored.

Instance Methods

`runtests`

```
$harness->runtests(@tests);
```

Accepts an array of `@tests` to be run. This should generally be the names of test files, but this is not required. Each element in `@tests` will be passed to `TAP::Parser::new()` as a `source`. See *TAP::Parser* for more information.

It is possible to provide aliases that will be displayed in place of the test name by supplying the test as a reference to an array containing `[ $test, $alias ]`:

```
$harness->runtests( [ 't/foo.t', 'Foo Once' ],
                    [ 't/foo.t', 'Foo Twice' ] );
```

Normally it is an error to attempt to run the same test twice. Aliases allow you to overcome this limitation by giving each run of the test a unique name.

Tests will be run in the order found.

If the environment variable `PERL_TEST_HARNESS_DUMP_TAP` is defined it should name a directory into which a copy of the raw TAP for each test will be written. TAP is written to files named for each test. Subdirectories will be created as needed.

Returns a *TAP::Parser::Aggregator* containing the test results.

summary

```
$harness->summary( $aggregator );
```

Output the summary for a *TAP::Parser::Aggregator*.

aggregate_tests

```
$harness->aggregate_tests( $aggregate, @tests );
```

Run the named tests and display a summary of result. Tests will be run in the order found.

Test results will be added to the supplied *TAP::Parser::Aggregator*. `aggregate_tests` may be called multiple times to run several sets of tests. Multiple `Test::Harness` instances may be used to pass results to a single aggregator so that different parts of a complex test suite may be run using different `TAP::Harness` settings. This is useful, for example, in the case where some tests should run in parallel but others are unsuitable for parallel execution.

```
my $formatter   = TAP::Formatter::Console->new;
my $ser_harness = TAP::Harness->new( { formatter => $formatter } );
my $par_harness = TAP::Harness->new(
    {   formatter => $formatter,
        jobs      => 9
    }
);
my $aggregator = TAP::Parser::Aggregator->new;

$aggregator->start();
$ser_harness->aggregate_tests( $aggregator, @ser_tests );
$par_harness->aggregate_tests( $aggregator, @par_tests );
$aggregator->stop();
$formatter->summary($aggregator);
```

Note that for simpler testing requirements it will often be possible to replace the above code with a single call to `runtests`.

Each element of the `@tests` array is either:

- the source name of a test to run
- a reference to a [source name, display name] array

In the case of a perl test suite, typically *source names* are simply the file names of the test scripts to run.

When you supply a separate display name it becomes possible to run a test more than once; the display name is effectively the alias by which the test is known inside the harness. The harness doesn't care if it runs the same test more than once when each invocation uses a different name.

make_scheduler

Called by the harness when it needs to create a *TAP::Parser::Scheduler*. Override in a subclass to provide an alternative scheduler. `make_scheduler` is passed the list of tests that was passed to `aggregate_tests`.

jobs

Gets or sets the number of concurrent test runs the harness is handling. By default, this value is 1 – for parallel testing, this should be set higher.

make_parser

Make a new parser and display formatter session. Typically used and/or overridden in subclasses.

```
my ( $parser, $session ) = $harness->make_parser;
```

finish_parser

Terminate use of a parser. Typically used and/or overridden in subclasses. The parser isn't destroyed as a result of this.

CONFIGURING

`TAP::Harness` is designed to be easy to configure.

Plugins

`TAP::Parser` plugins let you change the way TAP is *input* to and *output* from the parser.

*TAP::Parser::SourceHandler*s handle TAP *input*. You can configure them and load custom handlers using the `sources` parameter to "new".

*TAP::Formatter*s handle TAP *output*. You can load custom formatters by using the `formatter_class` parameter to "new". To configure a formatter, you currently need to instantiate it outside of *TAP::Harness* and pass it in with the `formatter` parameter to "new". This *may* be addressed by adding a *formatters* parameter to "new" in the future.

Module::Build

Module::Build version 0.30 supports TAP::Harness.

To load TAP::Harness plugins, you'll need to use the tap_harness_args parameter to new, typically from your Build.PL. For example:

```
Module::Build->new(
    module_name         => 'MyApp',
    test_file_exts      => [qw(.t .tap .txt)],
    use_tap_harness     => 1,
    tap_harness_args    => {
        sources => {
            MyCustom => {},
            File => {
                extensions => ['.tap', '.txt'],
            },
        },
        formatter_class => 'TAP::Formatter::HTML',
    },
    build_requires      => {
        'Module::Build' => '0.30',
        'TAP::Harness'  => '3.18',
    },
)->create_build_script;
```

See "new"

ExtUtils::MakeMaker

ExtUtils::MakeMaker does not support *TAP::Harness* out-of-the-box.

prove

prove supports TAP::Harness plugins, and has a plugin system of its own. See "FORMATTERS" in prove, "SOURCE HANDLERS" in prove and *App::Prove* for more details.

WRITING PLUGINS

If you can't configure TAP::Harness to do what you want, and you can't find an existing plugin, consider writing one.

The two primary use cases supported by *TAP::Harness* for plugins are *input* and *output*:

- Customize how TAP gets into the parser

  To do this, you can either extend an existing *TAP::Parser::SourceHandler*, or write your own. It's a pretty simple API, and they can be loaded and configured using the sources parameter to "new".

- Customize how TAP results are output from the parser

  To do this, you can either extend an existing *TAP::Formatter*, or write your own. Writing formatters are a bit more involved than writing a *SourceHandler*, as you'll

need to understand the *TAP::Parser* API. A good place to start is by understanding how “aggregate_tests” works.

Custom formatters can be loaded configured using the `formatter_class` parameter to “new”.

SUBCLASSING

If you can’t configure `TAP::Harness` to do exactly what you want, and writing a plugin isn’t an option, consider extending it. It is designed to be (mostly) easy to subclass, though the cases when sub-classing is necessary should be few and far between.

Methods

The following methods are ones you may wish to override if you want to subclass `TAP::Harness`.

- “new”
- “runtests”
- “summary”

REPLACING

If you like the `prove` utility and *TAP::Parser* but you want your own harness, all you need to do is write one and provide `new` and `runtests` methods. Then you can use the `prove` utility like so:

```
prove --harness My::Test::Harness
```

Note that while `prove` accepts a list of tests (or things to be tested), `new` has a fairly rich set of arguments. You’ll probably want to read over this code carefully to see how all of them are being used.

SEE ALSO

Test::Harness

Modules

- `TAP::Harness::Env` - Parsing harness related environmental variables where appropriate

TAP::Harness::Env

Parsing harness related environmental variables where appropriate

VERSION

Version 3.52

SYNOPSIS

```
my $harness = TAP::Harness::Env->create(\%extra_args)
```

DESCRIPTION

This module implements the environmental variables that *Test::Harness* uses with TAP::Harness, and instantiates the appropriate class with the appropriate arguments.

METHODS

- create(\%args)

  This function reads the environment and generates an appropriate argument hash from it. If given any arguments in `%extra_args`, these will override the environmental defaults. In accepts `harness_class` (which defaults to `TAP::Harness`), and any argument the harness class accepts.

ENVIRONMENTAL VARIABLES

- `HARNESS_PERL_SWITCHES`

  Setting this adds perl command line switches to each test file run.

  For example, `HARNESS_PERL_SWITCHES=-T` will turn on taint mode. `HARNESS_PERL_SWITCHES=-MDevel::Cover` will run `Devel::Cover` for each test.

- `HARNESS_VERBOSE`

  If true, `TAP::Harness` will output the verbose results of running its tests.

- `HARNESS_SUBCLASS`

  Specifies a TAP::Harness subclass to be used in place of TAP::Harness.

- `HARNESS_OPTIONS`

  Provide additional options to the harness. Currently supported options are:

  - `j<n>`

    Run <n> (default 9) parallel jobs.

  - `c`

    Try to color output. See *"new" in TAP::Formatter::Base*.

  - `a<file.tgz>`

    Will use TAP::Harness::Archive as the harness class, and save the TAP to `file.tgz`

- `fPackage-With-Dashes`

  Set the formatter_class of the harness being run. Since the `HARNESS_OPTIONS` is separated by `:`, we use `-` instead.

Multiple options may be separated by colons:

```
HARNESS_OPTIONS=j9:c make test
```

- `HARNESS_TIMER`

  Setting this to true will make the harness display the number of milliseconds each test took. You can also use `prove`'s `--timer` switch.

- `HARNESS_COLOR`

  Attempt to produce color output.

- `HARNESS_IGNORE_EXIT`

  If set to a true value instruct `TAP::Parser` to ignore exit and wait status from test scripts.

TAP::Object

Base class that provides common functionality to all `TAP::*` modules

VERSION

Version 3.52

SYNOPSIS

```
package TAP::Whatever;

use strict;

use base 'TAP::Object';

# new() implementation by TAP::Object
sub _initialize {
    my ( $self, @args) = @_;
    # initialize your object
    return $self;
}

# ... later ...
my $obj = TAP::Whatever->new(@args);
```

DESCRIPTION

`TAP::Object` provides a default constructor and exception model for all `TAP::*` classes. Exceptions are raised using *Carp*.

METHODS

Class Methods

new

Create a new object. Any arguments passed to `new` will be passed on to the "_initialize" method. Returns a new object.

Instance Methods

_initialize

Initializes a new object. This method is a stub by default, you should override it as appropriate.

Note: "new" expects you to return `$self` or raise an exception. See "_croak", and *Carp*.

_croak

Raise an exception using `croak` from *Carp*, eg:

```
$self->_croak( 'why me?', 'aaarrgh!' );
```

May also be called as a *class* method.

```
$class->_croak( 'this works too' );
```

_confess

Raise an exception using `confess` from *Carp*, eg:

```
$self->_confess( 'why me?', 'aaarrgh!' );
```

May also be called as a *class* method.

```
$class->_confess( 'this works too' );
```

_construct

Create a new instance of the specified class.

mk_methods

Create simple getter/setters.

```
__PACKAGE__->mk_methods(@method_names);
```

TAP::Parser

Parse TAP output

VERSION

Version 3.52

SYNOPSIS

```
use TAP::Parser;

my $parser = TAP::Parser->new( { source => $source } );

while ( my $result = $parser->next ) {
    print $result->as_string;
}
```

DESCRIPTION

`TAP::Parser` is designed to produce a proper parse of TAP output. For an example of how to run tests through this module, see the simple harnesses `examples/`.

There's a wiki dedicated to the Test Anything Protocol:

http://testanything.org

It includes the TAP::Parser Cookbook:

http://testanything.org/testing-with-tap/perl/tap::parser-cookbook.html

METHODS

Class Methods

new

```
my $parser = TAP::Parser->new(\%args);
```

Returns a new `TAP::Parser` object.

The arguments should be a hashref with *one* of the following keys:

- `source`

  CHANGED in 3.18

  This is the preferred method of passing input to the constructor.

The `source` is used to create a *TAP::Parser::Source* that is passed to the "iterator_factory_class" which in turn figures out how to handle the source and creates a TAP::Parser::Iterator for it. The iterator is used by the parser to read in the TAP stream.

To configure the *IteratorFactory* use the `sources` parameter below.

Note that `source`, `tap` and `exec` are *mutually exclusive*.

- `tap`

  CHANGED in 3.18

  The value should be the complete TAP output.

  The *tap* is used to create a *TAP::Parser::Source* that is passed to the "iterator_factory_class" which in turn figures out how to handle the source and creates a TAP::Parser::Iterator for it. The iterator is used by the parser to read in the TAP stream.

  To configure the *IteratorFactory* use the `sources` parameter below.

  Note that `source`, `tap` and `exec` are *mutually exclusive*.

- `exec`

  Must be passed an array reference.

  The *exec* array ref is used to create a *TAP::Parser::Source* that is passed to the "iterator_factory_class" which in turn figures out how to handle the source and creates a TAP::Parser::Iterator for it. The iterator is used by the parser to read in the TAP stream.

  By default the *TAP::Parser::SourceHandler::Executable* class will create a *TAP::Parser::Iterator::Process* object to handle the source. This passes the array reference strings as command arguments to *IPC::Open3::open3*:

  ```
  exec => [ '/usr/bin/ruby', 't/my_test.rb' ]
  ```

  If any `test_args` are given they will be appended to the end of the command argument list.

  To configure the *IteratorFactory* use the `sources` parameter below.

  Note that `source`, `tap` and `exec` are *mutually exclusive*.

The following keys are optional.

- `sources`

  NEW to 3.18.

  If set, `sources` must be a hashref containing the names of the *TAP::Parser::SourceHandler*s to load and/or configure. The values are a hash of configuration that will be accessible to the source handlers via *"config_for" in TAP::Parser::Source*.

  For example:

  ```
  sources => {
    Perl => { exec => '/path/to/custom/perl' },
  ```

(continues on next page)

(continued from previous page)

```
  File => { extensions => [ '.tap', '.txt' ] },
  MyCustom => { some => 'config' },
}
```

This will cause `TAP::Parser` to pass custom configuration to two of the built-in source handlers - *TAP::Parser::SourceHandler::Perl*, *TAP::Parser::SourceHandler::File* - and attempt to load the `MyCustom` class. See *"load_handlers" in TAP::Parser::IteratorFactory* for more detail.

The `sources` parameter affects how `source`, `tap` and `exec` parameters are handled.

See *TAP::Parser::IteratorFactory*, *TAP::Parser::SourceHandler* and subclasses for more details.

- `callback`

  If present, each callback corresponding to a given result type will be called with the result as the argument if the `run` method is used:

```
my %callbacks = (
    test    => \&test_callback,
    plan    => \&plan_callback,
    comment => \&comment_callback,
    bailout => \&bailout_callback,
    unknown => \&unknown_callback,
);

my $aggregator = TAP::Parser::Aggregator->new;
for my $file ( @test_files ) {
    my $parser = TAP::Parser->new(
        {
            source    => $file,
            callbacks => \%callbacks,
        }
    );
    $parser->run;
    $aggregator->add( $file, $parser );
}
```

- `switches`

  If using a Perl file as a source, optional switches may be passed which will be used when invoking the perl executable.

```
my $parser = TAP::Parser->new( {
    source   => $test_file,
    switches => [ '-Ilib' ],
} );
```

- `test_args`

  Used in conjunction with the `source` and `exec` option to supply a reference to an `@ARGV` style array of arguments to pass to the test program.

- `spool`

  If passed a filehandle will write a copy of all parsed TAP to that handle.

- `merge`

  If false, STDERR is not captured (though it is 'relayed' to keep it somewhat synchronized with STDOUT.)

  If true, STDERR and STDOUT are the same filehandle. This may cause breakage if STDERR contains anything resembling TAP format, but does allow exact synchronization.

  Subtleties of this behavior may be platform-dependent and may change in the future.

- `grammar_class`

  This option was introduced to let you easily customize which *grammar* class the parser should use. It defaults to *TAP::Parser::Grammar*.

  See also "make_grammar".

- `result_factory_class`

  This option was introduced to let you easily customize which *result* factory class the parser should use. It defaults to *TAP::Parser::ResultFactory*.

  See also "make_result".

- `iterator_factory_class`

  CHANGED in 3.18

  This option was introduced to let you easily customize which *iterator* factory class the parser should use. It defaults to *TAP::Parser::IteratorFactory*.

Instance Methods

next

```
my $parser = TAP::Parser->new( { source => $file } );
while ( my $result = $parser->next ) {
    print $result->as_string, "\n";
}
```

This method returns the results of the parsing, one result at a time. Note that it is destructive. You can't rewind and examine previous results.

If callbacks are used, they will be issued before this call returns.

Each result returned is a subclass of *TAP::Parser::Result*. See that module and related classes for more information on how to use them.

run

```
$parser->run;
```

This method merely runs the parser and parses all of the TAP.

make_grammar

Make a new *TAP::Parser::Grammar* object and return it. Passes through any arguments given.

The `grammar_class` can be customized, as described in "new".

make_result

Make a new *TAP::Parser::Result* object using the parser's *TAP::Parser::ResultFactory*, and return it. Passes through any arguments given.

The `result_factory_class` can be customized, as described in "new".

make_iterator_factory

NEW to 3.18.

Make a new *TAP::Parser::IteratorFactory* object and return it. Passes through any arguments given.

`iterator_factory_class` can be customized, as described in "new".

INDIVIDUAL RESULTS

If you've read this far in the docs, you've seen this:

```
while ( my $result = $parser->next ) {
    print $result->as_string;
}
```

Each result returned is a *TAP::Parser::Result* subclass, referred to as *result types*.

Result types

Basically, you fetch individual results from the TAP. The six types, with examples of each, are as follows:

- Version

  ```
  TAP version 12
  ```

- Plan

  ```
  1..42
  ```

- Pragma

```
pragma +strict
```

- Test

```
ok 3 - We should start with some foobar!
```

- Comment

```
# Hope we don't use up the foobar.
```

- Bailout

```
Bail out!  We ran out of foobar!
```

- Unknown

```
... yo, this ain't TAP! ...
```

Each result fetched is a result object of a different type. There are common methods to each result object and different types may have methods unique to their type. Sometimes a type method may be overridden in a subclass, but its use is guaranteed to be identical.

Common type methods

type

Returns the type of result, such as `comment` or `test`.

as_string

Prints a string representation of the token. This might not be the exact output, however. Tests will have test numbers added if not present, TODO and SKIP directives will be capitalized and, in general, things will be cleaned up. If you need the original text for the token, see the `raw` method.

raw

Returns the original line of text which was parsed.

is_plan

Indicates whether or not this is the test plan line.

is_test

Indicates whether or not this is a test line.

is_comment

Indicates whether or not this is a comment. Comments will generally only appear in the TAP stream if STDERR is merged to STDOUT. See the `merge` option.

is_bailout

Indicates whether or not this is bailout line.

is_yaml

Indicates whether or not the current item is a YAML block.

is_unknown

Indicates whether or not the current line could be parsed.

is_ok

```
if ( $result->is_ok ) { ... }
```

Reports whether or not a given result has passed. Anything which is **not** a test result returns true. This is merely provided as a convenient shortcut which allows you to do this:

```
my $parser = TAP::Parser->new( { source => $source } );
while ( my $result = $parser->next ) {
    # only print failing results
    print $result->as_string unless $result->is_ok;
}
```

plan methods

```
if ( $result->is_plan ) { ... }
```

If the above evaluates as true, the following methods will be available on the `$result` object.

plan

```
if ( $result->is_plan ) {
   print $result->plan;
}
```

This is merely a synonym for `as_string`.

directive

```
my $directive = $result->directive;
```

If a SKIP directive is included with the plan, this method will return it.

```
1..0 # SKIP: why bother?
```

explanation

```
my $explanation = $result->explanation;
```

If a SKIP directive was included with the plan, this method will return the explanation, if any.

pragma methods

```
if ( $result->is_pragma ) { ... }
```

If the above evaluates as true, the following methods will be available on the `$result` object.

pragmas

Returns a list of pragmas each of which is a + or - followed by the pragma name.

comment methods

```
if ( $result->is_comment ) { ... }
```

If the above evaluates as true, the following methods will be available on the `$result` object.

comment

```
if ( $result->is_comment ) {
    my $comment = $result->comment;
    print "I have something to say:  $comment";
}
```

bailout methods

```
if ( $result->is_bailout ) { ... }
```

If the above evaluates as true, the following methods will be available on the `$result` object.

explanation

```
if ( $result->is_bailout ) {
    my $explanation = $result->explanation;
    print "We bailed out because ($explanation)";
}
```

If, and only if, a token is a bailout token, you can get an "explanation" via this method. The explanation is the text after the mystical "Bail out!" words which appear in the tap output.

unknown methods

```
if ( $result->is_unknown ) { ... }
```

There are no unique methods for unknown results.

test methods

```
if ( $result->is_test ) { ... }
```

If the above evaluates as true, the following methods will be available on the `$result` object.

ok

```
my $ok = $result->ok;
```

Returns the literal text of the ok or not ok status.

number

```
my $test_number = $result->number;
```

Returns the number of the test, even if the original TAP output did not supply that number.

description

```
my $description = $result->description;
```

Returns the description of the test, if any. This is the portion after the test number but before the directive.

directive

```
my $directive = $result->directive;
```

Returns either `TODO` or `SKIP` if either directive was present for a test line.

explanation

```
my $explanation = $result->explanation;
```

If a test had either a `TODO` or `SKIP` directive, this method will return the accompanying explanation, if present.

```
not ok 17 - 'Pigs can fly' # TODO not enough acid
```

For the above line, the explanation is *not enough acid*.

is_ok

```
if ( $result->is_ok ) { ... }
```

Returns a boolean value indicating whether or not the test passed. Remember that for TODO tests, the test always passes.

Note: this was formerly `passed`. The latter method is deprecated and will issue a warning.

is_actual_ok

```
if ( $result->is_actual_ok ) { ... }
```

Returns a boolean value indicating whether or not the test passed, regardless of its TODO status.

Note: this was formerly `actual_passed`. The latter method is deprecated and will issue a warning.

is_unplanned

```
if ( $test->is_unplanned ) { ... }
```

If a test number is greater than the number of planned tests, this method will return true. Unplanned tests will *always* return false for `is_ok`, regardless of whether or not the test `has_todo` (see *TAP::Parser::Result::Test* for more information about this).

has_skip

```
if ( $result->has_skip ) { ... }
```

Returns a boolean value indicating whether or not this test had a SKIP directive.

has_todo

```
if ( $result->has_todo ) { ... }
```

Returns a boolean value indicating whether or not this test had a TODO directive.

Note that TODO tests *always* pass. If you need to know whether or not they really passed, check the `is_actual_ok` method.

in_todo

```
if ( $parser->in_todo ) { ... }
```

True while the most recent result was a TODO. Becomes true before the TODO result is returned and stays true until just before the next non- TODO test is returned.

TOTAL RESULTS

After parsing the TAP, there are many methods available to let you dig through the results and determine what is meaningful to you.

Individual Results

These results refer to individual tests which are run.

passed

```
my @passed = $parser->passed; # the test numbers which passed
my $passed = $parser->passed; # the number of tests which passed
```

This method lets you know which (or how many) tests passed. If a test failed but had a TODO directive, it will be counted as a passed test.

failed

```
my @failed = $parser->failed; # the test numbers which failed
my $failed = $parser->failed; # the number of tests which failed
```

This method lets you know which (or how many) tests failed. If a test passed but had a TODO directive, it will **NOT** be counted as a failed test.

actual_passed

```
# the test numbers which actually passed
my @actual_passed = $parser->actual_passed;

# the number of tests which actually passed
my $actual_passed = $parser->actual_passed;
```

This method lets you know which (or how many) tests actually passed, regardless of whether or not a TODO directive was found.

actual_ok

This method is a synonym for `actual_passed`.

actual_failed

```
# the test numbers which actually failed
my @actual_failed = $parser->actual_failed;

# the number of tests which actually failed
my $actual_failed = $parser->actual_failed;
```

This method lets you know which (or how many) tests actually failed, regardless of whether or not a TODO directive was found.

todo

```
my @todo = $parser->todo; # the test numbers with todo directives
my $todo = $parser->todo; # the number of tests with todo directives
```

This method lets you know which (or how many) tests had TODO directives.

todo_passed

```
# the test numbers which unexpectedly succeeded
my @todo_passed = $parser->todo_passed;

# the number of tests which unexpectedly succeeded
my $todo_passed = $parser->todo_passed;
```

This method lets you know which (or how many) tests actually passed but were declared as “TODO” tests.

todo_failed

```
# deprecated in favor of 'todo_passed'.  This method was horribly␣
 ↪misnamed.
```

This was a badly misnamed method. It indicates which TODO tests unexpectedly succeeded. Will now issue a warning and call `todo_passed`.

skipped

```
my @skipped = $parser->skipped; # the test numbers with SKIP␣
 ↪directives
my $skipped = $parser->skipped; # the number of tests with SKIP␣
 ↪directives
```

This method lets you know which (or how many) tests had SKIP directives.

Pragmas

pragma

Get or set a pragma. To get the state of a pragma:

```
if ( $p->pragma('strict') ) {
    # be strict
}
```

To set the state of a pragma:

```
$p->pragma('strict', 1); # enable strict mode
```

pragmas

Get a list of all the currently enabled pragmas:

```
my @pragmas_enabled = $p->pragmas;
```

Summary Results

These results are "meta" information about the total results of an individual test program.

plan

```
my $plan = $parser->plan;
```

Returns the test plan, if found.

good_plan

Deprecated. Use `is_good_plan` instead.

is_good_plan

```
if ( $parser->is_good_plan ) { ... }
```

Returns a boolean value indicating whether or not the number of tests planned matches the number of tests run.

Note: this was formerly `good_plan`. The latter method is deprecated and will issue a warning.

And since we're on that subject ...

tests_planned

```
print $parser->tests_planned;
```

Returns the number of tests planned, according to the plan. For example, a plan of '1..17' will mean that 17 tests were planned.

tests_run

```
print $parser->tests_run;
```

Returns the number of tests which actually were run. Hopefully this will match the number of `$parser->tests_planned`.

skip_all

Returns a true value (actually the reason for skipping) if all tests were skipped.

start_time

Returns the wall-clock time when the Parser was created.

end_time

Returns the wall-clock time when the end of TAP input was seen.

start_times

Returns the CPU times (like *"times" in perlfunc* when the Parser was created.

end_times

Returns the CPU times (like *"times" in perlfunc* when the end of TAP input was seen.

has_problems

```
if ( $parser->has_problems ) {
    ...
}
```

This is a 'catch-all' method which returns true if any tests have currently failed, any TODO tests unexpectedly succeeded, or any parse errors occurred.

version

```
$parser->version;
```

Once the parser is done, this will return the version number for the parsed TAP. Version numbers were introduced with TAP version 13 so if no version number is found version 12 is assumed.

exit

```
$parser->exit;
```

Once the parser is done, this will return the exit status. If the parser ran an executable, it returns the exit status of the executable.

wait

```
$parser->wait;
```

Once the parser is done, this will return the wait status. If the parser ran an executable, it returns the wait status of the executable. Otherwise, this merely returns the `exit` status.

ignore_exit

```
$parser->ignore_exit(1);
```

Tell the parser to ignore the exit status from the test when determining whether the test passed. Normally tests with non-zero exit status are considered to have failed even if all individual tests passed. In cases where it is not possible to control the exit value of the test script use this option to ignore it.

parse_errors

```
my @errors = $parser->parse_errors; # the parser errors
my $errors = $parser->parse_errors; # the number of parser_errors
```

Fortunately, all TAP output is perfect. In the event that it is not, this method will return parser errors. Note that a junk line which the parser does not recognize is `not` an error. This allows this parser to handle future versions of TAP. The following are all TAP errors reported by the parser:

- Misplaced plan

  The plan (for example, '1..5'), must only come at the beginning or end of the TAP output.

- No plan

  Gotta have a plan!

- More than one plan

  ```
  1..3
  ok 1 - input file opened
  not ok 2 - first line of the input valid # todo some data
  ok 3 read the rest of the file
  1..3
  ```

  Right. Very funny. Don't do that.

- Test numbers out of sequence

```
1..3
ok 1 - input file opened
not ok 2 - first line of the input valid # todo some data
ok 2 read the rest of the file
```

That last test line above should have the number '3' instead of '2'.

Note that it's perfectly acceptable for some lines to have test numbers and others to not have them. However, when a test number is found, it must be in sequence. The following is also an error:

```
1..3
ok 1 - input file opened
not ok - first line of the input valid # todo some data
ok 2 read the rest of the file
```

But this is not:

```
1..3
ok  - input file opened
not ok - first line of the input valid # todo some data
ok 3 read the rest of the file
```

get_select_handles

Get an a list of file handles which can be passed to `select` to determine the readiness of this parser.

delete_spool

Delete and return the spool.

```
my $fh = $parser->delete_spool;
```

CALLBACKS

As mentioned earlier, a "callback" key may be added to the `TAP::Parser` constructor. If present, each callback corresponding to a given result type will be called with the result as the argument if the `run` method is used. The callback is expected to be a subroutine reference (or anonymous subroutine) which is invoked with the parser result as its argument.

```
my %callbacks = (
    test    => \&test_callback,
    plan    => \&plan_callback,
    comment => \&comment_callback,
    bailout => \&bailout_callback,
    unknown => \&unknown_callback,
);
```

(continues on next page)

(continued from previous page)

```
my $aggregator = TAP::Parser::Aggregator->new;
for my $file ( @test_files ) {
    my $parser = TAP::Parser->new(
        {
            source    => $file,
            callbacks => \%callbacks,
        }
    );
    $parser->run;
    $aggregator->add( $file, $parser );
}
```

Callbacks may also be added like this:

```
$parser->callback( test => \&test_callback );
$parser->callback( plan => \&plan_callback );
```

The following keys allowed for callbacks. These keys are case-sensitive.

- `test`

  Invoked if `$result->is_test` returns true.

- `version`

  Invoked if `$result->is_version` returns true.

- `plan`

  Invoked if `$result->is_plan` returns true.

- `comment`

  Invoked if `$result->is_comment` returns true.

- `bailout`

  Invoked if `$result->is_unknown` returns true.

- `yaml`

  Invoked if `$result->is_yaml` returns true.

- `unknown`

  Invoked if `$result->is_unknown` returns true.

- `ELSE`

  If a result does not have a callback defined for it, this callback will be invoked. Thus, if all of the previous result types are specified as callbacks, this callback will *never* be invoked.

- `ALL`

  This callback will always be invoked and this will happen for each result after one of the above callbacks is invoked. For example, if *Term::ANSIColor* is loaded, you could use the following to color your test output:

```
my %callbacks = (
    test => sub {
        my $test = shift;
        if ( $test->is_ok && not $test->directive ) {
            # normal passing test
            print color 'green';
        }
        elsif ( !$test->is_ok ) {    # even if it's TODO
            print color 'white on_red';
        }
        elsif ( $test->has_skip ) {
            print color 'white on_blue';

        }
        elsif ( $test->has_todo ) {
            print color 'white';
        }
    },
    ELSE => sub {
        # plan, comment, and so on (anything which isn't a test
↪line)
        print color 'black on_white';
    },
    ALL => sub {
        # now print them
        print shift->as_string;
        print color 'reset';
        print "\n";
    },
);
```

- EOF

  Invoked when there are no more lines to be parsed. Since there is no accompanying *TAP::Parser::Result* object the `TAP::Parser` object is passed instead.

TAP GRAMMAR

If you're looking for an EBNF grammar, see *TAP::Parser::Grammar*.

BACKWARDS COMPATIBILITY

The Perl-QA list attempted to ensure backwards compatibility with *Test::Harness*. However, there are some minor differences.

Differences

- TODO plans

  A little-known feature of *Test::Harness* is that it supported TODO lists in the plan:

```
1..2 todo 2
ok 1 - We have liftoff
not ok 2 - Anti-gravity device activated
```

Under *Test::Harness*, test number 2 would *pass* because it was listed as a TODO test on the plan line. However, we are not aware of anyone actually using this feature and hard-coding test numbers is discouraged because it's very easy to add a test and break the test number sequence. This makes test suites very fragile. Instead, the following should be used:

```
1..2
ok 1 - We have liftoff
not ok 2 - Anti-gravity device activated # TODO
```

- 'Missing' tests

  It rarely happens, but sometimes a harness might encounter 'missing tests:

```
ok 1
ok 2
ok 15
ok 16
ok 17
```

  Test::Harness would report tests 3-14 as having failed. For the `TAP::Parser`, these tests are not considered failed because they've never run. They're reported as parse failures (tests out of sequence).

SUBCLASSING

If you find you need to provide custom functionality (as you would have using Test::Harness::Straps), you're in luck: `TAP::Parser` and friends are designed to be easily plugged-into and/or subclassed.

Before you start, it's important to know a few things:

1. All `TAP::*` objects inherit from *TAP::Object*.
2. Many `TAP::*` classes have a *SUBCLASSING* section to guide you.
3. Note that `TAP::Parser` is designed to be the central "maker" - ie: it is responsible for creating most new objects in the `TAP::Parser::*` namespace.

   This makes it possible for you to have a single point of configuring what subclasses should be used, which means that in many cases you'll find you only need to sub-class one of the parser's components.

   The exception to this rule are *SourceHandlers* & *Iterators*, but those are both created with customizable *IteratorFactory*.
4. By subclassing, you may end up overriding undocumented methods. That's not a bad thing per se, but be forewarned that undocumented methods may change without warning from one release to the next - we cannot guarantee backwards compatibility. If any *documented* method needs changing, it will be deprecated first, and changed in a later release.

Parser Components

Sources

A TAP parser consumes input from a single *raw source* of TAP, which could come from anywhere (a file, an executable, a database, an IO handle, a URI, etc..). The source gets bundled up in a *TAP::Parser::Source* object which gathers some meta data about it. The parser then uses a *TAP::Parser::IteratorFactory* to determine which *TAP::Parser::SourceHandler* to use to turn the raw source into a stream of TAP by way of "Iterators".

If you simply want `TAP::Parser` to handle a new source of TAP you probably don't need to subclass `TAP::Parser` itself. Rather, you'll need to create a new *TAP::Parser::SourceHandler* class, and just plug it into the parser using the *sources* param to "new". Before you start writing one, read through *TAP::Parser::IteratorFactory* to get a feel for how the system works first.

If you find you really need to use your own iterator factory you can still do so without sub-classing `TAP::Parser` by setting "iterator_factory_class".

If you just need to customize the objects on creation, subclass *TAP::Parser* and override "make_iterator_factory".

Note that `make_source` & `make_perl_source` have been *DEPRECATED* and are now removed.

Iterators

A TAP parser uses *iterators* to loop through the *stream* of TAP read in from the *source* it was given. There are a few types of Iterators available by default, all sub-classes of *TAP::Parser::Iterator*. Choosing which iterator to use is the responsibility of the *iterator factory*, though it simply delegates to the *Source Handler* it uses.

If you're writing your own *TAP::Parser::SourceHandler*, you may need to create your own iterators too. If so you'll need to subclass *TAP::Parser::Iterator*.

Note that "make_iterator" has been *DEPRECATED* and is now removed.

Results

A TAP parser creates *TAP::Parser::Result*s as it iterates through the input *stream*. There are quite a few result types available; choosing which class to use is the responsibility of the *result factory*.

To create your own result types you have two options:

- option 1

  Subclass *TAP::Parser::Result* and register your new result type/class with the default *TAP::Parser::ResultFactory*.

- option 2

  Subclass *TAP::Parser::ResultFactory* itself and implement your own *TAP::Parser::Result* creation logic. Then you'll need to customize the class used by your parser by setting the `result_factory_class` parameter. See "new" for more details.

If you need to customize the objects on creation, subclass *TAP::Parser* and override "make_result".

Grammar

TAP::Parser::Grammar is the heart of the parser. It tokenizes the TAP input *stream* and produces results. If you need to customize its behaviour you should probably familiarize yourself with the source first. Enough lecturing.

Subclass *TAP::Parser::Grammar* and customize your parser by setting the `grammar_class` parameter. See "new" for more details.

If you need to customize the objects on creation, subclass *TAP::Parser* and override "make_grammar"

ACKNOWLEDGMENTS

All of the following have helped. Bug reports, patches, (im)moral support, or just words of encouragement have all been forthcoming.

- Michael Schwern
- Andy Lester
- chromatic
- GEOFFR
- Shlomi Fish
- Torsten Schoenfeld
- Jerry Gay
- Aristotle
- Adam Kennedy
- Yves Orton
- Adrian Howard
- Sean & Lil
- Andreas J. Koenig
- Florian Ragwitz
- Corion
- Mark Stosberg
- Matt Kraai
- David Wheeler
- Alex Vandiver
- Cosimo Streppone
- Ville Skyttä

AUTHORS

Curtis “Ovid” Poe ovid@cpan.org

Andy Armstong andy@hexten.net

Eric Wilhelm @ <ewilhelm at cpan dot org>

Michael Peters <mpeters at plusthree dot com>

Leif Eriksen <leif dot eriksen at bigpond dot com>

Steve Purkis spurkis@cpan.org

Nicholas Clark nick@ccl4.org

Lee Johnson <notfadeaway at btinternet dot com>

Philippe Bruhat book@cpan.org

BUGS

Please report any bugs or feature requests to `bug-test-harness@rt.cpan.org`, or through the web interface at http://rt.cpan.org/NoAuth/ReportBug.html?Queue=Test-Harness. We will be notified, and then you’ll automatically be notified of progress on your bug as we make changes.

Obviously, bugs which include patches are best. If you prefer, you can patch against bleed by via anonymous checkout of the latest version:

```
git clone git://github.com/Perl-Toolchain-Gang/Test-Harness.git
```

COPYRIGHT & LICENSE

Copyright 2006-2008 Curtis “Ovid” Poe, all rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

Modules

- `TAP::Parser::Aggregator` - Aggregate TAP::Parser results
- `TAP::Parser::Grammar` - A grammar for the Test Anything Protocol.
- `TAP::Parser::Iterator` - Base class for TAP source iterators
- `TAP::Parser::IteratorFactory` - Figures out which SourceHandler objects to use for a given Source
- `TAP::Parser::Multiplexer` - Multiplex multiple TAP::Parsers
- `TAP::Parser::Result` - Base class for TAP::Parser output objects
- `TAP::Parser::ResultFactory` - Factory for creating TAP::Parser output objects
- `TAP::Parser::Scheduler` - Schedule tests during parallel testing
- `TAP::Parser::Source` - a TAP source & meta data about it

- `TAP::Parser::SourceHandler` - Base class for different TAP source handlers
- `TAP::Parser::YAMLish`

TAP::Parser::Aggregator

Aggregate TAP::Parser results

VERSION

Version 3.52

SYNOPSIS

```
use TAP::Parser::Aggregator;

my $aggregate = TAP::Parser::Aggregator->new;
$aggregate->add( 't/00-load.t', $load_parser );
$aggregate->add( 't/10-lex.t',  $lex_parser  );

my $summary = <<'END_SUMMARY';
Passed:  %s
Failed:  %s
Unexpectedly succeeded: %s
END_SUMMARY
printf $summary,
       scalar $aggregate->passed,
       scalar $aggregate->failed,
       scalar $aggregate->todo_passed;
```

DESCRIPTION

`TAP::Parser::Aggregator` collects parser objects and allows reporting/querying their aggregate results.

METHODS

Class Methods

new

```
my $aggregate = TAP::Parser::Aggregator->new;
```

Returns a new `TAP::Parser::Aggregator` object.

Instance Methods

add

```
$aggregate->add( $description => $parser );
```

The `$description` is usually a test file name (but only by convention.) It is used as a unique identifier (see e.g. "parsers".) Reusing a description is a fatal error.

The `$parser` is a *TAP::Parser* object.

parsers

```
my $count   = $aggregate->parsers;
my @parsers = $aggregate->parsers;
my @parsers = $aggregate->parsers(@descriptions);
```

In scalar context without arguments, this method returns the number of parsers aggregated. In list context without arguments, returns the parsers in the order they were added.

If `@descriptions` is given, these correspond to the keys used in each call to the add() method. Returns an array of the requested parsers (in the requested order) in list context or an array reference in scalar context.

Requesting an unknown identifier is a fatal error.

descriptions

Get an array of descriptions in the order in which they were added to the aggregator.

start

Call `start` immediately before adding any results to the aggregator. Among other times it records the start time for the test run.

stop

Call `stop` immediately after adding all test results to the aggregator.

elapsed

Elapsed returns a *Benchmark* object that represents the running time of the aggregated tests. In order for `elapsed` to be valid you must call `start` before running the tests and `stop` immediately afterwards.

elapsed_timestr

Returns a formatted string representing the runtime returned by `elapsed()`. This lets the caller not worry about Benchmark.

all_passed

Return true if all the tests passed and no parse errors were detected.

get_status

Get a single word describing the status of the aggregated tests. Depending on the outcome of the tests returns 'PASS', 'FAIL' or 'NOTESTS'. This token is understood by CPAN::Reporter.

Summary methods

Each of the following methods will return the total number of corresponding tests if called in scalar context. If called in list context, returns the descriptions of the parsers which contain the corresponding tests (see add for an explanation of description.

- failed
- parse_errors
- passed
- planned
- skipped
- todo
- todo_passed
- wait
- exit

For example, to find out how many tests unexpectedly succeeded (TODO tests which passed when they shouldn't):

```
my $count        = $aggregate->todo_passed;
my @descriptions = $aggregate->todo_passed;
```

Note that `wait` and `exit` are the totals of the wait and exit statuses of each of the tests. These values are totalled only to provide a true value if any of them are non-zero.

total

```
my $tests_run = $aggregate->total;
```

Returns the total number of tests run.

has_problems

```
if ( $parser->has_problems ) {
    ...
}
```

Identical to `has_errors`, but also returns true if any TODO tests unexpectedly succeeded. This is more akin to "warnings".

has_errors

```
if ( $parser->has_errors ) {
    ...
}
```

Returns true if *any* of the parsers failed. This includes:

- Failed tests
- Parse errors
- Bad exit or wait status

todo_failed

```
# deprecated in favor of 'todo_passed'.  This method was horribly␣
 ↪misnamed.
```

This was a badly misnamed method. It indicates which TODO tests unexpectedly succeeded. Will now issue a warning and call `todo_passed`.

See Also

TAP::Parser

TAP::Harness

TAP::Parser::Grammar

A grammar for the Test Anything Protocol.

VERSION

Version 3.52

SYNOPSIS

```
use TAP::Parser::Grammar;
my $grammar = $self->make_grammar({
  iterator => $tap_parser_iterator,
  parser   => $tap_parser,
  version  => 12,
});

my $result = $grammar->tokenize;
```

DESCRIPTION

`TAP::Parser::Grammar` tokenizes lines from a *TAP::Parser::Iterator* and constructs *TAP::Parser::Result* subclasses to represent the tokens.

Do not attempt to use this class directly. It won't make sense. It's mainly here to ensure that we will be able to have pluggable grammars when TAP is expanded at some future date (plus, this stuff was really cluttering the parser).

METHODS

Class Methods

new

```
my $grammar = TAP::Parser::Grammar->new({
    iterator => $iterator,
    parser   => $parser,
    version  => $version,
});
```

Returns *TAP::Parser* grammar object that will parse the TAP stream from the specified iterator. Both `iterator` and `parser` are required arguments. If `version` is not set it defaults to 12 (see "set_version" for more details).

Instance Methods

set_version

```
$grammar->set_version(13);
```

Tell the grammar which TAP syntax version to support. The lowest supported version is 12. Although 'TAP version' isn't valid version 12 syntax it is accepted so that higher version numbers may be parsed.

tokenize

```
my $token = $grammar->tokenize;
```

This method will return a *TAP::Parser::Result* object representing the current line of TAP.

token_types

```
my @types = $grammar->token_types;
```

Returns the different types of tokens which this grammar can parse.

syntax_for

```
my $syntax = $grammar->syntax_for($token_type);
```

Returns a pre-compiled regular expression which will match a chunk of TAP corresponding to the token type. For example (not that you should really pay attention to this, `$grammar->syntax_for('comment')` will return `qr/^#(.*)/`.

handler_for

```
my $handler = $grammar->handler_for($token_type);
```

Returns a code reference which, when passed an appropriate line of TAP, returns the lexed token corresponding to that line. As a result, the basic TAP parsing loop looks similar to the following:

```
my @tokens;
my $grammar = TAP::Grammar->new;
LINE: while ( defined( my $line = $parser->_next_chunk_of_tap ) ) {
    for my $type ( $grammar->token_types ) {
        my $syntax  = $grammar->syntax_for($type);
        if ( $line =~ $syntax ) {
            my $handler = $grammar->handler_for($type);
            push @tokens => $grammar->$handler($line);
            next LINE;
        }
    }
    push @tokens => $grammar->_make_unknown_token($line);
}
```

TAP GRAMMAR

NOTE: This grammar is slightly out of date. There's still some discussion about it and a new one will be provided when we have things better defined.

The *TAP::Parser* does not use a formal grammar because TAP is essentially a stream-based protocol. In fact, it's quite legal to have an infinite stream. For the same reason that we don't apply regexes to streams, we're not using a formal grammar here. Instead, we parse the TAP in lines.

For purposes for forward compatibility, any result which does not match the following grammar is currently referred to as *TAP::Parser::Result::Unknown*. It is *not* a parse error.

A formal grammar would look similar to the following:

```
(*
    For the time being, I'm cheating on the EBNF by allowing
    certain terms to be defined by POSIX character classes by
    using the following syntax:

      digit ::= [:digit:]
```

(continues on next page)

(continued from previous page)

```
    As far as I am aware, that's not valid EBNF.  Sue me.  I
    didn't know how to write "char" otherwise (Unicode issues).
    Suggestions welcome.
*)

tap            ::= version? { comment | unknown } leading_plan lines
                   |
                   lines trailing_plan {comment}

version        ::= 'TAP version ' positiveInteger {positiveInteger}
 ↪"\n"

leading_plan   ::= plan skip_directive? "\n"

trailing_plan  ::= plan "\n"

plan           ::= '1..' nonNegativeInteger

lines          ::= line {line}

line           ::= (comment | test | unknown | bailout ) "\n"

test           ::= status positiveInteger? description? directive?

status         ::= 'not '? 'ok '

description    ::= (character - (digit | '#')) {character - '#'}

directive      ::= todo_directive | skip_directive

todo_directive ::= hash_mark 'TODO' ' ' {character}

skip_directive ::= hash_mark 'SKIP' ' ' {character}

comment        ::= hash_mark {character}

hash_mark      ::= '#' {' '}

bailout        ::= 'Bail out!' {character}

unknown        ::= { (character - "\n") }

(* POSIX character classes and other terminals *)

digit              ::= [:digit:]
character          ::= ([:print:] - "\n")
positiveInteger    ::= ( digit - '0' ) {digit}
nonNegativeInteger ::= digit {digit}
```

SUBCLASSING

Please see *"SUBCLASSING" in TAP::Parser* for a subclassing overview.

If you *really* want to subclass *TAP::Parser*'s grammar the best thing to do is read through the code. There's no easy way of summarizing it here.

SEE ALSO

TAP::Object, *TAP::Parser*, *TAP::Parser::Iterator*, *TAP::Parser::Result*,

TAP::Parser::Iterator

Base class for TAP source iterators

VERSION

Version 3.52

SYNOPSIS

```
# to subclass:
use TAP::Parser::Iterator ();
use base 'TAP::Parser::Iterator';
sub _initialize {
  # see TAP::Object...
}

sub next_raw { ... }
sub wait     { ... }
sub exit     { ... }
```

DESCRIPTION

This is a simple iterator base class that defines *TAP::Parser*'s iterator API. Iterators are typically created from *TAP::Parser::SourceHandler*s.

METHODS

Class Methods

new

Create an iterator. Provided by *TAP::Object*.

Instance Methods

next

```
while ( my $item = $iter->next ) { ... }
```

Iterate through it, of course.

next_raw

Note: this method is abstract and should be overridden.

```
while ( my $item = $iter->next_raw ) { ... }
```

Iterate raw input without applying any fixes for quirky input syntax.

handle_unicode

If necessary switch the input stream to handle unicode. This only has any effect for I/O handle based streams.

The default implementation does nothing.

get_select_handles

Return a list of filehandles that may be used upstream in a select() call to signal that this Iterator is ready. Iterators that are not handle-based should return an empty list.

The default implementation does nothing.

wait

Note: this method is abstract and should be overridden.

```
my $wait_status = $iter->wait;
```

Return the `wait` status for this iterator.

exit

Note: this method is abstract and should be overridden.

```
my $wait_status = $iter->exit;
```

Return the `exit` status for this iterator.

SUBCLASSING

Please see *"SUBCLASSING" in TAP::Parser* for a subclassing overview.

You must override the abstract methods as noted above.

Example

TAP::Parser::Iterator::Array is probably the easiest example to follow. There's not much point repeating it here.

SEE ALSO

TAP::Object, *TAP::Parser*, *TAP::Parser::Iterator::Array*, *TAP::Parser::Iterator::Stream*, *TAP::Parser::Iterator::Process*,

Modules

- `TAP::Parser::Iterator::Array` - Iterator for array-based TAP sources
- `TAP::Parser::Iterator::Process` - Iterator for process-based TAP sources
- `TAP::Parser::Iterator::Stream` - Iterator for filehandle-based TAP sources

TAP::Parser::Iterator::Array

Iterator for array-based TAP sources

VERSION

Version 3.52

SYNOPSIS

```
use TAP::Parser::Iterator::Array;
my @data = ('foo', 'bar', baz');
my $it   = TAP::Parser::Iterator::Array->new(\@data);
my $line = $it->next;
```

DESCRIPTION

This is a simple iterator wrapper for arrays of scalar content, used by *TAP::Parser*. Unless you're writing a plugin or subclassing, you probably won't need to use this module directly.

METHODS

Class Methods

new

Create an iterator. Takes one argument: an `$array_ref`

Instance Methods

next

Iterate through it, of course.

next_raw

Iterate raw input without applying any fixes for quirky input syntax.

wait

Get the wait status for this iterator. For an array iterator this will always be zero.

exit

Get the exit status for this iterator. For an array iterator this will always be zero.

ATTRIBUTION

Originally ripped off from *Test::Harness*.

SEE ALSO

TAP::Object, *TAP::Parser*, *TAP::Parser::Iterator*,

TAP::Parser::Iterator::Process

Iterator for process-based TAP sources

VERSION

Version 3.52

SYNOPSIS

```
use TAP::Parser::Iterator::Process;
my %args = (
 command  => ['python', 'setup.py', 'test'],
 merge    => 1,
 setup    => sub { ... },
 teardown => sub { ... },
);
my $it   = TAP::Parser::Iterator::Process->new(\%args);
my $line = $it->next;
```

DESCRIPTION

This is a simple iterator wrapper for executing external processes, used by *TAP::Parser*. Unless you're writing a plugin or subclassing, you probably won't need to use this module directly.

METHODS

Class Methods

new

Create an iterator. Expects one argument containing a hashref of the form:

```
command  => \@command_to_execute
merge    => $attempt_merge_stderr_and_stdout?
setup    => $callback_to_setup_command
teardown => $callback_to_teardown_command
```

Tries to uses *IPC::Open3* & *IO::Select* to communicate with the spawned process if they are available. Falls back onto `open()`.

Instance Methods

next

Iterate through the process output, of course.

next_raw

Iterate raw input without applying any fixes for quirky input syntax.

wait

Get the wait status for this iterator's process.

exit

Get the exit status for this iterator's process.

handle_unicode

Upgrade the input stream to handle UTF8.

get_select_handles

Return a list of filehandles that may be used upstream in a select() call to signal that this Iterator is ready. Iterators that are not handle based should return an empty list.

ATTRIBUTION

Originally ripped off from *Test::Harness*.

SEE ALSO

TAP::Object, *TAP::Parser*, *TAP::Parser::Iterator*,

TAP::Parser::Iterator::Stream

Iterator for filehandle-based TAP sources

VERSION

Version 3.52

SYNOPSIS

```
use TAP::Parser::Iterator::Stream;
open( TEST, 'test.tap' );
my $it   = TAP::Parser::Iterator::Stream->new(\*TEST);
my $line = $it->next;
```

DESCRIPTION

This is a simple iterator wrapper for reading from filehandles, used by *TAP::Parser*. Unless you're writing a plugin or subclassing, you probably won't need to use this module directly.

METHODS

Class Methods

new

Create an iterator. Expects one argument containing a filehandle.

Instance Methods

next

Iterate through it, of course.

next_raw

Iterate raw input without applying any fixes for quirky input syntax.

wait

Get the wait status for this iterator. Always returns zero.

exit

Get the exit status for this iterator. Always returns zero.

ATTRIBUTION

Originally ripped off from *Test::Harness*.

SEE ALSO

TAP::Object, *TAP::Parser*, *TAP::Parser::Iterator*,

TAP::Parser::IteratorFactory

Figures out which SourceHandler objects to use for a given Source

VERSION

Version 3.52

SYNOPSIS

```
use TAP::Parser::IteratorFactory;
my $factory = TAP::Parser::IteratorFactory->new({ %config });
my $iterator  = $factory->make_iterator( $filename );
```

DESCRIPTION

This is a factory class that takes a *TAP::Parser::Source* and runs it through all the registered *TAP::Parser::SourceHandler*s to see which one should handle the source.

If you're a plugin author, you'll be interested in how to "register_handler"s, how "detect_source" works.

METHODS

Class Methods

new

Creates a new factory class:

```
my $sf = TAP::Parser::IteratorFactory->new( $config );
```

`$config` is optional. If given, sets "config" and calls "load_handlers".

register_handler

Registers a new *TAP::Parser::SourceHandler* with this factory.

```
__PACKAGE__->register_handler( $handler_class );
```

handlers

List of handlers that have been registered.

Instance Methods

config

```
my $cfg = $sf->config;
$sf->config({ Perl => { %config } });
```

Chaining getter/setter for the configuration of the available source handlers. This is a hashref keyed on handler class whose values contain config to be passed onto the handlers during detection & creation. Class names may be fully qualified or abbreviated, eg:

```
# these are equivalent
$sf->config({ 'TAP::Parser::SourceHandler::Perl' => { %config } });
$sf->config({ 'Perl' => { %config } });
```

load_handlers

```
$sf->load_handlers;
```

Loads the handler classes defined in "config". For example, given a config:

```
$sf->config({
  MySourceHandler => { some => 'config' },
});
```

`load_handlers` will attempt to load the `MySourceHandler` class by looking in `@INC` for it in this order:

```
TAP::Parser::SourceHandler::MySourceHandler
MySourceHandler
```

`croaks` on error.

make_iterator

```
my $iterator = $src_factory->make_iterator( $source );
```

Given a *TAP::Parser::Source*, finds the most suitable *TAP::Parser::SourceHandler* to use to create a *TAP::Parser::Iterator* (see "detect_source"). Dies on error.

detect_source

Given a *TAP::Parser::Source*, detects what kind of source it is and returns *one TAP::Parser::SourceHandler* (the most confident one). Dies on error.

The detection algorithm works something like this:

```
for (@registered_handlers) {
  # ask them how confident they are about handling this source
  $confidence{$handler} = $handler->can_handle( $source )
}
# choose the most confident handler
```

Ties are handled by choosing the first handler.

SUBCLASSING

Please see *"SUBCLASSING" in TAP::Parser* for a subclassing overview.

Example

If we've done things right, you'll probably want to write a new source, rather than sub-classing this (see *TAP::Parser::SourceHandler* for that).

But in case you find the need to...

```
package MyIteratorFactory;

use strict;

use base 'TAP::Parser::IteratorFactory';

# override source detection algorithm
sub detect_source {
  my ($self, $raw_source_ref, $meta) = @_;
  # do detective work, using $meta and whatever else...
}

1;
```

AUTHORS

Steve Purkis

ATTRIBUTION

Originally ripped off from *Test::Harness*.

Moved out of *TAP::Parser* & converted to a factory class to support extensible TAP source detective work by Steve Purkis.

SEE ALSO

TAP::Object, *TAP::Parser*, *TAP::Parser::SourceHandler*, *TAP::Parser::SourceHandler::File*, *TAP::Parser::SourceHandler::Perl*, *TAP::Parser::SourceHandler::RawTAP*, *TAP::Parser::SourceHandler::Handle*, *TAP::Parser::SourceHandler::Executable*

TAP::Parser::Multiplexer

Multiplex multiple TAP::Parsers

VERSION

Version 3.52

SYNOPSIS

```
use TAP::Parser::Multiplexer;

my $mux = TAP::Parser::Multiplexer->new;
$mux->add( $parser1, $stash1 );
$mux->add( $parser2, $stash2 );
while ( my ( $parser, $stash, $result ) = $mux->next ) {
    # do stuff
}
```

DESCRIPTION

`TAP::Parser::Multiplexer` gathers input from multiple TAP::Parsers. Internally it calls select on the input file handles for those parsers to wait for one or more of them to have input available.

See *TAP::Harness* for an example of its use.

METHODS

Class Methods

new

```
my $mux = TAP::Parser::Multiplexer->new;
```

Returns a new `TAP::Parser::Multiplexer` object.

Instance Methods

add

```
$mux->add( $parser, $stash );
```

Add a TAP::Parser to the multiplexer. `$stash` is an optional opaque reference that will be returned from `next` along with the parser and the next result.

parsers

```
my $count   = $mux->parsers;
```

Returns the number of parsers. Parsers are removed from the multiplexer when their input is exhausted.

next

Return a result from the next available parser. Returns a list containing the parser from which the result came, the stash that corresponds with that parser and the result.

```
my ( $parser, $stash, $result ) = $mux->next;
```

If `$result` is undefined the corresponding parser has reached the end of its input (and will automatically be removed from the multiplexer).

When all parsers are exhausted an empty list will be returned.

```
if ( my ( $parser, $stash, $result ) = $mux->next ) {
    if ( ! defined $result ) {
        # End of this parser
    }
    else {
        # Process result
    }
}
else {
    # All parsers finished
}
```

See Also

TAP::Parser

TAP::Harness

TAP::Parser::Result

Base class for TAP::Parser output objects

VERSION

Version 3.52

SYNOPSIS

```
# abstract class - not meant to be used directly
# see TAP::Parser::ResultFactory for preferred usage
```

(continues on next page)

(continued from previous page)

```
# directly:
use TAP::Parser::Result;
my $token  = {...};
my $result = TAP::Parser::Result->new( $token );
```

DESCRIPTION

This is a simple base class used by *TAP::Parser* to store objects that represent the current bit of test output data from TAP (usually a single line). Unless you're subclassing, you probably won't need to use this module directly.

METHODS

new

```
# see TAP::Parser::ResultFactory for preferred usage

# to use directly:
my $result = TAP::Parser::Result->new($token);
```

Returns an instance the appropriate class for the test token passed in.

Boolean methods

The following methods all return a boolean value and are to be overridden in the appropriate subclass.

- `is_plan`

  Indicates whether or not this is the test plan line.

  ```
  1..3
  ```

- `is_pragma`

  Indicates whether or not this is a pragma line.

  ```
  pragma +strict
  ```

- `is_test`

  Indicates whether or not this is a test line.

  ```
  ok 1 Is OK!
  ```

- `is_comment`

  Indicates whether or not this is a comment.

  ```
  # this is a comment
  ```

- `is_bailout`

  Indicates whether or not this is bailout line.

  ```
  Bail out! We're out of dilithium crystals.
  ```

- `is_version`

  Indicates whether or not this is a TAP version line.

  ```
  TAP version 4
  ```

- `is_unknown`

  Indicates whether or not the current line could be parsed.

  ```
  ... this line is junk ...
  ```

- `is_yaml`

  Indicates whether or not this is a YAML chunk.

raw

```
print $result->raw;
```

Returns the original line of text which was parsed.

type

```
my $type = $result->type;
```

Returns the "type" of a token, such as `comment` or `test`.

as_string

```
print $result->as_string;
```

Prints a string representation of the token. This might not be the exact output, however. Tests will have test numbers added if not present, TODO and SKIP directives will be capitalized and, in general, things will be cleaned up. If you need the original text for the token, see the `raw` method.

is_ok

```
if ( $result->is_ok ) { ... }
```

Reports whether or not a given result has passed. Anything which is **not** a test result returns true. This is merely provided as a convenient shortcut.

passed

Deprecated. Please use `is_ok` instead.

has_directive

```
if ( $result->has_directive ) {
    ...
}
```

Indicates whether or not the given result has a TODO or SKIP directive.

has_todo

```
if ( $result->has_todo ) {
    ...
}
```

Indicates whether or not the given result has a TODO directive.

has_skip

```
if ( $result->has_skip ) {
    ...
}
```

Indicates whether or not the given result has a SKIP directive.

set_directive

Set the directive associated with this token. Used internally to fake TODO tests.

SUBCLASSING

Please see *"SUBCLASSING" in TAP::Parser* for a subclassing overview.

Remember: if you want your subclass to be automatically used by the parser, you'll have to register it with *"register_type" in TAP::Parser::ResultFactory*.

If you're creating a completely new result *type*, you'll probably need to subclass *TAP::Parser::Grammar* too, or else it'll never get used.

Example

```
package MyResult;

use strict;

use base 'TAP::Parser::Result';
```

(continues on next page)

(continued from previous page)

```
# register with the factory:
TAP::Parser::ResultFactory->register_type( 'my_type' => __PACKAGE__
↪);

sub as_string { 'My results all look the same' }
```

SEE ALSO

TAP::Object, *TAP::Parser*, *TAP::Parser::ResultFactory*, *TAP::Parser::Result::Bailout*, *TAP::Parser::Result::Comment*, *TAP::Parser::Result::Plan*, *TAP::Parser::Result::Pragma*, *TAP::Parser::Result::Test*, *TAP::Parser::Result::Unknown*, *TAP::Parser::Result::Version*, *TAP::Parser::Result::YAML*,

Modules

- `TAP::Parser::Result::Bailout` - Bailout result token.
- `TAP::Parser::Result::Comment` - Comment result token.
- `TAP::Parser::Result::Plan` - Plan result token.
- `TAP::Parser::Result::Pragma` - TAP pragma token.
- `TAP::Parser::Result::Test` - Test result token.
- `TAP::Parser::Result::Unknown` - Unknown result token.
- `TAP::Parser::Result::Version` - TAP syntax version token.
- `TAP::Parser::Result::YAML` - YAML result token.

TAP::Parser::Result::Bailout

Bailout result token.

VERSION

Version 3.52

DESCRIPTION

This is a subclass of *TAP::Parser::Result*. A token of this class will be returned if a bail out line is encountered.

```
1..5
ok 1 - woo hooo!
Bail out! Well, so much for "woo hooo!"
```

OVERRIDDEN METHODS

Mainly listed here to shut up the pitiful screams of the pod coverage tests. They keep me awake at night.

- `as_string`

Instance Methods

explanation

```
if ( $result->is_bailout ) {
    my $explanation = $result->explanation;
    print "We bailed out because ($explanation)";
}
```

If, and only if, a token is a bailout token, you can get an "explanation" via this method. The explanation is the text after the mystical "Bail out!" words which appear in the tap output.

TAP::Parser::Result::Comment

Comment result token.

VERSION

Version 3.52

DESCRIPTION

This is a subclass of *TAP::Parser::Result*. A token of this class will be returned if a comment line is encountered.

```
1..1
ok 1 - woo hooo!
# this is a comment
```

OVERRIDDEN METHODS

Mainly listed here to shut up the pitiful screams of the pod coverage tests. They keep me awake at night.

- `as_string`

  Note that this method merely returns the comment preceded by a '# '.

Instance Methods

comment

```
if ( $result->is_comment ) {
    my $comment = $result->comment;
    print "I have something to say:  $comment";
}
```

TAP::Parser::Result::Plan

Plan result token.

VERSION

Version 3.52

DESCRIPTION

This is a subclass of *TAP::Parser::Result*. A token of this class will be returned if a plan line is encountered.

```
1..1
ok 1 - woo hooo!
```

1..1 is the plan. Gotta have a plan.

OVERRIDDEN METHODS

Mainly listed here to shut up the pitiful screams of the pod coverage tests. They keep me awake at night.

- as_string
- raw

Instance Methods

plan

```
if ( $result->is_plan ) {
   print $result->plan;
}
```

This is merely a synonym for as_string.

tests_planned

```
my $planned = $result->tests_planned;
```

Returns the number of tests planned. For example, a plan of 1..17 will cause this method to return '17'.

directive

```
my $directive = $plan->directive;
```

If a SKIP directive is included with the plan, this method will return it.

```
1..0 # SKIP: why bother?
```

has_skip

```
if ( $result->has_skip ) { ... }
```

Returns a boolean value indicating whether or not this test has a SKIP directive.

explanation

```
my $explanation = $plan->explanation;
```

If a SKIP directive was included with the plan, this method will return the explanation, if any.

todo_list

```
my $todo = $result->todo_list;
for ( @$todo ) {
    ...
}
```

TAP::Parser::Result::Pragma

TAP pragma token.

VERSION

Version 3.52

DESCRIPTION

This is a subclass of *TAP::Parser::Result*. A token of this class will be returned if a pragma is encountered.

```
TAP version 13
pragma +strict, -foo
```

Pragmas are only supported from TAP version 13 onwards.

OVERRIDDEN METHODS

Mainly listed here to shut up the pitiful screams of the pod coverage tests. They keep me awake at night.

- `as_string`
- `raw`

Instance Methods

`pragmas`

if ($result->is_pragma) { @pragmas = $result->pragmas; }

TAP::Parser::Result::Test

Test result token.

VERSION

Version 3.52

DESCRIPTION

This is a subclass of *TAP::Parser::Result*. A token of this class will be returned if a test line is encountered.

```
1..1
ok 1 - woo hooo!
```

OVERRIDDEN METHODS

This class is the workhorse of the *TAP::Parser* system. Most TAP lines will be test lines and if `$result->is_test`, then you have a bunch of methods at your disposal.

Instance Methods

`ok`

```
my $ok = $result->ok;
```

Returns the literal text of the ok or `not ` ok status.

`number`

```
my $test_number = $result->number;
```

Returns the number of the test, even if the original TAP output did not supply that number.

description

```
my $description = $result->description;
```

Returns the description of the test, if any. This is the portion after the test number but before the directive.

directive

```
my $directive = $result->directive;
```

Returns either `TODO` or `SKIP` if either directive was present for a test line.

explanation

```
my $explanation = $result->explanation;
```

If a test had either a `TODO` or `SKIP` directive, this method will return the accompanying explanation, if present.

```
not ok 17 - 'Pigs can fly' # TODO not enough acid
```

For the above line, the explanation is *not enough acid*.

is_ok

```
if ( $result->is_ok ) { ... }
```

Returns a boolean value indicating whether or not the test passed. Remember that for TODO tests, the test always passes.

If the test is unplanned, this method will always return false. See `is_unplanned`.

is_actual_ok

```
if ( $result->is_actual_ok ) { ... }
```

Returns a boolean value indicating whether or not the test passed, regardless of its TODO status.

actual_passed

Deprecated. Please use `is_actual_ok` instead.

todo_passed

```
if ( $test->todo_passed ) {
   # test unexpectedly succeeded
}
```

If this is a TODO test and an 'ok' line, this method returns true. Otherwise, it will always return false (regardless of passing status on non-todo tests).

This is used to track which tests unexpectedly succeeded.

todo_failed

```
# deprecated in favor of 'todo_passed'.  This method was horribly␣
↪misnamed.
```

This was a badly misnamed method. It indicates which TODO tests unexpectedly succeeded. Will now issue a warning and call `todo_passed`.

has_skip

```
if ( $result->has_skip ) { ... }
```

Returns a boolean value indicating whether or not this test has a SKIP directive.

has_todo

```
if ( $result->has_todo ) { ... }
```

Returns a boolean value indicating whether or not this test has a TODO directive.

as_string

```
print $result->as_string;
```

This method prints the test as a string. It will probably be similar, but not necessarily identical, to the original test line. Directives are capitalized, some whitespace may be trimmed and a test number will be added if it was not present in the original line. If you need the original text of the test line, use the `raw` method.

is_unplanned

```
if ( $test->is_unplanned ) { ... }
$test->is_unplanned(1);
```

If a test number is greater than the number of planned tests, this method will return true. Unplanned tests will *always* return false for `is_ok`, regardless of whether or not the test `has_todo`.

Note that if tests have a trailing plan, it is not possible to set this property for unplanned tests as we do not know it's unplanned until the plan is reached:

```
print <<'END';
ok 1
ok 2
```

(continues on next page)

(continued from previous page)

```
1..1
END
```

TAP::Parser::Result::Unknown

Unknown result token.

VERSION

Version 3.52

DESCRIPTION

This is a subclass of *TAP::Parser::Result*. A token of this class will be returned if the parser does not recognize the token line. For example:

```
1..5
VERSION 7
ok 1 - woo hooo!
... woo hooo! is cool!
```

In the above "TAP", the second and fourth lines will generate "Unknown" tokens.

OVERRIDDEN METHODS

Mainly listed here to shut up the pitiful screams of the pod coverage tests. They keep me awake at night.

- `as_string`
- `raw`

TAP::Parser::Result::Version

TAP syntax version token.

VERSION

Version 3.52

DESCRIPTION

This is a subclass of *TAP::Parser::Result*. A token of this class will be returned if a version line is encountered.

```
TAP version 13
ok 1
not ok 2
```

The first version of TAP to include an explicit version number is 13.

OVERRIDDEN METHODS

Mainly listed here to shut up the pitiful screams of the pod coverage tests. They keep me awake at night.

- `as_string`
- `raw`

Instance Methods

version

```
if ( $result->is_version ) {
   print $result->version;
}
```

This is merely a synonym for `as_string`.

TAP::Parser::Result::YAML

YAML result token.

VERSION

Version 3.52

DESCRIPTION

This is a subclass of *TAP::Parser::Result*. A token of this class will be returned if a YAML block is encountered.

```
1..1
ok 1 - woo hooo!
```

`1..1` is the plan. Gotta have a plan.

OVERRIDDEN METHODS

Mainly listed here to shut up the pitiful screams of the pod coverage tests. They keep me awake at night.

- `as_string`
- `raw`

Instance Methods

data

```
if ( $result->is_yaml ) {
   print $result->data;
}
```

Return the parsed YAML data for this result

TAP::Parser::ResultFactory

Factory for creating TAP::Parser output objects

SYNOPSIS

```
use TAP::Parser::ResultFactory;
my $token   = {...};
my $factory = TAP::Parser::ResultFactory->new;
my $result  = $factory->make_result( $token );
```

VERSION

Version 3.52

DESCRIPTION

This is a simple factory class which returns a *TAP::Parser::Result* subclass representing the current bit of test data from TAP (usually a single line). It is used primarily by *TAP::Parser::Grammar*. Unless you're subclassing, you probably won't need to use this module directly.

METHODS

Class Methods

new

Creates a new factory class. *Note:* You currently don't need to instantiate a factory in order to use it.

make_result

Returns an instance the appropriate class for the test token passed in.

```
my $result = TAP::Parser::ResultFactory->make_result($token);
```

Can also be called as an instance method.

class_for

Takes one argument: `$type`. Returns the class for this $type, or `croaks` with an error.

register_type

Takes two arguments: `$type`, `$class`

This lets you override an existing type with your own custom type, or register a completely new type, eg:

```
# create a custom result type:
package MyResult;
use strict;
use base 'TAP::Parser::Result';

# register with the factory:
TAP::Parser::ResultFactory->register_type( 'my_type' => __PACKAGE__
 ↪);

# use it:
my $r = TAP::Parser::ResultFactory->( { type => 'my_type' } );
```

Your custom type should then be picked up automatically by the *TAP::Parser*.

SUBCLASSING

Please see *"SUBCLASSING" in TAP::Parser* for a subclassing overview.

There are a few things to bear in mind when creating your own `ResultFactory`:

1. The factory itself is never instantiated (this *may* change in the future). This means that `_initialize` is never called.
2. `TAP::Parser::Result->new` is never called, $tokens are reblessed. This *will* change in a future version!
3. *TAP::Parser::Result* subclasses will register themselves with *TAP::Parser::ResultFactory* directly:

   ```
   package MyFooResult;
   TAP::Parser::ResultFactory->register_type( foo => __PACKAGE__ );
   ```

   Of course, it's up to you to decide whether or not to ignore them.

Example

```
package MyResultFactory;

use strict;

use MyResult;
```

(continues on next page)

(continued from previous page)

```
use base 'TAP::Parser::ResultFactory';

# force all results to be 'MyResult'
sub class_for {
  return 'MyResult';
}

1;
```

SEE ALSO

TAP::Parser, *TAP::Parser::Result*, *TAP::Parser::Grammar*

TAP::Parser::Scheduler

Schedule tests during parallel testing

VERSION

Version 3.52

SYNOPSIS

```
use TAP::Parser::Scheduler;
```

DESCRIPTION

METHODS

Class Methods

new

```
my $sched = TAP::Parser::Scheduler->new(tests => \@tests);
my $sched = TAP::Parser::Scheduler->new(
    tests => [ ['t/test_name.t','Test Description'], ... ],
    rules => \%rules,
);
```

Given 'tests' and optional 'rules' as input, returns a new `TAP::Parser::Scheduler` object. Each member of `@tests` should be either a a test file name, or a two element arrayref, where the first element is a test file name, and the second element is a test description. By default, we'll use the test name as the description.

The optional `rules` attribute provides direction on which tests should be run in parallel and which should be run sequentially. If no rule data structure is provided, a default data structure is used which makes every test eligible to be run in parallel:

```
{ par => '**' },
```

The rules data structure is documented more in the next section.

Rules data structure

The "rules" data structure is the the heart of the scheduler. It allows you to express simple rules like "run all tests in sequence" or "run all tests in parallel except these five tests.". However, the rules structure also supports glob-style pattern matching and recursive definitions, so you can also express arbitrarily complicated patterns.

The rule must only have one top level key: either 'par' for "parallel" or 'seq' for "sequence".

Values must be either strings with possible glob-style matching, or arrayrefs of strings or hashrefs which follow this pattern recursively.

Every element in an arrayref directly below a 'par' key is eligible to be run in parallel, while values directly below a 'seq' key must be run in sequence.

Rules examples

Here are some examples:

```
# All tests be run in parallel (the default rule)
{ par => '**' },

# Run all tests in sequence, except those starting with "p"
{ par => 't/p*.t' },

# Run all tests in parallel, except those starting with "p"
{
    seq => [
              { seq => 't/p*.t' },
              { par => '**'     },
           ],
}

# Run some  startup tests in sequence, then some parallel tests then
 ↪some
# teardown tests in sequence.
{
    seq => [
        { seq => 't/startup/*.t' },
        { par => ['t/a/*.t','t/b/*.t','t/c/*.t'], }
        { seq => 't/shutdown/*.t' },
    ],
},
```

Rules resolution

- By default, all tests are eligible to be run in parallel. Specifying any of your own rules removes this one.
- "First match wins". The first rule that matches a test will be the one that applies.
- Any test which does not match a rule will be run in sequence at the end of the run.
- The existence of a rule does not imply selecting a test. You must still specify the tests to run.
- Specifying a rule to allow tests to run in parallel does not make the run in parallel. You still need specify the number of parallel `jobs` in your Harness object.

Glob-style pattern matching for rules

We implement our own glob-style pattern matching. Here are the patterns it supports:

```
** is any number of characters, including /, within a pathname
* is zero or more characters within a filename/directory name
? is exactly one character within a filename/directory name
{foo,bar,baz} is any of foo, bar or baz.
\ is an escape character
```

Instance Methods

get_all

Get a list of all remaining tests.

get_job

Return the next available job as *TAP::Parser::Scheduler::Job* object or `undef` if none are available. Returns a *TAP::Parser::Scheduler::Spinner* if the scheduler still has pending jobs but none are available to run right now.

as_string

Return a human readable representation of the scheduling tree. For example:

```
my @tests = (qw{
    t/startup/foo.t
    t/shutdown/foo.t

    t/a/foo.t t/b/foo.t t/c/foo.t t/d/foo.t
});
my $sched = TAP::Parser::Scheduler->new(
    tests => \@tests,
    rules => {
        seq => [
            { seq => 't/startup/*.t' },
```

(continues on next page)

(continued from previous page)

```
            { par => ['t/a/*.t','t/b/*.t','t/c/*.t'] },
            { seq => 't/shutdown/*.t' },
        ],
    },
);
```

Produces:

```
par:
  seq:
    par:
      seq:
        par:
          seq:
            't/startup/foo.t'
        par:
          seq:
            't/a/foo.t'
          seq:
            't/b/foo.t'
          seq:
            't/c/foo.t'
        par:
          seq:
            't/shutdown/foo.t'
    't/d/foo.t'
```

Modules

- `TAP::Parser::Scheduler::Job` - A single testing job.
- `TAP::Parser::Scheduler::Spinner` - A no-op job.

TAP::Parser::Scheduler::Job

A single testing job.

VERSION

Version 3.52

SYNOPSIS

```
use TAP::Parser::Scheduler::Job;
```

DESCRIPTION

Represents a single test 'job'.

METHODS

Class Methods

new

```
my $job = TAP::Parser::Scheduler::Job->new(
    $filename, $description
);
```

Given the filename and description of a test as scalars, returns a new *TAP::Parser::Scheduler::Job* object.

Instance Methods

on_finish

```
$self->on_finish(\&method).
```

Register a closure to be called when this job is destroyed. The callback will be passed the `TAP::Parser::Scheduler::Job` object as it's only argument.

finish

```
$self->finish;
```

Called when a job is complete to unlock it. If a callback has been registered with `on_finish`, it calls it. Otherwise, it does nothing.

Attributes

```
$self->filename;
$self->description;
$self->context;
```

These are all "getters" which return the data set for these attributes during object construction.

filename

description

context

as_array_ref

For backwards compatibility in callbacks.

is_spinner

```
$self->is_spinner;
```

Returns false indicating that this is a real job rather than a 'spinner'. Spinners are returned when the scheduler still has pending jobs but can't (because of locking) return one right now.

TAP::Parser::Scheduler::Spinner

A no-op job.

VERSION

Version 3.52

SYNOPSIS

```
use TAP::Parser::Scheduler::Spinner;
```

DESCRIPTION

A no-op job. Returned by `TAP::Parser::Scheduler` as an instruction to the harness to spin (keep executing tests) while the scheduler can't return a real job.

METHODS

Class Methods

new

```
my $job = TAP::Parser::Scheduler::Spinner->new;
```

Ignores any arguments and returns a new `TAP::Parser::Scheduler::Spinner` object.

Instance Methods

is_spinner

Returns true indicating that is a 'spinner' job. Spinners are returned when the scheduler still has pending jobs but can't (because of locking) return one right now.

SEE ALSO

TAP::Parser::Scheduler, *TAP::Parser::Scheduler::Job*

TAP::Parser::Source

a TAP source & meta data about it

VERSION

Version 3.52

SYNOPSIS

```
use TAP::Parser::Source;
my $source = TAP::Parser::Source->new;
$source->raw( \'reference to raw TAP source' )
       ->config( \%config )
       ->merge( $boolean )
       ->switches( \@switches )
       ->test_args( \@args )
       ->assemble_meta;

do { ... } if $source->meta->{is_file};
# see assemble_meta for a full list of data available
```

DESCRIPTION

A TAP *source* is something that produces a stream of TAP for the parser to consume, such as an executable file, a text file, an archive, an IO handle, a database, etc. `TAP::Parser::Sources` encapsulate these *raw* sources, and provide some useful meta data about them. They are used by *TAP::Parser::SourceHandler*s, which do whatever is required to produce & capture a stream of TAP from the *raw* source, and package it up in a *TAP::Parser::Iterator* for the parser to consume.

Unless you're writing a new *TAP::Parser::SourceHandler*, a plugin or subclassing *TAP::Parser*, you probably won't need to use this module directly.

METHODS

Class Methods

new

```
my $source = TAP::Parser::Source->new;
```

Returns a new `TAP::Parser::Source` object.

Instance Methods

raw

```
my $raw = $source->raw;
$source->raw( $some_value );
```

Chaining getter/setter for the raw TAP source. This is a reference, as it may contain large amounts of data (eg: raw TAP).

meta

```
my $meta = $source->meta;
$source->meta({ %some_value });
```

Chaining getter/setter for meta data about the source. This defaults to an empty hashref. See “assemble_meta” for more info.

has_meta

True if the source has meta data.

config

```
my $config = $source->config;
$source->config({ %some_value });
```

Chaining getter/setter for the source’s configuration, if any has been provided by the user. How it’s used is up to you. This defaults to an empty hashref. See “config_for” for more info.

merge

```
my $merge = $source->merge;
$source->config( $bool );
```

Chaining getter/setter for the flag that dictates whether STDOUT and STDERR should be merged (where appropriate). Defaults to undef.

switches

```
my $switches = $source->switches;
$source->config([ @switches ]);
```

Chaining getter/setter for the list of command-line switches that should be passed to the source (where appropriate). Defaults to undef.

test_args

```
my $test_args = $source->test_args;
$source->config([ @test_args ]);
```

Chaining getter/setter for the list of command-line arguments that should be passed to the source (where appropriate). Defaults to undef.

assemble_meta

```
my $meta = $source->assemble_meta;
```

Gathers meta data about the “raw” source, stashes it in “meta” and returns it as a hashref. This is done so that the *TAP::Parser::SourceHandler*s don’t have to repeat common checks. Currently this includes:

```
  is_scalar => $bool,
  is_hash   => $bool,
  is_array  => $bool,

  # for scalars:
  length => $n
  has_newlines => $bool

  # only done if the scalar looks like a filename
  is_file => $bool,
  is_dir  => $bool,
  is_symlink => $bool,
  file => {
      # only done if the scalar looks like a filename
      basename => $string, # including ext
      dir      => $string,
      ext      => $string,
      lc_ext   => $string,
      # system checks
      exists  => $bool,
      stat    => [ ... ], # perldoc -f stat
      empty   => $bool,
      size    => $n,
      text    => $bool,
      binary  => $bool,
      read    => $bool,
      write   => $bool,
      execute => $bool,
      setuid  => $bool,
      setgid  => $bool,
      sticky  => $bool,
      is_file => $bool,
      is_dir  => $bool,
      is_symlink => $bool,
      # only done if the file's a symlink
      lstat      => [ ... ], # perldoc -f lstat
      # only done if the file's a readable text file
      shebang => $first_line,
  }

# for arrays:
size => $n,
```

shebang

Get the shebang line for a script file.

```
my $shebang = TAP::Parser::Source->shebang( $some_script );
```

May be called as a class method

config_for

```
my $config = $source->config_for( $class );
```

Returns "config" for the $class given. Class names may be fully qualified or abbreviated, eg:

```
# these are equivalent
$source->config_for( 'Perl' );
$source->config_for( 'TAP::Parser::SourceHandler::Perl' );
```

If a fully qualified $class is given, its abbreviated version is checked first.

AUTHORS

Steve Purkis.

SEE ALSO

TAP::Object, *TAP::Parser*, *TAP::Parser::IteratorFactory*, *TAP::Parser::SourceHandler*

TAP::Parser::SourceHandler

Base class for different TAP source handlers

VERSION

Version 3.52

SYNOPSIS

```
# abstract class - don't use directly!
# see TAP::Parser::IteratorFactory for general usage

# must be sub-classed for use
package MySourceHandler;
use base 'TAP::Parser::SourceHandler';
sub can_handle    { return $confidence_level }
sub make_iterator { return $iterator }

# see example below for more details
```

DESCRIPTION

This is an abstract base class for *TAP::Parser::Source* handlers / handlers.

A `TAP::Parser::SourceHandler` does whatever is necessary to produce & capture a stream of TAP from the *raw* source, and package it up in a *TAP::Parser::Iterator* for the parser to consume.

`SourceHandlers` must implement the *source detection & handling* interface used by *TAP::Parser::IteratorFactory*. At 2 methods, the interface is pretty simple: "can_handle" and "make_source".

Unless you're writing a new *TAP::Parser::SourceHandler*, a plugin, or subclassing *TAP::Parser*, you probably won't need to use this module directly.

METHODS

Class Methods

`can_handle`

Abstract method.

```
my $vote = $class->can_handle( $source );
```

`$source` is a *TAP::Parser::Source*.

Returns a number between `0` & `1` reflecting how confidently the raw source can be handled. For example, `0` means the source cannot handle it, `0.5` means it may be able to, and `1` means it definitely can. See *"detect_source" in TAP::Parser::IteratorFactory* for details on how this is used.

`make_iterator`

Abstract method.

```
my $iterator = $class->make_iterator( $source );
```

`$source` is a *TAP::Parser::Source*.

Returns a new *TAP::Parser::Iterator* object for use by the *TAP::Parser*. `croaks` on error.

SUBCLASSING

Please see *"SUBCLASSING" in TAP::Parser* for a subclassing overview, and any of the subclasses that ship with this module as an example. What follows is a quick overview.

Start by familiarizing yourself with *TAP::Parser::Source* and *TAP::Parser::IteratorFactory*. *TAP::Parser::SourceHandler::RawTAP* is the easiest sub-class to use as an example.

It's important to point out that if you want your subclass to be automatically used by *TAP::Parser* you'll have to and make sure it gets loaded somehow. If you're using prove you can write an *App::Prove* plugin. If you're using *TAP::Parser* or *TAP::Harness* directly (e.g. through a custom script, *ExtUtils::MakeMaker*, or Module::Build) you can use the

`config` option which will cause *"load_sources" in TAP::Parser::IteratorFactory* to load your subclass).

Don't forget to register your class with *"register_handler" in TAP::Parser::IteratorFactory*.

Example

```
package MySourceHandler;

use strict;

use MySourceHandler; # see TAP::Parser::SourceHandler
use TAP::Parser::IteratorFactory;

use base 'TAP::Parser::SourceHandler';

TAP::Parser::IteratorFactory->register_handler( __PACKAGE__ );

sub can_handle {
    my ( $class, $src ) = @_;
    my $meta   = $src->meta;
    my $config = $src->config_for( $class );

    if ($config->{accept_all}) {
        return 1.0;
    } elsif (my $file = $meta->{file}) {
        return 0.0 unless $file->{exists};
        return 1.0 if $file->{lc_ext} eq '.tap';
        return 0.9 if $file->{shebang} && $file->{shebang} =~ /^#!.
↪+tap/;
        return 0.5 if $file->{text};
        return 0.1 if $file->{binary};
    } elsif ($meta->{scalar}) {
        return 0.8 if $$raw_source_ref =~ /\d\.\.\d/;
        return 0.6 if $meta->{has_newlines};
    } elsif ($meta->{array}) {
        return 0.8 if $meta->{size} < 5;
        return 0.6 if $raw_source_ref->[0] =~ /foo/;
        return 0.5;
    } elsif ($meta->{hash}) {
        return 0.6 if $raw_source_ref->{foo};
        return 0.2;
    }

    return 0;
}

sub make_iterator {
    my ($class, $source) = @_;
    # this is where you manipulate the source and
    # capture the stream of TAP in an iterator
```

(continues on next page)

(continued from previous page)

```
    # either pick a TAP::Parser::Iterator::* or write your own...
    my $iterator = TAP::Parser::Iterator::Array->new([ 'foo', 'bar'␣
↪]);
    return $iterator;
}

1;
```

AUTHORS

TAPx Developers.

Source detection stuff added by Steve Purkis

SEE ALSO

TAP::Object, *TAP::Parser*, *TAP::Parser::Source*, *TAP::Parser::Iterator*, *TAP::Parser::IteratorFactory*, *TAP::Parser::SourceHandler::Executable*, *TAP::Parser::SourceHandler::Perl*, *TAP::Parser::SourceHandler::File*, *TAP::Parser::SourceHandler::Handle*, *TAP::Parser::SourceHandler::RawTAP*

Modules

- `TAP::Parser::SourceHandler::Executable` - Stream output from an executable TAP source
- `TAP::Parser::SourceHandler::File` - Stream TAP from a text file.
- `TAP::Parser::SourceHandler::Handle` - Stream TAP from an IO::Handle or a GLOB.
- `TAP::Parser::SourceHandler::Perl` - Stream TAP from a Perl executable
- `TAP::Parser::SourceHandler::RawTAP` - Stream output from raw TAP in a scalar/array ref.

TAP::Parser::SourceHandler::Executable

Stream output from an executable TAP source

VERSION

Version 3.52

SYNOPSIS

```
use TAP::Parser::Source;
use TAP::Parser::SourceHandler::Executable;

my $source = TAP::Parser::Source->new->raw(['/usr/bin/ruby',
```

(continues on next page)

(continued from previous page)

```
 ↪'mytest.rb']);
$source->assemble_meta;

my $class = 'TAP::Parser::SourceHandler::Executable';
my $vote  = $class->can_handle( $source );
my $iter  = $class->make_iterator( $source );
```

DESCRIPTION

This is an *executable* *TAP::Parser::SourceHandler* - it has 2 jobs:

1. Figure out if the *TAP::Parser::Source* it's given is an executable command ("can_handle").

2. Creates an iterator for executable commands ("make_iterator").

Unless you're writing a plugin or subclassing *TAP::Parser*, you probably won't need to use this module directly.

METHODS

Class Methods

`can_handle`

```
my $vote = $class->can_handle( $source );
```

Only votes if $source looks like an executable file. Casts the following votes:

```
0.9  if it's a hash with an 'exec' key
0.8  if it's a .bat file
0.75 if it's got an execute bit set
```

`make_iterator`

```
my $iterator = $class->make_iterator( $source );
```

Returns a new *TAP::Parser::Iterator::Process* for the source. `$source->raw` must be in one of the following forms:

```
{ exec => [ @exec ] }

[ @exec ]

$file
```

`croaks` on error.

iterator_class

The class of iterator to use, override if you're sub-classing. Defaults to *TAP::Parser::Iterator::Process*.

SUBCLASSING

Please see *"SUBCLASSING" in TAP::Parser* for a subclassing overview.

Example

```
package MyRubySourceHandler;

use strict;

use Carp qw( croak );
use TAP::Parser::SourceHandler::Executable;

use base 'TAP::Parser::SourceHandler::Executable';

# expect $handler->(['mytest.rb', 'cmdline', 'args']);
sub make_iterator {
  my ($self, $source) = @_;
  my @test_args = @{ $source->test_args };
  my $rb_file   = $test_args[0];
  croak("error: Ruby file '$rb_file' not found!") unless (-f $rb_
↪file);
  return $self->SUPER::raw_source(['/usr/bin/ruby', @test_args]);
}
```

SEE ALSO

TAP::Object, *TAP::Parser*, *TAP::Parser::IteratorFactory*, *TAP::Parser::SourceHandler*, *TAP::Parser::SourceHandler::Perl*, *TAP::Parser::SourceHandler::File*, *TAP::Parser::SourceHandler::Handle*, *TAP::Parser::SourceHandler::RawTAP*

TAP::Parser::SourceHandler::File

Stream TAP from a text file.

VERSION

Version 3.52

SYNOPSIS

```
use TAP::Parser::Source;
use TAP::Parser::SourceHandler::File;
```

(continues on next page)

(continued from previous page)

```
my $source = TAP::Parser::Source->new->raw( \'file.tap' );
$source->assemble_meta;

my $class = 'TAP::Parser::SourceHandler::File';
my $vote  = $class->can_handle( $source );
my $iter  = $class->make_iterator( $source );
```

DESCRIPTION

This is a *raw TAP stored in a file* *TAP::Parser::SourceHandler* - it has 2 jobs:

1. Figure out if the *raw* source it's given is a file containing raw TAP output. See *TAP::Parser::IteratorFactory* for more details.

2. Takes raw TAP from the text file given, and converts into an iterator.

Unless you're writing a plugin or subclassing *TAP::Parser*, you probably won't need to use this module directly.

METHODS

Class Methods

can_handle

```
my $vote = $class->can_handle( $source );
```

Only votes if $source looks like a regular file. Casts the following votes:

```
0.9 if it's a .tap file
0.9 if it has an extension matching any given in user config.
```

make_iterator

```
my $iterator = $class->make_iterator( $source );
```

Returns a new *TAP::Parser::Iterator::Stream* for the source. `croaks` on error.

iterator_class

The class of iterator to use, override if you're sub-classing. Defaults to *TAP::Parser::Iterator::Stream*.

CONFIGURATION

```
{
 extensions => [ @case_insensitive_exts_to_match ]
}
```

SUBCLASSING

Please see *"SUBCLASSING" in TAP::Parser* for a subclassing overview.

SEE ALSO

TAP::Object, *TAP::Parser*, *TAP::Parser::SourceHandler*, *TAP::Parser::SourceHandler::Executable*, *TAP::Parser::SourceHandler::Perl*, *TAP::Parser::SourceHandler::Handle*, *TAP::Parser::SourceHandler::RawTAP*

TAP::Parser::SourceHandler::Handle

Stream TAP from an IO::Handle or a GLOB.

VERSION

Version 3.52

SYNOPSIS

```
use TAP::Parser::Source;
use TAP::Parser::SourceHandler::Executable;

my $source = TAP::Parser::Source->new->raw( \*TAP_FILE );
$source->assemble_meta;

my $class = 'TAP::Parser::SourceHandler::Handle';
my $vote  = $class->can_handle( $source );
my $iter  = $class->make_iterator( $source );
```

DESCRIPTION

This is a *raw TAP stored in an IO Handle* *TAP::Parser::SourceHandler* class. It has 2 jobs:

1. Figure out if the *TAP::Parser::Source* it's given is an *IO::Handle* or GLOB containing raw TAP output ("can_handle").

2. Creates an iterator for IO::Handle's & globs ("make_iterator").

Unless you're writing a plugin or subclassing *TAP::Parser*, you probably won't need to use this module directly.

METHODS

Class Methods

can_handle

```
my $vote = $class->can_handle( $source );
```

Casts the following votes:

```
0.9 if $source is an IO::Handle
0.8 if $source is a glob
```

make_iterator

```
my $iterator = $class->make_iterator( $source );
```

Returns a new *TAP::Parser::Iterator::Stream* for the source.

iterator_class

The class of iterator to use, override if you're sub-classing. Defaults to *TAP::Parser::Iterator::Stream*.

SUBCLASSING

Please see *"SUBCLASSING" in TAP::Parser* for a subclassing overview.

SEE ALSO

TAP::Object, *TAP::Parser*, *TAP::Parser::Iterator*, *TAP::Parser::Iterator::Stream*, *TAP::Parser::IteratorFactory*, *TAP::Parser::SourceHandler*, *TAP::Parser::SourceHandler::Executable*, *TAP::Parser::SourceHandler::Perl*, *TAP::Parser::SourceHandler::File*, *TAP::Parser::SourceHandler::RawTAP*

TAP::Parser::SourceHandler::Perl

Stream TAP from a Perl executable

VERSION

Version 3.52

SYNOPSIS

```
use TAP::Parser::Source;
use TAP::Parser::SourceHandler::Perl;

my $source = TAP::Parser::Source->new->raw( \'script.pl' );
$source->assemble_meta;

my $class = 'TAP::Parser::SourceHandler::Perl';
my $vote  = $class->can_handle( $source );
my $iter  = $class->make_iterator( $source );
```

DESCRIPTION

This is a *Perl* *TAP::Parser::SourceHandler* - it has 2 jobs:

1. Figure out if the *TAP::Parser::Source* it's given is actually a Perl script ("can_handle").

2. Creates an iterator for Perl sources ("make_iterator").

Unless you're writing a plugin or subclassing *TAP::Parser*, you probably won't need to use this module directly.

METHODS

Class Methods

can_handle

```
my $vote = $class->can_handle( $source );
```

Only votes if $source looks like a file. Casts the following votes:

```
0.9  if it has a shebang ala "#!...perl"
0.3  if it has any shebang
0.8  if it's a .t file
0.9  if it's a .pl file
0.75 if it's in a 't' directory
0.25 by default (backwards compat)
```

make_iterator

```
my $iterator = $class->make_iterator( $source );
```

Constructs & returns a new *TAP::Parser::Iterator::Process* for the source. Assumes `$source->raw` contains a reference to the perl script. `croaks` if the file could not be found.

The command to run is built as follows:

```
$perl @switches $perl_script @test_args
```

The perl command to use is determined by "get_perl". The command generated is guaranteed to preserve:

```
PERL5LIB
PERL5OPT
Taint Mode, if set in the script's shebang
```

Note: the command generated will *not* respect any shebang line defined in your Perl script. This is only a problem if you have compiled a custom version of Perl or if you want to use a specific version of Perl for one test and a different version for another, for example:

```
#!/path/to/a/custom_perl --some --args
#!/usr/local/perl-5.6/bin/perl -w
```

Currently you need to write a plugin to get around this.

get_taint

Decode any taint switches from a Perl shebang line.

```
# $taint will be 't'
my $taint = TAP::Parser::SourceHandler::Perl->get_taint( '#!/usr/
 ↪bin/perl -t' );

# $untaint will be undefined
my $untaint = TAP::Parser::SourceHandler::Perl->get_taint( '#!/usr/
 ↪bin/perl' );
```

get_perl

Gets the version of Perl currently running the test suite.

SUBCLASSING

Please see *"SUBCLASSING" in TAP::Parser* for a subclassing overview.

Example

```
package MyPerlSourceHandler;

use strict;

use TAP::Parser::SourceHandler::Perl;

use base 'TAP::Parser::SourceHandler::Perl';

# use the version of perl from the shebang line in the test file
sub get_perl {
    my $self = shift;
    if (my $shebang = $self->shebang( $self->{file} )) {
        $shebang =~ /^#!(.*\bperl.*?)(?:(?:\s)|(?:$))/;
        return $1 if $1;
    }
    return $self->SUPER::get_perl(@_);
}
```

SEE ALSO

TAP::Object, *TAP::Parser*, *TAP::Parser::IteratorFactory*, *TAP::Parser::SourceHandler*, *TAP::Parser::SourceHandler::Executable*, *TAP::Parser::SourceHandler::File*, *TAP::Parser::SourceHandler::Handle*, *TAP::Parser::SourceHandler::RawTAP*

TAP::Parser::SourceHandler::RawTAP

Stream output from raw TAP in a scalar/array ref.

VERSION

Version 3.52

SYNOPSIS

```
use TAP::Parser::Source;
use TAP::Parser::SourceHandler::RawTAP;

my $source = TAP::Parser::Source->new->raw( \"1..1\nok 1\n" );
$source->assemble_meta;

my $class = 'TAP::Parser::SourceHandler::RawTAP';
my $vote  = $class->can_handle( $source );
my $iter  = $class->make_iterator( $source );
```

DESCRIPTION

This is a *raw TAP output* *TAP::Parser::SourceHandler* - it has 2 jobs:

1. Figure out if the *TAP::Parser::Source* it's given is raw TAP output ("can_handle").

2. Creates an iterator for raw TAP output ("make_iterator").

Unless you're writing a plugin or subclassing *TAP::Parser*, you probably won't need to use this module directly.

METHODS

Class Methods

can_handle

```
my $vote = $class->can_handle( $source );
```

Only votes if $source is an array, or a scalar with newlines. Casts the following votes:

```
0.9  if it's a scalar with '..' in it
0.7  if it's a scalar with 'ok' in it
0.3  if it's just a scalar with newlines
0.5  if it's an array
```

make_iterator

```
my $iterator = $class->make_iterator( $source );
```

Returns a new *TAP::Parser::Iterator::Array* for the source. `$source->raw` must be an array ref, or a scalar ref.

`croaks` on error.

SUBCLASSING

Please see *"SUBCLASSING" in TAP::Parser* for a subclassing overview.

SEE ALSO

TAP::Object, *TAP::Parser*, *TAP::Parser::IteratorFactory*, *TAP::Parser::SourceHandler*, *TAP::Parser::SourceHandler::Executable*, *TAP::Parser::SourceHandler::Perl*, *TAP::Parser::SourceHandler::File*, *TAP::Parser::SourceHandler::Handle*

TAP::Parser::YAMLish

Namespace containing the following modules:

- `TAP::Parser::YAMLish::Reader` - Read YAMLish data from iterator
- `TAP::Parser::YAMLish::Writer` - Write YAMLish data

TAP::Parser::YAMLish::Reader

Read YAMLish data from iterator

VERSION

Version 3.52

SYNOPSIS

DESCRIPTION

Note that parts of this code were derived from YAML::Tiny with the permission of Adam Kennedy.

METHODS

Class Methods

new

The constructor `new` creates and returns an empty `TAP::Parser::YAMLish::Reader` object.

```
my $reader = TAP::Parser::YAMLish::Reader->new;
```

Instance Methods

read

```
my $got = $reader->read($iterator);
```

Read YAMLish from a *TAP::Parser::Iterator* and return the data structure it represents.

get_raw

```
my $source = $reader->get_source;
```

Return the raw YAMLish source from the most recent `read`.

AUTHOR

Andy Armstrong, andy@hexten.net

Adam Kennedy wrote YAML::Tiny which provided the template and many of the YAML matching regular expressions for this module.

SEE ALSO

YAML::Tiny, *YAML*, YAML::Syck, Config::Tiny, CSS::Tiny, http://use.perl.org/~Alias/journal/29427

COPYRIGHT

Copyright 2007-2011 Andy Armstrong.

Portions copyright 2006-2008 Adam Kennedy.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

The full text of the license can be found in the LICENSE file included with this module.

TAP::Parser::YAMLish::Writer

Write YAMLish data

VERSION

Version 3.52

SYNOPSIS

```
use TAP::Parser::YAMLish::Writer;

my $data = {
    one => 1,
    two => 2,
    three => [ 1, 2, 3 ],
};

my $yw = TAP::Parser::YAMLish::Writer->new;

# Write to an array...
$yw->write( $data, \@some_array );

# ...an open file handle...
$yw->write( $data, $some_file_handle );

# ...a string ...
$yw->write( $data, \$some_string );

# ...or a closure
$yw->write( $data, sub {
    my $line = shift;
    print "$line\n";
} );
```

DESCRIPTION

Encodes a scalar, hash reference or array reference as YAMLish.

METHODS

Class Methods

new

```
my $writer = TAP::Parser::YAMLish::Writer->new;
```

The constructor new creates and returns an empty `TAP::Parser::YAMLish::Writer` object.

Instance Methods

write

```
$writer->write($obj, $output );
```

Encode a scalar, hash reference or array reference as YAML.

```
my $writer = sub {
    my $line = shift;
    print SOMEFILE "$line\n";
};

my $data = {
    one => 1,
    two => 2,
    three => [ 1, 2, 3 ],
};

my $yw = TAP::Parser::YAMLish::Writer->new;
$yw->write( $data, $writer );
```

The `$output` argument may be:

- a reference to a scalar to append YAML to
- the handle of an open file
- a reference to an array into which YAML will be pushed
- a code reference

If you supply a code reference the subroutine will be called once for each line of output with the line as its only argument. Passed lines will have no trailing newline.

AUTHOR

Andy Armstrong, andy@hexten.net

SEE ALSO

YAML::Tiny, *YAML*, YAML::Syck, Config::Tiny, CSS::Tiny, http://use.perl.org/~Alias/journal/29427

COPYRIGHT

Copyright 2007-2011 Andy Armstrong.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

The full text of the license can be found in the LICENSE file included with this module.

5.2.99 Template

Namespace containing the following modules:

- `Template::Stash`

Template::Stash

Namespace containing the following modules:

- `Template::Stash::XS` - The compiled stash of the Template Toolkit: `get` and `set`, which resolve a template variable such as `user.name` or `list.0.size` against the stash's data.

Template::Stash::XS

The compiled stash of the Template Toolkit: `get` and `set`, which resolve a template variable such as `user.name` or `list.0.size` against the stash's data.

A dotted variable is walked one element at a time: a hash key, a list index, an object method, a code reference called with its arguments, or a virtual method. `size`, `max`, `reverse` and `join` on a list, `keys`, `values` and `each` on a hash, and `length` and `defined` on a scalar are compiled; every other virtual method comes from `Template::Stash`'s `%LIST_OPS`, `%HASH_OPS` and `%SCALAR_OPS`.

This is a native implementation of the `Stash.xs` of Template Toolkit 3.106; its `Template/Stash/XS.pm` is used unchanged, and the Template Toolkit itself comes from CPAN. It is the stash the Template Toolkit uses by default.

Synopsis

```
use Template;
use Template::Stash::XS;

my $stash = Template::Stash::XS->new({ user => { name => 'Ann' } });
print $stash->get('user.name'), "\n";          # Ann
$stash->set('user.mail', 'ann@example.org');

my $tt = Template->new({ STASH => $stash });
```

Differences from upstream

- An object method that dies "Can't locate object method" no longer removes a value from the caller's argument list: after the third such method in one dotted variable, perl overwrites the caller's arguments, so `f('A', 'B', 'C', 'D', $stash->get('x.k.k.k.v'))` passes the error text in place of `'D'`.
- `set` called with a root that is not a reference dies "don't know how to assign to [ ].key"; perl reads the root as if it were a reference and crashes for undef, a number or a long string.

Functions

Template Toolkit

get

The value of a template variable: `$ident` is a name, a dotted name such as `'user.name'` (arguments in parentheses are dropped), or a list `[ name => \@args, name =>`

`\@args, ... ]`; `\@args` as a third argument is passed to a plain name that is a code ref or method.

set

Assign `$value` to a template variable named as for `get`, creating the hashes on the way; with a true fourth argument, only when the variable is not already true. Returns the value assigned, or an empty string when nothing was.

get

The value of a template variable: `$ident` is a name, a dotted name such as `'user.name'` (arguments in parentheses are dropped), or a list `[ name => \@args, name => \@args, ... ]`; `\@args` as a third argument is passed to a plain name that is a code ref or method.

Synopsis

```
$stash->get('title');
$stash->get('user.name');
$stash->get([ 'user', 0, 'greet', [ 'Hello' ] ]);
```

Edge cases

- A variable that is not defined is passed to `$stash->undefined`, whose value is returned (Template::Stash's returns an empty string, or dies under `STRICT`).
- A reference other than a list ref as `$ident` dies "Template::Stash::XS: get (arg 2) must be a scalar or listref".
- A true `_DEBUG` in the stash makes a missing variable die "NAME is undefined".

set

Assign `$value` to a template variable named as for `get`, creating the hashes on the way; with a true fourth argument, only when the variable is not already true. Returns the value assigned, or an empty string when nothing was.

Synopsis

```
$stash->set('title', 'Home');
$stash->set('user.mail', 'ann@example.org');   # creates user if␣
↪needed
$stash->set('title', 'Default', 1);            # only if not already␣
↪true
```

Edge cases

- A reference other than a list ref as `$ident` dies "Template::Stash::XS: set (arg 2) must be a scalar or listref".
- Names starting with `_` or `.` are private and are not assigned while `$Template::Stash::PRIVATE` is true.
- The last element on an object calls its method with `$value` as the last argument.

5.2.100 Term

Namespace containing the following modules:

- `Term::ANSIColor` - Color screen output using ANSI escape sequences
- `Term::Cap` - Perl termcap interface
- `Term::Complete` - Perl word completion module
- `Term::ReadKey` - Terminal modes (cooked, no-echo, cbreak, raw), reads that time out or do not block, and the terminal's size, speed and control characters.
- `Term::ReadLine` - Perl interface to various `readline` packages.
- `Term::Table` - Format a header and rows into a table

Term::ANSIColor

Color screen output using ANSI escape sequences

SYNOPSIS

```
use Term::ANSIColor;
print color('bold blue');
print "This text is bold blue.\n";
print color('reset');
print "This text is normal.\n";
print colored("Yellow on magenta.", 'yellow on_magenta'), "\n";
print "This text is normal.\n";
print colored(['yellow on_magenta'], 'Yellow on magenta.', "\n");
print colored(['red on_bright_yellow'], 'Red on bright yellow.', "\n
 ↪");
print colored(['bright_red on_black'], 'Bright red on black.', "\n
 ↪");
print "\n";

# Map escape sequences back to color names.
use Term::ANSIColor 1.04 qw(uncolor);
my @names = uncolor('01;31');
print join(q{ }, @names), "\n";

# Strip all color escape sequences.
use Term::ANSIColor 2.01 qw(colorstrip);
```

(continues on next page)

(continued from previous page)

```
print colorstrip("\e1mThis is bold\e[0m"), "\n";

# Determine whether a color is valid.
use Term::ANSIColor 2.02 qw(colorvalid);
my $valid = colorvalid('blue bold', 'on_magenta');
print "Color string is ", $valid ? "valid\n" : "invalid\n";

# Create new aliases for colors.
use Term::ANSIColor 4.00 qw(coloralias);
coloralias('alert', 'red');
print "Alert is ", coloralias('alert'), "\n";
print colored("This is in red.", 'alert'), "\n";

use Term::ANSIColor qw(:constants);
print BOLD, BLUE, "This text is in bold blue.\n", RESET;

use Term::ANSIColor qw(:constants);
{
    local $Term::ANSIColor::AUTORESET = 1;
    print BOLD BLUE "This text is in bold blue.\n";
    print "This text is normal.\n";
}

use Term::ANSIColor 2.00 qw(:pushpop);
print PUSHCOLOR RED ON_GREEN "This text is red on green.\n";
print PUSHCOLOR BRIGHT_BLUE "This text is bright blue on green.\n";
print RESET BRIGHT_BLUE "This text is just bright blue.\n";
print POPCOLOR "Back to red on green.\n";
print LOCALCOLOR GREEN ON_BLUE "This text is green on blue.\n";
print "This text is red on green.\n";
{
    local $Term::ANSIColor::AUTOLOCAL = 1;
    print ON_BLUE "This text is red on blue.\n";
    print "This text is red on green.\n";
}
print POPCOLOR "Back to whatever we started as.\n";
```

DESCRIPTION

This module has two interfaces, one through color() and colored() and the other through constants. It also offers the utility functions uncolor(), colorstrip(), colorvalid(), and coloralias(), which have to be explicitly imported to be used (see [“SYNOPSIS”).

If you are using Term::ANSIColor in a console command, consider supporting the CLICOLOR standard. See “Supporting CLICOLOR” for more information.

See “COMPATIBILITY” for the versions of Term::ANSIColor that introduced particular features and the versions of Perl that included them.

Supported Colors

Terminal emulators that support color divide into four types: ones that support only eight colors, ones that support sixteen, ones that support 256, and ones that support 24-bit color. This module provides the ANSI escape codes for all of them. These colors are referred to as ANSI colors 0 through 7 (normal), 8 through 15 (16-color), 16 through 255 (256-color), and true color (called direct-color by **xterm**).

Unfortunately, interpretation of colors 0 through 7 often depends on whether the emulator supports eight colors or sixteen colors. Emulators that only support eight colors (such as the Linux console) will display colors 0 through 7 with normal brightness and ignore colors 8 through 15, treating them the same as white. Emulators that support 16 colors, such as gnome-terminal, normally display colors 0 through 7 as dim or darker versions and colors 8 through 15 as normal brightness. On such emulators, the "normal" white (color 7) usually is shown as pale grey, requiring bright white (15) to be used to get a real white color. Bright black usually is a dark grey color, although some terminals display it as pure black. Some sixteen-color terminal emulators also treat normal yellow (color 3) as orange or brown, and bright yellow (color 11) as yellow.

Following the normal convention of sixteen-color emulators, this module provides a pair of attributes for each color. For every normal color (0 through 7), the corresponding bright color (8 through 15) is obtained by prepending the string `bright_` to the normal color name. For example, `red` is color 1 and `bright_red` is color 9. The same applies for background colors: `on_red` is the normal color and `on_bright_red` is the bright color. Capitalize these strings for the constant interface.

There is unfortunately no way to know whether the current emulator supports more than eight colors, which makes the choice of colors difficult. The most conservative choice is to use only the regular colors, which are at least displayed on all emulators. However, they will appear dark in sixteen-color terminal emulators, including most common emulators in UNIX X environments. If you know the display is one of those emulators, you may wish to use the bright variants instead. Even better, offer the user a way to configure the colors for a given application to fit their terminal emulator.

For 256-color emulators, this module additionally provides `ansi0` through `ansi15`, which are the same as colors 0 through 15 in sixteen-color emulators but use the 256-color escape syntax, `grey0` through `grey23` ranging from nearly black to nearly white, and a set of RGB colors. The RGB colors are of the form `rgb_RGB_` where *R*, *G*, and *B* are numbers from 0 to 5 giving the intensity of red, green, and blue. The grey and RGB colors are also available as `ansi16` through `ansi255` if you want simple names for all 256 colors. `on_` variants of all of these colors are also provided. These colors may be ignored completely on non-256-color terminals or may be misinterpreted and produce random behavior. Additional attributes such as blink, italic, or bold may not work with the 256-color palette.

For true color emulators, this module supports attributes of the form `r_NNN_g_NNN_b_NNN_` and `on_r_NNN_g_NNN_b_NNN_` for all values of *NNN* between 0 and 255. These represent foreground and background colors, respectively, with the RGB values given by the *NNN* numbers. These colors may be ignored completely on non-true-color terminals or may be misinterpreted and produce random behavior.

Function Interface

The function interface uses attribute strings to describe the colors and text attributes to assign to text. The recognized non-color attributes are clear, reset, bold, dark, faint, italic, underline, underscore, blink, reverse, and concealed. Clear and reset (reset to default attributes), dark and faint (dim and saturated), and underline and underscore are equivalent, so use whichever is the most intuitive to you.

Note that not all attributes are supported by all terminal types, and some terminals may not support any of these sequences. Dark and faint, italic, blink, and concealed in particular are frequently not implemented.

The recognized normal foreground color attributes (colors 0 to 7) are:

```
black  red  green  yellow  blue  magenta  cyan  white
```

The corresponding bright foreground color attributes (colors 8 to 15) are:

```
bright_black  bright_red      bright_green  bright_yellow
bright_blue   bright_magenta  bright_cyan   bright_white
```

The recognized normal background color attributes (colors 0 to 7) are:

```
on_black  on_red      on_green  on yellow
on_blue   on_magenta  on_cyan   on_white
```

The recognized bright background color attributes (colors 8 to 15) are:

```
on_bright_black  on_bright_red      on_bright_green  on_bright_
 ↪yellow
on_bright_blue   on_bright_magenta  on_bright_cyan   on_bright_white
```

For 256-color terminals, the recognized foreground colors are:

```
ansi0 .. ansi255
grey0 .. grey23
```

plus `rgb_RGB_` for *R*, *G*, and *B* values from 0 to 5, such as `rgb000` or `rgb515`. Similarly, the recognized background colors are:

```
on_ansi0 .. on_ansi255
on_grey0 .. on_grey23
```

plus `on_rgb_RGB_` for *R*, *G*, and *B* values from 0 to 5.

For true color terminals, the recognized foreground colors are `r_RRR_g_GGG_b_BBB_` for *RRR*, *GGG*, and *BBB* values between 0 and 255. Similarly, the recognized background colors are `on_r_RRR_g_GGG_b_BBB_` for *RRR*, *GGG*, and *BBB* values between 0 and 255.

For any of the above listed attributes, case is not significant.

Attributes, once set, last until they are unset (by printing the attribute `clear` or `reset`). Be careful to do this, or otherwise your attribute will last after your script is done running, and people get very annoyed at having their prompt and typing changed to weird colors.

- color(ATTR[, ATTR ...])

  color() takes any number of strings as arguments and considers them to be space-separated lists of attributes. It then forms and returns the escape sequence to set those attributes. It doesn't print it out, just returns it, so you'll have to print it yourself if you want to. This is so that you can save it as a string, pass it to something else, send it to a file handle, or do anything else with it that you might care to. color() throws an exception if given an invalid attribute.

- colored(STRING, ATTR[, ATTR ...])

- colored(ATTR-REF, STRING[, STRING...])

  As an aid in resetting colors, colored() takes a scalar as the first argument and any number of attribute strings as the second argument and returns the scalar wrapped in escape codes so that the attributes will be set as requested before the string and reset to normal after the string. Alternately, you can pass a reference to an array as the first argument, and then the contents of that array will be taken as attributes and color codes and the remainder of the arguments as text to colorize.

  Normally, colored() just puts attribute codes at the beginning and end of the string, but if you set $Term::ANSIColor::EACHLINE to some string, that string will be considered the line delimiter and the attribute will be set at the beginning of each line of the passed string and reset at the end of each line. This is often desirable if the output contains newlines and you're using background colors, since a background color that persists across a newline is often interpreted by the terminal as providing the default background color for the next line. Programs like pagers can also be confused by attributes that span lines. Normally you'll want to set $Term::ANSIColor::EACHLINE to `"\n"` to use this feature.

  Particularly consider setting $Term::ANSIColor::EACHLINE if you are interleaving output to standard output and standard error and you aren't flushing standard output (via autoflush() or setting `$|`). If you don't, the code to reset the color may unexpectedly sit in the standard output buffer rather than going to the display, causing standard error output to appear in the wrong color.

- uncolor(ESCAPE)

  uncolor() performs the opposite translation as color(), turning escape sequences into a list of strings corresponding to the attributes being set by those sequences. uncolor() will never return `ansi16` through `ansi255`, instead preferring the `grey` and `rgb` names (and likewise for `on_ansi16` through `on_ansi255`).

- colorstrip(STRING[, STRING ...])

  colorstrip() removes all color escape sequences from the provided strings, returning the modified strings separately in array context or joined together in scalar context. Its arguments are not modified.

- colorvalid(ATTR[, ATTR ...])

  colorvalid() takes attribute strings the same as color() and returns true if all attributes are known and false otherwise.

- coloralias(ALIAS[, ATTR ...])

  If ATTR is specified, it is interpreted as a list of space-separated strings naming attributes or existing aliases. In this case, coloralias() sets up an alias of ALIAS

for the set of attributes given by ATTR. From that point forward, ALIAS can be passed into color(), colored(), and colorvalid() and will have the same meaning as the sequence of attributes given in ATTR. One possible use of this facility is to give more meaningful names to the 256-color RGB colors. Only ASCII alphanumerics, ., _, and - are allowed in alias names.

If ATTR includes aliases, those aliases will be expanded at definition time and their values will be used to define the new alias. This means that if you define an alias A in terms of another alias B, and then later redefine alias B, the value of alias A will not change.

If ATTR is not specified, coloralias() returns the standard attribute or attributes to which ALIAS is aliased, if any, or undef if ALIAS does not exist. If it is aliased to multiple attributes, the return value will be a single string and the attributes will be separated by spaces.

This is the same facility used by the ANSI_COLORS_ALIASES environment variable (see "ENVIRONMENT" below) but can be used at runtime, not just when the module is loaded.

Later invocations of coloralias() with the same ALIAS will override earlier aliases. There is no way to remove an alias.

Aliases have no effect on the return value of uncolor().

WARNING: Aliases are global and affect all callers in the same process. There is no way to set an alias limited to a particular block of code or a particular object.

Constant Interface

Alternately, if you import `:constants`, you can use the following constants directly:

```
CLEAR           RESET             BOLD            DARK
FAINT           ITALIC            UNDERLINE       UNDERSCORE
BLINK           REVERSE           CONCEALED

BLACK           RED               GREEN           YELLOW
BLUE            MAGENTA           CYAN            WHITE
BRIGHT_BLACK    BRIGHT_RED        BRIGHT_GREEN    BRIGHT_YELLOW
BRIGHT_BLUE     BRIGHT_MAGENTA    BRIGHT_CYAN     BRIGHT_WHITE

ON_BLACK        ON_RED            ON_GREEN        ON_YELLOW
ON_BLUE         ON_MAGENTA        ON_CYAN         ON_WHITE
ON_BRIGHT_BLACK ON_BRIGHT_RED     ON_BRIGHT_GREEN ON_BRIGHT_YELLOW
ON_BRIGHT_BLUE  ON_BRIGHT_MAGENTA ON_BRIGHT_CYAN  ON_BRIGHT_WHITE
```

These are the same as color('attribute') and can be used if you prefer typing:

```
print BOLD BLUE ON_WHITE "Text", RESET, "\n";
```

to

```
print colored ("Text", 'bold blue on_white'), "\n";
```

(Note that the newline is kept separate to avoid confusing the terminal as described above since a background color is being used.)

If you import `:constants256`, you can use the following constants directly:

```
ANSI0 .. ANSI255
GREY0 .. GREY23

RGBXYZ (for X, Y, and Z values from 0 to 5, like RGB000 or RGB515)

ON_ANSI0 .. ON_ANSI255
ON_GREY0 .. ON_GREY23

ON_RGBXYZ (for X, Y, and Z values from 0 to 5)
```

Note that `:constants256` does not include the other constants, so if you want to mix both, you need to include `:constants` as well. You may want to explicitly import at least `RESET`, as in:

```
use Term::ANSIColor 4.00 qw(RESET :constants256);
```

True color and aliases are not supported by the constant interface.

When using the constants, if you don't want to have to remember to add the `, RESET` at the end of each print line, you can set $Term::ANSIColor::AUTORESET to a true value. Then, the display mode will automatically be reset if there is no comma after the constant. In other words, with that variable set:

```
print BOLD BLUE "Text\n";
```

will reset the display mode afterward, whereas:

```
print BOLD, BLUE, "Text\n";
```

will not. If you are using background colors, you will probably want to either use say() (in newer versions of Perl) or print the newline with a separate print statement to avoid confusing the terminal.

If $Term::ANSIColor::AUTOLOCAL is set (see below), it takes precedence over $Term::ANSIColor::AUTORESET, and the latter is ignored.

The subroutine interface has the advantage over the constants interface in that only two subroutines are exported into your namespace, versus thirty-eight in the constants interface, and aliases and true color attributes are supported. On the flip side, the constants interface has the advantage of better compile time error checking, since misspelled names of colors or attributes in calls to color() and colored() won't be caught until runtime whereas misspelled names of constants will be caught at compile time. So, pollute your namespace with almost two dozen subroutines that you may not even use that often, or risk a silly bug by mistyping an attribute. Your choice, TMTOWTDI after all.

The Color Stack

You can import :pushpop and maintain a stack of colors using PUSHCOLOR, POPCOLOR, and LOCALCOLOR. PUSHCOLOR takes the attribute string that starts its argument and pushes it onto a stack of attributes. POPCOLOR removes the top of the stack and restores the previous attributes set by the argument of a prior PUSHCOLOR. LOCALCOLOR surrounds its argument in a PUSHCOLOR and POPCOLOR so that the color resets afterward.

If $Term::ANSIColor::AUTOLOCAL is set, each sequence of color constants will be implicitly preceded by LOCALCOLOR. In other words, the following:

```
{
    local $Term::ANSIColor::AUTOLOCAL = 1;
    print BLUE "Text\n";
}
```

is equivalent to:

```
print LOCALCOLOR BLUE "Text\n";
```

If $Term::ANSIColor::AUTOLOCAL is set, it takes precedence over $Term::ANSIColor::AUTORESET, and the latter is ignored.

When using PUSHCOLOR, POPCOLOR, and LOCALCOLOR, it's particularly important to not put commas between the constants.

```
print PUSHCOLOR BLUE "Text\n";
```

will correctly push BLUE onto the top of the stack.

```
print PUSHCOLOR, BLUE, "Text\n";    # wrong!
```

will not, and a subsequent pop won't restore the correct attributes. PUSHCOLOR pushes the attributes set by its argument, which is normally a string of color constants. It can't ask the terminal what the current attributes are.

Supporting CLICOLOR

https://bixense.com/clicolors/ proposes a standard for enabling and disabling color output from console commands using two environment variables, CLICOLOR and CLICOLOR_FORCE. Term::ANSIColor cannot automatically support this standard, since the correct action depends on where the output is going and Term::ANSIColor may be used in a context where colors should always be generated even if CLICOLOR is set in the environment. But you can use the supported environment variable ANSI_COLORS_DISABLED to implement CLICOLOR in your own programs with code like this:

```
if (exists($ENV{CLICOLOR}) && $ENV{CLICOLOR} == 0) {
    if (!$ENV{CLICOLOR_FORCE}) {
        $ENV{ANSI_COLORS_DISABLED} = 1;
    }
}
```

If you are using the constant interface, be sure to include this code before you use any

color constants (such as at the very top of your script), since this environment variable is only honored the first time a color constant is seen.

Be aware that this will export ANSI_COLORS_DISABLED to any child processes of your program as well.

DIAGNOSTICS

- Bad color mapping %s

  (W) The specified color mapping from ANSI_COLORS_ALIASES is not valid and could not be parsed. It was ignored.

- Bad escape sequence %s

  (F) You passed an invalid ANSI escape sequence to uncolor().

- Bareword "%s" not allowed while "strict subs" in use

  (F) You probably mistyped a constant color name such as:

  ```
  $Foobar = FOOBAR . "This line should be blue\n";
  ```

  or:

  ```
  @Foobar = FOOBAR, "This line should be blue\n";
  ```

  This will only show up under use strict (another good reason to run under use strict).

- Cannot alias standard color %s

  (F) The alias name passed to coloralias() matches a standard color name. Standard color names cannot be aliased.

- Cannot alias standard color %s in %s

  (W) The same, but in ANSI_COLORS_ALIASES. The color mapping was ignored.

- Invalid alias name %s

  (F) You passed an invalid alias name to coloralias(). Alias names must consist only of alphanumerics, ., -, and _.

- Invalid alias name %s in %s

  (W) You specified an invalid alias name on the left hand of the equal sign in a color mapping in ANSI_COLORS_ALIASES. The color mapping was ignored.

- Invalid attribute name %s

  (F) You passed an invalid attribute name to color(), colored(), or coloralias().

- Invalid attribute name %s in %s

  (W) You specified an invalid attribute name on the right hand of the equal sign in a color mapping in ANSI_COLORS_ALIASES. The color mapping was ignored.

- Name "%s" used only once: possible typo

  (W) You probably mistyped a constant color name such as:

```
print FOOBAR "This text is color FOOBAR\n";
```

It's probably better to always use commas after constant names in order to force the next error.

- No comma allowed after filehandle

  (F) You probably mistyped a constant color name such as:

```
print FOOBAR, "This text is color FOOBAR\n";
```

  Generating this fatal compile error is one of the main advantages of using the constants interface, since you'll immediately know if you mistype a color name.

- No name for escape sequence %s

  (F) The ANSI escape sequence passed to uncolor() contains escapes which aren't recognized and can't be translated to names.

ENVIRONMENT

- ANSI_COLORS_ALIASES

  This environment variable allows the user to specify custom color aliases that will be understood by color(), colored(), and colorvalid(). None of the other functions will be affected, and no new color constants will be created. The custom colors are aliases for existing color names; no new escape sequences can be introduced. Only alphanumerics, `.`, `_`, and `-` are allowed in alias names.

  The format is:

```
ANSI_COLORS_ALIASES='newcolor1=oldcolor1,newcolor2=oldcolor2'
```

  Whitespace is ignored. The alias value can be a single attribute or a space-separated list of attributes.

  For example the Solarized colors can be mapped with:

```
ANSI_COLORS_ALIASES='\
    base00=bright_yellow, on_base00=on_bright_yellow,\
    base01=bright_green,  on_base01=on_bright_green, \
    base02=black,         on_base02=on_black,        \
    base03=bright_black,  on_base03=on_bright_black, \
    base0=bright_blue,    on_base0=on_bright_blue,   \
    base1=bright_cyan,    on_base1=on_bright_cyan,   \
    base2=white,          on_base2=on_white,         \
    base3=bright_white,   on_base3=on_bright_white,  \
    orange=bright_red,    on_orange=on_bright_red,   \
    violet=bright_magenta,on_violet=on_bright_magenta'
```

  This environment variable is read and applied when the Term::ANSIColor module is loaded and is then subsequently ignored. Changes to ANSI_COLORS_ALIASES after the module is loaded will have no effect. See coloralias() for an equivalent facility that can be used at runtime.

- ANSI_COLORS_DISABLED

  If this environment variable is set to a true value, all of the functions defined by this module (color(), colored(), and all of the constants) will not output any escape sequences and instead will just return the empty string or pass through the original text as appropriate. This is intended to support easy use of scripts using this module on platforms that don't support ANSI escape sequences.

- NO_COLOR

  If this environment variable is set to any value, it suppresses generation of escape sequences the same as if ANSI_COLORS_DISABLED is set to a true value. This implements the https://no-color.org/ informal standard. Programs that want to enable color despite NO_COLOR being set will need to unset that environment variable before any constant or function provided by this module is used.

COMPATIBILITY

Term::ANSIColor was first included with Perl in Perl 5.6.0.

The uncolor() function and support for ANSI_COLORS_DISABLED were added in Term::ANSIColor 1.04, included in Perl 5.8.0.

Support for dark was added in Term::ANSIColor 1.08, included in Perl 5.8.4.

The color stack, including the `:pushpop` import tag, PUSHCOLOR, POPCOLOR, LOCALCOLOR, and the $Term::ANSIColor::AUTOLOCAL variable, was added in Term::ANSIColor 2.00, included in Perl 5.10.1.

colorstrip() was added in Term::ANSIColor 2.01 and colorvalid() was added in Term::ANSIColor 2.02, both included in Perl 5.11.0.

Support for colors 8 through 15 (the `bright_` variants) was added in Term::ANSIColor 3.00, included in Perl 5.13.3.

Support for italic was added in Term::ANSIColor 3.02, included in Perl 5.17.1.

Support for colors 16 through 256 (the `ansi`, `rgb`, and `grey` colors), the `:constants256` import tag, the coloralias() function, and support for the ANSI_COLORS_ALIASES environment variable were added in Term::ANSIColor 4.00, included in Perl 5.17.8.

$Term::ANSIColor::AUTOLOCAL was changed to take precedence over $Term::ANSIColor::AUTORESET, rather than the other way around, in Term::ANSIColor 4.00, included in Perl 5.17.8.

`ansi16` through `ansi255`, as aliases for the `rgb` and `grey` colors, and the corresponding `on_ansi` names and `ANSI` and `ON_ANSI` constants were added in Term::ANSIColor 4.06, included in Perl 5.25.7.

Support for true color (the `rNNNgNNNbNNN` and `on_rNNNgNNNbNNN` attributes), defining aliases in terms of other aliases, and aliases mapping to multiple attributes instead of only a single attribute was added in Term::ANSIColor 5.00.

Support for NO_COLOR was added in Term::ANSIColor 5.01.

RESTRICTIONS

Both colored() and many uses of the color constants will add the reset escape sequence after a newline. If a program mixes colored output to standard output with output to standard error, this can result in the standard error text having the wrong color because the reset escape sequence hasn't yet been flushed to the display (since standard output to a terminal is line-buffered by default). To avoid this, either set autoflush() on STDOUT or set $Term::ANSIColor::EACHLINE to "\n".

It would be nice if one could leave off the commas around the constants entirely and just say:

```
print BOLD BLUE ON_WHITE "Text\n" RESET;
```

but the syntax of Perl doesn't allow this. You need a comma after the string. (Of course, you may consider it a bug that commas between all the constants aren't required, in which case you may feel free to insert commas unless you're using $Term::ANSIColor::AUTORESET or PUSHCOLOR/POPCOLOR.)

For easier debugging, you may prefer to always use the commas when not setting $Term::ANSIColor::AUTORESET or PUSHCOLOR/POPCOLOR so that you'll get a fatal compile error rather than a warning.

It's not possible to use this module to embed formatting and color attributes using Perl formats. They replace the escape character with a space (as documented in perlform(1)), resulting in garbled output from the unrecognized attribute. Even if there were a way around that problem, the format doesn't know that the non-printing escape sequence is zero-length and would incorrectly format the output. For formatted output using color or other attributes, either use sprintf() instead or use formline() and then add the color or other attributes after formatting and before output.

NOTES

The codes generated by this module are standard terminal control codes, complying with ECMA-048 and ISO 6429 (generally referred to as "ANSI color" for the color codes). The non-color control codes (bold, dark, italic, underline, and reverse) are part of the earlier ANSI X3.64 standard for control sequences for video terminals and peripherals.

Note that not all displays are ISO 6429-compliant, or even X3.64-compliant (or are even attempting to be so). This module will not work as expected on displays that do not honor these escape sequences, such as cmd.exe, 4nt.exe, and command.com under either Windows NT or Windows 2000. They may just be ignored, or they may display as an ESC character followed by some apparent garbage.

Jean Delvare provided the following table of different common terminal emulators and their support for the various attributes and others have helped me flesh it out:

```
            clear    bold     faint   under    blink   reverse␣
↪conceal
------------------------------------------------------------------
↪----
xterm         yes      yes      no      yes      yes      yes     ␣
↪yes
```

(continues on next page)

(continued from previous page)

```
linux          yes      yes      yes    bold       yes      yes      ␣
 ↪no
rxvt           yes      yes      no      yes   bold/black   yes      ␣
 ↪no
dtterm         yes      yes      yes     yes     reverse    yes      ␣
 ↪yes
teraterm       yes    reverse    no      yes     rev/red    yes      ␣
 ↪no
aixterm       kinda   normal     no      yes       no       yes      ␣
 ↪yes
PuTTY          yes     color     no      yes       no       yes      ␣
 ↪no
Windows        yes      no       no      no        no       yes      ␣
 ↪no
Cygwin SSH     yes      yes      no     color     color    color     ␣
 ↪yes
Terminal.app   yes      yes      no      yes       yes      yes      ␣
 ↪yes
```

Windows is Windows telnet, Cygwin SSH is the OpenSSH implementation under Cygwin on Windows NT, and Mac Terminal is the Terminal application in Mac OS X. Where the entry is other than yes or no, that emulator displays the given attribute as something else instead. Note that on an aixterm, clear doesn't reset colors; you have to explicitly set the colors back to what you want. More entries in this table are welcome.

Support for code 3 (italic) is rare and therefore not mentioned in that table. It is not believed to be fully supported by any of the terminals listed, although it's displayed as green in the Linux console, but it is reportedly supported by urxvt.

Note that codes 6 (rapid blink) and 9 (strike-through) are specified in ANSI X3.64 and ECMA-048 but are not commonly supported by most displays and emulators and therefore aren't supported by this module. ECMA-048 also specifies a large number of other attributes, including a sequence of attributes for font changes, Fraktur characters, double-underlining, framing, circling, and overlining. As none of these attributes are widely supported or useful, they also aren't currently supported by this module.

Most modern X terminal emulators support 256 colors. Known to not support those colors are aterm, rxvt, Terminal.app, and TTY/VC.

For information on true color support in various terminal emulators, see True Colour support.

AUTHORS

Original idea (using constants) by Zenin, reimplemented using subs by Russ Allbery rra@cpan.org, and then combined with the original idea by Russ with input from Zenin. 256-color support is based on work by Kurt Starsinic. Russ Allbery now maintains this module.

PUSHCOLOR, POPCOLOR, and LOCALCOLOR were contributed by openmethods.com voice solutions.

COPYRIGHT AND LICENSE

Copyright 1996-1998, 2000-2002, 2005-2006, 2008-2018, 2020 Russ Allbery rra@cpan.org

Copyright 1996 Zenin

Copyright 2012 Kurt Starsinic kstarsinic@gmail.com

This program is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

SEE ALSO

The CPAN module Term::ExtendedColor provides a different and more comprehensive interface for 256-color emulators that may be more convenient. The CPAN module Win32::Console::ANSI provides ANSI color (and other escape sequence) support in the Win32 Console environment. The CPAN module Term::Chrome provides a different interface using objects and operator overloading.

ECMA-048 is available on-line (at least at the time of this writing) at https://www.ecma-international.org/publications/standards/Ecma-048.htm.

ISO 6429 is available from ISO for a charge; the author of this module does not own a copy of it. Since the source material for ISO 6429 was ECMA-048 and the latter is available for free, there seems little reason to obtain the ISO standard.

The 256-color control sequences are documented at https://invisible-island.net/xterm/ctlseqs/ctlseqs.html (search for 256-color).

Information about true color support in various terminal emulators and test programs you can run to check the true color support in your terminal emulator are available at https://gist.github.com/XVilka/8346728.

CLICOLORS and NO_COLOR are useful standards to be aware of, and ideally follow, for any application using color. Term::ANSIColor complies with the latter.

The current version of this module is always available from its web site at https://www.eyrie.org/~eagle/software/ansicolor/. It is also part of the Perl core distribution as of 5.6.0.

Term::Cap

Perl termcap interface

SYNOPSIS

```
require Term::Cap;
$terminal = Term::Cap->Tgetent({ TERM => undef, OSPEED => $ospeed }
 ↪);
$terminal->Trequire(qw/ce ku kd/);
$terminal->Tgoto('cm', $col, $row, $FH);
$terminal->Tputs('dl', $count, $FH);
$terminal->Tpad($string, $count, $FH);
```

DESCRIPTION

These are low-level functions to extract and use capabilities from a terminal capability (termcap) database.

More information on the terminal capabilities will be found in the termcap manpage on most Unix-like systems.

METHODS

The output strings for **Tputs** are cached for counts of 1 for performance. **Tgoto** and **Tpad** do not cache. `$self->{_xx}` is the raw termcap data and `$self->{xx}` is the cached version.

```
print $terminal->Tpad($self->{_xx}, 1);
```

Tgoto, **Tputs**, and **Tpad** return the string and will also output the string to $FH if specified.

- **Tgetent**

  Returns a blessed object reference which the user can then use to send the control strings to the terminal using **Tputs** and **Tgoto**.

  The function extracts the entry of the specified terminal type *TERM* (defaults to the environment variable *TERM*) from the database.

  It will look in the environment for a *TERMCAP* variable. If found, and the value does not begin with a slash, and the terminal type name is the same as the environment string *TERM*, the *TERMCAP* string is used instead of reading a termcap file. If it does begin with a slash, the string is used as a path name of the termcap file to search. If *TERMCAP* does not begin with a slash and name is different from *TERM*, **Tgetent** searches the files `$HOME/.termcap`, `/etc/termcap`, and `/usr/share/misc/termcap`, in that order, unless the environment variable *TERMPATH* exists, in which case it specifies a list of file pathnames (separated by spaces or colons) to be searched **instead**. Whenever multiple files are searched and a tc field occurs in the requested entry, the entry it names must be found in the same file or one of the succeeding files. If there is a `:tc=...:` in the *TERMCAP* environment variable string it will continue the search in the files as above.

  The extracted termcap entry is available in the object as `$self->{TERMCAP}`.

  It takes a hash reference as an argument with two optional keys:

  – OSPEED

    The terminal output bit rate (often mistakenly called the baud rate) for this terminal - if not set a warning will be generated and it will be defaulted to 9600. *OSPEED* can be specified as either a POSIX termios/SYSV termio speeds (where 9600 equals 9600) or an old DSD-style speed (where 13 equals 9600).

  – TERM

    The terminal type whose termcap entry will be used - if not supplied it will default to $ENV{TERM}: if that is not set then **Tgetent** will croak.

  It calls `croak` on failure.

- **Tpad**

  Outputs a literal string with appropriate padding for the current terminal.

  It takes three arguments:

  - **$string**

    The literal string to be output. If it starts with a number and an optional '*' then the padding will be increased by an amount relative to this number, if the '*' is present then this amount will be multiplied by $cnt. This part of $string is removed before output/

  - **$cnt**

    Will be used to modify the padding applied to string as described above.

  - **$FH**

    An optional filehandle (or IO::Handle) that output will be printed to.

  The padded $string is returned.

- **Tputs**

  Output the string for the given capability padded as appropriate without any parameter substitution.

  It takes three arguments:

  - **$cap**

    The capability whose string is to be output.

  - **$cnt**

    A count passed to Tpad to modify the padding applied to the output string. If $cnt is zero or one then the resulting string will be cached.

  - **$FH**

    An optional filehandle (or IO::Handle) that output will be printed to.

  The appropriate string for the capability will be returned.

- **Tgoto**

  Tgoto decodes a cursor addressing string with the given parameters.

  There are four arguments:

  - **$cap**

    The name of the capability to be output.

  - **$col**

    The first value to be substituted in the output string (usually the column in a cursor addressing capability)

  - **$row**

    The second value to be substituted in the output string (usually the row in cursor addressing capabilities)

– **$FH**

An optional filehandle (or IO::Handle) to which the output string will be printed.

Substitutions are made with $col and $row in the output string with the following sprintf() line formats:

```
%%   output `%'
%d   output value as in printf %d
%2   output value as in printf %2d
%3   output value as in printf %3d
%.   output value as in printf %c
%+x  add x to value, then do %.

%>xy if value > x then add y, no output
%r   reverse order of two parameters, no output
%i   increment by one, no output
%B   BCD (16*(value/10)) + (value%10), no output

%n   exclusive-or all parameters with 0140 (Datamedia 2500)
%D   Reverse coding (value - 2*(value%16)), no output (Delta
 ↪Data)
```

The output string will be returned.

- **Trequire**

Takes a list of capabilities as an argument and will croak if one is not found.

EXAMPLES

```
use Term::Cap;

# Get terminal output speed
require POSIX;
my $termios = POSIX::Termios->new;
$termios->getattr;
my $ospeed = $termios->getospeed;

# Old-style ioctl code to get ospeed:
#     require 'ioctl.pl';
#     ioctl(TTY,$TIOCGETP,$sgtty);
#     ($ispeed,$ospeed) = unpack('cc',$sgtty);

# allocate and initialize a terminal structure
my $terminal = Term::Cap->Tgetent({ TERM => undef, OSPEED => $ospeed
 ↪});

# require certain capabilities to be available
$terminal->Trequire(qw/ce ku kd/);
```

(continues on next page)

(continued from previous page)

```
# Output Routines, if $FH is undefined these just return the string

# Tgoto does the % expansion stuff with the given args
$terminal->Tgoto('cm', $col, $row, $FH);

# Tputs doesn't do any % expansion.
$terminal->Tputs('dl', $count = 1, $FH);
```

COPYRIGHT AND LICENSE

Copyright 1995-2015 (c) perl5 porters.

This software is free software and can be modified and distributed under the same terms as Perl itself.

Please see the file README in the Perl source distribution for details of the Perl license.

AUTHOR

This module is part of the core Perl distribution and is also maintained for CPAN by Jonathan Stowe jns@gellyfish.co.uk.

The code is hosted on Github: https://github.com/jonathanstowe/Term-Cap please feel free to fork, submit patches etc, etc there.

SEE ALSO

termcap(5)

Term::Complete

Perl word completion module

SYNOPSIS

```
$input = Complete('prompt_string', \@completion_list);
$input = Complete('prompt_string', @completion_list);
```

DESCRIPTION

This routine provides word completion on the list of words in the array (or array ref).

The tty driver is put into raw mode and restored using an operating system specific command, in UNIX-like environments `stty`.

The following command characters are defined:

- <tab>

  Attempts word completion. Cannot be changed.

- ^D

  Prints completion list. Defined by *$Term::Complete::complete*.

- ^U

  Erases the current input. Defined by *$Term::Complete::kill*.

- <del>, <bs>

  Erases one character. Defined by *$Term::Complete::erase1* and *$Term::Complete::erase2*.

DIAGNOSTICS

Bell sounds when word completion fails.

BUGS

The completion character <tab> cannot be changed.

AUTHOR

Wayne Thompson

Term::ReadKey

Terminal modes (cooked, no-echo, cbreak, raw), reads that time out or do not block, and the terminal's size, speed and control characters.

Synopsis

```
use Term::ReadKey;

ReadMode 'cbreak';                  # one key at a time, no echo
my $key = ReadKey(0.5);             # wait half a second at most
ReadMode 'restore';

my ($cols, $rows) = GetTerminalSize();
my %cc = GetControlChars();         # INTERRUPT => "\cC", ...
```

How it works

`ReadMode` changes the handle's termios settings, saving the ones it found on first use; mode 0 (`restore`) puts those back and forgets them. `ReadKey` and `ReadLine` with a positive timeout wait with `select()`, and with a negative one switch the handle to non-blocking for the one read. `GetTerminalSize` asks the terminal (TIOCGWINSZ), then `$COLUMNS`/`$LINES`, then `resize` and `stty size`.

This is a native implementation of TermReadKey 2.38's `ReadKey.xs` and the `cchars.h` its genchars.pl writes, as they build on Linux (termios, `O_NDELAY` and `select()`, no `poll()`); its `Term/ReadKey.pm`, which ReadKey.pm.PL writes, is used unchanged.

Differences from upstream

Three, each where perl 5.44 with ReadKey.xs aborts or returns memory it never wrote (t/81-xs-native/Term-ReadKey/150-or-better.t):

- A timed `ReadKey`/`ReadLine` on a closed handle reads nothing, as an untimed one does; perl5 aborts ("bit out of range 0 - FD_SETSIZE on fd_set"). One on a descriptor of 1024 or above waits as on any other; perl5 aborts the same way.
- `GetSpeed` on a handle that is not a terminal returns (0, 0); perl5 returns two numbers from uninitialised memory, different each time.
- `SetTerminalSize` with a width or height of ten characters or more (1000000000, -100000000) sets `COLUMNS`/`LINES` for child processes as for any other; perl5 aborts ("buffer overflow detected").

Functions

Terminal modes

`selectfile`

Wait until a handle has input, for at most `delay` seconds.

`SetReadMode`

Put a terminal handle into one of the modes 0 to 5.

`setnodelay`

Switch a handle's descriptor to non-blocking reads (`mode` true) or back.

`pollfile`

The `poll()` wait of builds that have it; this one dies "pollfile is not supported on this architecture".

`Win32PeekChar`

The Windows console read; here it dies "Win32PeekChar is not supported on this architecture".

Terminal properties

`blockoptions`

The ways this build can read without blocking, as bits: 1 non-blocking descriptors, 2 `poll()`, 4 `select()`, 8 the Windows console. Here 5.

termoptions

The terminal interface this build drives: 1 termios, 2 termio, 3 sgtty, 4 the `stty` program, 5 the Windows console. Here 1.

termsizeoptions

The ways this build can ask a terminal its size, as bits: 1 OS/2's VIO, 2 TIOCGWINSZ, 4 TIOCGSIZE, 8 the Windows console. Here 2.

GetTermSizeWin32

The Windows console's size; here it dies "TermSizeWin32 is not implemented on this architecture".

GetTermSizeVIO

OS/2's screen size; here it dies "TermSizeVIO is not implemented on this architecture".

GetTermSizeGWINSZ

A terminal's size from the TIOCGWINSZ ioctl.

GetTermSizeGSIZE

The TIOCGSIZE size of systems without TIOCGWINSZ; here it dies "TermSizeGSIZE is not implemented on this architecture".

SetTerminalSize

Tell a terminal its size.

GetSpeed

A terminal's input and output speeds in baud.

GetControlChars

A terminal's control characters, as name/character pairs.

SetControlChars

Change a terminal's control characters.

selectfile

Wait until a handle has input, for at most `delay` seconds.

Synopsis

```
my $ready = Term::ReadKey::selectfile(\*STDIN, 0.25);
```

What you get back

1 when the handle's buffer already holds input, -1 when `select()` says the descriptor is readable (or fails), 0 when the time ran out. `ReadKey` and `ReadLine` call it for a positive timeout.

SetReadMode

Put a terminal handle into one of the modes 0 to 5.

Synopsis

```
Term::ReadKey::SetReadMode(3, \*STDIN);   # ReadMode 'cbreak'
```

How it works

The numeric half of `ReadMode`, which maps the names (`restore`, `normal`, `noecho`, `cbreak`, `raw`, `ultra-raw`) to 0 to 5. The first change of a descriptor saves its settings; mode 0 restores them and forgets them, so the next change saves afresh. Any other number dies "ReadMode N is not implemented on this architecture.".

setnodelay

Switch a handle's descriptor to non-blocking reads (`mode` true) or back.

Synopsis

```
Term::ReadKey::setnodelay(\*STDIN, 1);
my $c = getc STDIN;                    # undef if nothing is waiting
Term::ReadKey::setnodelay(\*STDIN, 0);
```

What you get back

Always 0. `ReadKey` and `ReadLine` call it around a read with a negative timeout.

GetTermSizeGWINSZ

A terminal's size from the TIOCGWINSZ ioctl.

Synopsis

```
my ($cols, $rows, $xpix, $ypix) = Term::ReadKey::GetTermSizeGWINSZ(\
 ↪*STDOUT);
```

What you get back

Width and height in characters and in pixels (0 where the terminal does not say), or an empty list when the handle is not a terminal. `GetTerminalSize` calls it first and falls back from an empty list.

SetTerminalSize

Tell a terminal its size.

Synopsis

```
SetTerminalSize(132, 50, 0, 0, \*STDOUT);
```

What you get back

0. The size is the terminal's record, read by the programs on it (a change sends them SIGWINCH); the window does not change. `COLUMNS` and `LINES` are set for child processes, not in `%ENV`. A handle that is not a terminal dies "TIOCSWINSZ ioctl call to set terminal size failed: " and the system's reason.

GetSpeed

A terminal's input and output speeds in baud.

Synopsis

```
my ($in, $out) = GetSpeed();          # e.g. (38400, 38400)
```

What you get back

Two numbers, decimal: a speed outside the table of 50 to 115200 baud comes back as the system's own code for it. A handle that is not a terminal gives (0, 0).

GetControlChars

A terminal's control characters, as name/character pairs.

Synopsis

```
my %cc = GetControlChars(\*STDIN);
printf "interrupt is ^%c\n", ord($cc{INTERRUPT}) + 64;
```

What you get back

A list for a hash: DISCARD, EOF, EOL, EOL2, ERASE, ERASEWORD, INTERRUPT, KILL, MIN, QUIT, QUOTENEXT, REPRINT, START, STOP, SUSPEND, SWITCH and TIME, each with its character as a one-character string (MIN and TIME are counts, as characters). A handle that is not a terminal dies “Unable to read terminal settings in GetControlChars”.

SetControlChars

Change a terminal’s control characters.

Synopsis

```
SetControlChars(INTERRUPT => "\cX", EOF => 4, \*STDIN);
```

How it works

Takes name/value pairs as `GetControlChars` returns them, and an odd last argument as the handle (STDIN without one). A value that is a number is the character’s code; any other the first character of its string. An unknown name dies “Invalid control character passed to SetControlChars” before any change is made; a handle that is not a terminal dies “Unable to read terminal settings in SetControlChars”.

Term::ReadLine

Perl interface to various `readline` packages. If no real package is found, substitutes stubs instead of basic functions.

SYNOPSIS

```
use Term::ReadLine;
my $term = Term::ReadLine->new('Simple Perl calc');
my $prompt = "Enter your arithmetic expression: ";
my $OUT = $term->OUT || \*STDOUT;
while ( defined ($_ = $term->readline($prompt)) ) {
  my $res = eval($_);
  warn $@ if $@;
  print $OUT $res, "\n" unless $@;
  $term->addhistory($_) if /\S/;
}
```

DESCRIPTION

This package is just a front end to some other packages. It's a stub to set up a common interface to the various ReadLine implementations found on CPAN (under the `Term::ReadLine::*` namespace).

Minimal set of supported functions

All the supported functions should be called as methods, i.e., either as

```
$term = Term::ReadLine->new('name');
```

or as

```
$term->addhistory('row');
```

where $term is a return value of Term::ReadLine->new().

- `ReadLine`

  returns the actual package that executes the commands. Among possible values are `Term::ReadLine::Gnu`, `Term::ReadLine::Perl`, `Term::ReadLine::Stub`.

- `new`

  returns the handle for subsequent calls to following functions. Argument is the name of the application. Optionally can be followed by two arguments for `IN` and `OUT` filehandles. These arguments should be globs.

- `readline`

  gets an input line, *possibly* with actual `readline` support. Trailing newline is removed. Returns `undef` on `EOF`.

- `addhistory`

  adds the line to the history of input, from where it can be used if the actual `readline` is present.

- `IN`, `OUT`

  return the filehandles for input and output or `undef` if `readline` input and output cannot be used for Perl.

- `MinLine`

  If argument is specified, it is an advice on minimal size of line to be included into history. `undef` means do not include anything into history. Returns the old value.

- `findConsole`

  returns an array with two strings that give most appropriate names for files for input and output using conventions `"<$in"`, `">out"`.

  The strings returned may not be useful for 3-argument open().

- Attribs

returns a reference to a hash which describes internal configuration of the package. Names of keys in this hash conform to standard conventions with the leading `rl_` stripped.

- `Features`

  Returns a reference to a hash with keys being features present in current implementation. Several optional features are used in the minimal interface: `appname` should be present if the first argument to `new` is recognized, and `minline` should be present if `MinLine` method is not dummy. `autohistory` should be present if lines are put into history automatically (maybe subject to `MinLine`), and `addhistory` if `addhistory` method is not dummy.

  If `Features` method reports a feature `attribs` as present, the method `Attribs` is not dummy.

Additional supported functions

Actually `Term::ReadLine` can use some other package, that will support a richer set of commands.

All these commands are callable via method interface and have names which conform to standard conventions with the leading `rl_` stripped.

The stub package included with the perl distribution allows some additional methods:

- `tkRunning`

  makes Tk event loop run when waiting for user input (i.e., during `readline` method).

- `event_loop`

  Registers call-backs to wait for user input (i.e., during `readline` method). This supersedes tkRunning.

  The first call-back registered is the call back for waiting. It is expected that the callback will call the current event loop until there is something waiting to get on the input filehandle. The parameter passed in is the return value of the second call back.

  The second call-back registered is the call back for registration. The input filehandle (often STDIN, but not necessarily) will be passed in.

  For example, with AnyEvent:

```
$term->event_loop(sub {
  my $data = shift;
  $data->[1] = AE::cv();
  $data->[1]->recv();
}, sub {
  my $fh = shift;
  my $data = [];
  $data->[0] = AE::io($fh, 0, sub { $data->[1]->send() });
  $data;
});
```

The second call-back is optional if the call back is registered prior to the call to $term->readline.

Deregistration is done in this case by calling event_loop with `undef` as its parameter:

```
$term->event_loop(undef);
```

This will cause the data array ref to be removed, allowing normal garbage collection to clean it up. With AnyEvent, that will cause $data->[0] to be cleaned up, and AnyEvent will automatically cancel the watcher at that time. If another loop requires more than that to clean up a file watcher, that will be up to the caller to handle.

- `ornaments`

  makes the command line stand out by using termcap data. The argument to `ornaments` should be 0, 1, or a string of a form `"aa,bb,cc,dd"`. Four components of this string should be names of *terminal capacities*, first two will be issued to make the prompt standout, last two to make the input line standout.

- `newTTY`

  takes two arguments which are input filehandle and output filehandle. Switches to use these filehandles.

One can check whether the currently loaded ReadLine package supports these methods by checking for corresponding `Features`.

EXPORTS

None

ENVIRONMENT

The environment variable `PERL_RL` governs which ReadLine clone is loaded. If the value is false, a dummy interface is used. If the value is true, it should be tail of the name of the package to use, such as `Perl` or `Gnu`.

As a special case, if the value of this variable is space-separated, the tail might be used to disable the ornaments by setting the tail to be `o=0` or `ornaments=0`. The head should be as described above, say

If the variable is not set, or if the head of space-separated list is empty, the best available package is loaded.

```
export "PERL_RL=Perl o=0" # Use Perl ReadLine sans ornaments
export "PERL_RL= o=0"     # Use best available ReadLine sans␣
↪ornaments
```

(Note that processing of `PERL_RL` for ornaments is in the discretion of the particular used `Term::ReadLine::*` package).

Modules

- `Term::ReadLine::Gnu` - Peta::ReadLine under the name programs know

Term::ReadLine::Gnu

Peta::ReadLine under the name programs know

SYNOPSIS

```
use Term::ReadLine;
my $term = Term::ReadLine->new('demo');
print $term->ReadLine;    # Term::ReadLine::Gnu
```

DESCRIPTION

`Term::ReadLine::Gnu` is an alias of *Peta::ReadLine*: the same methods, the same `Attribs` hash, the same `Term::ReadLine::Gnu::XS` functions, under the name that *Term::ReadLine* loads first and that programs such as CPAN.pm and perl's debugger test for before they install their completers. `$term->ReadLine` answers `Term::ReadLine::Gnu` for an object made through this name and `Peta::ReadLine` for one made through that.

Term::Table

Format a header and rows into a table

DESCRIPTION

This is used by some failing tests to provide diagnostics about what has gone wrong. This module is able to format rows of data into tables.

SYNOPSIS

```
use Term::Table;

my $table = Term::Table->new(
    max_width      => 80,    # Defaults to terminal size
    pad            => 4,     # Extra padding between table and max-
↪width (defaults to 4)
    allow_overflow => 0,     # Default is 0, when off an exception␣
↪will be thrown if the table is too big
    collapse       => 1,     # Do not show empty columns

    header => ['name', 'age', 'hair color'],
    rows   => [
        ['Fred Flintstone',  2000000, 'black'],
        ['Wilma Flintstone', 1999995, 'red'],
        ...
```

(continues on next page)

(continued from previous page)

```
    ],
);

say $_ for $table->render;
```

This prints a table like this:

```
+------------------+---------+------------+
| name             | age     | hair color |
+------------------+---------+------------+
| Fred Flintstone  | 2000000 | black      |
| Wilma Flintstone | 1999995 | red        |
| ...              | ...     | ...        |
+------------------+---------+------------+
```

INTERFACE

```
use Term::Table;
my $table = Term::Table->new(...);
```

OPTIONS

- header => [...]

  If you want a header specify it here. This takes an arrayref with each columns heading.

- rows => [[...], [...], ...]

  This should be an arrayref containing an arrayref per row.

- collapse => $bool

  Use this if you want to hide empty columns, that is any column that has no data in any row. Having a header for the column will not effect collapse.

- max_width => $num

  Set the maximum width of the table, the table may not be this big, but it will be no bigger. If none is specified it will attempt to find the width of your terminal and use that, otherwise it falls back to the terminal width or `80`.

- pad => $num

  Defaults to 4, extra padding for row width calculations. Default is for legacy support. Set this to `0` to turn padding off.

- allow_overflow => $bool

  Defaults to `0`. If this is off then an exception will be thrown if the table cannot be made to fit inside the max-width. If this is set to 1 then the table will be rendered anyway, larger than max-width, if it is not possible to stay within the max-width. In other words this turns max-width from a hard-limit to a soft recommendation.

- sanitize => $bool

  This will sanitize all the data in the table such that newlines, control characters, and all whitespace except for ASCII 20 ' ' are replaced with escape sequences. This prevents newlines, tabs, and similar whitespace from disrupting the table.

  Note: newlines are marked as \n, but a newline is also inserted into the data so that it typically displays in a way that is useful to humans.

  Example:

```
my $field = "foo\nbar\nbaz\n";

print join "\n" => table(
    sanitize => 1,
    rows => [
        [$field,      'col2'     ],
        ['row2 col1', 'row2 col2']
    ]
);
```

  Prints:

```
+-----------------+-----------+
| foo\n           | col2      |
| bar\n           |           |
| baz\n           |           |
|                 |           |
| row2 col1       | row2 col2 |
+-----------------+-----------+
```

  So it marks the newlines by inserting the escape sequence, but it also shows the data across as many lines as it would normally display.

- mark_tail => $bool

  This will replace the last whitespace character of any trailing whitespace with its escape sequence. This makes it easier to notice trailing whitespace when comparing values.

- show_header => $bool

  Set this to false to hide the header. This defaults to true if the header is set, false if no header is provided.

- auto_columns => $bool

  Set this to true to automatically add columns that are not named in the header. This defaults to false if a header is provided, and defaults to true when there is no header.

- no_collapse => [$col_num_a, $col_num_b, ...]
- no_collapse => [$col_name_a, $col_name_b, ...]
- no_collapse => { $col_num_a => 1, $col_num_b => 1, ... }
- no_collapse => { $col_name_a => 1, $col_name_b => 1, ... }

Specify (by number and/or name) columns that should not be removed when empty. The 'name' form only works when a header is specified. There is currently no protection to insure that names you specify are actually in the header, invalid names are ignored, patches to fix this will be happily accepted.

NOTE ON UNICODE/WIDE CHARACTERS

Some unicode characters, such as 婧 (U+5A67) are wider than others. These will render just fine if you `use utf8;` as necessary, and *Unicode::GCString* is installed, however if the module is not installed there will be anomalies in the table:

```
+-----+-----+---+
| a   | b   | c |
+-----+-----+---+
| 婧 | x   | y |
| x   | y   | z |
| x   | 婧 | z |
+-----+-----+---+
```

SOURCE

The source code repository for `Term-Table` can be found at https://github.com/exodist/Term-Table/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright 2016 Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Modules

- `Term::Table::Cell` - Representation of a cell in a table.
- `Term::Table::CellStack` - Combine several cells into one (vertical)
- `Term::Table::HashBase` - Build hash based classes.
- `Term::Table::LineBreak` - Break up lines for use in tables.
- `Term::Table::Util` - Utilities for Term::Table.

Term::Table::Cell

Representation of a cell in a table.

DESCRIPTION

This package is used to represent a cell in a table.

SOURCE

The source code repository for Term-Table can be found at `http://github.com/exodist/Term-Table/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright 2016 Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Term::Table::CellStack

Combine several cells into one (vertical)

DESCRIPTION

This package is used to represent a merged-cell in a table (vertical).

SOURCE

The source code repository for Term-Table can be found at `http://github.com/exodist/Term-Table/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright 2016 Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Term::Table::HashBase

Build hash based classes.

SYNOPSIS

A class:

```
package My::Class;
use strict;
use warnings;

# Generate 3 accessors
use Term::Table::HashBase qw/foo -bar ^baz <bat >ban +boo/;

# Chance to initialize defaults
sub init {
    my $self = shift;    # No other args
    $self->{+FOO} ||= "foo";
    $self->{+BAR} ||= "bar";
    $self->{+BAZ} ||= "baz";
    $self->{+BAT} ||= "bat";
    $self->{+BAN} ||= "ban";
    $self->{+BOO} ||= "boo";
}

sub print {
    print join ", " => map { $self->{$_} } FOO, BAR, BAZ, BAT, BAN,␣
 ↪BOO;
}
```

Subclass it

```
package My::Subclass;
use strict;
use warnings;

# Note, you should subclass before loading HashBase.
use base 'My::Class';
```

(continues on next page)

(continued from previous page)

```
use Term::Table::HashBase qw/bub/;

sub init {
    my $self = shift;

    # We get the constants from the base class for free.
    $self->{+FOO} ||= 'SubFoo';
    $self->{+BUB} ||= 'bub';

    $self->SUPER::init();
}
```

use it:

```
package main;
use strict;
use warnings;
use My::Class;

# These are all functionally identical
my $one   = My::Class->new(foo => 'MyFoo', bar => 'MyBar');
my $two   = My::Class->new({foo => 'MyFoo', bar => 'MyBar'});
my $three = My::Class->new(['MyFoo', 'MyBar']);

# Readers!
my $foo = $one->foo;    # 'MyFoo'
my $bar = $one->bar;    # 'MyBar'
my $baz = $one->baz;    # Defaulted to: 'baz'
my $bat = $one->bat;    # Defaulted to: 'bat'
# '>ban' means setter only, no reader
# '+boo' means no setter or reader, just the BOO constant

# Setters!
$one->set_foo('A Foo');

#'-bar' means read-only, so the setter will throw an exception (but␣
 ↪is defined).
$one->set_bar('A bar');

# '^baz' means deprecated setter, this will warn about the setter␣
 ↪being
# deprecated.
$one->set_baz('A Baz');

# '<bat' means no setter defined at all
# '+boo' means no setter or reader, just the BOO constant

$one->{+FOO} = 'xxx';
```

DESCRIPTION

This package is used to generate classes based on hashrefs. Using this class will give you a `new()` method, as well as generating accessors you request. Generated accessors will be getters, `set_ACCESSOR` setters will also be generated for you. You also get constants for each accessor (all caps) which return the key into the hash for that accessor. Single inheritance is also supported.

THIS IS A BUNDLED COPY OF HASHBASE

This is a bundled copy of Object::HashBase. This file was generated using the `/home/exodist/perl5/perlbrew/perls/main/bin/hashbase_inc.pl` script.

METHODS

PROVIDED BY HASH BASE

- $it = $class->new(%PAIRS)
- $it = $class->new(\%PAIRS)
- $it = $class->new(\@ORDERED_VALUES)

  Create a new instance.

  HashBase will not export `new()` if there is already a `new()` method in your packages inheritance chain.

  If you do not want this method you can define your own you just have to declare it before loading *Term::Table::HashBase*.

```
package My::Package;

# predeclare new() so that HashBase does not give us one.
sub new;

use Term::Table::HashBase qw/foo bar baz/;

# Now we define our own new method.
sub new { ... }
```

  This makes it so that HashBase sees that you have your own `new()` method. Alternatively you can define the method before loading HashBase instead of just declaring it, but that scatters your use statements.

  The most common way to create an object is to pass in key/value pairs where each key is an attribute and each value is what you want assigned to that attribute. No checking is done to verify the attributes or values are valid, you may do that in `init()` if desired.

  If you would like, you can pass in a hashref instead of pairs. When you do so the hashref will be copied, and the copy will be returned blessed as an object. There is no way to ask HashBase to bless a specific hashref.

  In some cases an object may only have 1 or 2 attributes, in which case a hashref may be too verbose for your liking. In these cases you can pass in an arrayref with

only values. The values will be assigned to attributes in the order the attributes were listed. When there is inheritance involved the attributes from parent classes will come before subclasses.

HOOKS

- $self->init()

  This gives you the chance to set some default values to your fields. The only argument is `$self` with its indexes already set from the constructor.

  Note: Term::Table::HashBase checks for an init using `$class->can('init')` during construction. It DOES NOT call `can()` on the created object. Also note that the result of the check is cached, it is only ever checked once, the first time an instance of your class is created. This means that adding an `init()` method AFTER the first construction will result in it being ignored.

ACCESSORS

READ/WRITE

To generate accessors you list them when using the module:

```
use Term::Table::HashBase qw/foo/;
```

This will generate the following subs in your namespace:

- foo()

  Getter, used to get the value of the `foo` field.

- set_foo()

  Setter, used to set the value of the `foo` field.

- FOO()

  Constant, returns the field `foo`'s key into the class hashref. Subclasses will also get this function as a constant, not simply a method, that means it is copied into the subclass namespace.

  The main reason for using these constants is to help avoid spelling mistakes and similar typos. It will not help you if you forget to prefix the '+' though.

READ ONLY

```
use Term::Table::HashBase qw/-foo/;
```

- set_foo()

  Throws an exception telling you the attribute is read-only. This is exported to override any active setters for the attribute in a parent class.

DEPRECATED SETTER

```
use Term::Table::HashBase qw/^foo/;
```

- set_foo()

  This will set the value, but it will also warn you that the method is deprecated.

NO SETTER

```
use Term::Table::HashBase qw/<foo/;
```

Only gives you a reader, no `set_foo` method is defined at all.

NO READER

```
use Term::Table::HashBase qw/>foo/;
```

Only gives you a write (`set_foo`), no `foo` method is defined at all.

CONSTANT ONLY

```
use Term::Table::HashBase qw/+foo/;
```

This does not create any methods for you, it just adds the `FOO` constant.

SUBCLASSING

You can subclass an existing HashBase class.

```
use base 'Another::HashBase::Class';
use Term::Table::HashBase qw/foo bar baz/;
```

The base class is added to `@ISA` for you, and all constants from base classes are added to subclasses automatically.

GETTING A LIST OF ATTRIBUTES FOR A CLASS

Term::Table::HashBase provides a function for retrieving a list of attributes for an Term::Table::HashBase class.

- @list = Term::Table::HashBase::attr_list($class)
- @list = $class->Term::Table::HashBase::attr_list()

  Either form above will work. This will return a list of attributes defined on the object. This list is returned in the attribute definition order, parent class attributes are listed before subclass attributes. Duplicate attributes will be removed before the list is returned.

  Note: This list is used in the `$class->new(\@ARRAY)` constructor to determine the attribute to which each value will be paired.

SOURCE

The source code repository for HashBase can be found at `http://github.com/Test-More/HashBase/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright 2017 Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Term::Table::LineBreak

Break up lines for use in tables.

DESCRIPTION

This is meant for internal use. This package takes long lines of text and splits them so that they fit in table rows.

SYNOPSIS

```
use Term::Table::LineBreak;

my $lb = Term::Table::LineBreak->new(string => $STRING);

$lb->break($SIZE);
while (my $part = $lb->next) {
    ...
}
```

SOURCE

The source code repository for Term-Table can be found at `http://github.com/exodist/Term-Table/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright 2016 Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Term::Table::Util

Utilities for Term::Table.

DESCRIPTION

This package exports some tools used by Term::Table.

EXPORTS

CONSTANTS

- $bool = USE_GCS

  True if *Unicode::GCString* is installed.

- $bool = USE_TERM_READKEY

  True if *Term::ReadKey* is installed.

UTILITIES

- $width = term_size()

  Get the width of the terminal.

  If the `$TABLE_TERM_SIZE` environment variable is set then that value will be returned.

  This will default to 80 if there is no good way to get the size, or if the size is unreasonably small.

  If *Term::ReadKey* is installed it will be used.

- $width = uni_length($string)

  Get the width (in columns) of the specified string. When *Unicode::GCString* is installed this will work on unicode strings, otherwise it will just use `length($string)`.

SOURCE

The source code repository for Term-Table can be found at `http://github.com/exodist/Term-Table/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright 2016 Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

5.2.101 Test

provides a simple framework for writing test scripts

SYNOPSIS

```
use strict;
use Test;

# use a BEGIN block so we print our plan before MyModule is loaded
BEGIN { plan tests => 14, todo => [3,4] }

# load your module...
use MyModule;

# Helpful notes.  All note-lines must start with a "#".
print "# I'm testing MyModule version $MyModule::VERSION\n";

ok(0); # failure
ok(1); # success

ok(0); # ok, expected failure (see todo list, above)
ok(1); # surprise success!

ok(0,1);             # failure: '0' ne '1'
ok('broke','fixed'); # failure: 'broke' ne 'fixed'
ok('fixed','fixed'); # success: 'fixed' eq 'fixed'
ok('fixed',qr/x/);   # success: 'fixed' =~ qr/x/
```

(continues on next page)

(continued from previous page)

```
ok(sub { 1+1 }, 2);  # success: '2' eq '2'
ok(sub { 1+1 }, 3);  # failure: '2' ne '3'

my @list = (0,0);
ok @list, 3, "\@list=".join(',',@list);      #extra notes
ok 'segmentation fault', '/(?i)success/';    #regex match

skip(
  $^O =~ m/MSWin/ ? "Skip if MSWin" : 0,  # whether to skip
  $foo, $bar  # arguments just like for ok(...)
);
skip(
  $^O =~ m/MSWin/ ? 0 : "Skip unless MSWin",  # whether to skip
  $foo, $bar  # arguments just like for ok(...)
);
```

DESCRIPTION

This module simplifies the task of writing test files for Perl modules, such that their output is in the format that *Test::Harness* expects to see.

QUICK START GUIDE

To write a test for your new (and probably not even done) module, create a new file called `t/test.t` (in a new `t` directory). If you have multiple test files, to test the "foo", "bar", and "baz" feature sets, then feel free to call your files `t/foo.t`, `t/bar.t`, and `t/baz.t`

Functions

This module defines three public functions, `plan(...)`, `ok(...)`, and `skip(...)`. By default, all three are exported by the `use Test;` statement.

- `plan(...)`

  ```
  BEGIN { plan %theplan; }
  ```

  This should be the first thing you call in your test script. It declares your testing plan, how many there will be, if any of them should be allowed to fail, and so on.

  Typical usage is just:

  ```
  use Test;
  BEGIN { plan tests => 23 }
  ```

  These are the things that you can put in the parameters to plan:

  - `tests => _number_`

    The number of tests in your script. This means all ok() and skip() calls.

  - `todo => [_1,5,14_]`

A reference to a list of tests which are allowed to fail. See "TODO TESTS".

- `onfail => sub { ... }`
- `onfail => \&some_sub`

  A subroutine reference to be run at the end of the test script, if any of the tests fail. See "ONFAIL".

You must call `plan(...)` once and only once. You should call it in a `BEGIN {...}` block, like so:

```
BEGIN { plan tests => 23 }
```

- `ok(...)`

```
ok(1 + 1 == 2);
ok($have, $expect);
ok($have, $expect, $diagnostics);
```

This function is the reason for `Test`'s existence. It's the basic function that handles printing "ok" or "not ok", along with the current test number. (That's what `Test::Harness` wants to see.)

In its most basic usage, `ok(...)` simply takes a single scalar expression. If its value is true, the test passes; if false, the test fails. Examples:

```
# Examples of ok(scalar)

ok( 1 + 1 == 2 );           # ok if 1 + 1 == 2
ok( $foo =~ /bar/ );        # ok if $foo contains 'bar'
ok( baz($x + $y) eq 'Armondo' );    # ok if baz($x + $y) returns
                                    # 'Armondo'
ok( @a == @b );             # ok if @a and @b are the same
                            # length
```

The expression is evaluated in scalar context. So the following will work:

```
ok( @stuff );                       # ok if @stuff has any
                                    # elements
ok( !grep !defined $_, @stuff );    # ok if everything in @stuff
                                    # is defined.
```

A special case is if the expression is a subroutine reference (in either `sub {...}` syntax or `\&foo` syntax). In that case, it is executed and its value (true or false) determines if the test passes or fails. For example,

```
ok( sub {   # See whether sleep works at least passably
  my $start_time = time;
  sleep 5;
  time() - $start_time  >= 4
});
```

In its two-argument form, `ok(_arg1_, _arg2_)` compares the two scalar values to see if they match. They match if both are undefined, or if *arg2* is a regex that

matches *arg1*, or if they compare equal with eq.

```
# Example of ok(scalar, scalar)

ok( "this", "that" );                # not ok, 'this' ne 'that'
ok( "", undef );                     # not ok, "" is defined
```

The second argument is considered a regex if it is either a regex object or a string that looks like a regex. Regex objects are constructed with the qr// operator in recent versions of perl. A string is considered to look like a regex if its first and last characters are "/", or if the first character is "m" and its second and last characters are both the same non-alphanumeric non-whitespace character. These regexp

Regex examples:

```
ok( 'JaffO', '/Jaff/' );    # ok, 'JaffO' =~ /Jaff/
ok( 'JaffO', 'm|Jaff|' );   # ok, 'JaffO' =~ m|Jaff|
ok( 'JaffO', qr/Jaff/ );    # ok, 'JaffO' =~ qr/Jaff/;
ok( 'JaffO', '/(?i)jaff/ ); # ok, 'JaffO' =~ /jaff/i;
```

If either (or both!) is a subroutine reference, it is run and used as the value for comparing. For example:

```
ok sub {
    open(OUT, '>', 'x.dat') || die $!;
    print OUT "\x{e000}";
    close OUT;
    my $bytecount = -s 'x.dat';
    unlink 'x.dat' or warn "Can't unlink : $!";
    return $bytecount;
  },
  4
;
```

The above test passes two values to `ok(arg1, arg2)` – the first a coderef, and the second is the number 4. Before ok compares them, it calls the coderef, and uses its return value as the real value of this parameter. Assuming that `$bytecount` returns 4, ok ends up testing `4 eq 4`. Since that's true, this test passes.

Finally, you can append an optional third argument, in `ok(_arg1_,_arg2_, _note_)`, where *note* is a string value that will be printed if the test fails. This should be some useful information about the test, pertaining to why it failed, and/or a description of the test. For example:

```
ok( grep($_ eq 'something unique', @stuff), 1,
    "Something that should be unique isn't!\n".
    '@stuff = '.join ', ', @stuff
  );
```

Unfortunately, a note cannot be used with the single argument style of `ok()`. That is, if you try `ok(_arg1_, _note_)`, then `Test` will interpret this as `ok(_arg1_, _arg2_)`, and probably end up testing `_arg1_ eq _arg2_` – and that's not what you want!

All of the above special cases can occasionally cause some problems. See "BUGS and CAVEATS".

- `skip(_skip_if_true_, _args..._)`

  This is used for tests that under some conditions can be skipped. It's basically equivalent to:

```
if( $skip_if_true ) {
  ok(1);
} else {
  ok( args... );
}
```

  ...except that the `ok(1)` emits not just "`ok _testnum_`" but actually "`ok _testnum_ # _skip_if_true_value_`".

  The arguments after the *skip_if_true* are what is fed to `ok(...)` if this test isn't skipped.

  Example usage:

```
my $if_MSWin =
  $^O =~ m/MSWin/ ? 'Skip if under MSWin' : '';

# A test to be skipped if under MSWin (i.e., run except under
# MSWin)
skip($if_MSWin, thing($foo), thing($bar) );
```

  Or, going the other way:

```
my $unless_MSWin =
  $^O =~ m/MSWin/ ? '' : 'Skip unless under MSWin';

# A test to be skipped unless under MSWin (i.e., run only under
# MSWin)
skip($unless_MSWin, thing($foo), thing($bar) );
```

  The tricky thing to remember is that the first parameter is true if you want to *skip* the test, not *run* it; and it also doubles as a note about why it's being skipped. So in the first codeblock above, read the code as "skip if MSWin – (otherwise) test whether `thing($foo)` is `thing($bar)`" or for the second case, "skip unless MSWin...".

  Also, when your *skip_if_reason* string is true, it really should (for backwards compatibility with older Test.pm versions) start with the string "Skip", as shown in the above examples.

  Note that in the above cases, `thing($foo)` and `thing($bar)` *are* evaluated – but as long as the `skip_if_true` is true, then we `skip(...)` just tosses out their value (i.e., not bothering to treat them like values to `ok(...)`. But if you need to *not* eval the arguments when skipping the test, use this format:

```
skip( $unless_MSWin,
  sub {
    # This code returns true if the test passes.
    # (But it doesn't even get called if the test is skipped.)
    thing($foo) eq thing($bar)
  }
);
```

or even this, which is basically equivalent:

```
skip( $unless_MSWin,
  sub { thing($foo) }, sub { thing($bar) }
);
```

That is, both are like this:

```
if( $unless_MSWin ) {
  ok(1);  # but it actually appends "# $unless_MSWin"
          #  so that Test::Harness can tell it's a skip
} else {
  # Not skipping, so actually call and evaluate...
  ok( sub { thing($foo) }, sub { thing($bar) } );
}
```

TEST TYPES

- NORMAL TESTS

  These tests are expected to succeed. Usually, most or all of your tests are in this category. If a normal test doesn't succeed, then that means that something is *wrong*.

- SKIPPED TESTS

  The `skip(...)` function is for tests that might or might not be possible to run, depending on the availability of platform-specific features. The first argument should evaluate to true (think "yes, please skip") if the required feature is *not* available. After the first argument, `skip(...)` works exactly the same way as `ok(...)` does.

- TODO TESTS

  TODO tests are designed for maintaining an **executable TODO list**. These tests are *expected to fail.* If a TODO test does succeed, then the feature in question shouldn't be on the TODO list, now should it?

  Packages should NOT be released with succeeding TODO tests. As soon as a TODO test starts working, it should be promoted to a normal test, and the newly working feature should be documented in the release notes or in the change log.

ONFAIL

```
BEGIN { plan test => 4, onfail => sub { warn "CALL 911!" } }
```

Although test failures should be enough, extra diagnostics can be triggered at the end of a test run. `onfail` is passed an array ref of hash refs that describe each test failure. Each hash will contain at least the following fields: `package`, `repetition`, and `result`. (You shouldn't rely on any other fields being present.) If the test had an expected value or a diagnostic (or "note") string, these will also be included.

The *optional* `onfail` hook might be used simply to print out the version of your package and/or how to report problems. It might also be used to generate extremely sophisticated diagnostics for a particularly bizarre test failure. However it's not a panacea. Core dumps or other unrecoverable errors prevent the `onfail` hook from running. (It is run inside an `END` block.) Besides, `onfail` is probably over-kill in most cases. (Your test code should be simpler than the code it is testing, yes?)

BUGS and CAVEATS

- `ok(...)`'s special handing of strings which look like they might be regexes can also cause unexpected behavior. An innocent:

```
ok( $fileglob, '/path/to/some/*stuff/' );
```

  will fail, since Test.pm considers the second argument to be a regex! The best bet is to use the one-argument form:

```
ok( $fileglob eq '/path/to/some/*stuff/' );
```

- `ok(...)`'s use of string `eq` can sometimes cause odd problems when comparing numbers, especially if you're casting a string to a number:

```
$foo = "1.0";
ok( $foo, 1 );      # not ok, "1.0" ne 1
```

  Your best bet is to use the single argument form:

```
ok( $foo == 1 );    # ok "1.0" == 1
```

- As you may have inferred from the above documentation and examples, `ok`'s prototype is `($;$$)` (and, incidentally, `skip`'s is `($;$$$)`). This means, for example, that you can do `ok @foo, @bar` to compare the *size* of the two arrays. But don't be fooled into thinking that `ok @foo, @bar` means a comparison of the contents of two arrays – you're comparing *just* the number of elements of each. It's so easy to make that mistake in reading `ok @foo, @bar` that you might want to be very explicit about it, and instead write `ok scalar(@foo), scalar(@bar)`.

- This almost definitely doesn't do what you expect:

```
ok $thingy->can('some_method');
```

  Why? Because `can` returns a coderef to mean "yes it can (and the method is this...)", and then `ok` sees a coderef and thinks you're passing a function that you want it to call and consider the truth of the result of! I.e., just like:

```
ok $thingy->can('some_method')->();
```

  What you probably want instead is this:

```
ok $thingy->can('some_method') && 1;
```

If the `can` returns false, then that is passed to `ok`. If it returns true, then the larger expression `$thingy->can('some_method') && 1` returns 1, which `ok` sees as a simple signal of success, as you would expect.

- The syntax for `skip` is about the only way it can be, but it's still quite confusing. Just start with the above examples and you'll be okay.

  Moreover, users may expect this:

```
skip $unless_mswin, foo($bar), baz($quux);
```

  to not evaluate `foo($bar)` and `baz($quux)` when the test is being skipped. But in reality, they *are* evaluated, but `skip` just won't bother comparing them if `$unless_mswin` is true.

  You could do this:

```
skip $unless_mswin, sub{foo($bar)}, sub{baz($quux)};
```

  But that's not terribly pretty. You may find it simpler or clearer in the long run to just do things like this:

```
if( $^O =~ m/MSWin/ ) {
  print "# Yay, we're under $^O\n";
  ok foo($bar), baz($quux);
  ok thing($whatever), baz($stuff);
  ok blorp($quux, $whatever);
  ok foo($barzbarz), thang($quux);
} else {
  print "# Feh, we're under $^O.  Watch me skip some tests...\n
 ↪";
  for(1 .. 4) { skip "Skip unless under MSWin" }
}
```

  But be quite sure that `ok` is called exactly as many times in the first block as `skip` is called in the second block.

ENVIRONMENT

If `PERL_TEST_DIFF` environment variable is set, it will be used as a command for comparing unexpected multiline results. If you have GNU diff installed, you might want to set `PERL_TEST_DIFF` to `diff -u`. If you don't have a suitable program, you might install the `Text::Diff` module and then set `PERL_TEST_DIFF` to be `perl \-MText::Diff -e 'print diff(@ARGV)'`. If `PERL_TEST_DIFF` isn't set but the `Algorithm::Diff` module is available, then it will be used to show the differences in multiline results.

NOTE

A past developer of this module once said that it was no longer being actively developed. However, rumors of its demise were greatly exaggerated. Feedback and suggestions are quite welcome.

Be aware that the main value of this module is its simplicity. Note that there are already more ambitious modules out there, such as *Test::More* and Test::Unit.

Some earlier versions of this module had docs with some confusing typos in the description of `skip(...)`.

SEE ALSO

Test::Harness

Test::Simple, *Test::More*, *Devel::Cover*

Test::Builder for building your own testing library.

Test::Unit is an interesting XUnit-style testing library.

Test::Inline lets you embed tests in code.

AUTHOR

Copyright (c) 1998-2000 Joshua Nathaniel Pritikin.

Copyright (c) 2001-2002 Michael G. Schwern.

Copyright (c) 2002-2004 Sean M. Burke.

Current maintainer: Jesse Vincent. jesse@bestpractical.com

This package is free software and is provided “as is” without express or implied warranty. It may be used, redistributed and/or modified under the same terms as Perl itself.

Modules

- `Test::Builder` - Backend for building test libraries
- `Test::Harness` - Run Perl standard test scripts with statistics
- `Test::LeakTrace` - Find the scalars, arrays, hashes and code a block of Perl leaves behind.
- `Test::More` - yet another framework for writing test scripts
- `Test::Simple` - Basic utilities for writing tests.
- `Test::Tester` - Ease testing test modules built with Test::Builder
- `Test::use`

Test::Builder

Backend for building test libraries

SYNOPSIS

```
package My::Test::Module;
use base 'Test::Builder::Module';

my $CLASS = __PACKAGE__;

sub ok {
    my($test, $name) = @_;
    my $tb = $CLASS->builder;

    $tb->ok($test, $name);
}
```

DESCRIPTION

Test::Simple and *Test::More* have proven to be popular testing modules, but they're not always flexible enough. Test::Builder provides a building block upon which to write your own test libraries *which can work together*.

Construction

- **new**

  ```
  my $Test = Test::Builder->new;
  ```

  Returns a Test::Builder object representing the current state of the test.

  Since you only run one test per program `new` always returns the same Test::Builder object. No matter how many times you call `new()`, you're getting the same object. This is called a singleton. This is done so that multiple modules share such global information as the test counter and where test output is going.

  If you want a completely new Test::Builder object different from the singleton, use `create`.

- **create**

  ```
  my $Test = Test::Builder->create;
  ```

  Ok, so there can be more than one Test::Builder object and this is how you get it. You might use this instead of `new()` if you're testing a Test::Builder based module, but otherwise you probably want `new`.

  NOTE: the implementation is not complete. `level`, for example, is still shared by **all** Test::Builder objects, even ones created using this method. Also, the method name may change in the future.

- **subtest**

  ```
  $builder->subtest($name, \&subtests, @args);
  ```

  See documentation of `subtest` in Test::More.

`subtest` also, and optionally, accepts arguments which will be passed to the subtests reference.

- **name**

```
diag $builder->name;
```

Returns the name of the current builder. Top level builders default to `$0` (the name of the executable). Child builders are named via the `child` method. If no name is supplied, will be named "Child of $parent->name".

- **reset**

```
$Test->reset;
```

Reinitializes the Test::Builder singleton to its original state. Mostly useful for tests run in persistent environments where the same test might be run multiple times in the same process.

Setting up tests

These methods are for setting up tests and declaring how many there are. You usually only want to call one of these methods.

- **plan**

```
$Test->plan('no_plan');
$Test->plan( skip_all => $reason );
$Test->plan( tests => $num_tests );
```

A convenient way to set up your tests. Call this and Test::Builder will print the appropriate headers and take the appropriate actions.

If you call `plan()`, don't call any of the other methods below.

- **expected_tests**

```
my $max = $Test->expected_tests;
$Test->expected_tests($max);
```

Gets/sets the number of tests we expect this test to run and prints out the appropriate headers.

- **no_plan**

```
$Test->no_plan;
```

Declares that this test will run an indeterminate number of tests.

- **done_testing**

```
$Test->done_testing();
$Test->done_testing($num_tests);
```

Declares that you are done testing, no more tests will be run after this point.

If a plan has not yet been output, it will do so.

$num_tests is the number of tests you planned to run. If a numbered plan was already declared, and if this contradicts, a failing test will be run to reflect the planning mistake. If no_plan was declared, this will override.

If done_testing() is called twice, the second call will issue a failing test.

If $num_tests is omitted, the number of tests run will be used, like no_plan.

done_testing() is, in effect, used when you'd want to use no_plan, but safer. You'd use it like so:

```
$Test->ok($a == $b);
$Test->done_testing();
```

Or to plan a variable number of tests:

```
for my $test (@tests) {
    $Test->ok($test);
}
$Test->done_testing(scalar @tests);
```

- **has_plan**

```
$plan = $Test->has_plan
```

Find out whether a plan has been defined. $plan is either undef (no plan has been set), no_plan (indeterminate # of tests) or an integer (the number of expected tests).

- **skip_all**

```
$Test->skip_all;
$Test->skip_all($reason);
```

Skips all the tests, using the given $reason. Exits immediately with 0.

- **exported_to**

```
my $pack = $Test->exported_to;
$Test->exported_to($pack);
```

Tells Test::Builder what package you exported your functions to.

This method isn't terribly useful since modules which share the same Test::Builder object might get exported to different packages and only the last one will be honored.

Running tests

These actually run the tests, analogous to the functions in Test::More.

They all return true if the test passed, false if the test failed.

$name is always optional.

- **ok**

```
$Test->ok($test, $name);
```

Your basic test. Pass if `$test` is true, fail if $test is false. Just like Test::Simple's `ok()`.

- **is_eq**

```
$Test->is_eq($got, $expected, $name);
```

Like Test::More's `is()`. Checks if `$got eq $expected`. This is the string version.

`undef` only ever matches another `undef`.

- **is_num**

```
$Test->is_num($got, $expected, $name);
```

Like Test::More's `is()`. Checks if `$got == $expected`. This is the numeric version.

`undef` only ever matches another `undef`.

- **isnt_eq**

```
$Test->isnt_eq($got, $dont_expect, $name);
```

Like *Test::More*'s `isnt()`. Checks if `$got ne $dont_expect`. This is the string version.

- **isnt_num**

```
$Test->isnt_num($got, $dont_expect, $name);
```

Like *Test::More*'s `isnt()`. Checks if `$got ne $dont_expect`. This is the numeric version.

- **like**

```
$Test->like($thing, qr/$regex/, $name);
$Test->like($thing, '/$regex/', $name);
```

Like *Test::More*'s `like()`. Checks if $thing matches the given `$regex`.

- **unlike**

```
$Test->unlike($thing, qr/$regex/, $name);
$Test->unlike($thing, '/$regex/', $name);
```

Like *Test::More*'s `unlike()`. Checks if $thing **does not match** the given `$regex`.

- **cmp_ok**

```
$Test->cmp_ok($thing, $type, $that, $name);
```

Works just like *Test::More*'s `cmp_ok()`.

```
$Test->cmp_ok($big_num, '!=', $other_big_num);
```

Other Testing Methods

These are methods which are used in the course of writing a test but are not themselves tests.

- **BAIL_OUT**

```
$Test->BAIL_OUT($reason);
```

  Indicates to the *Test::Harness* that things are going so badly all testing should terminate. This includes running any additional test scripts.

  It will exit with 255.

- **skip**

```
$Test->skip;
$Test->skip($why);
```

  Skips the current test, reporting `$why`.

- **todo_skip**

```
$Test->todo_skip;
$Test->todo_skip($why);
```

  Like `skip()`, only it will declare the test as failing and TODO. Similar to

```
print "not ok $tnum # TODO $why\n";
```

Test building utility methods

These methods are useful when writing your own test methods.

- **maybe_regex**

```
$Test->maybe_regex(qr/$regex/);
$Test->maybe_regex('/$regex/');
```

  This method used to be useful back when Test::Builder worked on Perls before 5.6 which didn't have qr//. Now its pretty useless.

  Convenience method for building testing functions that take regular expressions as arguments.

  Takes a quoted regular expression produced by `qr//`, or a string representing a regular expression.

  Returns a Perl value which may be used instead of the corresponding regular expression, or `undef` if its argument is not recognized.

  For example, a version of `like()`, sans the useful diagnostic messages, could be written as:

```
sub laconic_like {
    my ($self, $thing, $regex, $name) = @_;
    my $usable_regex = $self->maybe_regex($regex);
    die "expecting regex, found '$regex'\n"
        unless $usable_regex;
    $self->ok($thing =~ m/$usable_regex/, $name);
}
```

- **is_fh**

```
my $is_fh = $Test->is_fh($thing);
```

Determines if the given `$thing` can be used as a filehandle.

Test style

- **level**

```
$Test->level($how_high);
```

How far up the call stack should `$Test` look when reporting where the test failed.

Defaults to 1.

Setting `$Test::Builder::Level` overrides. This is typically useful localized:

```
sub my_ok {
    my $test = shift;

    local $Test::Builder::Level = $Test::Builder::Level + 1;
    $TB->ok($test);
}
```

To be polite to other functions wrapping your own you usually want to increment `$Level` rather than set it to a constant.

- **use_numbers**

```
$Test->use_numbers($on_or_off);
```

Whether or not the test should output numbers. That is, this if true:

```
ok 1
ok 2
ok 3
```

or this if false

```
ok
ok
ok
```

Most useful when you can't depend on the test output order, such as when threads or forking is involved.

Defaults to on.

- **no_diag**

```
$Test->no_diag($no_diag);
```

If set true no diagnostics will be printed. This includes calls to `diag()`.

- **no_ending**

```
$Test->no_ending($no_ending);
```

Normally, Test::Builder does some extra diagnostics when the test ends. It also changes the exit code as described below.

If this is true, none of that will be done.

- **no_header**

```
$Test->no_header($no_header);
```

If set to true, no "1..N" header will be printed.

Output

Controlling where the test output goes.

It's ok for your test to change where STDOUT and STDERR point to, Test::Builder's default output settings will not be affected.

- **diag**

```
$Test->diag(@msgs);
```

Prints out the given `@msgs`. Like `print`, arguments are simply appended together.

Normally, it uses the `failure_output()` handle, but if this is for a TODO test, the `todo_output()` handle is used.

Output will be indented and marked with a # so as not to interfere with test output. A newline will be put on the end if there isn't one already.

We encourage using this rather than calling print directly.

Returns false. Why? Because `diag()` is often used in conjunction with a failing test (`ok() || diag()`) it "passes through" the failure.

```
return ok(...) || diag(...);
```

- **note**

```
$Test->note(@msgs);
```

Like `diag()`, but it prints to the `output()` handle so it will not normally be seen by the user except in verbose mode.

- **explain**

```
my @dump = $Test->explain(@msgs);
```

Will dump the contents of any references in a human readable format. Handy for things like...

```
is_deeply($have, $want) || diag explain $have;
```

or

```
is_deeply($have, $want) || note explain $have;
```

- **output**
- **failure_output**
- **todo_output**

```
my $filehandle = $Test->output;
$Test->output($filehandle);
$Test->output($filename);
$Test->output(\$scalar);
```

These methods control where Test::Builder will print its output. They take either an open `$filehandle`, a `$filename` to open and write to or a `$scalar` reference to append to. It will always return a `$filehandle`.

output is where normal "ok/not ok" test output goes.

Defaults to STDOUT.

failure_output is where diagnostic output on test failures and `diag()` goes. It is normally not read by Test::Harness and instead is displayed to the user.

Defaults to STDERR.

`todo_output` is used instead of `failure_output()` for the diagnostics of a failing TODO test. These will not be seen by the user.

Defaults to STDOUT.

- reset_outputs

```
$tb->reset_outputs;
```

Resets all the output filehandles back to their defaults.

- carp

```
$tb->carp(@message);
```

Warns with `@message` but the message will appear to come from the point where the original test function was called (`$tb->caller`).

- croak

```
$tb->croak(@message);
```

Dies with `@message` but the message will appear to come from the point where the original test function was called (`$tb->caller`).

Test Status and Info

- **no_log_results**

  This will turn off result long-term storage. Calling this method will make `details` and `summary` useless. You may want to use this if you are running enough tests to fill up all available memory.

```
Test::Builder->new->no_log_results();
```

  There is no way to turn it back on.

- **current_test**

```
my $curr_test = $Test->current_test;
$Test->current_test($num);
```

  Gets/sets the current test number we're on. You usually shouldn't have to set this.

  If set forward, the details of the missing tests are filled in as 'unknown'. if set backward, the details of the intervening tests are deleted. You can erase history if you really want to.

- **is_passing**

```
my $ok = $builder->is_passing;
```

  Indicates if the test suite is currently passing.

  More formally, it will be false if anything has happened which makes it impossible for the test suite to pass. True otherwise.

  For example, if no tests have run `is_passing()` will be true because even though a suite with no tests is a failure you can add a passing test to it and start passing.

  Don't think about it too much.

- **summary**

```
my @tests = $Test->summary;
```

  A simple summary of the tests so far. True for pass, false for fail. This is a logical pass/fail, so todos are passes.

  Of course, test #1 is $tests[0], etc...

- **details**

```
my @tests = $Test->details;
```

  Like `summary()`, but with a lot more detail.

```
$tests[$test_num - 1] =
        { 'ok'       => is the test considered a pass?
          actual_ok  => did it literally say 'ok'?
          name       => name of the test (if any)
          type       => type of test (if any, see below).
          reason     => reason for the above (if any)
        };
```

'ok' is true if Test::Harness will consider the test to be a pass.

'actual_ok' is a reflection of whether or not the test literally printed 'ok' or 'not ok'. This is for examining the result of 'todo' tests.

'name' is the name of the test.

'type' indicates if it was a special test. Normal tests have a type of ''. Type can be one of the following:

```
skip        see skip()
todo        see todo()
todo_skip   see todo_skip()
unknown     see below
```

Sometimes the Test::Builder test counter is incremented without it printing any test output, for example, when `current_test()` is changed. In these cases, Test::Builder doesn't know the result of the test, so its type is 'unknown'. These details for these tests are filled in. They are considered ok, but the name and actual_ok is left `undef`.

For example "not ok 23 - hole count # TODO insufficient donuts" would result in this structure:

```
$tests[22] =    # 23 - 1, since arrays start from 0.
  { ok        => 1,   # logically, the test passed since its
↪todo
    actual_ok => 0,   # in absolute terms, it failed
    name      => 'hole count',
    type      => 'todo',
    reason    => 'insufficient donuts'
  };
```

- **todo**

```
my $todo_reason = $Test->todo;
my $todo_reason = $Test->todo($pack);
```

If the current tests are considered "TODO" it will return the reason, if any. This reason can come from a `$TODO` variable or the last call to `todo_start()`.

Since a TODO test does not need a reason, this function can return an empty string even when inside a TODO block. Use `$Test->in_todo` to determine if you are currently inside a TODO block.

`todo()` is about finding the right package to look for `$TODO` in. It's pretty good at guessing the right package to look at. It first looks for the caller based on `$Level`

+ 1, since `todo()` is usually called inside a test function. As a last resort it will use `exported_to()`.

Sometimes there is some confusion about where `todo()` should be looking for the `$TODO` variable. If you want to be sure, tell it explicitly what $pack to use.

- **find_TODO**

```
my $todo_reason = $Test->find_TODO();
my $todo_reason = $Test->find_TODO($pack);
```

Like `todo()` but only returns the value of `$TODO` ignoring `todo_start()`.

Can also be used to set `$TODO` to a new value while returning the old value:

```
my $old_reason = $Test->find_TODO($pack, 1, $new_reason);
```

- **in_todo**

```
my $in_todo = $Test->in_todo;
```

Returns true if the test is currently inside a TODO block.

- **todo_start**

```
$Test->todo_start();
$Test->todo_start($message);
```

This method allows you declare all subsequent tests as TODO tests, up until the `todo_end` method has been called.

The `TODO:` and `$TODO` syntax is generally pretty good about figuring out whether or not we're in a TODO test. However, often we find that this is not possible to determine (such as when we want to use `$TODO` but the tests are being executed in other packages which can't be inferred beforehand).

Note that you can use this to nest "todo" tests

```
$Test->todo_start('working on this');
# lots of code
$Test->todo_start('working on that');
# more code
$Test->todo_end;
$Test->todo_end;
```

This is generally not recommended, but large testing systems often have weird internal needs.

We've tried to make this also work with the TODO: syntax, but it's not guaranteed and its use is also discouraged:

```
TODO: {
    local $TODO = 'We have work to do!';
    $Test->todo_start('working on this');
    # lots of code
```

(continues on next page)

(continued from previous page)

```
    $Test->todo_start('working on that');
    # more code
    $Test->todo_end;
    $Test->todo_end;
}
```

Pick one style or another of "TODO" to be on the safe side.

- `todo_end`

```
$Test->todo_end;
```

Stops running tests as "TODO" tests. This method is fatal if called without a preceding `todo_start` method call.

- **caller**

```
my $package = $Test->caller;
my($pack, $file, $line) = $Test->caller;
my($pack, $file, $line) = $Test->caller($height);
```

Like the normal `caller()`, except it reports according to your `level()`.

`$height` will be added to the `level()`.

If `caller()` winds up off the top of the stack it report the highest context.

EXIT CODES

If all your tests passed, Test::Builder will exit with zero (which is normal). If anything failed it will exit with how many failed. If you run less (or more) tests than you planned, the missing (or extras) will be considered failures. If no tests were ever run Test::Builder will throw a warning and exit with 255. If the test died, even after having successfully completed all its tests, it will still be considered a failure and will exit with 255.

So the exit codes are...

```
0                   all tests successful
255                 test died or all passed but wrong # of tests run
any other number    how many failed (including missing or extras)
```

If you fail more than 254 tests, it will be reported as 254.

THREADS

In perl 5.8.1 and later, Test::Builder is thread-safe. The test number is shared by all threads. This means if one thread sets the test number using `current_test()` they will all be affected.

While versions earlier than 5.8.1 had threads they contain too many bugs to support.

Test::Builder is only thread-aware if threads.pm is loaded *before* Test::Builder.

You can directly disable thread support with one of the following:

```
$ENV{T2_NO_IPC} = 1
```

or

```
no Test2::IPC;
```

or

```
Test2::API::test2_ipc_disable()
```

MEMORY

An informative hash, accessible via `details()`, is stored for each test you perform. So memory usage will scale linearly with each test run. Although this is not a problem for most test suites, it can become an issue if you do large (hundred thousands to million) combinatorics tests in the same run.

In such cases, you are advised to either split the test file into smaller ones, or use a reverse approach, doing "normal" (code) compares and triggering `fail()` should anything go unexpected.

Future versions of Test::Builder will have a way to turn history off.

EXAMPLES

CPAN can provide the best examples. *Test::Simple*, *Test::More*, Test::Exception and Test::Differences all use Test::Builder.

SEE ALSO

INTERNALS

Test2, *Test2::API*

LEGACY

Test::Simple, *Test::More*

EXTERNAL

Test::Harness

AUTHORS

Original code by chromatic, maintained by Michael G Schwern schwern@pobox.com

MAINTAINERS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright 2002-2008 by chromatic chromatic@wgz.org and Michael G Schwern schwern@pobox.com.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Modules

- `Test::Builder::Formatter` - Test::Builder subclass of Test2::Formatter::TAP
- `Test::Builder::Module` - Base class for test modules
- `Test::Builder::Tester` - test testsuites that have been built with Test::Builder
- `Test::Builder::TodoDiag` - Test::Builder subclass of Test2::Event::Diag

Test::Builder::Formatter

Test::Builder subclass of Test2::Formatter::TAP

DESCRIPTION

This is what takes events and turns them into TAP.

SYNOPSIS

```
use Test::Builder; # Loads Test::Builder::Formatter for you
```

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test::Builder::Module

Base class for test modules

SYNOPSIS

```
# Emulates Test::Simple
package Your::Module;

my $CLASS = __PACKAGE__;

use parent 'Test::Builder::Module';
@EXPORT = qw(ok);

sub ok ($;$) {
    my $tb = $CLASS->builder;
    return $tb->ok(@_);
}

1;
```

DESCRIPTION

This is a superclass for *Test::Builder*-based modules. It provides a handful of common functionality and a method of getting at the underlying *Test::Builder* object.

Importing

Test::Builder::Module is a subclass of *Exporter* which means your module is also a subclass of Exporter. @EXPORT, @EXPORT_OK, etc... all act normally.

A few methods are provided to do the `use Your::Module tests => 23` part for you.

import

Test::Builder::Module provides an `import()` method which acts in the same basic way as *Test::More*'s, setting the plan and controlling exporting of functions and variables. This allows your module to set the plan independent of *Test::More*.

All arguments passed to `import()` are passed onto `Your::Module->builder->plan()` with the exception of `import =>[qw(things to import)]`.

```
use Your::Module import => [qw(this that)], tests => 23;
```

says to import the functions `this()` and `that()` as well as set the plan to be 23 tests.

`import()` also sets the `exported_to()` attribute of your builder to be the caller of the `import()` function.

Additional behaviors can be added to your `import()` method by overriding `import_extra()`.

import_extra

```
Your::Module->import_extra(\@import_args);
```

`import_extra()` is called by `import()`. It provides an opportunity for you to add behaviors to your module based on its import list.

Any extra arguments which shouldn't be passed on to `plan()` should be stripped off by this method.

See *Test::More* for an example of its use.

NOTE This mechanism is *VERY ALPHA AND LIKELY TO CHANGE* as it feels like a bit of an ugly hack in its current form.

Builder

Test::Builder::Module provides some methods of getting at the underlying Test::Builder object.

builder

```
my $builder = Your::Class->builder;
```

This method returns the *Test::Builder* object associated with Your::Class. It is not a constructor so you can call it as often as you like.

This is the preferred way to get the *Test::Builder* object. You should *not* get it via `Test::Builder->new` as was previously recommended.

The object returned by `builder()` may change at runtime so you should call `builder()` inside each function rather than store it in a global.

```
sub ok {
    my $builder = Your::Class->builder;

    return $builder->ok(@_);
}
```

SEE ALSO

Test2::Manual::Tooling::TestBuilder describes the improved options for writing testing modules provided by *Test2*.

Test::Builder::Tester

test testsuites that have been built with Test::Builder

SYNOPSIS

```
use Test::Builder::Tester tests => 1;
use Test::More;

test_out("not ok 1 - foo");
test_fail(+1);
fail("foo");
test_test("fail works");
```

DESCRIPTION

A module that helps you test testing modules that are built with *Test::Builder*.

The testing system is designed to be used by performing a three step process for each test you wish to test. This process starts with using `test_out` and `test_err` in advance to declare what the testsuite you are testing will output with *Test::Builder* to stdout and stderr.

You then can run the test(s) from your test suite that call *Test::Builder*. At this point the output of *Test::Builder* is safely captured by *Test::Builder::Tester* rather than being interpreted as real test output.

The final stage is to call `test_test` that will simply compare what you predeclared to what *Test::Builder* actually outputted, and report the results back with a "ok" or "not ok" (with debugging) to the normal output.

Functions

These are the six methods that are exported as default.

- test_out
- test_err

  Procedures for predeclaring the output that your test suite is expected to produce until `test_test` is called. These procedures automatically assume that each line terminates with "\n". So

  ```
  test_out("ok 1","ok 2");
  ```

  is the same as

```
test_out("ok 1\nok 2");
```

which is even the same as

```
test_out("ok 1");
test_out("ok 2");
```

Once `test_out` or `test_err` (or `test_fail` or `test_diag`) have been called, all further output from *Test::Builder* will be captured by *Test::Builder::Tester*. This means that you will not be able perform further tests to the normal output in the normal way until you call `test_test` (well, unless you manually meddle with the output filehandles)

- test_fail

  Because the standard failure message that *Test::Builder* produces whenever a test fails will be a common occurrence in your test error output, and because it has changed between Test::Builder versions, rather than forcing you to call `test_err` with the string all the time like so

  ```
  test_err("# Failed test ($0 at line ".line_num(+1).")");
  ```

  `test_fail` exists as a convenience function that can be called instead. It takes one argument, the offset from the current line that the line that causes the fail is on.

  ```
  test_fail(+1);
  ```

  This means that the example in the synopsis could be rewritten more simply as:

  ```
  test_out("not ok 1 - foo");
  test_fail(+1);
  fail("foo");
  test_test("fail works");
  ```

- test_diag

  As most of the remaining expected output to the error stream will be created by *Test::Builder*'s `diag` function, *Test::Builder::Tester* provides a convenience function `test_diag` that you can use instead of `test_err`.

  The `test_diag` function prepends comment hashes and spacing to the start and newlines to the end of the expected output passed to it and adds it to the list of expected error output. So, instead of writing

  ```
  test_err("# Couldn't open file");
  ```

  you can write

  ```
  test_diag("Couldn't open file");
  ```

  Remember that *Test::Builder*'s diag function will not add newlines to the end of output and test_diag will. So to check

```
Test::Builder->new->diag("foo\n","bar\n");
```

You would do

```
test_diag("foo","bar")
```

without the newlines.

- test_test

  Actually performs the output check testing the tests, comparing the data (with eq) that we have captured from *Test::Builder* against what was declared with `test_out` and `test_err`.

  This takes name/value pairs that affect how the test is run.

  - title (synonym 'name', 'label')

    The name of the test that will be displayed after the ok or `not ok`.

  - skip_out

    Setting this to a true value will cause the test to ignore if the output sent by the test to the output stream does not match that declared with `test_out`.

  - skip_err

    Setting this to a true value will cause the test to ignore if the output sent by the test to the error stream does not match that declared with `test_err`.

  As a convenience, if only one argument is passed then this argument is assumed to be the name of the test (as in the above examples.)

  Once `test_test` has been run test output will be redirected back to the original filehandles that *Test::Builder* was connected to (probably STDOUT and STDERR,) meaning any further tests you run will function normally and cause success/errors for *Test::Harness*.

- line_num

  A utility function that returns the line number that the function was called on. You can pass it an offset which will be added to the result. This is very useful for working out the correct text of diagnostic functions that contain line numbers.

  Essentially this is the same as the `__LINE__` macro, but the `line_num(+3)` idiom is arguably nicer.

In addition to the six exported functions there exists one function that can only be accessed with a fully qualified function call.

- color

  When `test_test` is called and the output that your tests generate does not match that which you declared, `test_test` will print out debug information showing the two conflicting versions. As this output itself is debug information it can be confusing which part of the output is from `test_test` and which was the original output from your original tests. Also, it may be hard to spot things like extraneous whitespace at the end of lines that may cause your test to fail even though the output looks similar.

To assist you `test_test` can colour the background of the debug information to disambiguate the different types of output. The debug output will have its background coloured green and red. The green part represents the text which is the same between the executed and actual output, the red shows which part differs.

The `color` function determines if colouring should occur or not. Passing it a true or false value will enable or disable colouring respectively, and the function called with no argument will return the current setting.

To enable colouring from the command line, you can use the Text::Builder::Tester::Color module like so:

```
perl -Mlib=Text::Builder::Tester::Color test.t
```

Or by including the *Test::Builder::Tester::Color* module directly in the PERL5LIB.

BUGS

Test::Builder::Tester does not handle plans well. It has never done anything special with plans. This means that plans from outside Test::Builder::Tester will affect Test::Builder::Tester, worse plans when using Test::Builder::Tester will affect overall testing. At this point there are no plans to fix this bug as people have come to depend on it, and Test::Builder::Tester is now discouraged in favor of `Test2::API::intercept()`. See https://github.com/Test-More/test-more/issues/667

Calls `Test::Builder->no_ending` turning off the ending tests. This is needed as otherwise it will trip out because we've run more tests than we strictly should have and it'll register any failures we had that we were testing for as real failures.

The color function doesn't work unless *Term::ANSIColor* is compatible with your terminal. Additionally, Win32::Console::ANSI must be installed on windows platforms for color output.

Bugs (and requests for new features) can be reported to the author though GitHub: https://github.com/Test-More/test-more/issues

AUTHOR

Copyright Mark Fowler mark@twoshortplanks.com 2002, 2004.

Some code taken from *Test::More* and Test::Catch, written by Michael G Schwern schwern@pobox.com. Hence, those parts Copyright Micheal G Schwern 2001. Used and distributed with permission.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

MAINTAINERS

- Chad Granum exodist@cpan.org

NOTES

Thanks to Richard Clamp richardc@unixbeard.net for letting me use his testing system to try this module out on.

SEE ALSO

Test::Builder, *Test::Builder::Tester::Color*, *Test::More*.

Modules

- `Test::Builder::Tester::Color` - turn on colour in Test::Builder::Tester

Test::Builder::Tester::Color

turn on colour in Test::Builder::Tester

SYNOPSIS

```
When running a test script

  perl -MTest::Builder::Tester::Color test.t
```

DESCRIPTION

Importing this module causes the subroutine color in Test::Builder::Tester to be called with a true value causing colour highlighting to be turned on in debug output.

The sole purpose of this module is to enable colour highlighting from the command line.

AUTHOR

Copyright Mark Fowler mark@twoshortplanks.com 2002.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

BUGS

This module will have no effect unless Term::ANSIColor is installed.

SEE ALSO

Test::Builder::Tester, *Term::ANSIColor*

Test::Builder::TodoDiag

Test::Builder subclass of Test2::Event::Diag

DESCRIPTION

This is used to encapsulate diag messages created inside TODO.

SYNOPSIS

You do not need to use this directly.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test::Harness

Run Perl standard test scripts with statistics

VERSION

Version 3.52

SYNOPSIS

```
use Test::Harness;

runtests(@test_files);
```

DESCRIPTION

Although, for historical reasons, the *Test::Harness* distribution takes its name from this module it now exists only to provide *TAP::Harness* with an interface that is somewhat backwards compatible with *Test::Harness* 2.xx. If you're writing new code consider using *TAP::Harness* directly instead.

Emulation is provided for `runtests` and `execute_tests` but the pluggable 'Straps' interface that previous versions of *Test::Harness* supported is not reproduced here. Straps is now available as a stand alone module: Test::Harness::Straps.

See *TAP::Parser*, *TAP::Harness* for the main documentation for this distribution.

FUNCTIONS

The following functions are available.

runtests(@test_files)

This runs all the given *@test_files* and divines whether they passed or failed based on their output to STDOUT (details above). It prints out each individual test which failed along with a summary report and a how long it all took.

It returns true if everything was ok. Otherwise it will `die()` with one of the messages in the DIAGNOSTICS section.

execute_tests(tests => \@test_files, out => *FH)

Runs all the given `@test_files` (just like `runtests()`) but doesn't generate the final report. During testing, progress information will be written to the currently selected output filehandle (usually `STDOUT`), or to the filehandle given by the `out` parameter. The *out* is optional.

Returns a list of two values, `$total` and `$failed`, describing the results. `$total` is a hash ref summary of all the tests run. Its keys and values are this:

```
bonus           Number of individual todo tests unexpectedly passed
max             Number of individual tests ran
ok              Number of individual tests passed
sub_skipped     Number of individual tests skipped
todo            Number of individual todo tests

files           Number of test files ran
good            Number of test files passed
bad             Number of test files failed
tests           Number of test files originally given
skipped         Number of test files skipped
```

If `$total->{bad} == 0` and `$total->{max} > 0`, you've got a successful test.

`$failed` is a hash ref of all the test scripts that failed. Each key is the name of a test script, each value is another hash representing how that script failed. Its keys are these:

```
name        Name of the test which failed
estat       Script's exit value
wstat       Script's wait status
max         Number of individual tests
failed      Number which failed
canon       List of tests which failed (as string).
```

`$failed` should be empty if everything passed.

EXPORT

`&runtests` is exported by `Test::Harness` by default.

`&execute_tests`, `$verbose`, `$switches` and `$debug` are exported upon request.

ENVIRONMENT VARIABLES THAT TAP::HARNESS::COMPATIBLE SETS

`Test::Harness` sets these before executing the individual tests.

- `HARNESS_ACTIVE`

  This is set to a true value. It allows the tests to determine if they are being executed through the harness or by any other means.

- `HARNESS_VERSION`

  This is the version of `Test::Harness`.

ENVIRONMENT VARIABLES THAT AFFECT TEST::HARNESS

- `HARNESS_PERL_SWITCHES`

  Setting this adds perl command line switches to each test file run.

  For example, `HARNESS_PERL_SWITCHES=-T` will turn on taint mode. `HARNESS_PERL_SWITCHES=-MDevel::Cover` will run `Devel::Cover` for each test.

  `-w` is always set. You can turn this off in the test with `BEGIN { $^W = 0 }`.

- `HARNESS_TIMER`

  Setting this to true will make the harness display the number of milliseconds each test took. You can also use `prove`'s `--timer` switch.

- `HARNESS_VERBOSE`

  If true, `Test::Harness` will output the verbose results of running its tests. Setting `$Test::Harness::verbose` will override this, or you can use the `-v` switch in the `prove` utility.

- `HARNESS_OPTIONS`

  Provide additional options to the harness. Currently supported options are:

  – `j<n>`

    Run <n> (default 9) parallel jobs.

  – `c`

    Try to color output. See *"new" in TAP::Formatter::Base*.

  – `a<file.tgz>`

    Will use TAP::Harness::Archive as the harness class, and save the TAP to `file.tgz`

- `fPackage-With-Dashes`

  Set the formatter_class of the harness being run. Since the `HARNESS_OPTIONS` is separated by `:`, we use `-` instead.

Multiple options may be separated by colons:

```
HARNESS_OPTIONS=j9:c make test
```

- `HARNESS_SUBCLASS`

  Specifies a TAP::Harness subclass to be used in place of TAP::Harness.

- `HARNESS_SUMMARY_COLOR_SUCCESS`

  Determines the *Term::ANSIColor* for the summary in case it is successful. This color defaults to `'green'`.

- `HARNESS_SUMMARY_COLOR_FAIL`

  Determines the *Term::ANSIColor* for the failure in case it is successful. This color defaults to `'red'`.

Taint Mode

Normally when a Perl program is run in taint mode the contents of the `PERL5LIB` environment variable do not appear in `@INC`.

Because `PERL5LIB` is often used during testing to add build directories to `@INC` `Test::Harness` passes the names of any directories found in `PERL5LIB` as -I switches. The net effect of this is that `PERL5LIB` is honoured even in taint mode.

SEE ALSO

TAP::Harness

BUGS

Please report any bugs or feature requests to `bug-test-harness at rt.cpan.org`, or through the web interface at http://rt.cpan.org/NoAuth/ReportBug.html?Queue=Test-Harness. I will be notified, and then you'll automatically be notified of progress on your bug as I make changes.

AUTHORS

Andy Armstrong `<andy@hexten.net>`

Test::Harness 2.64 (maintained by Andy Lester and on which this module is based) has this attribution:

```
Either Tim Bunce or Andreas Koenig, we don't know. What we know for
sure is, that it was inspired by Larry Wall's F<TEST> script that␣
↪came
with perl distributions for ages. Numerous anonymous contributors
```

(continues on next page)

(continued from previous page)

```
exist.  Andreas Koenig held the torch for many years, and then
Michael G Schwern.
```

LICENCE AND COPYRIGHT

Copyright (c) 2007-2011, Andy Armstrong <andy@hexten.net>. All rights reserved.

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself. See perlartistic.

Test::LeakTrace

Find the scalars, arrays, hashes and code a block of Perl leaves behind.

Each of the functions runs a block and reports what it allocated and did not free: the values themselves, a count, or a report naming the file and line that created each one. The test functions turn that into a test, so a leak becomes a failing test.

Synopsis

```
use Test::More;
use Test::LeakTrace;

no_leaks_ok { my @a = (1 .. 10); my %h = (a => 1) } 'no leaks';

leaks_cmp_ok { my $x; $x = \$x } '<', 2, 'one cycle at most';

my $count = leaked_count { my $x; $x = \$x };      # 1
my @refs  = leaked_refs  { my $x; $x = \$x };      # (\$x)
my @info  = leaked_info  { my $x; $x = \$x };      # ([\$x, $file,
 ↪$line])

leaktrace { my $x; $x = \$x } -verbose;            # report on␣
 ↪STDERR
done_testing;
```

Reporting modes

`leaktrace BLOCK MODE` reports each leaked value on STDERR. `-simple`, the default, prints its type, address and the line that created it; `-lines` adds the source lines around that line; `-sv_dump` dumps the value as `Devel::Peek` does; `-verbose` is `-lines` and `-sv_dump` together; `-silent` prints nothing. A code reference as MODE is called once per leaked value with a reference to it, the file and the line.

`Test::LeakTrace::Script` runs the whole program under a trace: `perl -MTest::LeakTrace::Script=-verbose script.pl`, or the mode in `TEST_LEAKTRACE`.

This is a native implementation of the XS half of Test::LeakTrace 0.17; its `Test/LeakTrace.pm` and `Test/LeakTrace/Script.pm` are used unchanged. Everything

listed above is exported by default.

Differences from upstream

None.

Modules

- `Test::LeakTrace::Script` - A LeakTrace interface for whole scripts

Test::LeakTrace::Script

A LeakTrace interface for whole scripts

SYNOPSIS

```
#!perl -w
use Test::LeakTrace::Script sub{
        my($svref, $file, $line) = @_;

        warn "leaked $svref from $file line $line.\n";
};
```

DESCRIPTION

This is a interface to `Test::LeakTrace` for whole scripts.

INTERFACE

Command line interface

```
$ perl -MTest::LeakTrace::Script script.pl

$ perl -MTest::LeakTrace::Script=-verbose script.pl

$ TEST_LEAKTRACE=-lines script.pl
```

ENVIRONMENT VARIABLES

TEST_LEAKTRACE=mode

-simple (DEFAULT)

-sv_dump

-lines

-verbose

SEE ALSO

Test::LeakTrace.

Test::More

yet another framework for writing test scripts

SYNOPSIS

```
use Test::More tests => 23;
# or
use Test::More skip_all => $reason;
# or
use Test::More;   # see done_testing()

require_ok( 'Some::Module' );

# Various ways to say "ok"
ok($got eq $expected, $test_name);

is  ($got, $expected, $test_name);
isnt($got, $expected, $test_name);

# Rather than print STDERR "# here's what went wrong\n"
diag("here's what went wrong");

like  ($got, qr/expected/, $test_name);
unlike($got, qr/expected/, $test_name);

cmp_ok($got, '==', $expected, $test_name);

is_deeply($got_complex_structure, $expected_complex_structure,
 ↪$test_name);

SKIP: {
    skip $why, $how_many unless $have_some_feature;

    ok( foo(),       $test_name );
    is( foo(42), 23, $test_name );
};

TODO: {
    local $TODO = $why;

    ok( foo(),       $test_name );
    is( foo(42), 23, $test_name );
};

can_ok($module, @methods);
```

(continues on next page)

(continued from previous page)

```
isa_ok($object, $class);

pass($test_name);
fail($test_name);

BAIL_OUT($why);

# UNIMPLEMENTED!!!
my @status = Test::More::status;
```

DESCRIPTION

STOP! If you're just getting started writing tests, have a look at *Test2::Suite* first.

This is a drop in replacement for Test::Simple which you can switch to once you get the hang of basic testing.

The purpose of this module is to provide a wide range of testing utilities. Various ways to say "ok" with better diagnostics, facilities to skip tests, test future features and compare complicated data structures. While you can do almost anything with a simple ok() function, it doesn't provide good diagnostic output.

I love it when a plan comes together

Before anything else, you need a testing plan. This basically declares how many tests your script is going to run to protect against premature failure.

The preferred way to do this is to declare a plan when you `use Test::More`.

```
use Test::More tests => 23;
```

There are cases when you will not know beforehand how many tests your script is going to run. In this case, you can declare your tests at the end.

```
use Test::More;

... run your tests ...

done_testing( $number_of_tests_run );
```

NOTE `done_testing()` should never be called in an `END { ... }` block.

Sometimes you really don't know how many tests were run, or it's too difficult to calculate. In which case you can leave off $number_of_tests_run.

In some cases, you'll want to completely skip an entire testing script.

```
use Test::More skip_all => $skip_reason;
```

Your script will declare a skip with the reason why you skipped and exit immediately with a zero (success). See *Test::Harness* for details.

If you want to control what functions Test::More will export, you have to use the 'import' option. For example, to import everything but 'fail', you'd do:

```
use Test::More tests => 23, import => ['!fail'];
```

Alternatively, you can use the `plan()` function. Useful for when you have to calculate the number of tests.

```
use Test::More;
plan tests => keys %Stuff * 3;
```

or for deciding between running the tests at all:

```
use Test::More;
if( $^O eq 'MacOS' ) {
    plan skip_all => 'Test irrelevant on MacOS';
}
else {
    plan tests => 42;
}
```

- **done_testing**

  ```
  done_testing();
  done_testing($number_of_tests);
  ```

  If you don't know how many tests you're going to run, you can issue the plan when you're done running tests.

  $number_of_tests is the same as `plan()`, it's the number of tests you expected to run. You can omit this, in which case the number of tests you ran doesn't matter, just the fact that your tests ran to conclusion.

  This is safer than and replaces the "no_plan" plan.

  Note: You must never put `done_testing()` inside an `END { ... }` block. The plan is there to ensure your test does not exit before testing has completed. If you use an END block you completely bypass this protection.

Test names

By convention, each test is assigned a number in order. This is largely done automatically for you. However, it's often very useful to assign a name to each test. Which would you rather see:

```
ok 4
not ok 5
ok 6
```

or

```
ok 4 - basic multi-variable
not ok 5 - simple exponential
ok 6 - force == mass * acceleration
```

The later gives you some idea of what failed. It also makes it easier to find the test in your script, simply search for "simple exponential".

All test functions take a name argument. It's optional, but highly suggested that you use it.

I'm ok, you're not ok.

The basic purpose of this module is to print out either "ok #" or "not ok #" depending on if a given test succeeded or failed. Everything else is just gravy.

All of the following print "ok" or "not ok" depending on if the test succeeded or failed. They all also return true or false, respectively.

- **ok**

```
ok($got eq $expected, $test_name);
```

  This simply evaluates any expression (`$got eq $expected` is just a simple example) and uses that to determine if the test succeeded or failed. A true expression passes, a false one fails. Very simple.

  For example:

```
ok( $exp{9} == 81,                   'simple exponential' );
ok( Film->can('db_Main'),            'set_db()' );
ok( $p->tests == 4,                  'saw tests' );
ok( !grep(!defined $_, @items),      'all items defined' );
```

  (Mnemonic: "This is ok.")

  $test_name is a very short description of the test that will be printed out. It makes it very easy to find a test in your script when it fails and gives others an idea of your intentions. $test_name is optional, but we **very** strongly encourage its use.

  Should an `ok()` fail, it will produce some diagnostics:

```
not ok 18 - sufficient mucus
#   Failed test 'sufficient mucus'
#   in foo.t at line 42.
```

  This is the same as *Test::Simple*'s `ok()` routine.

- **is**
- **isnt**

```
is  ( $got, $expected, $test_name );
isnt( $got, $expected, $test_name );
```

  Similar to `ok()`, `is()` and `isnt()` compare their two arguments with `eq` and `ne` respectively and use the result of that to determine if the test succeeded or failed. So these:

```
# Is the ultimate answer 42?
is( ultimate_answer(), 42,          "Meaning of Life" );

# $foo isn't empty
isnt( $foo, '',     "Got some foo" );
```

are similar to these:

```
ok( ultimate_answer() eq 42,        "Meaning of Life" );
ok( $foo ne '',     "Got some foo" );
```

undef will only ever match undef. So you can test a value against undef like this:

```
is($not_defined, undef, "undefined as expected");
```

(Mnemonic: "This is that." "This isn't that.")

So why use these? They produce better diagnostics on failure. ok() cannot know what you are testing for (beyond the name), but is() and isnt() know what the test was and why it failed. For example this test:

```
my $foo = 'waffle';  my $bar = 'yarblokos';
is( $foo, $bar,   'Is foo the same as bar?' );
```

Will produce something like this:

```
not ok 17 - Is foo the same as bar?
#   Failed test 'Is foo the same as bar?'
#   in foo.t at line 139.
#          got: 'waffle'
#     expected: 'yarblokos'
```

So you can figure out what went wrong without rerunning the test.

You are encouraged to use is() and isnt() over ok() where possible, however do not be tempted to use them to find out if something is true or false!

```
# XXX BAD!
is( exists $brooklyn{tree}, 1, 'A tree grows in Brooklyn' );
```

This does not check if exists $brooklyn{tree} is true, it checks if it returns 1. Very different. Similar caveats exist for false and 0. In these cases, use ok().

```
ok( exists $brooklyn{tree},    'A tree grows in Brooklyn' );
```

A simple call to isnt() usually does not provide a strong test but there are cases when you cannot say much more about a value than that it is different from some other value:

```
new_ok $obj, "Foo";

my $clone = $obj->clone;
isa_ok $obj, "Foo", "Foo->clone";
```

(continues on next page)

(continued from previous page)

```
isnt $obj, $clone, "clone() produces a different object";
```

Historically we supported an `isn't()` function as an alias of `isnt()`, however in Perl 5.37.9 support for the use of apostrophe as a package separator was deprecated and by Perl 5.42.0 support for it will have been removed completely. Accordingly use of `isn't()` is also deprecated, and will produce warnings when used unless 'deprecated' warnings are specifically disabled in the scope where it is used. You are strongly advised to migrate to using `isnt()` instead.

- **like**

```
like( $got, qr/expected/, $test_name );
```

Similar to `ok()`, `like()` matches $got against the regex `qr/expected/`.

So this:

```
like($got, qr/expected/, 'this is like that');
```

is similar to:

```
ok( $got =~ m/expected/, 'this is like that');
```

(Mnemonic "This is like that".)

The second argument is a regular expression. It may be given as a regex reference (i.e. `qr//`) or (for better compatibility with older perls) as a string that looks like a regex (alternative delimiters are currently not supported):

```
like( $got, '/expected/', 'this is like that' );
```

Regex options may be placed on the end (`'/expected/i'`).

Its advantages over `ok()` are similar to that of `is()` and `isnt()`. Better diagnostics on failure.

- **unlike**

```
unlike( $got, qr/expected/, $test_name );
```

Works exactly as `like()`, only it checks if $got **does not** match the given pattern.

- **cmp_ok**

```
cmp_ok( $got, $op, $expected, $test_name );
```

Halfway between `ok()` and `is()` lies `cmp_ok()`. This allows you to compare two arguments using any binary perl operator. The test passes if the comparison is true and fails otherwise.

```
# ok( $got eq $expected );
cmp_ok( $got, 'eq', $expected, 'this eq that' );
```

(continues on next page)

(continued from previous page)

```
# ok( $got == $expected );
cmp_ok( $got, '==', $expected, 'this == that' );

# ok( $got && $expected );
cmp_ok( $got, '&&', $expected, 'this && that' );
...etc...
```

Its advantage over `ok()` is when the test fails you'll know what $got and $expected were:

```
not ok 1
#   Failed test in foo.t at line 12.
#     '23'
#         &&
#     undef
```

It's also useful in those cases where you are comparing numbers and `is()`'s use of eq will interfere:

```
cmp_ok( $big_hairy_number, '==', $another_big_hairy_number );
```

It's especially useful when comparing greater-than or smaller-than relation between values:

```
cmp_ok( $some_value, '<=', $upper_limit );
```

- **can_ok**

```
can_ok($module, @methods);
can_ok($object, @methods);
```

Checks to make sure the $module or $object can do these @methods (works with functions, too).

```
can_ok('Foo', qw(this that whatever));
```

is almost exactly like saying:

```
ok( Foo->can('this') &&
    Foo->can('that') &&
    Foo->can('whatever')
  );
```

only without all the typing and with a better interface. Handy for quickly testing an interface.

No matter how many @methods you check, a single `can_ok()` call counts as one test. If you desire otherwise, use:

```
foreach my $meth (@methods) {
    can_ok('Foo', $meth);
}
```

- **isa_ok**

```
isa_ok($object,   $class, $object_name);
isa_ok($subclass, $class, $object_name);
isa_ok($ref,      $type,  $ref_name);
```

Checks to see if the given `$object->isa($class)`. Also checks to make sure the object was defined in the first place. Handy for this sort of thing:

```
my $obj = Some::Module->new;
isa_ok( $obj, 'Some::Module' );
```

where you'd otherwise have to write

```
my $obj = Some::Module->new;
ok( defined $obj && $obj->isa('Some::Module') );
```

to safeguard against your test script blowing up.

You can also test a class, to make sure that it has the right ancestor:

```
isa_ok( 'Vole', 'Rodent' );
```

It works on references, too:

```
isa_ok( $array_ref, 'ARRAY' );
```

The diagnostics of this test normally just refer to 'the object'. If you'd like them to be more specific, you can supply an $object_name (for example 'Test customer').

- **new_ok**

```
my $obj = new_ok( $class );
my $obj = new_ok( $class => \@args );
my $obj = new_ok( $class => \@args, $object_name );
```

A convenience function which combines creating an object and calling `isa_ok()` on that object.

It is basically equivalent to:

```
my $obj = $class->new(@args);
isa_ok $obj, $class, $object_name;
```

If @args is not given, an empty list will be used.

This function only works on `new()` and it assumes `new()` will return just a single object which isa `$class`.

- **subtest**

```
subtest $name => \&code, @args;
```

`subtest()` runs the &code as its own little test with its own plan and its own result. The main test counts this as a single test using the result of the whole subtest to determine if its ok or not ok.

For example...

```
use Test::More tests => 3;

pass("First test");

subtest 'An example subtest' => sub {
    plan tests => 2;

    pass("This is a subtest");
    pass("So is this");
};

pass("Third test");
```

This would produce.

```
1..3
ok 1 - First test
    # Subtest: An example subtest
    1..2
    ok 1 - This is a subtest
    ok 2 - So is this
ok 2 - An example subtest
ok 3 - Third test
```

A subtest may call `skip_all`. No tests will be run, but the subtest is considered a skip.

```
subtest 'skippy' => sub {
    plan skip_all => 'cuz I said so';
    pass('this test will never be run');
};
```

Returns true if the subtest passed, false otherwise.

Due to how subtests work, you may omit a plan if you desire. This adds an implicit `done_testing()` to the end of your subtest. The following two subtests are equivalent:

```
subtest 'subtest with implicit done_testing()', sub {
    ok 1, 'subtests with an implicit done testing should work';
    ok 1, '... and support more than one test';
    ok 1, '... no matter how many tests are run';
};

subtest 'subtest with explicit done_testing()', sub {
    ok 1, 'subtests with an explicit done testing should work';
    ok 1, '... and support more than one test';
    ok 1, '... no matter how many tests are run';
    done_testing();
};
```

Extra arguments given to `subtest` are passed to the callback. For example:

```
sub my_subtest {
    my $range = shift;
    ...
}

for my $range (1, 10, 100, 1000) {
    subtest "testing range $range", \&my_subtest, $range;
}
```

- **pass**
- **fail**

```
pass($test_name);
fail($test_name);
```

Sometimes you just want to say that the tests have passed. Usually the case is you've got some complicated condition that is difficult to wedge into an `ok()`. In this case, you can simply use `pass()` (to declare the test ok) or fail (for not ok). They are synonyms for `ok(1)` and `ok(0)`.

Use these very, very, very sparingly.

Module tests

Sometimes you want to test if a module, or a list of modules, can successfully load. For example, you'll often want a first test which simply loads all the modules in the distribution to make sure they work before going on to do more complicated testing.

For such purposes we have `use_ok` and `require_ok`.

- **require_ok**

```
require_ok($module);
require_ok($file);
```

Tries to `require` the given $module or $file. If it loads successfully, the test will pass. Otherwise it fails and displays the load error.

`require_ok` will guess whether the input is a module name or a filename.

No exception will be thrown if the load fails.

```
# require Some::Module
require_ok "Some::Module";

# require "Some/File.pl";
require_ok "Some/File.pl";

# stop testing if any of your modules will not load
for my $module (@module) {
    require_ok $module or BAIL_OUT "Can't load $module";
}
```

- **use_ok**

```
BEGIN { use_ok($module); }
BEGIN { use_ok($module, @imports); }
```

Like `require_ok`, but it will `use` the $module in question and only loads modules, not files.

If you just want to test a module can be loaded, use `require_ok`.

If you just want to load a module in a test, we recommend simply using `use` directly. It will cause the test to stop.

It's recommended that you run `use_ok()` inside a BEGIN block so its functions are exported at compile-time and prototypes are properly honored.

If @imports are given, they are passed through to the use. So this:

```
BEGIN { use_ok('Some::Module', qw(foo bar)) }
```

is like doing this:

```
use Some::Module qw(foo bar);
```

Version numbers can be checked like so:

```
# Just like "use Some::Module 1.02"
BEGIN { use_ok('Some::Module', 1.02) }
```

Don't try to do this:

```
BEGIN {
    use_ok('Some::Module');

    ...some code that depends on the use...
    ...happening at compile time...
}
```

because the notion of "compile-time" is relative. Instead, you want:

```
BEGIN { use_ok('Some::Module') }
BEGIN { ...some code that depends on the use... }
```

If you want the equivalent of `use Foo ()`, use a module but not import anything, use `require_ok`.

```
BEGIN { require_ok "Foo" }
```

Complex data structures

Not everything is a simple eq check or regex. There are times you need to see if two data structures are equivalent. For these instances Test::More provides a handful of useful functions.

NOTE I'm not quite sure what will happen with filehandles.

- **is_deeply**

```
is_deeply( $got, $expected, $test_name );
```

Similar to `is()`, except that if $got and $expected are references, it does a deep comparison walking each data structure to see if they are equivalent. If the two structures are different, it will display the place where they start differing.

`is_deeply()` compares the dereferenced values of references, the references themselves (except for their type) are ignored. This means aspects such as blessing and ties are not considered "different".

`is_deeply()` currently has very limited handling of function reference and globs. It merely checks if they have the same referent. This may improve in the future.

Test::Differences and Test::Deep provide more in-depth functionality along these lines.

NOTE is_deeply() has limitations when it comes to comparing strings and refs:

```
my $path = path('.');
my $hash = {};
is_deeply( $path, "$path" ); # ok
is_deeply( $hash, "$hash" ); # fail
```

This happens because is_deeply will unoverload all arguments unconditionally. It is probably best not to use is_deeply with overloading. For legacy reasons this is not likely to ever be fixed. If you would like a much better tool for this you should see *Test2::Suite* Specifically *Test2::Tools::Compare* has an `is()` function that works like `is_deeply` with many improvements.

Diagnostics

If you pick the right test function, you'll usually get a good idea of what went wrong when it failed. But sometimes it doesn't work out that way. So here we have ways for you to write your own diagnostic messages which are safer than just `print STDERR`.

- **diag**

```
diag(@diagnostic_message);
```

Prints a diagnostic message which is guaranteed not to interfere with test output. Like `print` @diagnostic_message is simply concatenated together.

Returns false, so as to preserve failure.

Handy for this sort of thing:

```
ok( grep(/foo/, @users), "There's a foo user" ) or
    diag("Since there's no foo, check that /etc/bar is set up␣
↪right");
```

which would produce:

```
not ok 42 - There's a foo user
#   Failed test 'There's a foo user'
#   in foo.t at line 52.
# Since there's no foo, check that /etc/bar is set up right.
```

You might remember `ok() or diag()` with the mnemonic `open() or die()`.

NOTE The exact formatting of the diagnostic output is still changing, but it is guaranteed that whatever you throw at it won't interfere with the test.

- **note**

```
note(@diagnostic_message);
```

Like `diag()`, except the message will not be seen when the test is run in a harness. It will only be visible in the verbose TAP stream.

Handy for putting in notes which might be useful for debugging, but don't indicate a problem.

```
note("Tempfile is $tempfile");
```

- **explain**

```
my @dump = explain @diagnostic_message;
```

Will dump the contents of any references in a human readable format. Usually you want to pass this into `note` or `diag`.

Handy for things like...

```
is_deeply($have, $want) || diag explain $have;
```

or

```
note explain \%args;
Some::Class->method(%args);
```

Conditional tests

Sometimes running a test under certain conditions will cause the test script to die. A certain function or method isn't implemented (such as `fork()` on MacOS), some resource isn't available (like a net connection) or a module isn't available. In these cases it's necessary to skip tests, or declare that they are supposed to fail but will work in the future (a todo test).

For more details on the mechanics of skip and todo tests see *Test::Harness*.

The way Test::More handles this is with a named block. Basically, a block of tests which can be skipped over or made todo. It's best if I just show you...

- **SKIP: BLOCK**

```
SKIP: {
    skip $why, $how_many if $condition;

    ...normal testing code goes here...
}
```

This declares a block of tests that might be skipped, $how_many tests there are, $why and under what $condition to skip them. An example is the easiest way to illustrate:

```
SKIP: {
    eval { require HTML::Lint };

    skip "HTML::Lint not installed", 2 if $@;

    my $lint = new HTML::Lint;
    isa_ok( $lint, "HTML::Lint" );

    $lint->parse( $html );
    is( $lint->errors, 0, "No errors found in HTML" );
}
```

If the user does not have HTML::Lint installed, the whole block of code *won't be run at all.* Test::More will output special ok's which Test::Harness interprets as skipped, but passing, tests.

It's important that $how_many accurately reflects the number of tests in the SKIP block so the # of tests run will match up with your plan. If your plan is `no_plan` $how_many is optional and will default to 1.

It's perfectly safe to nest SKIP blocks. Each SKIP block must have the label `SKIP`, or Test::More can't work its magic.

You don't skip tests which are failing because there's a bug in your program, or for which you don't yet have code written. For that you use TODO. Read on.

- **TODO: BLOCK**

```
TODO: {
    local $TODO = $why if $condition;

    ...normal testing code goes here...
}
```

Declares a block of tests you expect to fail and $why. Perhaps it's because you haven't fixed a bug or haven't finished a new feature:

```
TODO: {
    local $TODO = "URI::Geller not finished";

    my $card = "Eight of clubs";
    is( URI::Geller->your_card, $card, 'Is THIS your card?' );
```

(continues on next page)

(continued from previous page)

```
    my $spoon;
    URI::Geller->bend_spoon;
    is( $spoon, 'bent',    "Spoon bending, that's original" );
}
```

With a todo block, the tests inside are expected to fail. Test::More will run the tests normally, but print out special flags indicating they are "todo". *Test::Harness* will interpret failures as being ok. Should anything succeed, it will report it as an unexpected success. You then know the thing you had todo is done and can remove the TODO flag.

The nice part about todo tests, as opposed to simply commenting out a block of tests, is that it is like having a programmatic todo list. You know how much work is left to be done, you're aware of what bugs there are, and you'll know immediately when they're fixed.

Once a todo test starts succeeding, simply move it outside the block. When the block is empty, delete it.

Note that, if you leave $TODO unset or undef, Test::More reports failures as normal. This can be useful to mark the tests as expected to fail only in certain conditions, e.g.:

```
TODO: {
    local $TODO = "$^O doesn't work yet. :(" if !_os_is_
↪supported($^O);

    ...
}
```

- **todo_skip**

```
TODO: {
    todo_skip $why, $how_many if $condition;

    ...normal testing code...
}
```

With todo tests, it's best to have the tests actually run. That way you'll know when they start passing. Sometimes this isn't possible. Often a failing test will cause the whole program to die or hang, even inside an `eval BLOCK` with and using `alarm`. In these extreme cases you have no choice but to skip over the broken tests entirely.

The syntax and behavior is similar to a `SKIP: BLOCK` except the tests will be marked as failing but todo. *Test::Harness* will interpret them as passing.

- When do I use SKIP vs. TODO?

  If it's something the user might not be able to do, use SKIP. This includes optional modules that aren't installed, running under an OS that doesn't have some feature (like `fork()` or symlinks), or maybe you need an Internet connection and one isn't available.

  If it's something the programmer hasn't done yet, use TODO. This is for any code

you haven't written yet, or bugs you have yet to fix, but want to put tests in your testing script (always a good idea).

Test control

- **BAIL_OUT**

```
BAIL_OUT($reason);
```

Indicates to the harness that things are going so badly all testing should terminate. This includes the running of any additional test scripts.

This is typically used when testing cannot continue such as a critical module failing to compile or a necessary external utility not being available such as a database connection failing.

The test will exit with 255.

For even better control look at Test::Most.

Discouraged comparison functions

The use of the following functions is discouraged as they are not actually testing functions and produce no diagnostics to help figure out what went wrong. They were written before `is_deeply()` existed because I couldn't figure out how to display a useful diff of two arbitrary data structures.

These functions are usually used inside an `ok()`.

```
ok( eq_array(\@got, \@expected) );
```

`is_deeply()` can do that better and with diagnostics.

```
is_deeply( \@got, \@expected );
```

They may be deprecated in future versions.

- **eq_array**

```
my $is_eq = eq_array(\@got, \@expected);
```

Checks if two arrays are equivalent. This is a deep check, so multi-level structures are handled correctly.

- **eq_hash**

```
my $is_eq = eq_hash(\%got, \%expected);
```

Determines if the two hashes contain the same keys and values. This is a deep check.

- **eq_set**

```
my $is_eq = eq_set(\@got, \@expected);
```

Similar to `eq_array()`, except the order of the elements is **not** important. This is a deep check, but the irrelevancy of order only applies to the top level.

```
ok( eq_set(\@got, \@expected) );
```

Is better written:

```
is_deeply( [sort @got], [sort @expected] );
```

NOTE By historical accident, this is not a true set comparison. While the order of elements does not matter, duplicate elements do.

NOTE `eq_set()` does not know how to deal with references at the top level. The following is an example of a comparison which might not work:

```
eq_set([\1, \2], [\2, \1]);
```

Test::Deep contains much better set comparison functions.

Extending and Embedding Test::More

Sometimes the Test::More interface isn't quite enough. Fortunately, Test::More is built on top of *Test::Builder* which provides a single, unified backend for any test library to use. This means two test libraries which both use *Test::Builder* **can** be used together in the same program.

If you simply want to do a little tweaking of how the tests behave, you can access the underlying *Test::Builder* object like so:

- **builder**

```
my $test_builder = Test::More->builder;
```

Returns the *Test::Builder* object underlying Test::More for you to play with.

EXIT CODES

If all your tests passed, *Test::Builder* will exit with zero (which is normal). If anything failed it will exit with how many failed. If you run less (or more) tests than you planned, the missing (or extras) will be considered failures. If no tests were ever run *Test::Builder* will throw a warning and exit with 255. If the test died, even after having successfully completed all its tests, it will still be considered a failure and will exit with 255.

So the exit codes are...

```
0                   all tests successful
255                 test died or all passed but wrong # of tests run
any other number    how many failed (including missing or extras)
```

If you fail more than 254 tests, it will be reported as 254.

NOTE This behavior may go away in future versions.

COMPATIBILITY

Test::More works with Perls as old as 5.8.1.

Thread support is not very reliable before 5.10.1, but that's because threads are not very reliable before 5.10.1.

Although Test::More has been a core module in versions of Perl since 5.6.2, Test::More has evolved since then, and not all of the features you're used to will be present in the shipped version of Test::More. If you are writing a module, don't forget to indicate in your package metadata the minimum version of Test::More that you require. For instance, if you want to use `done_testing()` but want your test script to run on Perl 5.10.0, you will need to explicitly require Test::More > 0.88.

Key feature milestones include:

- subtests

  Subtests were released in Test::More 0.94, which came with Perl 5.12.0. Subtests did not implicitly call `done_testing()` until 0.96; the first Perl with that fix was Perl 5.14.0 with 0.98.

- `done_testing()`

  This was released in Test::More 0.88 and first shipped with Perl in 5.10.1 as part of Test::More 0.92.

- `cmp_ok()`

  Although `cmp_ok()` was introduced in 0.40, 0.86 fixed an important bug to make it safe for overloaded objects; the fixed first shipped with Perl in 5.10.1 as part of Test::More 0.92.

- `new_ok()` `note()` and `explain()`

  These were was released in Test::More 0.82, and first shipped with Perl in 5.10.1 as part of Test::More 0.92.

There is a full version history in the Changes file, and the Test::More versions included as core can be found using *Module::CoreList*:

```
$ corelist -a Test::More
```

CAVEATS and NOTES

- UTF-8 / "Wide character in print"

  If you use UTF-8 or other non-ASCII characters with Test::More you might get a "Wide character in print" warning. Using `binmode STDOUT, ":utf8"` will not fix it. *Test::Builder* (which powers Test::More) duplicates STDOUT and STDERR. So any changes to them, including changing their output disciplines, will not be seen by Test::More.

  One work around is to apply encodings to STDOUT and STDERR as early as possible and before Test::More (or any other Test module) loads.

  ```
  use open ':std', ':encoding(UTF-8)';
  use Test::More;
  ```

A more direct work around is to change the filehandles used by *Test::Builder*.

```
my $builder = Test::More->builder;
binmode $builder->output,         ":encoding(UTF-8)";
binmode $builder->failure_output, ":encoding(UTF-8)";
binmode $builder->todo_output,    ":encoding(UTF-8)";
```

- Overloaded objects

  String overloaded objects are compared **as strings** (or in `cmp_ok()`'s case, strings or numbers as appropriate to the comparison op). This prevents Test::More from piercing an object's interface allowing better blackbox testing. So if a function starts returning overloaded objects instead of bare strings your tests won't notice the difference. This is good.

  However, it does mean that functions like `is_deeply()` cannot be used to test the internals of string overloaded objects. In this case I would suggest Test::Deep which contains more flexible testing functions for complex data structures.

- Threads

  Test::More will only be aware of threads if `use threads` has been done *before* Test::More is loaded. This is ok:

  ```
  use threads;
  use Test::More;
  ```

  This may cause problems:

  ```
  use Test::More
  use threads;
  ```

  5.8.1 and above are supported. Anything below that has too many bugs.

HISTORY

This is a case of convergent evolution with Joshua Pritikin's *Test* module. I was largely unaware of its existence when I'd first written my own `ok()` routines. This module exists because I can't figure out how to easily wedge test names into Test's interface (along with a few other problems).

The goal here is to have a testing utility that's simple to learn, quick to use and difficult to trip yourself up with while still providing more flexibility than the existing Test.pm. As such, the names of the most common routines are kept tiny, special cases and magic side-effects are kept to a minimum. WYSIWYG.

SEE ALSO

ALTERNATIVES

Test2::Suite is the most recent and modern set of tools for testing.

Test::Simple if all this confuses you and you just want to write some tests. You can upgrade to Test::More later (it's forward compatible).

Test::Legacy tests written with Test.pm, the original testing module, do not play well with other testing libraries. Test::Legacy emulates the Test.pm interface and does play well with others.

ADDITIONAL LIBRARIES

Test::Differences for more ways to test complex data structures. And it plays well with Test::More.

Test::Class is like xUnit but more perlish.

Test::Deep gives you more powerful complex data structure testing.

Test::Inline shows the idea of embedded testing.

Mock::Quick The ultimate mocking library. Easily spawn objects defined on the fly. Can also override, block, or reimplement packages as needed.

Test::FixtureBuilder Quickly define fixture data for unit tests.

OTHER COMPONENTS

Test::Harness is the test runner and output interpreter for Perl. It's the thing that powers `make test` and where the `prove` utility comes from.

BUNDLES

Test::Most Most commonly needed test functions and features.

ENVIRONMENT VARIABLES

See *Test2::Env* for a list of meaningful environment variables.

AUTHORS

Michael G Schwern schwern@pobox.com with much inspiration from Joshua Pritikin's Test module and lots of help from Barrie Slaymaker, Tony Bowden, blackstar.co.uk, chromatic, Fergal Daly and the perl-qa gang.

MAINTAINERS

- Chad Granum exodist@cpan.org

BUGS

See https://github.com/Test-More/test-more/issues to report and view bugs.

SOURCE

The source code repository for Test::More can be found at https://github.com/Test-More/test-more/.

COPYRIGHT

Copyright 2001-2008 by Michael G Schwern schwern@pobox.com.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test::Simple

Basic utilities for writing tests.

SYNOPSIS

```
use Test::Simple tests => 1;

ok( $foo eq $bar, 'foo is bar' );
```

DESCRIPTION

** If you are unfamiliar with testing **read Test::Tutorial first!** **

This is an extremely simple, extremely basic module for writing tests suitable for CPAN modules and other pursuits. If you wish to do more complicated testing, use the Test::More module (a drop-in replacement for this one).

The basic unit of Perl testing is the ok. For each thing you want to test your program will print out an "ok" or "not ok" to indicate pass or fail. You do this with the `ok()` function (see below).

The only other constraint is you must pre-declare how many tests you plan to run. This is in case something goes horribly wrong during the test and your test program aborts, or skips a test or whatever. You do this like so:

```
use Test::Simple tests => 23;
```

You must have a plan.

- **ok**

  ```
  ok( $foo eq $bar, $name );
  ok( $foo eq $bar );
  ```

  `ok()` is given an expression (in this case `$foo eq $bar`). If it's true, the test passed. If it's false, it didn't. That's about it.

  `ok()` prints out either "ok" or "not ok" along with a test number (it keeps track of that for you).

  ```
  # This produces "ok 1 - Hell not yet frozen over" (or not ok)
  ok( get_temperature($hell) > 0, 'Hell not yet frozen over' );
  ```

If you provide a $name, that will be printed along with the "ok/not ok" to make it easier to find your test when if fails (just search for the name). It also makes it easier for the next guy to understand what your test is for. It's highly recommended you use test names.

All tests are run in scalar context. So this:

```
ok( @stuff, 'I have some stuff' );
```

will do what you mean (fail if stuff is empty)

Test::Simple will start by printing number of tests run in the form "1..M" (so "1..5" means you're going to run 5 tests). This strange format lets *Test::Harness* know how many tests you plan on running in case something goes horribly wrong.

If all your tests passed, Test::Simple will exit with zero (which is normal). If anything failed it will exit with how many failed. If you run less (or more) tests than you planned, the missing (or extras) will be considered failures. If no tests were ever run Test::Simple will throw a warning and exit with 255. If the test died, even after having successfully completed all its tests, it will still be considered a failure and will exit with 255.

So the exit codes are...

```
0                   all tests successful
255                 test died or all passed but wrong # of tests run
any other number    how many failed (including missing or extras)
```

If you fail more than 254 tests, it will be reported as 254.

This module is by no means trying to be a complete testing system. It's just to get you started. Once you're off the ground its recommended you look at *Test::More*.

EXAMPLE

Here's an example of a simple .t file for the fictional Film module.

```
use Test::Simple tests => 5;

use Film;  # What you're testing.

my $btaste = Film->new({ Title    => 'Bad Taste',
                         Director => 'Peter Jackson',
                         Rating   => 'R',
                         NumExplodingSheep => 1
                       });
ok( defined($btaste) && ref $btaste eq 'Film',     'new() works' );

ok( $btaste->Title      eq 'Bad Taste',     'Title() get'    );
ok( $btaste->Director   eq 'Peter Jackson', 'Director() get' );
ok( $btaste->Rating     eq 'R',             'Rating() get'   );
ok( $btaste->NumExplodingSheep == 1,        'NumExplodingSheep() get
↪' );
```

It will produce output like this:

```
1..5
ok 1 - new() works
ok 2 - Title() get
ok 3 - Director() get
not ok 4 - Rating() get
#   Failed test 'Rating() get'
#   in t/film.t at line 14.
ok 5 - NumExplodingSheep() get
# Looks like you failed 1 tests of 5
```

Indicating the Film::Rating() method is broken.

CAVEATS

Test::Simple will only report a maximum of 254 failures in its exit code. If this is a problem, you probably have a huge test script. Split it into multiple files. (Otherwise blame the Unix folks for using an unsigned short integer as the exit status).

Because VMS's exit codes are much, much different than the rest of the universe, and perl does horrible mangling to them that gets in my way, it works like this on VMS.

```
0      SS$_NORMAL        all tests successful
4      SS$_ABORT         something went wrong
```

Unfortunately, I can't differentiate any further.

NOTES

Test::Simple is **explicitly** tested all the way back to perl 5.6.0.

Test::Simple is thread-safe in perl 5.8.1 and up.

HISTORY

This module was conceived while talking with Tony Bowden in his kitchen one night about the problems I was having writing some really complicated feature into the new Testing module. He observed that the main problem is not dealing with these edge cases but that people hate to write tests **at all**. What was needed was a dead simple module that took all the hard work out of testing and was really, really easy to learn. Paul Johnson simultaneously had this idea (unfortunately, he wasn't in Tony's kitchen). This is it.

SEE ALSO

- *Test::More*

  More testing functions! Once you outgrow Test::Simple, look at *Test::More*. Test::Simple is 100% forward compatible with *Test::More* (i.e. you can just use *Test::More* instead of Test::Simple in your programs and things will still work).

Look in *Test::More*'s SEE ALSO for more testing modules.

AUTHORS

Idea by Tony Bowden and Paul Johnson, code by Michael G Schwern schwern@pobox.com, wardrobe by Calvin Klein.

MAINTAINERS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright 2001-2008 by Michael G Schwern schwern@pobox.com.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test::Tester

Ease testing test modules built with Test::Builder

SYNOPSIS

```
use Test::Tester tests => 6;

use Test::MyStyle;

check_test(
  sub {
    is_mystyle_eq("this", "that", "not eq");
  },
  {
    ok => 0, # expect this to fail
    name => "not eq",
    diag => "Expected: 'this'\nGot: 'that'",
  }
);
```

or

```
use Test::Tester tests => 6;

use Test::MyStyle;

check_test(
  sub {
    is_mystyle_qr("this", "that", "not matching");
  },
  {
    ok => 0, # expect this to fail
```

(continues on next page)

(continued from previous page)

```
    name => "not matching",
    diag => qr/Expected: 'this'\s+Got: 'that'/,
  }
);
```

or

```
use Test::Tester;

use Test::More tests => 3;
use Test::MyStyle;

my ($premature, @results) = run_tests(
  sub {
    is_database_alive("dbname");
  }
);

# now use Test::More::like to check the diagnostic output

like($results[0]->{diag}, "/^Database ping took \\d+ seconds$"/,
 ↪"diag");
```

DESCRIPTION

If you have written a test module based on Test::Builder then Test::Tester allows you to test it with the minimum of effort.

HOW TO USE (THE EASY WAY)

From version 0.08 Test::Tester no longer requires you to included anything special in your test modules. All you need to do is

```
use Test::Tester;
```

in your test script **before** any other Test::Builder based modules and away you go.

Other modules based on Test::Builder can be used to help with the testing. In fact you can even use functions from your module to test other functions from the same module (while this is possible it is probably not a good idea, if your module has bugs, then using it to test itself may give the wrong answers).

The easiest way to test is to do something like

```
check_test(
  sub { is_mystyle_eq("this", "that", "not eq") },
  {
    ok => 0, # we expect the test to fail
    name => "not eq",
    diag => "Expected: 'this'\nGot: 'that'",
```

(continues on next page)

(continued from previous page)

```
  }
);
```

this will execute the is_mystyle_eq test, capturing its results and checking that they are what was expected.

You may need to examine the test results in a more flexible way, for example, the diagnostic output may be quite long or complex or it may involve something that you cannot predict in advance like a timestamp. In this case you can get direct access to the test results:

```
my ($premature, @results) = run_tests(
  sub {
    is_database_alive("dbname");
  }
);

like($result[0]->{diag}, "/^Database ping took \\d+ seconds$"/,
 ↪"diag");
```

or

```
check_test(
  sub { is_mystyle_qr("this", "that", "not matching") },
  {
    ok => 0, # we expect the test to fail
    name => "not matching",
    diag => qr/Expected: 'this'\s+Got: 'that'/,
  }
);
```

We cannot predict how long the database ping will take so we use Test::More's like() test to check that the diagnostic string is of the right form.

HOW TO USE (THE HARD WAY)

This is here for backwards compatibility only

Make your module use the Test::Tester::Capture object instead of the Test::Builder one. How to do this depends on your module but assuming that your module holds the Test::Builder object in $Test and that all your test routines access it through $Test then providing a function something like this

```
sub set_builder
{
  $Test = shift;
}
```

should allow your test scripts to do

```
Test::YourModule::set_builder(Test::Tester->capture);
```

and after that any tests inside your module will captured.

TEST RESULTS

The result of each test is captured in a hash. These hashes are the same as the hashes returned by Test::Builder->details but with a couple of extra fields.

These fields are documented in *Test::Builder* in the details() function

- ok

  Did the test pass?

- actual_ok

  Did the test really pass? That is, did the pass come from Test::Builder->ok() or did it pass because it was a TODO test?

- name

  The name supplied for the test.

- type

  What kind of test? Possibilities include, skip, todo etc. See *Test::Builder* for more details.

- reason

  The reason for the skip, todo etc. See *Test::Builder* for more details.

These fields are exclusive to Test::Tester.

- diag

  Any diagnostics that were output for the test. This only includes diagnostics output **after** the test result is declared.

  Note that Test::Builder ensures that any diagnostics end in a \n and it in earlier versions of Test::Tester it was essential that you have the final \n in your expected diagnostics. From version 0.10 onward, Test::Tester will add the \n if you forgot it. It will not add a \n if you are expecting no diagnostics. See below for help tracking down hard to find space and tab related problems.

- depth

  This allows you to check that your test module is setting the correct value for $Test::Builder::Level and thus giving the correct file and line number when a test fails. It is calculated by looking at caller() and $Test::Builder::Level. It should count how many subroutines there are before jumping into the function you are testing. So for example in

  ```
  run_tests( sub { my_test_function("a", "b") } );
  ```

  the depth should be 1 and in

  ```
  sub deeper { my_test_function("a", "b") }

  run_tests(sub { deeper() });
  ```

depth should be 2, that is 1 for the sub {} and one for deeper(). This might seem a little complex but if your tests look like the simple examples in this doc then you don't need to worry as the depth will always be 1 and that's what Test::Tester expects by default.

Note: if you do not specify a value for depth in check_test() then it automatically compares it against 1, if you really want to skip the depth test then pass in undef.

Note: depth will not be correctly calculated for tests that run from a signal handler or an END block or anywhere else that hides the call stack.

Some of Test::Tester's functions return arrays of these hashes, just like Test::Builder->details. That is, the hash for the first test will be array element 1 (not 0). Element 0 will not be a hash it will be a string which contains any diagnostic output that came before the first test. This should usually be empty, if it's not, it means something output diagnostics before any test results showed up.

SPACES AND TABS

Appearances can be deceptive, especially when it comes to emptiness. If you are scratching your head trying to work out why Test::Tester is saying that your diagnostics are wrong when they look perfectly right then the answer is probably whitespace. From version 0.10 on, Test::Tester surrounds the expected and got diag values with single quotes to make it easier to spot trailing whitespace. So in this example

```
# Got diag (5 bytes):
# 'abcd '
# Expected diag (4 bytes):
# 'abcd'
```

it is quite clear that there is a space at the end of the first string. Another way to solve this problem is to use colour and inverse video on an ANSI terminal, see below COLOUR below if you want this.

Unfortunately this is sometimes not enough, neither colour nor quotes will help you with problems involving tabs, other non-printing characters and certain kinds of problems inherent in Unicode. To deal with this, you can switch Test::Tester into a mode whereby all "tricky" characters are shown as \{xx}. Tricky characters are those with ASCII code less than 33 or higher than 126. This makes the output more difficult to read but much easier to find subtle differences between strings. To turn on this mode either call `show_space()` in your test script or set the `TESTTESTERSPACE` environment variable to be a true value. The example above would then look like

```
# Got diag (5 bytes):
# abcd\x{20}
# Expected diag (4 bytes):
# abcd
```

COLOUR

If you prefer to use colour as a means of finding tricky whitespace characters then you can set the TESTTESTCOLOUR environment variable to a comma separated pair of colours, the first for the foreground, the second for the background. For example "white,red" will print white text on a red background. This requires the Term::ANSIColor module. You can specify any colour that would be acceptable to the Term::ANSIColor::color function.

If you spell colour differently, that's no problem. The TESTTESTERCOLOR variable also works (if both are set then the British spelling wins out).

EXPORTED FUNCTIONS

($premature, @results) = run_tests(\&test_sub)

\&test_sub is a reference to a subroutine.

run_tests runs the subroutine in $test_sub and captures the results of any tests inside it. You can run more than 1 test inside this subroutine if you like.

$premature is a string containing any diagnostic output from before the first test.

@results is an array of test result hashes.

cmp_result(\%result, \%expect, $name)

\%result is a ref to a test result hash.

\%expect is a ref to a hash of expected values for the test result.

cmp_result compares the result with the expected values. If any differences are found it outputs diagnostics. You may leave out any field from the expected result and cmp_result will not do the comparison of that field.

cmp_results(\@results, \@expects, $name)

\@results is a ref to an array of test results.

\@expects is a ref to an array of hash refs.

cmp_results checks that the results match the expected results and if any differences are found it outputs diagnostics. It first checks that the number of elements in \@results and \@expects is the same. Then it goes through each result checking it against the expected result as in cmp_result() above.

($premature, @results) = check_tests(\&test_sub, \@expects, $name)

\&test_sub is a reference to a subroutine.

\@expect is a ref to an array of hash refs which are expected test results.

check_tests combines run_tests and cmp_tests into a single call. It also checks if the tests died at any stage.

It returns the same values as run_tests, so you can further examine the test results if you need to.

($premature, @results) = check_test(\&test_sub, \%expect, $name)

\&test_sub is a reference to a subroutine.

\%expect is a ref to an hash of expected values for the test result.

check_test is a wrapper around check_tests. It combines run_tests and cmp_tests into a single call, checking if the test died. It assumes that only a single test is run inside \&test_sub and include a test to make sure this is true.

It returns the same values as run_tests, so you can further examine the test results if you need to.

show_space()

Turn on the escaping of characters as described in the SPACES AND TABS section.

HOW IT WORKS

Normally, a test module (let's call it Test:MyStyle) calls Test::Builder->new to get the Test::Builder object. Test::MyStyle calls methods on this object to record information about test results. When Test::Tester is loaded, it replaces Test::Builder's new() method with one which returns a Test::Tester::Delegate object. Most of the time this object behaves as the real Test::Builder object. Any methods that are called are delegated to the real Test::Builder object so everything works perfectly. However once we go into test mode, the method calls are no longer passed to the real Test::Builder object, instead they go to the Test::Tester::Capture object. This object seems exactly like the real Test::Builder object, except, instead of outputting test results and diagnostics, it just records all the information for later analysis.

CAVEATS

Support for calling Test::Builder->note is minimal. It's implemented as an empty stub, so modules that use it will not crash but the calls are not recorded for testing purposes like the others. Patches welcome.

SEE ALSO

Test::Builder the source of testing goodness. *Test::Builder::Tester* for an alternative approach to the problem tackled by Test::Tester - captures the strings output by Test::Builder. This means you cannot get separate access to the individual pieces of information and you must predict **exactly** what your test will output.

AUTHOR

This module is copyright 2005 Fergal Daly fergal@esatclear.ie, some parts are based on other people's work.

Plan handling lifted from Test::More. written by Michael G Schwern schwern@pobox.com.

Test::Tester::Capture is a cut down and hacked up version of Test::Builder. Test::Builder was written by chromatic chromatic@wgz.org and Michael G Schwern schwern@pobox.com.

LICENSE

Under the same license as Perl itself

See https://dev.perl.org/licenses/

Modules

- `Test::Tester::Capture` - Help testing test modules built with Test::Builder
- `Test::Tester::CaptureRunner` - Help testing test modules built with Test::Builder

Test::Tester::Capture

Help testing test modules built with Test::Builder

DESCRIPTION

This is a subclass of Test::Builder that overrides many of the methods so that they don't output anything. It also keeps track of its own set of test results so that you can use Test::Builder based modules to perform tests on other Test::Builder based modules.

AUTHOR

Most of the code here was lifted straight from Test::Builder and then had chunks removed by Fergal Daly fergal@esatclear.ie.

LICENSE

Under the same license as Perl itself

See https://dev.perl.org/licenses/

Test::Tester::CaptureRunner

Help testing test modules built with Test::Builder

DESCRIPTION

This stuff if needed to allow me to play with other ways of monitoring the test results.

AUTHOR

Copyright 2003 by Fergal Daly fergal@esatclear.ie.

LICENSE

Under the same license as Perl itself

See https://dev.perl.org/licenses/

Test::use

Namespace containing the following modules:

- `Test::use::ok` - Alternative to Test::More::use_ok

Test::use::ok

Alternative to Test::More::use_ok

SYNOPSIS

```
use ok 'Some::Module';
```

DESCRIPTION

According to the **Test::More** documentation, it is recommended to run `use_ok()` inside a `BEGIN` block, so functions are exported at compile-time and prototypes are properly honored.

That is, instead of writing this:

```
use_ok( 'Some::Module' );
use_ok( 'Other::Module' );
```

One should write this:

```
BEGIN { use_ok( 'Some::Module' ); }
BEGIN { use_ok( 'Other::Module' ); }
```

However, people often either forget to add `BEGIN`, or mistakenly group `use_ok` with other tests in a single `BEGIN` block, which can create subtle differences in execution order.

With this module, simply change all `use_ok` in test scripts to `use ok`, and they will be executed at `BEGIN` time. The explicit space after `use` makes it clear that this is a single compile-time action.

SEE ALSO

Test::More

MAINTAINER

- Chad Granum exodist@cpan.org

CC0 1.0 Universal

To the extent possible under law, 唐鳳 has waived all copyright and related or neighboring rights to *Test::use::ok*.

This work is published from Taiwan.

https://creativecommons.org/publicdomain/zero/1.0/

5.2.102 Test2

Framework for writing test tools that all work together.

DESCRIPTION

Test2 is a new testing framework produced by forking *Test::Builder*, completely refactoring it, adding many new features and capabilities.

WHAT IS NEW?

- Easier to test new testing tools.

  From the beginning Test2 was built with introspection capabilities. With Test::Builder it was difficult at best to capture test tool output for verification. Test2 Makes it easy with `Test2::API::intercept()`.

- Better diagnostics capabilities.

  Test2 uses an *Test2::API::Context* object to track filename, line number, and tool details. This object greatly simplifies tracking for where errors should be reported.

- Event driven.

  Test2 based tools produce events which get passed through a processing system before being output by a formatter. This event system allows for rich plugin and extension support.

- More complete API.

  Test::Builder only provided a handful of methods for generating lines of TAP. Test2 took inventory of everything people were doing with Test::Builder that required hacking it up. Test2 made public API functions for nearly all the desired functionality people didn't previously have.

- Support for output other than TAP.

  Test::Builder assumed everything would end up as TAP. Test2 makes no such assumption. Test2 provides ways for you to specify alternative and custom formatters.

- Subtest implementation is more sane.

  The Test::Builder implementation of subtests was certifiably insane. Test2 uses a stacked event hub system that greatly improves how subtests are implemented.

- Support for threading/forking.

  Test2 support for forking and threading can be turned on using *Test2::IPC*. Once turned on threading and forking operate sanely and work as one would expect.

GETTING STARTED

If you are interested in writing tests using new tools then you should look at *Test2::Suite*. *Test2::Suite* is a separate cpan distribution that contains many tools implemented on Test2.

If you are interested in writing new tools you should take a look at *Test2::API* first.

NAMESPACE LAYOUT

This describes the namespace layout for the Test2 ecosystem. Not all the namespaces listed here are part of the Test2 distribution, some are implemented in *Test2::Suite*.

Test2::Tools::

This namespace is for sets of tools. Modules in this namespace should export tools like `ok()` and `is()`. Most things written for Test2 should go here. Modules in this namespace **MUST NOT** export subs from other tools. See the "Test2::Bundle::" namespace if you want to do that.

Test2::Plugin::

This namespace is for plugins. Plugins are modules that change or enhance the behavior of Test2. An example of a plugin is a module that sets the encoding to utf8 globally. Another example is a module that causes a bail-out event after the first test failure.

Test2::Bundle::

This namespace is for bundles of tools and plugins. Loading one of these may load multiple tools and plugins. Modules in this namespace should not implement tools directly. In general modules in this namespace should load tools and plugins, then re-export things into the consumers namespace.

Test2::Require::

This namespace is for modules that cause a test to be skipped when conditions do not allow it to run. Examples would be modules that skip the test on older perls, or when non-essential modules have not been installed.

Test2::Formatter::

Formatters live under this namespace. *Test2::Formatter::TAP* is the only formatter currently. It is acceptable for third party distributions to create new formatters under this namespace.

Test2::Event::

Events live under this namespace. It is considered acceptable for third party distributions to add new event types in this namespace.

Test2::Hub::

Hub subclasses (and some hub utility objects) live under this namespace. It is perfectly reasonable for third party distributions to add new hub subclasses in this namespace.

Test2::IPC::

The IPC subsystem lives in this namespace. There are not many good reasons to add anything to this namespace, with exception of IPC drivers.

Test2::IPC::Driver::

IPC drivers live in this namespace. It is fine to create new IPC drivers and to put them in this namespace.

Test2::Util::

This namespace is for general utilities used by testing tools. Please be considerate when adding new modules to this namespace.

Test2::API::

This is for Test2 API and related packages.

Test2::

The Test2:: namespace is intended for extensions and frameworks. Tools, Plugins, etc should not go directly into this namespace. However extensions that are used to build tools and plugins may go here.

In short: If the module exports anything that should be run directly by a test script it should probably NOT go directly into `Test2::XXX`.

SEE ALSO

Test2::API - Primary API functions.

Test2::API::Context - Detailed documentation of the context object.

Test2::IPC - The IPC system used for threading/fork support.

Test2::Formatter - Formatters such as TAP live here.

Test2::Event - Events live in this namespace.

Test2::Hub - All events eventually funnel through a hub. Custom hubs are how `intercept()` and `run_subtest()` are implemented.

CONTACTING US

Many Test2 developers and users lurk on irc://irc.perl.org/#perl-qa and irc://irc.perl.org/#toolchain. We also have a slack team that can be joined by anyone with an `@cpan.org` email address https://perl-test2.slack.com/ If you do not have an `@cpan.org` email you can ask for a slack invite by emailing Chad Granum exodist@cpan.org.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Modules

- `Test2::API` - Primary interface for writing Test2 based testing tools.
- `Test2::AsyncSubtest` - Object representing an async subtest.
- `Test2::Bundle` - Documentation for bundles.
- `Test2::Compare` - Test2 extension for writing deep comparison tools.
- `Test2::Env` - Documentation for environment variables used or set by Test2.
- `Test2::Event` - Base class for events
- `Test2::EventFacet` - Base class for all event facets.
- `Test2::Formatter` - Namespace for formatters.
- `Test2::Handle` - Base class for Test2 handles used in V# bundles.
- `Test2::Hub` - The conduit through which all events flow.
- `Test2::IPC` - Turn on IPC for threading or forking support.
- `Test2::Manual` - Documentation hub for Test2 and Test2-Suite.
- `Test2::Mock` - Module for managing mocked classes and instances.
- `Test2::Plugin` - Documentation for plugins

- `Test2::Require` - Base class and documentation for skip-unless type test packages.
- `Test2::Suite` - Distribution with a rich set of tools built upon the Test2 framework.
- `Test2::Todo` - TODO extension for Test2.
- `Test2::Tools` - Documentation for Tools.
- `Test2::Util` - Tools used by Test2 and friends.
- `Test2::V0` - 0Th edition of the Test2 recommended bundle.
- `Test2::V1` - V1 edition of the Test2 recommended bundle.
- `Test2::Workflow` - A test workflow is a way of structuring tests using composable units.

Test2::API

Primary interface for writing Test2 based testing tools.

INTERNALS NOTE

The internals of this package are subject to change at any time! The public methods provided will not change in backwards-incompatible ways (once there is a stable release), but the underlying implementation details might. **Do not break encapsulation here!**

Currently the implementation is to create a single instance of the *Test2::API::Instance* Object. All class methods defer to the single instance. There is no public access to the singleton, and that is intentional. The class methods provided by this package provide the only functionality publicly exposed.

This is done primarily to avoid the problems Test::Builder had by exposing its singleton. We do not want anyone to replace this singleton, rebless it, or directly muck with its internals. If you need to do something and cannot because of the restrictions placed here, then please report it as an issue. If possible, we will create a way for you to implement your functionality without exposing things that should not be exposed.

DESCRIPTION

This package exports all the functions necessary to write and/or verify testing tools. Using these building blocks you can begin writing test tools very quickly. You are also provided with tools that help you to test the tools you write.

SYNOPSIS

WRITING A TOOL

The `context()` method is your primary interface into the Test2 framework.

```
package My::Ok;
use Test2::API qw/context/;

our @EXPORT = qw/my_ok/;
use base 'Exporter';

# Just like ok() from Test::More
sub my_ok($;$) {
    my ($bool, $name) = @_;
    my $ctx = context(); # Get a context
    $ctx->ok($bool, $name);
    $ctx->release; # Release the context
    return $bool;
}
```

See *Test2::API::Context* for a list of methods available on the context object.

TESTING YOUR TOOLS

The `intercept { ... }` tool lets you temporarily intercept all events generated by the test system:

```
use Test2::API qw/intercept/;

use My::Ok qw/my_ok/;

my $events = intercept {
    # These events are not displayed
    my_ok(1, "pass");
    my_ok(0, "fail");
};
```

As of version 1.302178 this now returns an arrayref that is also an instance of *Test2::API::InterceptResult*. See the *Test2::API::InterceptResult* documentation for details on how to best use it.

OTHER API FUNCTIONS

```
use Test2::API qw{
    test2_init_done
    test2_stack
    test2_set_is_end
    test2_get_is_end
    test2_ipc
    test2_formatter_set
    test2_formatter
    test2_is_testing_done
};

my $init  = test2_init_done();
```

(continues on next page)

(continued from previous page)

```
my $stack = test2_stack();
my $ipc   = test2_ipc();

test2_formatter_set($FORMATTER)
my $formatter = test2_formatter();

... And others ...
```

ENVIRONMENT VARIABLES

See *Test2::Env* for a list of meaningful environment variables.

MAIN API EXPORTS

All exports are optional. You must specify subs to import.

```
use Test2::API qw/context intercept run_subtest/;
```

This is the list of exports that are most commonly needed. If you are simply writing a tool, then this is probably all you need. If you need something and you cannot find it here, then you can also look at "OTHER API EXPORTS".

These exports lack the 'test2_' prefix because of how important/common they are. Exports in the "OTHER API EXPORTS" section have the 'test2_' prefix to ensure they stand out.

context(...)

Usage:

- $ctx = context()
- $ctx = context(%params)

The `context()` function will always return the current context. If there is already a context active, it will be returned. If there is not an active context, one will be generated. When a context is generated it will default to using the file and line number where the currently running sub was called from.

Please see *"CRITICAL DETAILS" in Test2::API::Context* for important rules about what you can and cannot do with a context once it is obtained.

Note This function will throw an exception if you ignore the context object it returns.

Note On perls 5.14+ a depth check is used to ensure there are no context leaks. This cannot be safely done on older perls due to https://rt.perl.org/Public/Bug/Display.html?id=127774 You can forcefully enable it either by setting `$ENV{T2_CHECK_DEPTH} = 1` or `$Test2::API::DO_DEPTH_CHECK = 1` **BEFORE** loading *Test2::API*.

OPTIONAL PARAMETERS

All parameters to `context` are optional.

- level => $int

  If you must obtain a context in a sub deeper than your entry point you can use this to tell it how many EXTRA stack frames to look back. If this option is not provided the default of `0` is used.

```
sub third_party_tool {
    my $sub = shift;
    ... # Does not obtain a context
    $sub->();
    ...
}

third_party_tool(sub {
    my $ctx = context(level => 1);
    ...
    $ctx->release;
});
```

- wrapped => $int

  Use this if you need to write your own tool that wraps a call to `context()` with the intent that it should return a context object.

```
sub my_context {
    my %params = ( wrapped => 0, @_ );
    $params{wrapped}++;
    my $ctx = context(%params);
    ...
    return $ctx;
}

sub my_tool {
    my $ctx = my_context();
    ...
    $ctx->release;
}
```

  If you do not do this, then tools you call that also check for a context will notice that the context they grabbed was created at the same stack depth, which will trigger protective measures that warn you and destroy the existing context.

- stack => $stack

  Normally `context()` looks at the global hub stack. If you are maintaining your own *Test2::API::Stack* instance you may pass it in to be used instead of the global one.

- hub => $hub

  Use this parameter if you want to obtain the context for a specific hub instead of

whatever one happens to be at the top of the stack.

- on_init => sub { ... }

  This lets you provide a callback sub that will be called **ONLY** if your call to `context()` generated a new context. The callback **WILL NOT** be called if `context()` is returning an existing context. The only argument passed into the callback will be the context object itself.

```
sub foo {
    my $ctx = context(on_init => sub { 'will run' });

    my $inner = sub {
        # This callback is not run since we are getting the
↪existing
        # context from our parent sub.
        my $ctx = context(on_init => sub { 'will NOT run' });
        $ctx->release;
    }
    $inner->();

    $ctx->release;
}
```

- on_release => sub { ... }

  This lets you provide a callback sub that will be called when the context instance is released. This callback will be added to the returned context even if an existing context is returned. If multiple calls to context add callbacks, then all will be called in reverse order when the context is finally released.

```
sub foo {
    my $ctx = context(on_release => sub { 'will run second' });

    my $inner = sub {
        my $ctx = context(on_release => sub { 'will run first' }
↪);

        # Neither callback runs on this release
        $ctx->release;
    }
    $inner->();

    # Both callbacks run here.
    $ctx->release;
}
```

release($;$)

Usage:

- release $ctx;
- release $ctx, ...;

This is intended as a shortcut that lets you release your context and return a value in one statement. This function will get your context, and an optional return value. It will release your context, then return your value. Scalar context is always assumed.

```
sub tool {
    my $ctx = context();
    ...

    return release $ctx, 1;
}
```

This tool is most useful when you want to return the value you get from calling a function that needs to see the current context:

```
my $ctx = context();
my $out = some_tool(...);
$ctx->release;
return $out;
```

We can combine the last 3 lines of the above like so:

```
my $ctx = context();
release $ctx, some_tool(...);
```

context_do(&;@)

Usage:

```
sub my_tool {
    context_do {
        my $ctx = shift;

        my (@args) = @_;

        $ctx->ok(1, "pass");

        ...

        # No need to call $ctx->release, done for you on scope exit.
    } @_;
}
```

Using this inside your test tool takes care of a lot of boilerplate for you. It will ensure a context is acquired. It will capture and rethrow any exception. It will ensure the context is released when you are done. It preserves the subroutine call context (list, scalar, void).

This is the safest way to write a test tool. The only two downsides to this are a slight performance decrease, and some extra indentation in your source. If the indentation is a problem for you then you can take a peek at the next section.

no_context(&;$)

Usage:

- no_context { ... };
- no_context { ... } $hid;

```
sub my_tool(&) {
    my $code = shift;
    my $ctx = context();
    ...

    no_context {
        # Things in here will not see our current context, they
↪get a new
        # one.

        $code->();
    };

    ...
    $ctx->release;
};
```

This tool will hide a context for the provided block of code. This means any tools run inside the block will get a completely new context if they acquire one. The new context will be inherited by tools nested below the one that acquired it.

This will normally hide the current context for the top hub. If you need to hide the context for a different hub you can pass in the optional `$hid` parameter.

intercept(&)

Usage:

```
my $events = intercept {
    ok(1, "pass");
    ok(0, "fail");
    ...
};
```

This function takes a codeblock as its only argument, and it has a prototype. It will execute the codeblock, intercepting any generated events in the process. It will return an array reference with all the generated event objects. All events should be subclasses of *Test2::Event*.

As of version 1.302178 the events array that is returned is blssed as an *Test2::API::InterceptResult* instance. *Test2::API::InterceptResult* Provides a helpful interface for filtering and/or inspecting the events list overall, or individual events within the list.

This is intended to help you test your test code. This is not intended for people simply writing tests.

run_subtest(...)

Usage:

```
run_subtest($NAME, \&CODE, $BUFFERED, @ARGS)

# or

run_subtest($NAME, \&CODE, \%PARAMS, @ARGS)
```

This will run the provided codeblock with the args in `@args`. This codeblock will be run as a subtest. A subtest is an isolated test state that is condensed into a single *Test2::Event::Subtest* event, which contains all events generated inside the subtest.

ARGUMENTS:

- $NAME

  The name of the subtest.

- \&CODE

  The code to run inside the subtest.

- $BUFFERED or \%PARAMS

  If this is a simple scalar then it will be treated as a boolean for the 'buffered' setting. If this is a hash reference then it will be used as a parameters hash. The param hash will be used for hub construction (with the specified keys removed).

  Keys that are removed and used by run_subtest:

  - 'buffered' => $bool

    Toggle buffered status.

  - 'inherit_trace' => $bool

    Normally the subtest hub is pushed and the sub is allowed to generate its own root context for the hub. When this setting is turned on a root context will be created for the hub that shares the same trace as the current context.

    Set this to true if your tool is producing subtests without user-specified subs.

  - 'no_fork' => $bool

    Defaults to off. Normally forking inside a subtest will actually fork the subtest, resulting in 2 final subtest events. This parameter will turn off that behavior, only the original process/thread will return a final subtest event.

- @ARGS

  Any extra arguments you want passed into the subtest code.

BUFFERED VS UNBUFFERED (OR STREAMED)

Normally all events inside and outside a subtest are sent to the formatter immediately by the hub. Sometimes it is desirable to hold off sending events within a subtest until the subtest is complete. This usually depends on the formatter being used.

- Things not affected by this flag

  In both cases events are generated and stored in an array. This array is eventually used to populate the `subevents` attribute on the *Test2::Event::Subtest* event that is generated at the end of the subtest. This flag has no effect on this part; it always happens.

  At the end of the subtest, the final *Test2::Event::Subtest* event is sent to the formatter.

- Things that are affected by this flag

  The `buffered` attribute of the *Test2::Event::Subtest* event will be set to the value of this flag. This means any formatter, listener, etc which looks at the event will know if it was buffered.

- Things that are formatter dependent

  Events within a buffered subtest may or may not be sent to the formatter as they happen. If a formatter fails to specify then the default is to **NOT SEND** the events as they are generated, instead the formatter can pull them from the `subevents` attribute.

  A formatter can specify by implementing the `hide_buffered()` method. If this method returns true then events generated inside a buffered subtest will not be sent independently of the final subtest event.

An example of how this is used is the *Test2::Formatter::TAP* formatter. For unbuffered subtests the events are rendered as they are generated. At the end of the subtest, the final subtest event is rendered, but the `subevents` attribute is ignored. For buffered subtests the opposite occurs, the events are NOT rendered as they are generated, instead the `subevents` attribute is used to render them all at once. This is useful when running subtests tests in parallel, since without it the output from subtests would be interleaved together.

OTHER API EXPORTS

Exports in this section are not commonly needed. These all have the 'test2_' prefix to help ensure they stand out. You should look at the "MAIN API EXPORTS" section before looking here. This section is one where "Great power comes with great responsibility". It is possible to break things badly if you are not careful with these.

All exports are optional. You need to list which ones you want at import time:

```
use Test2::API qw/test2_init_done .../;
```

STATUS AND INITIALIZATION STATE

These provide access to internal state and object instances.

- $bool = test2_init_done()

  This will return true if the stack and IPC instances have already been initialized. It will return false if they have not. Init happens as late as possible. It happens as soon as a tool requests the IPC instance, the formatter, or the stack.

- $bool = test2_load_done()

  This will simply return the boolean value of the loaded flag. If Test2 has finished loading this will be true, otherwise false. Loading is considered complete the first time a tool requests a context.

- test2_set_is_end()

- test2_set_is_end($bool)

  This is used to toggle Test2's belief that the END phase has already started. With no arguments this will set it to true. With arguments it will set it to the first argument's value.

  This is used to prevent the use of `caller()` in END blocks which can cause segfaults. This is only necessary in some persistent environments that may have multiple END phases.

- $bool = test2_get_is_end()

  Check if Test2 believes it is the END phase.

- $stack = test2_stack()

  This will return the global *Test2::API::Stack* instance. If this has not yet been initialized it will be initialized now.

- $bool = test2_is_testing_done()

  This will return true if testing is complete and no other events should be sent. This is useful in things like warning handlers where you might want to turn warnings into events, but need them to start acting like normal warnings when testing is done.

  ```
  $SIG{__WARN__} = sub {
      my ($warning) = @_;

      if (test2_is_testing_done()) {
          warn @_;
      }
      else {
          my $ctx = context();
          ...
          $ctx->release
      }
  }
  ```

- test2_ipc_disable

Disable IPC.

- $bool = test2_ipc_disabled

  Check if IPC is disabled.

- test2_ipc_wait_enable()
- test2_ipc_wait_disable()
- $bool = test2_ipc_wait_enabled()

  These can be used to turn IPC waiting on and off, or check the current value of the flag.

  Waiting is turned on by default. Waiting will cause the parent process/thread to wait until all child processes and threads are finished before exiting. You will almost never want to turn this off.

- $bool = test2_no_wait()
- test2_no_wait($bool)

  DISCOURAGED: This is a confusing interface, it is better to use `test2_ipc_wait_enable()`, `test2_ipc_wait_disable()` and `test2_ipc_wait_enabled()`.

  This can be used to get/set the no_wait status. Waiting is turned on by default. Waiting will cause the parent process/thread to wait until all child processes and threads are finished before exiting. You will almost never want to turn this off.

- $fh = test2_stdout()
- $fh = test2_stderr()

  These functions return the filehandles that test output should be written to. They are primarily useful when writing a custom formatter and code that turns events into actual output (TAP, etc.). They will return a dupe of the original filehandles that formatted output can be sent to regardless of whatever state the currently running test may have left STDOUT and STDERR in.

- test2_reset_io()

  Re-dupe the internal filehandles returned by `test2_stdout()` and `test2_stderr()` from the current STDOUT and STDERR. You shouldn't need to do this except in very peculiar situations (for example, you're testing a new formatter and you need control over where the formatter is sending its output.)

BEHAVIOR HOOKS

These are hooks that allow you to add custom behavior to actions taken by Test2 and tools built on top of it.

- test2_add_callback_exit(sub { ... })

  This can be used to add a callback that is called after all testing is done. This is too late to add additional results, the main use of this callback is to set the exit code.

```
test2_add_callback_exit(
    sub {
        my ($context, $exit, \$new_exit) = @_;
        ...
    }
);
```

The `$context` passed in will be an instance of *Test2::API::Context*. The `$exit` argument will be the original exit code before anything modified it. `$$new_exit` is a reference to the new exit code. You may modify this to change the exit code. Please note that `$$new_exit` may already be different from `$exit`

- test2_add_callback_post_load(sub { ... })

  Add a callback that will be called when Test2 is finished loading. This means the callback will be run once, the first time a context is obtained. If Test2 has already finished loading then the callback will be run immediately.

- test2_add_callback_testing_done(sub { ... })

  This adds your coderef as a follow-up to the root hub after Test2 is finished loading.

  This is essentially a helper to do the following:

```
test2_add_callback_post_load(sub {
    my $stack = test2_stack();
    $stack->top; # Ensure we have a hub
    my ($hub) = Test2::API::test2_stack->all;

    $hub->set_active(1);

    $hub->follow_up(sub { ... }); # <-- Your coderef here
});
```

- test2_add_callback_context_acquire(sub { ... })

  Add a callback that will be called every time someone tries to acquire a context. This will be called on EVERY call to `context()`. It gets a single argument, a reference to the hash of parameters being used the construct the context. This is your chance to change the parameters by directly altering the hash.

```
test2_add_callback_context_acquire(sub {
    my $params = shift;
    $params->{level}++;
});
```

  This is a very scary API function. Please do not use this unless you need to. This is here for *Test::Builder* and backwards compatibility. This has you directly manipulate the hash instead of returning a new one for performance reasons.

- test2_add_callback_context_init(sub { ... })

  Add a callback that will be called every time a new context is created. The callback will receive the newly created context as its only argument.

- test2_add_callback_context_release(sub { ... })

Add a callback that will be called every time a context is released. The callback will receive the released context as its only argument.

- test2_add_callback_pre_subtest(sub { ... })

  Add a callback that will be called every time a subtest is going to be run. The callback will receive the subtest name, coderef, and any arguments.

- @list = test2_list_context_acquire_callbacks()

  Return all the context acquire callback references.

- @list = test2_list_context_init_callbacks()

  Returns all the context init callback references.

- @list = test2_list_context_release_callbacks()

  Returns all the context release callback references.

- @list = test2_list_exit_callbacks()

  Returns all the exit callback references.

- @list = test2_list_post_load_callbacks()

  Returns all the post load callback references.

- @list = test2_list_pre_subtest_callbacks()

  Returns all the pre-subtest callback references.

- test2_add_uuid_via(sub { ... })

- $sub = test2_add_uuid_via()

  This allows you to provide a UUID generator. If provided UUIDs will be attached to all events, hubs, and contexts. This is useful for storing, tracking, and linking these objects.

  The sub you provide should always return a unique identifier. Most things will expect a proper UUID string, however nothing in Test2::API enforces this.

  The sub will receive exactly 1 argument, the type of thing being tagged 'context', 'hub', or 'event'. In the future additional things may be tagged, in which case new strings will be passed in. These are purely informative, you can (and usually should) ignore them.

- test2_add_pending_diag($diag)

- test2_add_pending_diag($diag1, $diag2)

  Add a diagnostics message that will be issued the next time a context in which a failure occurred is released.

  This can also be thought of like this: "If the next bit causes a failed assertion, add this diagnostics message".

- @diags = test2_get_pending_diags()

  Get all the current pending diagnostics messages.

- @diags = test2_clear_pending_diags()

  Clear any pending diagnostics, returning them.

IPC AND CONCURRENCY

These let you access, or specify, the IPC system internals.

- $bool = test2_has_ipc()

  Check if IPC is enabled.

- $ipc = test2_ipc()

  This will return the global *Test2::IPC::Driver* instance. If this has not yet been initialized it will be initialized now.

- test2_ipc_add_driver($DRIVER)

  Add an IPC driver to the list. This will add the driver to the start of the list.

- @drivers = test2_ipc_drivers()

  Get the list of IPC drivers.

- $bool = test2_ipc_polling()

  Check if polling is enabled.

- test2_ipc_enable_polling()

  Turn on polling. This will cull events from other processes and threads every time a context is created.

- test2_ipc_disable_polling()

  Turn off IPC polling.

- test2_ipc_enable_shm()

  Legacy, this is currently a no-op that returns 0;

- test2_ipc_set_pending($uniq_val)

  Tell other processes and events that an event is pending. `$uniq_val` should be a unique value no other thread/process will generate.

  Note: After calling this `test2_ipc_get_pending()` will return 1. This is intentional, and not avoidable.

- $pending = test2_ipc_get_pending()

  This returns -1 if there is no way to check (assume yes)

  This returns 0 if there are (most likely) no pending events.

  This returns 1 if there are (likely) pending events. Upon return it will reset, nothing else will be able to see that there were pending events.

- $timeout = test2_ipc_get_timeout()

- test2_ipc_set_timeout($timeout)

  Get/Set the timeout value for the IPC system. This timeout is how long the IPC system will wait for child processes and threads to finish before aborting.

  The default value is `30` seconds.

MANAGING FORMATTERS

These let you access, or specify, the formatters that can/should be used.

- $formatter = test2_formatter

  This will return the global formatter class. This is not an instance. By default the formatter is set to *Test2::Formatter::TAP*.

  You can override this default using the `T2_FORMATTER` environment variable.

  Normally 'Test2::Formatter::' is prefixed to the value in the environment variable:

```
$ T2_FORMATTER='TAP' perl test.t     # Use the␣
↪Test2::Formatter::TAP formatter
$ T2_FORMATTER='Foo' perl test.t     # Use the␣
↪Test2::Formatter::Foo formatter
```

  If you want to specify a full module name you use the '+' prefix:

```
$ T2_FORMATTER='+Foo::Bar' perl test.t     # Use the Foo::Bar␣
↪formatter
```

- test2_formatter_set($class_or_instance)

  Set the global formatter class. This can only be set once. **Note:** This will override anything specified in the 'T2_FORMATTER' environment variable.

- @formatters = test2_formatters()

  Get a list of all loaded formatters.

- test2_formatter_add($class_or_instance)

  Add a formatter to the list. Last formatter added is used at initialization. If this is called after initialization a warning will be issued.

TIME STAMPS

You can enable or disable timestamps in trace facets. They are disabled by default for compatibility and performance reasons.

- test2_enable_trace_stamps()

  Enable stamps in traces.

- test2_disable_trace_stamps()

  Disable stamps in traces.

- $bool = test2_trace_stamps_enabled()

  Check status of trace stamps.

OTHER EXAMPLES

See the `/Examples/` directory included in this distribution.

SEE ALSO

Test2::API::Context - Detailed documentation of the context object.

Test2::IPC - The IPC system used for threading/fork support.

Test2::Formatter - Formatters such as TAP live here.

Test2::Event - Events live in this namespace.

Test2::Hub - All events eventually funnel through a hub. Custom hubs are how `intercept()` and `run_subtest()` are implemented.

MAGIC

This package has an END block. This END block is responsible for setting the exit code based on the test results. This end block also calls the callbacks that can be added to this package.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Modules

- `Test2::API::Breakage` - What breaks at what version
- `Test2::API::Context` - Object to represent a testing context.
- `Test2::API::Instance` - Object used by Test2::API under the hood
- `Test2::API::InterceptResult` - Representation of a list of events.

- `Test2::API::Stack` - Object to manage a stack of *Test2::Hub* instances.

Test2::API::Breakage

What breaks at what version

DESCRIPTION

This module provides lists of modules that are broken, or have been broken in the past, when upgrading *Test::Builder* to use *Test2*.

FUNCTIONS

These can be imported, or called as methods on the class.

-
- This returns key/value pairs. The key is the module name, the value is the version number. If the installed version of the module is at or below the specified one then an upgrade would be a good idea, but not strictly necessary.
-
- This returns key/value pairs. The key is the module name, the value is the version number. If the installed version of the module is at or below the specified one then an upgrade is required for the module to work properly.
-
- This returns key/value pairs. The key is the module name, the value is the version number. If the installed version of the module is at or below the specified one then the module will not work. A newer version may work, but is not tested or verified.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::API::Context

Object to represent a testing context.

DESCRIPTION

The context object is the primary interface for authors of testing tools written with *Test2*. The context object represents the context in which a test takes place (File and Line Number), and provides a quick way to generate events from that context. The context object also takes care of sending events to the correct *Test2::Hub* instance.

SYNOPSIS

In general you will not be creating contexts directly. To obtain a context you should always use `context()` which is exported by the *Test2::API* module.

```
use Test2::API qw/context/;

sub my_ok {
    my ($bool, $name) = @_;
    my $ctx = context();

    if ($bool) {
        $ctx->pass($name);
    }
    else {
        $ctx->fail($name);
    }

    $ctx->release; # You MUST do this!
    return $bool;
}
```

Context objects make it easy to wrap other tools that also use context. Once you grab a context, any tool you call before releasing your context will inherit it:

```
sub wrapper {
    my ($bool, $name) = @_;
    my $ctx = context();
    $ctx->diag("wrapping my_ok");

    my $out = my_ok($bool, $name);
    $ctx->release; # You MUST do this!
    return $out;
}
```

CRITICAL DETAILS

- you MUST always use the context() sub from Test2::API

  Creating your own context via `Test2::API::Context->new()` will almost never produce a desirable result. Use `context()` which is exported by *Test2::API*.

  There are a handful of cases where a tool author may want to create a new context by hand, which is why the `new` method exists. Unless you really know what you are doing you should avoid this.

- You MUST always release the context when done with it

  Releasing the context tells the system you are done with it. This gives it a chance to run any necessary callbacks or cleanup tasks. If you forget to release the context it will try to detect the problem and warn you about it.

- You MUST NOT pass context objects around

  When you obtain a context object it is made specifically for your tool and any tools nested within. If you pass a context around you run the risk of polluting other tools with incorrect context information.

  If you are certain that you want a different tool to use the same context you may pass it a snapshot. `$ctx->snapshot` will give you a shallow clone of the context that is safe to pass around or store.

- You MUST NOT store or cache a context for later

  As long as a context exists for a given hub, all tools that try to get a context will get the existing instance. If you try to store the context you will pollute other tools with incorrect context information.

  If you are certain that you want to save the context for later, you can use a snapshot. `$ctx->snapshot` will give you a shallow clone of the context that is safe to pass around or store.

  `context()` has some mechanisms to protect you if you do cause a context to persist beyond the scope in which it was obtained. In practice you should not rely on these protections, and they are fairly noisy with warnings.

- You SHOULD obtain your context as soon as possible in a given tool

  You never know what tools you call from within your own tool will need a context. Obtaining the context early ensures that nested tools can find the context you want them to find.

METHODS

- $ctx->done_testing;

  Note that testing is finished. If no plan has been set this will generate a Plan event.

- $clone = $ctx->snapshot()

  This will return a shallow clone of the context. The shallow clone is safe to store for later.

- $ctx->release()

This will release the context. This runs cleanup tasks, and several important hooks. It will also restore `$!`, `$?`, and `$@` to what they were when the context was created.

Note: If a context is acquired more than once an internal refcount is kept. `release()` decrements the ref count, none of the other actions of `release()` will occur unless the refcount hits 0. This means only the last call to `release()` will reset `$?`, `$!`, `$@`, and run the cleanup tasks.

- $ctx->throw($message)

  This will throw an exception reporting to the file and line number of the context. This will also release the context for you.

- $ctx->alert($message)

  This will issue a warning from the file and line number of the context.

- $stack = $ctx->stack()

  This will return the *Test2::API::Stack* instance the context used to find the current hub.

- $hub = $ctx->hub()

  This will return the *Test2::Hub* instance the context recognizes as the current one to which all events should be sent.

- $dbg = $ctx->trace()

  This will return the *Test2::EventFacet::Trace* instance used by the context.

- $ctx->do_in_context(\&code, @args);

  Sometimes you have a context that is not current, and you want things to use it as the current one. In these cases you can call `$ctx->do_in_context(sub { ... })`. The codeblock will be run, and anything inside of it that looks for a context will find the one on which the method was called.

  This **DOES NOT** affect context on other hubs, only the hub used by the context will be affected.

```
my $ctx = ...;
$ctx->do_in_context(sub {
    my $ctx = context(); # returns the $ctx the sub is called on
});
```

  Note: The context will actually be cloned, the clone will be used instead of the original. This allows the thread id, process id, and error variables to be correct without modifying the original context.

- $ctx->restore_error_vars()

  This will set `$!`, `$?`, and `$@` to what they were when the context was created. There is no localization or anything done here, calling this method will actually set these vars.

- $! = $ctx->errno()

  The (numeric) value of `$!` when the context was created.

- $? = $ctx->child_error()

  The value of `$?` when the context was created.

- $@ = $ctx->eval_error()

  The value of `$@` when the context was created.

EVENT PRODUCTION METHODS

Which one do I use?

The `pass*` and `fail*` are optimal if they meet your situation, using one of them will always be the most optimal. That said they are optimal by eliminating many features.

Method such as ok, and `note` are shortcuts for generating common 1-task events based on the old API, however they are forward compatible, and easy to use. If these meet your needs then go ahead and use them, but please check back often for alternatives that may be added.

If you want to generate new style events, events that do many things at once, then you want the `*ev2*` methods. These let you directly specify which facets you wish to use.

- $event = $ctx->pass()
- $event = $ctx->pass($name)

  This will send and return an *Test2::Event::Pass* event. You may optionally provide a `$name` for the assertion.

  The *Test2::Event::Pass* is a specially crafted and optimized event, using this will help the performance of passing tests.

- $true = $ctx->pass_and_release()
- $true = $ctx->pass_and_release($name)

  This is a combination of `pass()` and `release()`. You can use this if you do not plan to do anything with the context after sending the event. This helps write more clear and compact code.

```
sub shorthand {
    my ($bool, $name) = @_;
    my $ctx = context();
    return $ctx->pass_and_release($name) if $bool;

    ... Handle a failure ...
}

sub longform {
    my ($bool, $name) = @_;
    my $ctx = context();

    if ($bool) {
        $ctx->pass($name);
        $ctx->release;
        return 1;
```

(continues on next page)

(continued from previous page)

```
    }

    ... Handle a failure ...
}
```

- my $event = $ctx->fail()
- my $event = $ctx->fail($name)
- my $event = $ctx->fail($name, @diagnostics)

  This lets you send an *Test2::Event::Fail* event. You may optionally provide a `$name` and `@diagnostics` messages.

  Diagnostics messages can be simple strings, data structures, or instances of *Test2::EventFacet::Info::Table* (which are converted inline into the *Test2::EventFacet::Info* structure).

- my $false = $ctx->fail_and_release()
- my $false = $ctx->fail_and_release($name)
- my $false = $ctx->fail_and_release($name, @diagnostics)

  This is a combination of `fail()` and `release()`. This can be used to write clearer and shorter code.

```
sub shorthand {
    my ($bool, $name) = @_;
    my $ctx = context();
    return $ctx->fail_and_release($name) unless $bool;

    ... Handle a success ...
}

sub longform {
    my ($bool, $name) = @_;
    my $ctx = context();

    unless ($bool) {
        $ctx->pass($name);
        $ctx->release;
        return 1;
    }

    ... Handle a success ...
}
```

- $event = $ctx->ok($bool, $name)
- $event = $ctx->ok($bool, $name, \@on_fail)

  NOTE: Use of this method is discouraged in favor of `pass()` and `fail()` which produce *Test2::Event::Pass* and *Test2::Event::Fail* events. These newer event types are faster and less crufty.

This will create an *Test2::Event::Ok* object for you. If `$bool` is false then an *Test2::Event::Diag* event will be sent as well with details about the failure. If you do not want automatic diagnostics you should use the `send_event()` method directly.

The third argument `\@on_fail`) is an optional set of diagnostics to be sent in the event of a test failure. Unlike with `fail()` these diagnostics must be plain strings, data structures are not supported.

- $event = $ctx->note($message)

  Send an *Test2::Event::Note*. This event prints a message to STDOUT.

- $event = $ctx->diag($message)

  Send an *Test2::Event::Diag*. This event prints a message to STDERR.

- $event = $ctx->plan($max)

- $event = $ctx->plan(0, 'SKIP', $reason)

  This can be used to send an *Test2::Event::Plan* event. This event usually takes either a number of tests you expect to run. Optionally you can set the expected count to 0 and give the 'SKIP' directive with a reason to cause all tests to be skipped.

- $event = $ctx->skip($name, $reason);

  Send an *Test2::Event::Skip* event.

- $event = $ctx->bail($reason)

  This sends an *Test2::Event::Bail* event. This event will completely terminate all testing.

- $event = $ctx->send_ev2(%facets)

  This lets you build and send a V2 event directly from facets. The event is returned after it is sent.

  This example sends a single assertion, a note (comment for stdout in Test::Builder talk) and sets the plan to 1.

```
my $event = $ctx->send_event(
    plan   => {count => 1},
    assert => {pass  => 1, details => "A passing assert"},
    info => [{tag => 'NOTE', details => "This is a note"}],
);
```

- $event = $ctx->build_e2(%facets)

  This is the same as `send_ev2()`, except it builds and returns the event without sending it.

- $event = $ctx->send_ev2_and_release($Type, %parameters)

  This is a combination of `send_ev2()` and `release()`.

```
sub shorthand {
    my $ctx = context();
    return $ctx->send_ev2_and_release(assert => {pass => 1,␣
↪details => 'foo'});
```

(continues on next page)

(continued from previous page)

```
}

sub longform {
    my $ctx = context();
    my $event = $ctx->send_ev2(assert => {pass => 1, details =>
↪'foo'});
    $ctx->release;
    return $event;
}
```

- $event = $ctx->send_event($Type, %parameters)

  It is better to use send_ev2() in new code.

  This lets you build and send an event of any type. The `$Type` argument should be the event package name with `Test2::Event::` left off, or a fully qualified package name prefixed with a '+'. The event is returned after it is sent.

```
my $event = $ctx->send_event('Ok', ...);
```

  or

```
my $event = $ctx->send_event('+Test2::Event::Ok', ...);
```

- $event = $ctx->build_event($Type, %parameters)

  It is better to use build_ev2() in new code.

  This is the same as `send_event()`, except it builds and returns the event without sending it.

- $event = $ctx->send_event_and_release($Type, %parameters)

  It is better to use send_ev2_and_release() in new code.

  This is a combination of `send_event()` and `release()`.

```
sub shorthand {
    my $ctx = context();
    return $ctx->send_event_and_release(Pass => { name => 'foo'␣
↪});
}

sub longform {
    my $ctx = context();
    my $event = $ctx->send_event(Pass => { name => 'foo' });
    $ctx->release;
    return $event;
}
```

HOOKS

There are 2 types of hooks, init hooks, and release hooks. As the names suggest, these hooks are triggered when contexts are created or released.

INIT HOOKS

These are called whenever a context is initialized. That means when a new instance is created. These hooks are **NOT** called every time something requests a context, just when a new one is created.

GLOBAL

This is how you add a global init callback. Global callbacks happen for every context for any hub or stack.

```
Test2::API::test2_add_callback_context_init(sub {
    my $ctx = shift;
    ...
});
```

PER HUB

This is how you add an init callback for all contexts created for a given hub. These callbacks will not run for other hubs.

```
$hub->add_context_init(sub {
    my $ctx = shift;
    ...
});
```

PER CONTEXT

This is how you specify an init hook that will only run if your call to `context()` generates a new context. The callback will be ignored if `context()` is returning an existing context.

```
my $ctx = context(on_init => sub {
    my $ctx = shift;
    ...
});
```

RELEASE HOOKS

These are called whenever a context is released. That means when the last reference to the instance is about to be destroyed. These hooks are **NOT** called every time `$ctx->release` is called.

GLOBAL

This is how you add a global release callback. Global callbacks happen for every context for any hub or stack.

```
Test2::API::test2_add_callback_context_release(sub {
    my $ctx = shift;
    ...
});
```

PER HUB

This is how you add a release callback for all contexts created for a given hub. These callbacks will not run for other hubs.

```
$hub->add_context_release(sub {
    my $ctx = shift;
    ...
});
```

PER CONTEXT

This is how you add release callbacks directly to a context. The callback will **ALWAYS** be added to the context that gets returned, it does not matter if a new one is generated, or if an existing one is returned.

```
my $ctx = context(on_release => sub {
    my $ctx = shift;
    ...
});
```

THIRD PARTY META-DATA

This object consumes *Test2::Util::ExternalMeta* which provides a consistent way for you to attach meta-data to instances of this class. This is useful for tools, plugins, and other extensions.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org
- Kent Fredric kentnl@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::API::Instance

Object used by Test2::API under the hood

DESCRIPTION

This object encapsulates the global shared state tracked by *Test2*. A single global instance of this package is stored (and obscured) by the *Test2::API* package.

There is no reason to directly use this package. This package is documented for completeness. This package can change, or go away completely at any time. Directly using, or monkeypatching this package is not supported in any way shape or form.

SYNOPSIS

```
use Test2::API::Instance;

my $obj = Test2::API::Instance->new;
```

- $pid = $obj->pid

  PID of this instance.

- $obj->tid

  Thread ID of this instance.

- $obj->reset()

  Reset the object to defaults.

- $obj->load()

  Set the internal state to loaded, and run and stored post-load callbacks.

- $bool = $obj->loaded

  Check if the state is set to loaded.

- $arrayref = $obj->post_load_callbacks

  Get the post-load callbacks.

- $obj->add_post_load_callback(sub { ... })

  Add a post-load callback. If `load()` has already been called then the callback will be immediately executed. If `load()` has not been called then the callback will be stored and executed later when `load()` is called.

- $hashref = $obj->contexts()

  Get a hashref of all active contexts keyed by hub id.

- $arrayref = $obj->context_acquire_callbacks

  Get all context acquire callbacks.

- $arrayref = $obj->context_init_callbacks

  Get all context init callbacks.

- $arrayref = $obj->context_release_callbacks

  Get all context release callbacks.

- $arrayref = $obj->pre_subtest_callbacks

  Get all pre-subtest callbacks.

- $obj->add_context_init_callback(sub { ... })

  Add a context init callback. Subs are called every time a context is created. Subs get the newly created context as their only argument.

- $obj->add_context_release_callback(sub { ... })

  Add a context release callback. Subs are called every time a context is released. Subs get the released context as their only argument. These callbacks should not call release on the context.

- $obj->add_pre_subtest_callback(sub { ... })

  Add a pre-subtest callback. Subs are called every time a subtest is going to be run. Subs get the subtest name, coderef, and any arguments.

- $obj->set_exit()

  This is intended to be called in an `END { ... }` block. This will look at test state and set $?. This will also call any end callbacks, and wait on child processes/threads.

- $obj->set_ipc_pending($val)

  Tell other processes and threads there is a pending event. `$val` should be a unique value no other thread/process will generate.

  Note: This will also make the current process see a pending event.

- $pending = $obj->get_ipc_pending()

  This returns -1 if it is not possible to know.

  This returns 0 if there are no pending events.

  This returns 1 if there are pending events.

- $timeout = $obj->ipc_timeout;

- $obj->set_ipc_timeout($timeout);

  How long to wait for child processes and threads before aborting.

- $drivers = $obj->ipc_drivers

  Get the list of IPC drivers.

- $obj->add_ipc_driver($DRIVER_CLASS)

  Add an IPC driver to the list. The most recently added IPC driver will become the global one during initialization. If a driver is added after initialization has occurred a warning will be generated:

```
"IPC driver $driver loaded too late to be used as the global␣
↪ipc driver"
```

- $bool = $obj->ipc_polling

  Check if polling is enabled.

- $obj->enable_ipc_polling

  Turn on polling. This will cull events from other processes and threads every time a context is created.

- $obj->disable_ipc_polling

  Turn off IPC polling.

- $bool = $obj->no_wait

- $bool = $obj->set_no_wait($bool)

  Get/Set no_wait. This option is used to turn off process/thread waiting at exit.

- $arrayref = $obj->exit_callbacks

  Get the exit callbacks.

- $obj->add_exit_callback(sub { ... })

  Add an exit callback. This callback will be called by `set_exit()`.

- $bool = $obj->finalized

  Check if the object is finalized. Finalization happens when either `ipc()`, `stack()`, or `format()` are called on the object. Once finalization happens these fields are considered unchangeable (not enforced here, enforced by *Test2*).

- $ipc = $obj->ipc

  Get the one true IPC instance.

- $obj->ipc_disable

  Turn IPC off

- $bool = $obj->ipc_disabled

  Check if IPC is disabled

- $stack = $obj->stack

  Get the one true hub stack.

- $formatter = $obj->formatter

  Get the global formatter. By default this is the `'Test2::Formatter::TAP'` package. This could be any package that implements the `write()` method. This can also be an instantiated object.

- $bool = $obj->formatter_set()

  Check if a formatter has been set.

- $obj->add_formatter($class)

- $obj->add_formatter($obj)

  Add a formatter. The most recently added formatter will become the global one during initialization. If a formatter is added after initialization has occurred a warning will be generated:

```
"Formatter $formatter loaded too late to be used as the global␣
↪formatter"
```

- $obj->set_add_uuid_via(sub { ... })

- $sub = $obj->add_uuid_via()

  This allows you to provide a UUID generator. If provided UUIDs will be attached to all events, hubs, and contexts. This is useful for storing, tracking, and linking these objects.

  The sub you provide should always return a unique identifier. Most things will expect a proper UUID string, however nothing in Test2::API enforces this.

  The sub will receive exactly 1 argument, the type of thing being tagged 'context', 'hub', or 'event'. In the future additional things may be tagged, in which case new strings will be passed in. These are purely informative, you can (and usually should) ignore them.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::API::InterceptResult

Representation of a list of events.

DESCRIPTION

This class represents a list of events, normally obtained using `intercept()` from *Test2::API*.

This class is intended for people who with to verify the results of test tools they write.

This class provides methods to normalize, summarize, or map the list of events. The output of these operations makes verifying your testing tools and the events they generate significantly easier. In most cases this spares you from needing a deep understanding of the event/facet model.

SYNOPSIS

Usually you get an instance of this class when you use `intercept()` from *Test2::API*.

```
use Test2::V1 '-iPp';
use Test2::API qw/intercept/;

my $events = intercept {
    ok(1, "pass");
    ok(0, "fail");
    todo "broken" => sub { ok(0, "fixme") };
    plan 3;
};

# This is typically the most useful construct
# squash_info() merges assertions and diagnostics that are␣
↪associated
#   (and returns a new instance with the modifications)
# flatten() condenses the facet data into the key details for each␣
↪event
#   (and returns those structures in an arrayref)
is(
    $events->squash_info->flatten(),
    [
        {
            causes_failure => 0,

            name => 'pass',
            pass => 1,

            trace_file => 'xxx.t',
            trace_line => 5,
        },
```

(continues on next page)

(continued from previous page)

```
        {
            causes_failure => 1,

            name => 'fail',
            pass => 0,

            trace_file => 'xxx.t',
            trace_line => 6,

            # There can be more than one diagnostics message so this
↪is
            # always an array when present.
            diag => ["Failed test 'fail'\nat xxx.t line 6."],
        },
        {
            causes_failure => 0,

            name => 'fixme',
            pass => 0,

            trace_file => 'xxx.t',
            trace_line => 7,

            # There can be more than one diagnostics message or todo
            # reason, so these are always an array when present.
            todo => ['broken'],

            # Diag message was turned into a note since the
↪assertion was
            # TODO
            note => ["Failed test 'fixme'\nat xxx.t line 7."],
        },
        {
            causes_failure => 0,

            plan => 3,

            trace_file => 'xxx.t',
            trace_line => 8,
        },
    ],
    "Flattened events look like we expect"
);
```

See *Test2::API::InterceptResult::Event* for a full description of what `flatten()` provides for each event.

METHODS

Please note that no methods modify the original instance unless asked to do so.

CONSTRUCTION

- $events = Test2::API::InterceptResult->new(@EVENTS)
- $events = Test2::API::InterceptResult->new_from_ref(\@EVENTS)

  These create a new instance of Test2::API::InterceptResult from the given events.

  In the first form a new blessed arrayref is returned. In the 'new_from_ref' form the reference you pass in is directly blessed.

  Both of these will throw an exception if called in void context. This is mainly important for the 'filtering' methods listed below which normally return a new instance, they throw an exception in such cases as it probably means someone meant to filter the original in place.

- $clone = $events->clone()

  Make a clone of the original events. Note that this is a deep copy, the entire structure is duplicated. This uses `dclone` from *Storable* to achieve the deep clone.

NORMALIZATION

- @events = $events->event_list

  This returns all the events in list-form.

- $hub = $events->hub

  This returns a new *Test2::Hub* instance that has processed all the events contained in the instance. This gives you a simple way to inspect the state changes your events cause.

- $state = $events->state

  This returns a summary of the state of a hub after processing all the events.

```
{
    count        => 2,      # Number of assertions made
    failed       => 1,      # Number of test failures seen
    is_passing   => 0,      # Boolean, true if the test would␣
↪be passing
                            # after the events are processed.

    plan         => 2,      # Plan, either a number, undef,
↪'SKIP', or 'NO PLAN'
    follows_plan => 1,      # True if there is a plan and it␣
↪was followed.
                            # False if the plan and assertions␣
↪did not
                            # match, undef if no plan was␣
↪present in the
```

(continues on next page)

(continued from previous page)

```
                            # event list.

    bailed_out    => undef,  # undef unless there was a bail-out␣
↪in the
                            # events in which case this will be␣
↪a string
                            # explaining why there was a␣
↪bailout, if no
                            # reason was given this will simply␣
↪be set to
                            # true (1).

    skip_reason   => undef,  # If there was a skip_all this will␣
↪give the
                            # reason.
}
```

- $new = $events->upgrade
- $events->upgrade(in_place => $BOOL)

  Note: This normally returns a new instance, leaving the original unchanged. If you call it in void context it will throw an exception. If you want to modify the original you must pass in the `in_place => 1` option. You may call this in void context when you ask to modify it in place. The in-place form returns the instance that was modified so you can chain methods.

  This will create a clone of the list where all events have been converted into *Test2::API::InterceptResult::Event* instances. This is extremely helpful as *Test2::API::InterceptResult::Event* provide a much better interface for working with events. This allows you to avoid thinking about legacy event types.

  This also means your tests against the list are not fragile if the tool you are testing randomly changes what type of events it generates (IE Changing from *Test2::Event::Ok* to *Test2::Event::Pass*, both make assertions and both will normalize to identical (or close enough) *Test2::API::InterceptResult::Event* instances.

  Really you almost always want this, the only reason it is not done automatically is to make sure the `intercept()` tool is backwards compatible.

- $new = $events->squash_info
- $events->squash_info(in_place => $BOOL)

  Note: This normally returns a new instance, leaving the original unchanged. If you call it in void context it will throw an exception. If you want to modify the original you must pass in the `in_place => 1` option. You may call this in void context when you ask to modify it in place. The in-place form returns the instance that was modified so you can chain methods.

  Note: All events in the new or modified instance will be converted to *Test2::API::InterceptResult::Event* instances. There is no way to avoid this, the squash operation requires the upgraded event class.

  Test::More and many other legacy tools would send notes, diags, and assertions

as separate events. A subtest in *Test::More* would send a note with the subtest name, the subtest assertion, and finally a diagnostics event if the subtest failed. This method will normalize things by squashing the note and diag into the same event as the subtest (This is different from putting them into the subtest, which is not what happens).

FILTERING

Note: These normally return new instances, leaving the originals unchanged. If you call them in void context they will throw exceptions. If you want to modify the originals you must pass in the `in_place => 1` option. You may call these in void context when you ask to modify them in place. The in-place forms return the instance that was modified so you can chain methods.

%PARAMS

These all accept the same 2 optional parameters:

- in_place => $BOOL

  When true the method will modify the instance in place instead of returning a new instance.

- args => \@ARGS

  If you wish to pass parameters into the event method being used for filtering, you may do so here.

METHODS

- $events->grep($CALL, %PARAMS)

  This is essentially:

  ```
  Test2::API::InterceptResult->new(
      grep { $_->$CALL( @{$PARAMS{args}} ) } $self->event_list,
  );
  ```

  Note: that $CALL is called on an upgraded version of the event, though the events returned will be the original ones, not the upgraded ones.

  $CALL may be either the name of a method on *Test2::API::InterceptResult::Event*, or a coderef.

- $events->asserts(%PARAMS)

  This is essentially:

  ```
  $events->grep(has_assert => @{$PARAMS{args}})
  ```

  It returns a new instance containing only the events that made assertions.

- $events->subtests(%PARAMS)

  This is essentially:

```
$events->grep(has_subtest => @{$PARAMS{args}})
```

It returns a new instance containing only the events that have subtests.

- $events->diags(%PARAMS)

  This is essentially:

```
$events->grep(has_diags => @{$PARAMS{args}})
```

  It returns a new instance containing only the events that have diags.

- $events->notes(%PARAMS)

  This is essentially:

```
$events->grep(has_notes => @{$PARAMS{args}})
```

  It returns a new instance containing only the events that have notes.

- $events->errors(%PARAMS)

  Note: Errors are NOT failing assertions. Failing assertions are a different thing.

  This is essentially:

```
$events->grep(has_errors => @{$PARAMS{args}})
```

  It returns a new instance containing only the events that have errors.

- $events->plans(%PARAMS)

  This is essentially:

```
$events->grep(has_plan => @{$PARAMS{args}})
```

  It returns a new instance containing only the events that set the plan.

- $events->causes_fail(%PARAMS)
- $events->causes_failure(%PARAMS)

  These are essentially:

```
$events->grep(causes_fail    => @{$PARAMS{args}})
$events->grep(causes_failure => @{$PARAMS{args}})
```

  Note: `causes_fail()` and `causes_failure()` are both aliases for each other in events, so these methods are effectively aliases here as well.

  It returns a new instance containing only the events that cause failure.

MAPPING

These methods **ALWAYS** return an arrayref.

Note: No methods on *Test2::API::InterceptResult::Event* alter the event in any way.

Important Notes about Events:

Test2::API::InterceptResult::Event was tailor-made to be used in event-lists. Most methods that are not applicable to a given event will return an empty list, so you normally do not need to worry about unwanted undef values or exceptions being thrown. Mapping over event methods is an intended use, so it works well to produce lists.

Exceptions to the rule:

Some methods such as `causes_fail` always return a boolean true or false for all events. Any method prefixed with `the_` conveys the intent that the event should have exactly 1 of something, so those will throw an exception when that condition is not true.

- $arrayref = $events->map($CALL, %PARAMS)

  This is essentially:

```
[ map { $_->$CALL(@{ $PARAMS{args} }) } $events->upgrade->event_
↪list ];
```

  $CALL may be either the name of a method on *Test2::API::InterceptResult::Event*, or a coderef.

- $arrayref = $events->flatten(%PARAMS)

  This is essentially:

```
[ map { $_->flatten(@{ $PARAMS{args} }) } $events->upgrade->
↪event_list ];
```

  It returns a new list of flattened structures.

  See *Test2::API::InterceptResult::Event* for details on what `flatten()` returns.

- $arrayref = $events->briefs(%PARAMS)

  This is essentially:

```
[ map { $_->briefs(@{ $PARAMS{args} }) } $events->upgrade->
↪event_list ];
```

  It returns a new list of event briefs.

  See *Test2::API::InterceptResult::Event* for details on what `brief()` returns.

- $arrayref = $events->summaries(%PARAMS)

  This is essentially:

```
[ map { $_->summaries(@{ $PARAMS{args} }) } $events->upgrade->
↪event_list ];
```

  It returns a new list of event summaries.

  See *Test2::API::InterceptResult::Event* for details on what `summary()` returns.

- $arrayref = $events->subtest_results(%PARAMS)

  This is essentially:

```
[ map { $_->subtest_result(@{ $PARAMS{args} }) } $events->
↪upgrade->event_list ];
```

It returns a new list of event summaries.

See *Test2::API::InterceptResult::Event* for details on what `subtest_result()` returns.

- $arrayref = $events->diag_messages(%PARAMS)

  This is essentially:

```
[ map { $_->diag_messages(@{ $PARAMS{args} }) } $events->
↪upgrade->event_list ];
```

  It returns a new list of diagnostic messages (strings).

  See *Test2::API::InterceptResult::Event* for details on what `diag_messages()` returns.

- $arrayref = $events->note_messages(%PARAMS)

  This is essentially:

```
[ map { $_->note_messages(@{ $PARAMS{args} }) } $events->
↪upgrade->event_list ];
```

  It returns a new list of notification messages (strings).

  See *Test2::API::InterceptResult::Event* for details on what `note_messages()` returns.

- $arrayref = $events->error_messages(%PARAMS)

  This is essentially:

```
[ map { $_->error_messages(@{ $PARAMS{args} }) } $events->
↪upgrade->event_list ];
```

  It returns a new list of error messages (strings).

  See *Test2::API::InterceptResult::Event* for details on what `error_messages()` returns.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Modules

- `Test2::API::InterceptResult::Event` - Representation of an event for use in testing other test tools.
- `Test2::API::InterceptResult::Hub` - Hub used by InterceptResult.
- `Test2::API::InterceptResult::Squasher` - Encapsulation of the algorithm that squashes diags into assertions.

Test2::API::InterceptResult::Event

Representation of an event for use in testing other test tools.

DESCRIPTION

`intercept { ... }` from *Test2::API* returns an instance of *Test2::API::InterceptResult* which is a blessed arrayref of *Test2::API::InterceptResult::Event* objects.

This POD documents the methods of these events, which are mainly provided for you to use when testing your test tools.

SYNOPSIS

```
use Test2::V1 '-iPp';
use Test2::API qw/intercept/;

my $events = intercept {
    ok(1, "A passing assertion");
    plan(1);
};

# This will convert all events into instances of
# Test2::API::InterceptResult::Event. Until we do this they are the
# original Test::Event::* instances
$events->upgrade(in_place => 1);

# Now we can get individual events in this form
my $assert = $events->[0];
my $plan   = $events->[1];

# Or we can operate on all events at once:
my $flattened = $events->flatten;
```

(continues on next page)

(continued from previous page)

```
is(
    $flattened,
    [
      {
        causes_failure => 0,

        name => 'A passing assertion',
        pass => 1,

        trace_file => 'xxx.t',
        trace_line => 5,
      },
      {
        causes_failure => 0,

        plan => 1,

        trace_file => 'xxx.t',
        trace_line => 6,
      },
    ],
    "Flattened both events and returned an arrayref of the results
);
```

METHODS

!!! IMPORTANT NOTES ON DESIGN !!!

Please pay attention to what these return, many return a scalar when applicable or an empty list when not (as opposed to undef). Many also always return a list of 0 or more items. Some always return a scalar. Note that none of the methods care about context, their behavior is consistent regardless of scalar, list, or void context.

This was done because this class was specifically designed to be used in a list and generate more lists in bulk operations. Sometimes in a map you want nothing to show up for the event, and you do not want an undef in its place. In general single event instances are not going to be used alone, though that is allowed.

As a general rule any method prefixed with `the_` implies the event should have exactly 1 of the specified item, and and exception will be thrown if there are 0, or more than 1 of the item.

ATTRIBUTES

- $hashref = $event->facet_data

  This will return the facet data hashref, which is all Test2 cares about for any given event.

- $class = $event->result_class

This is normally *Test2::API::InterceptResult*. This is set at construction so that subtest results can be turned into instances of it on demand.

DUPLICATION

- $copy = $event->clone

  Create a deep copy of the event. Modifying either event will not affect the other.

CONDENSED MULTI-FACET DATA

- $bool = $event->causes_failure
- $bool = $event->causes_fail

  These are both aliases of the same functionality.

  This will always return either a true value, or a false value. This never returns a list.

  This method may be relatively slow (still super fast) because it determines pass or fail by creating an instance of *Test2::Hub* and asking it to process the event, and then asks the hub for its pass/fail state. This is slower than building in logic to do the check, but it is more reliable as it will always tell you what the hub thinks, so the logic will never be out of date relative to the Test2 logic that actually cares.

- STRING_OR_EMPTY_LIST = $event->brief

  Not all events have a brief, some events are not rendered by the formatter, others have no "brief" data worth seeing. When this is the case an empty list is returned. This is done intentionally so it can be used in a map operation without having `undef` being included in the result.

  When a brief can be generated it is always a single 1-line string, and is returned as-is, not in a list.

  Possible briefs:

```
# From control facets
"BAILED OUT"
"BAILED OUT: $why"

# From error facets
"ERROR"
"ERROR: $message"
"ERROR: $partial_message [...]"
"ERRORS: $first_error_message [...]"

# From assert facets
"PASS"
"FAIL"
"PASS with amnesty"
"FAIL with amnesty"

# From plan facets
```

(continues on next page)

(continued from previous page)

```
"PLAN $count"
"NO PLAN"
"SKIP ALL"
"SKIP ALL: $why"
```

Note that only the first applicable brief is returned. This is essentially a poor-mans TAP that only includes facets that could (but not necessarily do) cause a failure.

- $hashref = $event->flatten
- $hashref = $event->flatten(include_subevents => 1)

  This ALWAYS returns a hashref. This puts all the most useful data for the most interesting facets into a single hashref for easy validation.

  If there are no meaningful facets this will return an empty hashref.

  If given the 'include_subevents' parameter it will also include subtest data:

  Here is a list of EVERY possible field. If a field is not applicable it will not be present.

  – always present

  ```
  causes_failure => 1,    # Always present
  ```

  – Present if the event has a trace facet

  ```
  trace_line    => 42,
  trace_file    => 'Foo/Bar.pm',
  trace_details => 'Extra trace details',    # usually not␣
  ↪present
  ```

  – If an assertion is present

  ```
  pass => 0,
  name => "1 + 1 = 2, so math works",
  ```

  – If a plan is present:

  ```
  plan => $count_or_SKIP_ALL_or_NO_PLAN,
  ```

  – If amnesty facets are present

  You get an array for each type that is present.

  ```
  todo => [    # Yes you could be under multiple todos, this␣
  ↪will list them all.
      "I will fix this later",
      "I promise to fix these",
  ],

  skip => ["This will format the main drive, do not run"],

  ... => ["Other amnesty"]
  ```

– If Info (note/diag) facets are present

  You get an arrayref for any that are present, the key is not defined if they are not present.

```
diag => [
    "Test failed at Foo/Bar.pm line 42",
    "You forgot to tie your boots",
],

note => ["Your boots are red"],

...  => ["Other info"],
```

– If error facets are present

  Always an arrayref

```
error => [
    "non fatal error (does not cause test failure, just an␣
↪FYI",
    "FATAL: This is a fatal error (causes failure)",
],

# Errors can have alternative tags, but in practice are␣
↪always 'error',
# listing this for completeness.
... => [ ... ]
```

– Present if the event is a subtest

```
subtest => {
    count      => 2,    # Number of assertions made
    failed     => 1,    # Number of test failures seen
    is_passing => 0,    # Boolean, true if the test would be␣
↪passing
                        # after the events are processed.

    plan         => 2,  # Plan, either a number, undef, 'SKIP
↪', or 'NO PLAN'
    follows_plan => 1,  # True if there is a plan and it was␣
↪followed.
                        # False if the plan and assertions␣
↪did not
                        # match, undef if no plan was␣
↪present in the
                        # event list.

    bailed_out => "foo",    # if there was a bail-out in the
                            # events in this will be a␣
↪string explaining
                            # why there was a bailout, if no␣
```

(continues on next page)

(continued from previous page)

```
↪reason was
                                # given this will simply be set␣
↪to true (1).

    skip_reason => "foo",    # If there was a skip_all this␣
↪will give the
                                # reason.
},
```

if (`include_subtest => 1`) was provided as a parameter then the following will be included. This is the result of turning all subtest child events into an *Test2::API::InterceptResult* instance and calling the `flatten` method on it.

```
subevents => Test2::API::InterceptResult->new(@child_events)-
↪>flatten(...),
```

– If a bail-out is being requested

If no reason was given this will be set to 1.

```
bailed_out => "reason",
```

- $hashref = $event->summary()

This returns a limited summary. See `flatten()`, which is usually a better option.

```
{
    brief => $event->brief || '',

    causes_failure => $event->causes_failure,

    trace_line    => $event->trace_line,
    trace_file    => $event->trace_file,
    trace_tool    => $event->trace_subname,
    trace_details => $event->trace_details,

    facets => [ sort keys(%{$event->{+FACET_DATA}}) ],
}
```

DIRECT ARBITRARY FACET ACCESS

- @list_of_facets = $event->facet($name)

This always returns a list of 0 or more items. This fetches the facet instances from the event. For facets like 'assert' this will always return 0 or 1 item. For events like 'info' (diags, notes) this will return 0 or more instances, once for each instance of the facet.

These will be blessed into the proper *Test2::EventFacet* subclass. If no subclass can be found it will be blessed as an Test2::API::InterceptResult::Facet generic facet class.

- $undef_or_facet = $event->the_facet($name)

  If you know you will have exactly 1 instance of a facet you can call this.

  If you are correct and there is exactly one instance of the facet it will always return the hashref.

  If there are 0 instances of the facet this will return undef, not an empty list.

  If there are more than 1 instance this will throw an exception because your assumption was incorrect.

TRACE FACET

- @list_of_facets = $event->trace

  TODO

- $undef_or_hashref = $event->the_trace

  This returns the trace hashref, or undef if it is not present.

- $undef_or_arrayref = $event->frame

  If a trace is present, and has a caller frame, this will be an arrayref:

```
[$package, $file, $line, $subname]
```

  If the trace is not present, or has no caller frame this will return undef.

- $undef_or_string = $event->trace_details

  This is usually undef, but occasionally has a string that overrides the file/line number debugging a trace usually provides on test failure.

- $undef_or_string = $event->trace_package

  Same as `(caller())[0]`, the first element of the trace frame.

  Will be undef if not present.

- $undef_or_string = $event->trace_file

  Same as `(caller())[1]`, the second element of the trace frame.

  Will be undef if not present.

- $undef_or_integer = $event->trace_line

  Same as `(caller())[2]`, the third element of the trace frame.

  Will be undef if not present.

- $undef_or_string = $event->trace_subname

- $undef_or_string = $event->trace_tool

  Aliases for the same thing

  Same as `(caller($level))[4]`, the fourth element of the trace frame.

  Will be undef if not present.

- $undef_or_string = $event->trace_signature

  A string that is a unique signature for the trace. If a single context generates multiple events they will all have the same signature. This can be used to tie assertions and diagnostics sent as separate events together after the fact.

ASSERT FACET

- $bool = $event->has_assert

  Returns true if the event has an assert facet, false if it does not.

- $undef_or_hashref = $event->the_assert

  Returns the assert facet if present, undef if it is not.

- @list_of_facets = $event->assert

  TODO

- EMPTY_LIST_OR_STRING = $event->assert_brief

  Returns a string giving a brief of the assertion if an assertion is present. Returns an empty list if no assertion is present.

SUBTESTS (PARENT FACET)

- $bool = $event->has_subtest

  True if a subetest is present in this event.

- $undef_or_hashref = $event->the_subtest

  Get the one subtest if present, otherwise undef.

- @list_of_facets = $event->subtest

  TODO

- EMPTY_LIST_OR_OBJECT = $event->subtest_result

  Returns an empty list if there is no subtest.

  Get an instance of *Test2::API::InterceptResult* representing the subtest.

CONTROL FACET (BAILOUT, ENCODING)

- $bool = $event->has_bailout

  True if there was a bailout

- $undef_hashref = $event->the_bailout

  Return the control facet if it requested a bailout.

- EMPTY_LIST_OR_HASHREF = $event->bailout

  Get a list of 0 or 1 hashrefs. The hashref will be the control facet if a bail-out was requested.

- EMPTY_LIST_OR_STRING = $event->bailout_brief

  Get the brief of the bailout if present.

- EMPTY_LIST_OR_STRING = $event->bailout_reason

  Get the reason for the bailout, an empty string if no reason was provided, or an empty list if there was no bailout.

PLAN FACET

TODO

- $bool = $event->has_plan
- $undef_or_hashref = $event->the_plan
- @list_if_hashrefs = $event->plan
- EMPTY_LIST_OR_STRING $event->plan_brief

AMNESTY FACET (TODO AND SKIP)

TODO

- $event->has_amnesty
- $event->the_amnesty
- $event->amnesty
- $event->amnesty_reasons
- $event->has_todos
- $event->todos
- $event->todo_reasons
- $event->has_skips
- $event->skips
- $event->skip_reasons
- $event->has_other_amnesty
- $event->other_amnesty
- $event->other_amnesty_reasons

ERROR FACET (CAPTURED EXCEPTIONS)

TODO

- $event->has_errors
- $event->the_errors
- $event->errors
- $event->error_messages

- $event->error_brief

INFO FACET (DIAG, NOTE)

TODO

- $event->has_info
- $event->the_info
- $event->info
- $event->info_messages
- $event->has_diags
- $event->diags
- $event->diag_messages
- $event->has_notes
- $event->notes
- $event->note_messages
- $event->has_other_info
- $event->other_info
- $event->other_info_messages

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::API::InterceptResult::Hub

Hub used by InterceptResult.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::API::InterceptResult::Squasher

Encapsulation of the algorithm that squashes diags into assertions.

DESCRIPTION

Internal use only, please ignore.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::API::Stack

Object to manage a stack of *Test2::Hub* instances.

INTERNALS NOTE

The internals of this package are subject to change at any time! The public methods provided will not change in backwards incompatible ways, but the underlying implementation details might. **Do not break encapsulation here!**

DESCRIPTION

This module is used to represent and manage a stack of *Test2::Hub* objects. Hubs are usually in a stack so that you can push a new hub into place that can intercept and handle events differently than the primary hub.

SYNOPSIS

```
my $stack = Test2::API::Stack->new;
my $hub = $stack->top;
```

METHODS

- $stack = Test2::API::Stack->new()

  This will create a new empty stack instance. All arguments are ignored.

- $hub = $stack->new_hub()
- $hub = $stack->new_hub(%params)
- $hub = $stack->new_hub(%params, class => $class)

  This will generate a new hub and push it to the top of the stack. Optionally you can provide arguments that will be passed into the constructor for the *Test2::Hub* object.

  If you specify the `'class' => $class` argument, the new hub will be an instance of the specified class.

  Unless your parameters specify `'formatter'` or `'ipc'` arguments, the formatter and IPC instance will be inherited from the current top hub. You can set the parameters to `undef` to avoid having a formatter or IPC instance.

  If there is no top hub, and you do not ask to leave IPC and formatter undef, then a new formatter will be created, and the IPC instance from *Test2::API* will be used.

- $hub = $stack->top()

  This will return the top hub from the stack. If there is no top hub yet this will create it.

- $hub = $stack->peek()

  This will return the top hub from the stack. If there is no top hub yet this will return undef.

- $stack->cull

  This will call `$hub->cull` on all hubs in the stack.

- @hubs = $stack->all

  This will return all the hubs in the stack as a list.

- $stack->clear

  This will completely remove all hubs from the stack. Normally you do not want to do this, but there are a few valid reasons for it.

- $stack->push($hub)

  This will push the new hub onto the stack.

- $stack->pop($hub)

  This will pop a hub from the stack, if the hub at the top of the stack does not match the hub you expect (passed in as an argument) it will throw an exception.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::AsyncSubtest

Object representing an async subtest.

DESCRIPTION

Regular subtests have a limited scope, they start, events are generated, then they close and send an *Test2::Event::Subtest* event. This is a problem if you want the subtest to keep receiving events while other events are also being generated. This class implements subtests that stay open until you decide to close them.

This is mainly useful for tools that start a subtest in one process and then spawn children. In many cases it is nice to let the parent process continue instead of waiting on the children.

SYNOPSIS

```
use Test2::AsyncSubtest;

my $ast = Test2::AsyncSubtest->new(name => foo);

$ast->run(sub {
    ok(1, "Event in parent" );
});

ok(1, "Event outside of subtest");

$ast->run_fork(sub {
    ok(1, "Event in child process");
});

...

$ast->finish;

done_testing;
```

CONSTRUCTION

```
my $ast = Test2::AsyncSubtest->new( ... );
```

- name => $name (required)

  Name of the subtest. This construction argument is required.

- send_to => $hub (optional)

  Hub to which the final subtest event should be sent. This must be an instance of *Test2::Hub* or a subclass. If none is specified then the current top hub will be used.

- trace => $trace (optional)

File/Line to which errors should be attributed. This must be an instance of *Test2::Util::Trace*. If none is specified then the file/line where the constructor was called will be used.

- hub => $hub (optional)

  Use this to specify a hub the subtest should use. By default a new hub is generated. This must be an instance of *Test2::AsyncSubtest::Hub*.

METHODS

SIMPLE ACCESSORS

- $bool = $ast->active

  True if the subtest is active. The subtest is active if its hub appears in the global hub stack. This is true when `$ast->run(...)` us running.

- $arrayref = $ast->children

  Get an arrayref of child processes/threads. Numerical items are PIDs, blessed items are *threads* instances.

- $arrayref = $ast->events

  Get an arrayref of events that have been sent to the subtests hub.

- $bool = $ast->finished

  True if `finished()` has already been called.

- $hub = $ast->hub

  The hub created for the subtest.

- $int = $ast->id

  Attach/Detach counter. Used internally, not useful to users.

- $str = $ast->name

  Name of the subtest.

- $pid = $ast->pid

  PID in which the subtest was created.

- $tid = $ast->tid

  Thread ID in which the subtest was created.

- $hub = $ast->send_to

  Hub to which the final subtest event should be sent.

- $arrayref = $ast->stack

  Stack of async subtests at the time this one was created. This is mainly for internal use.

- $trace = $ast->trace

  Test2::Util::Trace instance used for error reporting.

INTERFACE

- $ast->attach($id)

  Attach a subtest in a child/process to the original.

  Note: `my $id = $ast->cleave` must have been called in the parent process/thread before the child was started, the id it returns must be used in the call to `$ast->attach($id)`

- $id = $ast->cleave

  Prepare a slot for a child process/thread to attach. This must be called BEFORE the child process or thread is started. The ID returned is used by `attach()`.

  This must only be called in the original process/thread.

- $ctx = $ast->context

  Get an *Test2::API::Context* instance that can be used to send events to the context in which the hub was created. This is not a canonical context, you should not call `$ctx->release` on it.

- $ast->detach

  Detach from the parent in a child process/thread. This should be called just before the child exits.

- $ast->finish

- $ast->finish(%options)

  Finish the subtest, wait on children, and send the final subtest event.

  This must only be called in the original process/thread.

  Note: This calls `$ast->wait`.

  These are the options:

  - collapse => 1

    This intelligently allows a subtest to be empty.

    If no events bump the test count then the subtest no final plan will be added. The subtest will not be considered a failure (normally an empty subtest is a failure).

    If there are no events at all the subtest will be collapsed into an *Test2::Event::Ok* event.

  - silent => 1

    This will prevent finish from generating a final *Test2::Event::Subtest* event. This effectively ends the subtest without it affecting the parent subtest (or top level test).

  - no_plan => 1

    This will prevent a final plan from being added to the subtest for you when none is directly specified.

  - skip => “reason”

    This will issue an *Test2::Event::Skip* instead of a subtest. This will throw an exception if any events have been seen, or if state implies events have occurred.

- $out = $ast->fork

  This is a slightly higher level interface to fork. Running it will fork your code in-place just like `fork()`. It will return a pid in the parent, and an *Test2::Util::Guard* instance in the child. An exception will be thrown if fork fails.

  It is recommended that you use `$ast->run_fork(sub { ... })` instead.

- $bool = $ast->pending

  True if there are child processes, threads, or subtests that depend on this one.

- $bool = $ast->ready

  This is essentially `!$ast->pending`.

- $ast->run(sub { ... })

  Run the provided codeblock inside the subtest. This will push the subtest hub onto the stack, run the code, then pop the hub off the stack.

- $pid = $ast->run_fork(sub { ... })

  Same as `$ast->run()`, except that the codeblock is run in a child process.

  You do not need to directly call `wait($pid)`, that will be done for you when `$ast->wait`, or `$ast->finish` are called.

- my $thr = $ast->run_thread(sub { ... });

  ** **DISCOURAGED** ** Threads cause problems. This method remains for anyone who REALLY wants it, but it is no longer supported. Tests for this functionality do not even run unless the AUTHOR_TESTING or T2_DO_THREAD_TESTS env vars are enabled.

  Same as `$ast->run()`, except that the codeblock is run in a child thread.

  You do not need to directly call `$thr->join`, that is done for you when `$ast->wait`, or `$ast->finish` are called.

- $passing = $ast->start

  Push the subtest hub onto the stack. Returns the current pass/fail status of the subtest.

- $ast->stop

  Pop the subtest hub off the stack. Returns the current pass/fail status of the subtest.

- $ast->wait

  Wait on all threads/processes that were started using `$ast->fork`, `$ast->run_fork`, or `$ast->run_thread`.

SOURCE

The source code repository for Test2-AsyncSubtest can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist7@gmail.com.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Modules

- `Test2::AsyncSubtest::Event`
- `Test2::AsyncSubtest::Hub` - Hub used by async subtests.

Test2::AsyncSubtest::Event

Namespace containing the following modules:

- `Test2::AsyncSubtest::Event::Attach` - Event to attach a subtest to the parent.
- `Test2::AsyncSubtest::Event::Detach` - Event to detach a subtest from the parent.

Test2::AsyncSubtest::Event::Attach

Event to attach a subtest to the parent.

DESCRIPTION

Used internally by *Test2::AsyncSubtest*. No user serviceable parts inside.

SOURCE

The source code repository for Test2-AsyncSubtest can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist7@gmail.com.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::AsyncSubtest::Event::Detach

Event to detach a subtest from the parent.

DESCRIPTION

Used internally by *Test2::AsyncSubtest*. No user serviceable parts inside.

SOURCE

The source code repository for Test2-AsyncSubtest can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist7@gmail.com.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::AsyncSubtest::Hub

Hub used by async subtests.

DESCRIPTION

This is a subclass of *Test2::Hub::Subtest* used for async subtests.

SYNOPSIS

You should not use this directly.

METHODS

- $ast = $hub->ast

  Get the *Test2::AsyncSubtest* object to which this hub is bound.

SOURCE

The source code repository for Test2-AsyncSubtest can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist7@gmail.com.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Bundle

Documentation for bundles.

DESCRIPTION

Bundles are collections of Tools and Plugins. Bundles should not provide any tools or behaviors of their own, they should simply combine the tools and behaviors of other packages.

FAQ

- Should my bundle subclass Test2::Bundle?

  No. Currently this class is empty. Eventually we may want to add behavior, in which case we do not want anyone to already be subclassing it.

HOW DO I WRITE A BUNDLE?

Writing a bundle can be very simple:

```
package Test2::Bundle::MyBundle;
use strict;
use warnings;

use Test2::Plugin::ExitSummary; # Load a plugin

use Test2::Tools::Basic qw/ok plan done_testing/;

# Re-export the tools
our @EXPORTS = qw/ok plan done_testing/;
use base 'Exporter';

1;
```

If you want to do anything more complex you should look into Import::Into and Symbol::Move.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Modules

- `Test2::Bundle::Extended` - Old name for Test2::V0
- `Test2::Bundle::More` - ALMOST a drop-in replacement for Test::More.
- `Test2::Bundle::Simple` - ALMOST a drop-in replacement for Test::Simple.

Test2::Bundle::Extended

Old name for Test2::V0

*** DEPRECATED ***

This bundle has been renamed to *Test2::V0*, in which the `':v1'` tag has been removed as unnecessary.

Note: *Test2::V1* is the latest bundle, you probably want to look at that.

DESCRIPTION

This is the big-daddy bundle. This bundle includes nearly every tool, and several plugins, that the Test2 author uses. This bundle is used extensively to test *Test2::Suite* itself.

SYNOPSIS

```
use Test2::Bundle::Extended ':v1';

ok(1, "pass");

...

done_testing;
```

RESOLVING CONFLICTS WITH MOOSE

```
use Test2::Bundle::Extended '!meta';
```

Moose and *Test2::Bundle::Extended* both export very different `meta()` subs. Adding `'!meta'` to the import args will prevent the sub from being imported. This bundle also exports the sub under the name `meta_check()` so you can use that spelling as an alternative.

TAGS

- :v1
- :DEFAULT

  The following are all identical:

```
use Test2::Bundle::Extended;

use Test2::Bundle::Extended ':v1';

use Test2::Bundle::Extended ':DEFAULT';
```

RENAMING ON IMPORT

```
use Test2::Bundle::Extended ':v1', '!ok', ok => {-as => 'my_ok'};
```

This bundle uses Importer for exporting, as such you can use any arguments it accepts.

Explanation:

- ':v1'

  Use the default tag, all default exports.

- '!ok'

  Do not export `ok()`

- ok => {-as => 'my_ok'}

  Actually, go ahead and import `ok()` but under the name `my_ok()`.

If you did not add the `'!ok'` argument then you would have both `ok()` and `my_ok()`

PRAGMAS

All of these can be disabled via individual import arguments, or by the `-no_pragmas` argument.

```
use Test2::Bundle::Extended -no_pragmas => 1;
```

STRICT

strict is turned on for you. You can disable this with the `-no_strict` or `-no_pragmas` import arguments:

```
use Test2::Bundle::Extended -no_strict => 1;
```

WARNINGS

warnings are turned on for you. You can disable this with the `-no_warnings` or `-no_pragmas` import arguments:

```
use Test2::Bundle::Extended -no_warnings => 1;
```

UTF8

This is actually done via the *Test2::Plugin::UTF8* plugin, see the "PLUGINS" section for details.

Note: `-no_pragmas => 1` will turn off the entire plugin.

PLUGINS

SRAND

See *Test2::Plugin::SRand*.

This will set the random seed to today's date. You can provide an alternate seed with the `-srand` import option:

```
use Test2::Bundle::Extended -srand => 1234;
```

UTF8

See *Test2::Plugin::UTF8*.

This will set the file, and all output handles (including formatter handles), to utf8. This will turn on the utf8 pragma for the current scope.

This can be disabled using the `-no_utf8 => 1` or `-no_pragmas => 1` import arguments.

```
use Test2::Bundle::Extended -no_utf8 => 1;
```

EXIT SUMMARY

See *Test2::Plugin::ExitSummary*.

This plugin has no configuration.

API FUNCTIONS

See *Test2::API* for these

- $ctx = context()
- $events = intercept { ... }

TOOLS

TARGET

See *Test2::Tools::Target*.

You can specify a target class with the `-target` import argument. If you do not provide a target then `$CLASS` and `CLASS()` will not be imported.

```
use Test2::Bundle::Extended -target => 'My::Class';

print $CLASS;  # My::Class
print CLASS(); # My::Class
```

Or you can specify names:

```
use Test2::Bundle::Extended -target => { pkg => 'Some::Package' };

pkg()->xxx; # Call 'xxx' on Some::Package
$pkg->xxx;  # Same
```

- $CLASS

  Package variable that contains the target class name.

- $class = CLASS()

  Constant function that returns the target class name.

DEFER

See *Test2::Tools::Defer*.

- def $func => @args;
- do_def()

BASIC

See *Test2::Tools::Basic*.

- ok($bool, $name)
- pass($name)
- fail($name)
- diag($message)
- note($message)
- $todo = todo($reason)
- todo $reason => sub { ... }
- skip($reason, $count)
- plan($count)
- skip_all($reason)
- done_testing()
- bail_out($reason)

COMPARE

See *Test2::Tools::Compare*.

- is($got, $want, $name)
- isnt($got, $do_not_want, $name)
- like($got, qr/match/, $name)
- unlike($got, qr/mismatch/, $name)
- $check = match(qr/pattern/)
- $check = mismatch(qr/pattern/)
- $check = validator(sub { return $bool })
- $check = hash { ... }
- $check = array { ... }
- $check = bag { ... }
- $check = object { ... }
- $check = meta { ... }
- $check = number($num)
- $check = string($str)
- $check = check_isa($class_name)
- $check = in_set(@things)
- $check = not_in_set(@things)
- $check = check_set(@things)
- $check = item($thing)
- $check = item($idx => $thing)
- $check = field($name => $val)
- $check = call($method => $expect)
- $check = call_list($method => $expect)
- $check = call_hash($method => $expect)
- $check = prop($name => $expect)
- $check = check($thing)
- $check = T()
- $check = F()
- $check = D()
- $check = DF()
- $check = E()
- $check = DNE()

- $check = FDNE()
- $check = U()
- $check = L()
- $check = exact_ref($ref)
- end()
- etc()
- filter_items { grep { ... } @_ }
- $check = event $type => ...
- @checks = fail_events $type => ...

CLASSIC COMPARE

See *Test2::Tools::ClassicCompare*.

- cmp_ok($got, $op, $want, $name)

SUBTEST

See *Test2::Tools::Subtest*.

- subtest $name => sub { ... }

  (Note: This is called `subtest_buffered()` in the Tools module.)

CLASS

See *Test2::Tools::Class*.

- can_ok($thing, @methods)
- isa_ok($thing, @classes)
- DOES_ok($thing, @roles)

ENCODING

See *Test2::Tools::Encoding*.

- set_encoding($encoding)

EXPORTS

See *Test2::Tools::Exports*.

- imported_ok('function', '$scalar', ...)
- not_imported_ok('function', '$scalar', ...)

REF

See *Test2::Tools::Ref*.

- ref_ok($ref, $type)
- ref_is($got, $want)
- ref_is_not($got, $do_not_want)

MOCK

See *Test2::Tools::Mock*.

- $control = mock ...
- $bool = mocked($thing)

EXCEPTION

See *Test2::Tools::Exception*.

- $exception = dies { ... }
- $bool = lives { ... }
- $bool = try_ok { ... }

WARNINGS

See *Test2::Tools::Warnings*.

- $count = warns { ... }
- $warning = warning { ... }
- $warnings_ref = warnings { ... }
- $bool = no_warnings { ... }

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Bundle::More

ALMOST a drop-in replacement for Test::More.

DESCRIPTION

This bundle is intended to be a (mostly) drop-in replacement for *Test::More*. See "KEY DIFFERENCES FROM Test::More" for details.

SYNOPSIS

```
use Test2::Bundle::More;

ok(1, "pass");

...

done_testing;
```

PLUGINS

This loads *Test2::Plugin::ExitSummary*.

TOOLS

These are from *Test2::Tools::Basic*. See *Test2::Tools::Basic* for details.

- ok($bool, $name)
- pass($name)
- fail($name)
- skip($why, $count)
- $todo = todo($why)
- diag($message)
- note($message)
- plan($count)
- skip_all($why)
- done_testing()

- BAIL_OUT($why)

These are from *Test2::Tools::ClassicCompare*. See *Test2::Tools::ClassicCompare* for details.

- is($got, $want, $name)
- isnt($got, $donotwant, $name)
- like($got, qr/match/, $name)
- unlike($got, qr/mismatch/, $name)
- is_deeply($got, $want, "Deep compare")
- cmp_ok($got, $op, $want, $name)

These are from *Test2::Tools::Class*. See *Test2::Tools::Class* for details.

- isa_ok($thing, @classes)
- can_ok($thing, @subs)

This is from *Test2::Tools::Subtest*. It is called `subtest_streamed()` in that package.

- subtest $name => sub { ... }

KEY DIFFERENCES FROM Test::More

- You cannot plan at import.

  THIS WILL **NOT** WORK:

  ```
  use Test2::Bundle::More tests => 5;
  ```

  Instead you must plan in a separate statement:

  ```
  use Test2::Bundle::More;
  plan 5;
  ```

- You have three subs imported for use in planning

  Use `plan($count)`, `skip_all($reason)`, or `done_testing()` for your planning.

- isa_ok accepts different arguments

  `isa_ok` in Test::More was:

  ```
  isa_ok($thing, $isa, $alt_thing_name);
  ```

  This was very inconsistent with tools like `can_ok($thing, @subs)`.

  In Test2::Bundle::More, `isa_ok()` takes a `$thing` and a list of `@isa`.

  ```
  isa_ok($thing, $class1, $class2, ...);
  ```

THESE FUNCTIONS AND VARIABLES HAVE BEEN REMOVED

- $TODO

  See `todo()`.

- use_ok()
- require_ok()

  These are not necessary. Use `use` and `require` directly. If there is an error loading the module the test will catch the error and fail.

- todo_skip()

  Not necessary.

- eq_array()
- eq_hash()
- eq_set()

  Discouraged in Test::More.

- explain()

  This started a fight between Test developers, who may now each write their own implementations in *Test2*. (See explain in Test::Most vs *Test::More*. Hint: Test::Most wrote it first, then Test::More added it, but broke compatibility).

- new_ok()

  Not necessary.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Bundle::Simple

ALMOST a drop-in replacement for Test::Simple.

DESCRIPTION

This bundle is intended to be a (mostly) drop-in replacement for *Test::Simple*. See "KEY DIFFERENCES FROM Test::Simple" for details.

SYNOPSIS

```
use Test2::Bundle::Simple;

ok(1, "pass");

done_testing;
```

PLUGINS

This loads *Test2::Plugin::ExitSummary*.

TOOLS

These are all from *Test2::Tools::Basic*.

- ok($bool, $name)

  Run a test. If bool is true, the test passes. If bool is false, it fails.

- plan($count)

  Tell the system how many tests to expect.

- skip_all($reason)

  Tell the system to skip all the tests (this will exit the script).

- done_testing();

  Tell the system that all tests are complete. You can use this instead of setting a plan.

KEY DIFFERENCES FROM Test::Simple

- You cannot plan at import.

  THIS WILL **NOT** WORK:

  ```
  use Test2::Bundle::Simple tests => 5;
  ```

  Instead you must plan in a separate statement:

  ```
  use Test2::Bundle::Simple;
  plan 5;
  ```

- You have three subs imported for use in planning

  Use `plan($count)`, `skip_all($reason)`, or `done_testing()` for your planning.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Compare

Test2 extension for writing deep comparison tools.

DESCRIPTION

This library is the driving force behind deep comparison tools such as `Test2::Tools::Compare::is()` and `Test2::Tools::ClassicCompare::is_deeply()`.

SYNOPSIS

```
package Test2::Tools::MyCheck;

use Test2::Compare::MyCheck;
use Test2::Compare qw/compare/;

sub MyCheck {
    my ($got, $exp, $name, @diag) = @_;
    my $ctx = context();

    my $delta = compare($got, $exp, \&convert);

    if ($delta) {
```

(continues on next page)

(continued from previous page)

```
        $ctx->fail($name, $delta->diag, @diag);
    }
    else {
        $ctx->ok(1, $name);
    }

    $ctx->release;
    return !$delta;
}

sub convert {
    my $thing = shift;
    return $thing if blessed($thing) && $thing->isa(
↪'Test2::Compare::MyCheck');

    return Test2::Compare::MyCheck->new(stuff => $thing);
}
```

See *Test2::Compare::Base* for details about writing a custom check.

EXPORTS

- $delta = compare($got, $expect, \&convert)

  This will compare the structures in `$got` with those in `$expect`, The convert sub should convert vanilla structures inside `$expect` into checks. If there are differences in the structures they will be reported back as an *Test2::Compare::Delta* tree.

- $build = get_build()

  Get the current global build, if any.

- push_build($build)

  Set the current global build.

- $build = pop_build($build)

  Unset the current global build. This will throw an exception if the build passed in is different from the current global.

- build($class, sub { ... })

  Run the provided codeblock with a new instance of `$class` as the current build. Returns the new build.

- $check = convert($thing)

- $check = convert($thing, $config)

  This convert function is used by `strict_convert()` and `relaxed_convert()` under the hood. It can also be used as the basis for other convert functions.

  If you want to use it with a custom configuration you should wrap it in another sub like so:

```
sub my_convert {
    my $thing_to_convert = shift;
    return convert(
        $thing_to_convert,
        { ... }
    );
}
```

Or the short variant:

```
sub my_convert { convert($_[0], { ... }) }
```

There are several configuration options, here they are with the default setting listed first:

- implicit_end => 1

  This option toggles array/hash boundaries. If this is true then no extra hash keys or array indexes will be allowed. This setting affects generated compare objects as well as any passed in.

- use_regex => 1

  This option toggles regex matching. When true (default) regexes are converted to checks such that values must match the regex. When false regexes will be compared to see if they are identical regexes.

- use_code => 0

  This option toggles code matching. When false (default) coderefs in structures must be the same coderef as specified. When true coderefs will be run to verify the value being checked.

- $check = strict_convert($thing)

  Convert `$thing` to an *Test2::Compare::** object. This will behave strictly which means it uses these settings:

  - implicit_end => 1

    Array bounds will be checked when this object is used in a comparison. No unexpected hash keys can be present.

  - use_code => 0

    Sub references will be compared as refs (IE are these sub refs the same ref?)

  - use_regex => 0

    Regexes will be compared directly (IE are the regexes the same?)

- $compare = relaxed_convert($thing)

  Convert `$thing` to an *Test2::Compare::** object. This will be relaxed which means it uses these settings:

  - implicit_end => 0

    Array bounds will not be checked when this object is used in a comparison. Unexpected hash keys can be present.

- use_code => 1

  Sub references will be run to verify a value.

- use_regex => 1

  Values will be checked against any regexes provided.

WRITING A VARIANT OF IS/LIKE

```
use Test2::Compare qw/compare convert/;

sub my_like($$;$@) {
    my ($got, $exp, $name, @diag) = @_;
    my $ctx = context();

    # A custom converter that does the same thing as the one used by like()
    my $convert = sub {
        my $thing = shift;
        return convert(
            $thing,
            {
                implicit_end => 0,
                use_code     => 1,
                use_regex    => 1,
            }
        );
    };

    my $delta = compare($got, $exp, $convert);

    if ($delta) {
        $ctx->fail($name, $delta->diag, @diag);
    }
    else {
        $ctx->ok(1, $name);
    }

    $ctx->release;
    return !$delta;
}
```

The work of a comparison tool is done by 3 entities:

- compare()

  The `compare()` function takes the structure you got, the specification you want to check against, and a `\&convert` sub that will convert anything that is not an instance of an *Test2::Compare::Base* subclass into one.

  This tool will use the `\&convert` function on the specification, and then produce an *Test2::Compare::Delta* structure that outlines all the ways the structure you got deviates from the specification.

- \&convert

  Converts anything that is not an instance of an *Test2::Compare::Base* subclass, and turns it into one. The objects this produces are able to check that a structure matches a specification.

- $delta

  An instance of *Test2::Compare::Delta* is ultimately returned. This object represents all the ways in with the structure you got deviated from the specification. The delta is a tree and may contain child deltas for nested structures.

  The delta is capable of rendering itself as a table, use `@lines = $delta->diag` to get the table (lines in `@lines` will not be terminated with `"\n"`).

The `convert()` function provided by this package contains all the specification behavior of `like()` and `is()`. It is intended to be wrapped in a sub that passes in a configuration hash, which allows you to control the behavior.

You are free to write your own `$check = compare($thing)` function, it just needs to accept a single argument, and produce a single instance of an *Test2::Compare::Base* subclass.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Modules

- `Test2::Compare::Array` - Internal representation of an array comparison.
- `Test2::Compare::Bag` - Internal representation of a bag comparison.
- `Test2::Compare::Base` - Base class for comparison classes.
- `Test2::Compare::Bool` - Compare two values as booleans
- `Test2::Compare::Custom` - Custom field check for comparisons.

- `Test2::Compare::DeepRef` - Ref comparison
- `Test2::Compare::Delta` - Representation of differences between nested data structures.
- `Test2::Compare::Event` - Event specific Object subclass.
- `Test2::Compare::EventMeta` - Meta class for events in deep comparisons
- `Test2::Compare::Float` - Compare two values as numbers with tolerance.
- `Test2::Compare::Hash` - Representation of a hash in a deep comparison.
- `Test2::Compare::Isa` - Check if the value is an instance of the class.
- `Test2::Compare::Meta` - Check library for meta-checks
- `Test2::Compare::Negatable` - Poor mans 'role' for compare objects that can be negated.
- `Test2::Compare::Number` - Compare two values as numbers
- `Test2::Compare::Object` - Representation of an object during deep comparison.
- `Test2::Compare::OrderedSubset` - Internal representation of an ordered subset.
- `Test2::Compare::Pattern` - Use a pattern to validate values in a deep comparison.
- `Test2::Compare::Ref` - Ref comparison
- `Test2::Compare::Regex` - Regex direct comparison
- `Test2::Compare::Scalar` - Representation of a Scalar Ref in deep comparisons
- `Test2::Compare::Set` - Allows a field to be matched against a set of checks.
- `Test2::Compare::String` - Compare two values as strings
- `Test2::Compare::Undef` - Check that something is undefined
- `Test2::Compare::Wildcard` - Placeholder check.

Test2::Compare::Array

Internal representation of an array comparison.

DESCRIPTION

This module is an internal representation of an array for comparison purposes.

METHODS

- $ref = $arr->inref()

  If the instance was constructed from an actual array, this will return the reference to that array.

- $bool = $arr->ending

- $arr->set_ending($bool)

  Set this to true if you would like to fail when the array being validated has more items than the check. That is, if you check indexes 0-3 but the array has values for indexes 0-4, it will fail and list that last item in the array as unexpected. If set to false then it is assumed you do not care about extra items.

- $hashref = $arr->items()

  Returns the hashref of `key => val` pairs to be checked in the array.

- $arr->set_items($hashref)

  Accepts a hashref to permit indexes to be skipped if desired.

  Note: that there is no validation when using `set_items`, it is better to use the `add_item` interface.

- $arrayref = $arr->order()

  Returns an arrayref of all indexes that will be checked, in order.

- $arr->set_order($arrayref)

  Sets the order in which indexes will be checked.

  Note: that there is no validation when using `set_order`, it is better to use the `add_item` interface.

- $name = $arr->name()

  Always returns the string `"<ARRAY>"`.

- $bool = $arr->verify(got => $got, exists => $bool)

  Check if `$got` is an array reference or not.

- $idx = $arr->top_index()

  Returns the topmost index which is checked. This will return undef if there are no items, or `0` if there is only 1 item.

- $arr->add_item($item)

  Push an item onto the list of values to be checked.

- $arr->add_item($idx => $item)

  Add an item to the list of values to be checked at the specified index.

- $arr->add_filter(sub { ... })

  Add a filter sub. The filter receives all remaining values of the array being checked, and should return the values that should still be checked. The filter will be run between the last item added and the next item added.

- @deltas = $arr->deltas(got => $got, convert => \&convert, seen => \%seen)

  Find the differences between the expected array values and those in the `$got` arrayref.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Compare::Bag

Internal representation of a bag comparison.

DESCRIPTION

This module is an internal representation of a bag for comparison purposes.

METHODS

- $bool = $arr->ending
- $arr->set_ending($bool)

  Set this to true if you would like to fail when the array being validated has more items than the check. That is, if you check for 4 items but the array has 5 values, it will fail and list that unmatched item in the array as unexpected. If set to false then it is assumed you do not care about extra items.

- $arrayref = $arr->items()

  Returns the arrayref of values to be checked in the array.

- $arr->set_items($arrayref)

  Accepts an arrayref.

  Note: that there is no validation when using `set_items`, it is better to use the `add_item` interface.

- $name = $arr->name()

  Always returns the string `"<BAG>"`.

- $bool = $arr->verify(got => $got, exists => $bool)

  Check if `$got` is an array reference or not.

- $arr->add_item($item)

  Push an item onto the list of values to be checked.

- @deltas = $arr->deltas(got => $got, convert => \&convert, seen => \%seen)

  Find the differences between the expected bag values and those in the `$got` arrayref.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org
- Gianni Ceccarelli dakkar@thenautilus.net

AUTHORS

- Chad Granum exodist@cpan.org
- Gianni Ceccarelli dakkar@thenautilus.net

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

Copyright 2018 Gianni Ceccarelli dakkar@thenautilus.net

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Compare::Base

Base class for comparison classes.

DESCRIPTION

All comparison classes for Test2::Compare should inherit from this base class.

SYNOPSIS

```
package Test2::Compare::MyCheck;
use strict;
use warnings;

use base 'Test2::Compare::Base';
use Test2::Util::HashBase qw/stuff/;

sub name { 'STUFF' }

sub operator {
    my $self = shift;
    my ($got) = @_;
    return 'eq';
}

sub verify {
    my $self = shift;
    my $params = @_;

    # Always check if $got exists! This method must return false if no
    # value at all was received.
    return 0 unless $params{exists};

    my $got = $params{got};

    # Returns true if both values match. This includes undef, 0, and other
    # false-y values!
    return $got eq $self->stuff;
}
```

METHODS

Some of these must be overridden, others can be.

- $dclass = $check->delta_class

  Returns the delta subclass that should be used. By default *Test2::Compare::Delta* is used.

- @deltas = $check->deltas(id => $id, exists => $bool, got => $got, convert => \&convert, seen => \%seen)

  Should return child deltas.

- @lines = $check->got_lines($got)

  This is your chance to provide line numbers for errors in the `$got` structure.

- $op = $check->operator()

- $op = $check->operator($got)

Returns the operator that was used to compare the check with the received data in `$got`. If there was no value for got then there will be no arguments, undef will only be an argument if undef was seen in `$got`. This is how you can tell the difference between a missing value and an undefined one.

- $bool = $check->verify(id => $id, exists => $bool, got => $got, convert => \&convert, seen => \%seen)

  Return true if there is a shallow match, that is both items are arrayrefs, both items are the same string or same number, etc. This should not recurse, as deep checks are done in `$check->deltas()`.

- $name = $check->name

  Get the name of the check.

- $display = $check->render

  What should be displayed in a table for this check, usually the name or value.

- $delta = $check->run(id => $id, exists => $bool, got => $got, convert => \&convert, seen => \%seen)

  This is where the checking is done, first a shallow check using `$check->verify`, then checking `$check->deltas()`. `\%seen` is used to prevent cycles.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Compare::Bool

Compare two values as booleans

DESCRIPTION

Check if two values have the same boolean result (both true, or both false).

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Compare::Custom

Custom field check for comparisons.

DESCRIPTION

Sometimes you want to do something complicated or unusual when validating a field nested inside a deep data structure. You could pull it out of the structure and test it separately, or you can use this to embed the check. This provides a way for you to write custom checks for fields in deep comparisons.

SYNOPSIS

```
my $cus = Test2::Compare::Custom->new(
    name => 'IsRef',
    operator => 'ref(...)',
    stringify_got => 1,
    code => sub {
        my %args = @_;
        return $args{got} ? 1 : 0;
    },
);

# Pass
```

(continues on next page)

(continued from previous page)

```
is(
    { a => 1, ref => {},   b => 2 },
    { a => 1, ref => $cus, b => 2 },
    "This will pass"
);

# Fail
is(
    {a => 1, ref => 'notref', b => 2},
    {a => 1, ref => $cus,     b => 2},
    "This will fail"
);
```

ARGUMENTS

Your custom sub will be passed 4 arguments in a hash:

```
code => sub {
    my %args = @_;
    # provides got, exists, operator, name
    return ref($args{got}) ? 1 : 0;
},
```

$_ is also localized to `got` to make it easier for those who need to use regexes.

- got
- $_

  The value to be checked.

- exists

  This will be a boolean. This will be true if `got` exists at all. If `exists` is false then it means `got` is not simply undef, but doesn't exist at all (think checking the value of a hash key that does not exist).

- operator

  The operator specified at construction.

- name

  The name provided at construction.

METHODS

- $code = $cus->code

  Returns the coderef provided at construction.

- $name = $cus->name

  Returns the name provided at construction.

- $op = $cus->operator

  Returns the operator provided at construction.

- $stringify = $cus->stringify_got

  Returns the stringify_got flag provided at construction.

- $bool = $cus->verify(got => $got, exists => $bool)

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org
- Daniel Böhmer dboehmer@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Compare::DeepRef

Ref comparison

DESCRIPTION

Used to compare two refs in a deep comparison.

SYNOPSIS

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Compare::Delta

Representation of differences between nested data structures.

DESCRIPTION

This is used by *Test2::Compare*. When data structures are compared a delta will be returned. Deltas are a tree data structure that represent all the differences between two other data structures.

METHODS

CLASS METHODS

- $class->add_column($NAME => sub { ... })
- $class->add_column($NAME, %PARAMS)

  This can be used to add columns to the table that it produced when a comparison fails. The first argument should always be the column name, which must be unique.

  The first form simply takes a coderef that produces the value that should be displayed in the column for any given delta. The arguments passed into the sub are the delta, and the row ID.

```
Test2::Compare::Delta->add_column(
    Foo => sub {
        my ($delta, $id) = @_;
        return $delta->... ? 'foo' : 'bar'
    },
);
```

  The second form allows you some extra options. The `'value'` key is required, and must be a coderef. All other keys are optional.

```
Test2::Compare::Delta->add_column(
    'Foo',    # column name
    value => sub { ... },    # how to get the cell value
    alias       => 'FOO',    # Display name (used in table␣
↪header)
    no_collapse => $bool,    # Show column even if it has no␣
↪values?
);
```

- $bool = $class->remove_column($NAME)

  This will remove the specified column. This will return true if the column existed and was removed. This will return false if the column did not exist. No exceptions are thrown. If a missing column is a problem then you need to check the return yourself.

- $class->set_column_alias($NAME, $ALIAS)

  This can be used to change the table header, overriding the default column names with new ones.

ATTRIBUTES

- $bool = $delta->verified
- $delta->set_verified($bool)

  This will be true if the delta itself matched, if the delta matched then the problem is in the delta's children, not the delta itself.

- $aref = $delta->id
- $delta->set_id([$type, $name])

  ID for the delta, used to produce the path into the data structure. An example is `['HASH' => 'foo']` which means the delta is in the path `...->{'foo'}`. Valid types are `HASH`, `ARRAY`, `SCALAR`, `META`, and `METHOD`.

- $val = $delta->got
- $delta->set_got($val)

  Deltas are produced by comparing a received data structure 'got' against a check data structure 'check'. The 'got' attribute contains the value that was received for comparison.

- $check = $delta->chk
- $check = $delta->check
- $delta->set_chk($check)
- $delta->set_check($check)

  Deltas are produced by comparing a received data structure 'got' against a check data structure 'check'. The 'check' attribute contains the value that was expected in the comparison.

  `check` and `chk` are aliases for the same attribute.

- $aref = $delta->children
- $delta->set_children([$delta1, $delta2, ...])

  A Delta may have child deltas. If it does then this is an arrayref with those children.
- $dne = $delta->dne
- $delta->set_dne($dne)

  Sometimes a comparison results in one side or the other not existing at all, in which case this is set to the name of the attribute that does not exist. This can be set to 'got' or 'check'.
- $e = $delta->exception
- $delta->set_exception($e)

  This will be set to the exception in cases where the comparison failed due to an exception being thrown.

OTHER

- $string = $delta->render_got

  Renders the string that should be used in a table to represent the received value in a comparison.
- $string = $delta->render_check

  Renders the string that should be used in a table to represent the expected value in a comparison.
- $bool = $delta->should_show

  This will return true if the delta should be shown in the table. This is normally true for any unverified delta. This will also be true for deltas that contain extra useful debug information.
- $aref = $delta->filter_visible

  This will produce an arrayref of `[ $path => $delta ]` for all deltas that should be displayed in the table.
- $aref = $delta->table_header

  This returns an array ref of the headers for the table.
- $string = $delta->table_op

  This returns the operator that should be shown in the table.
- $string = $delta->table_check_lines

  This returns the defined lines (extra debug info) that should be displayed.
- $string = $delta->table_got_lines

  This returns the generated lines (extra debug info) that should be displayed.
- $aref = $delta->table_rows

  This returns an arrayref of table rows, each row is itself an arrayref.

- @table_lines = $delta->table

  Returns all the lines of the table that should be displayed.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Compare::Event

Event specific Object subclass.

DESCRIPTION

This module is used to represent an expected event in a deep comparison.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Compare::EventMeta

Meta class for events in deep comparisons

DESCRIPTION

This is used in deep comparisons of event objects. You should probably never use this directly.

DEFINED CHECKS

- file

  File that generated the event.

- line

  Line where the event was generated.

- package

  Package that generated the event.

- subname

  Name of the tool that generated the event.

- debug

  The debug information that will be printed in event of a failure.

- tid

  Thread ID of the thread that generated the event.

- pid

  Process ID of the process that generated the event.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Compare::Float

Compare two values as numbers with tolerance.

DESCRIPTION

This is used to compare two numbers. You can also check that two numbers are not the same.

This is similar to Test2::Compare::Number, with extra checks to work around floating point representation issues.

The optional 'tolerance' parameter controls how close the two numbers must be to be considered equal. Tolerance defaults to 1e-08.

Note: This will fail if the received value is undefined. It must be a number.

Note: This will fail if the comparison generates a non-numeric value warning (which will not be shown). This is because it must get a number. The warning is not shown as it will report to a useless line and filename. However, the test diagnostics show both values.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Andrew Grangaard spazm@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Compare::Hash

Representation of a hash in a deep comparison.

DESCRIPTION

In deep comparisons this class is used to represent a hash.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Compare::Isa

Check if the value is an instance of the class.

DESCRIPTION

This is used to check if the got value is an instance of the expected class.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org
- TOYAMA Nao nanto@moon.email.ne.jp

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Compare::Meta

Check library for meta-checks

DESCRIPTION

Sometimes in a deep comparison you want to run extra checks against an item down the chain. This library allows you to write a check that verifies several attributes of an item.

DEFINED CHECKS

- blessed

  Lets you check that an item is blessed, and that it is blessed into the expected class.

- reftype

  Lets you check the reftype of the item.

- isa

  Lets you check if the item is an instance of the expected class.

- this

  Lets you check the item itself.

- size

  Lets you check the size of the item. For an arrayref this is the number of elements. For a hashref this is the number of keys. For everything else this is undef.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Compare::Negatable

Poor mans 'role' for compare objects that can be negated.

DESCRIPTION

Using this package inside an *Test2::Compare::Base* subclass will overload `!$obj` and import `clone_negate()` and `toggle_negate()`.

WHY?

Until perl 5.18 the 'fallback' parameter to *overload* would not be inherited, so we cannot use inheritance for the behavior we actually want. This module works around the problem by emulating the `use overload` call we want for each consumer class.

ATTRIBUTES

- $bool = $obj->negate
- $obj->set_negate($bool)
- $attr = NEGATE()

  The NEGATE attribute will be added via *Test2::Util::HashBase*.

METHODS

- $clone = $obj->clone_negate()

  Create a shallow copy of the object, and call `toggle_negate` on it.

- $obj->toggle_negate()

  Toggle the negate attribute. If the attribute was on it will now be off, if it was off it will now be on.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Compare::Number

Compare two values as numbers

DESCRIPTION

This is used to compare two numbers. You can also check that two numbers are not the same.

Note: This will fail if the received value is undefined. It must be a number.

Note: This will fail if the comparison generates a non-numeric value warning (which will not be shown). This is because it must get a number. The warning is not shown as it will report to a useless line and filename. However, the test diagnostics show both values.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Compare::Object

Representation of an object during deep comparison.

DESCRIPTION

This class lets you specify an expected object in a deep comparison. You can check the fields/elements of the underlying reference, call methods to verify results, and do meta checks for object type and ref type.

METHODS

- $class = $obj->meta_class

  The meta-class to be used when checking the object type. This is mainly listed because it is useful to override for specialized object subclasses.

  This normally just returns *Test2::Compare::Meta*.

- $class = $obj->object_base

  The base-class to be expected when checking the object type. This is mainly listed because it is useful to override for specialized object subclasses.

  This normally just returns 'UNIVERSAL'.

- $obj->add_prop(...)

  Add a meta-property to check, see *Test2::Compare::Meta*. This method just delegates.

- $obj->add_field(...)

  Add a hash-field to check, see *Test2::Compare::Hash*. This method just delegates.

- $obj->add_item(...)

  Add an array item to check, see *Test2::Compare::Array*. This method just delegates.

- $obj->add_call($method, $check)

- $obj->add_call($method, $check, $name)

- $obj->add_call($method, $check, $name, $context)

  Add a method call check. This will call the specified method on your object and verify the result. `$method` may be a method name, an array ref, or a coderef.

  If it's an arrayref, the first element must be the method name, and the rest are arguments that will be passed to it.

In the case of a coderef it can be helpful to provide an alternate name. When no name is provided the name is either `$method` or the string '\&CODE'.

If `$context` is `'list'`, the method will be invoked in list context, and the result will be an arrayref.

If `$context` is `'hash'`, the method will be invoked in list context, and the result will be a hashref (this will warn if the method returns an odd number of values).

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Compare::OrderedSubset

Internal representation of an ordered subset.

DESCRIPTION

This module is used to ensure an array has all the expected items int he expected order. It ignores any unexpected items mixed into the array. It only cares that all the expected values are present, and in order, everything else is noise.

METHODS

- $ref = $arr->inref()

  If the instance was constructed from an actual array, this will have the reference to that array.

- $arrayref = $arr->items()

- $arr->set_items($arrayref)

  All the expected items, in order.

- $name = $arr->name()

  Always returns the string "<ORDERED SUBSET>".

- $bool = $arr->verify(got => $got, exists => $bool)

  Check if $got is an array reference or not.

- $arr->add_item($item)

  Add an item to the list of values to check.

- @deltas = $arr->deltas(got => $got, convert => \&convert, seen => \%seen)

  Find the differences between the expected array values and those in the $got arrayref.

SOURCE

The source code repository for Test2-Suite can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See http://dev.perl.org/licenses/

Test2::Compare::Pattern

Use a pattern to validate values in a deep comparison.

DESCRIPTION

This allows you to use a regex to validate a value in a deep comparison. Sometimes a value just needs to look right, it may not need to be exact. An example is a memory address that might change from run to run.

SOURCE

The source code repository for Test2-Suite can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Compare::Ref

Ref comparison

DESCRIPTION

Used to compare two refs in a deep comparison.

SYNOPSIS

```
my $ref = {};
my $check = Test2::Compare::Ref->new(input => $ref);

# Passes
is( [$ref], [$check], "The array contains the exact ref we want" );

# Fails, they both may be empty hashes, but we are looking for a␣
↪specific
# reference.
is( [{}], [$check], "This will fail");
```

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Compare::Regex

Regex direct comparison

DESCRIPTION

Used to compare two regexes. This compares the stringified form of each regex.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Compare::Scalar

Representation of a Scalar Ref in deep comparisons

DESCRIPTION

This is used in deep comparisons to represent a scalar reference.

SYNOPSIS

```
my $sr = Test2::Compare::Scalar->new(item => 'foo');

is([\'foo'], $sr, "pass");
is([\'bar'], $sr, "fail, different value");
is(['foo'],  $sr, "fail, not a ref");
```

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Compare::Set

Allows a field to be matched against a set of checks.

DESCRIPTION

This module is used by the `check_set` function in the *Test2::Tools::Compare* plugin.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Compare::String

Compare two values as strings

DESCRIPTION

This is used to compare two items after they are stringified. You can also check that two strings are not equal.

Note: This will fail if the received value is undefined, it must be defined.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Compare::Undef

Check that something is undefined

DESCRIPTION

Make sure something is undefined in a comparison. You can also check that something is defined.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Compare::Wildcard

Placeholder check.

DESCRIPTION

This module is used as a temporary placeholder for values that still need to be converted. This is necessary to carry forward the filename and line number which would be lost in the conversion otherwise.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Env

Documentation for environment variables used or set by Test2.

DESCRIPTION

This is a list of environment variables that are either set by, or read by Test2.

AUTHOR_TESTING

This env var is read by Test2. When set Test2 will run tests that are normally skipped unless a module author is doing extra author-specific testing.

AUTOMATED_TESTING

This env var is read by Test2. When set this indicates the tests are run by an automated system and no human interaction is possible.

See *Test2::Require::AuthorTesting*.

EXTENDED_TESTING

This env var is read by Test2. When set it indicates some extended testing that should normally be skipped will be run.

See *Test2::Require::ExtendedTesting*.

HARNESS_ACTIVE

This env var is read by Test2. It is usually set by `prove` (*Test::Harness*) or `yath` (App::Yath).

NONINTERACTIVE_TESTING

This env var is read by Test2. When set this indicates the testing will not be interactive.

See *Test2::Require::NonInteractiveTesting*.

RELEASE_TESTING

This env var is read by Test2. When set this indicates that release testing is being done, which may run more tests than normal.

See *Test2::Require::ReleaseTesting*.

T2_FORMATTER

This can be used to set the formatter that Test2 will use. If set to a string without a '+' prefix, then 'Test2::Formatter::' will be added to the start of the module name. If '+' is present it will be stripped and no further modification will be made to the module name.

T2_IN_PRELOAD

Test2 sets this when preload mode is active. This is mainly used by App::Yath and similar tools that preload Test2, then fork to run tests.

TABLE_TERM_SIZE

This is used to set a terminal width for things like diagnostic message tables.

TEST2_ACTIVE

Test2 sets this variable when tests are running.

TEST2_ENABLE_PLUGINS

This can be used to force plugins to be loaded when the Test2 API is loaded. It takes a list of one or more plugin names separated by comma. If the module name does not have a '+' in front of it then the `Test2::Plugin::` namespace is assumed and added. If a '+' is present at the start of a module name it will be stripped and no further modification will be made.

Examples:

```
TEST2_ENABLE_PLUGINS=BailOnFail
Test2_ENABLE_PLUGINS=SRand,+My::Plugin::Name
```

TEST_ACTIVE

Set by Test2 when tests are running.

TS_MAX_DELTA

Used to determine how many max lines of output will be provided when is() finds a deep data structure mismatch.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Event

Base class for events

DESCRIPTION

Base class for all event objects that get passed through *Test2*.

SYNOPSIS

```
package Test2::Event::MyEvent;
use strict;
use warnings;

# This will make our class an event subclass (required)
use base 'Test2::Event';

# Add some accessors (optional)
# You are not obligated to use HashBase, you can use any object tool␣
 ↪you
# want, or roll your own accessors.
use Test2::Util::HashBase qw/foo bar baz/;

# Use this if you want the legacy API to be written for you, for␣
 ↪this to
# work you will need to implement a facet_data() method.
use Test2::Util::Facets2Legacy;
```

(continues on next page)

(continued from previous page)

```
# Chance to initialize some defaults
sub init {
    my $self = shift;
    # no other args in @_

    $self->set_foo('xxx') unless defined $self->foo;

    ...
}

# This is the new way for events to convey data to the Test2 system
sub facet_data {
    my $self = shift;

    # Get common facets such as 'about', 'trace' 'amnesty', and
↪'meta'
    my $facet_data = $self->common_facet_data();

    # Are you making an assertion?
    $facet_data->{assert} = {pass => 1, details => 'my assertion'};
    ...

    return $facet_data;
}

1;
```

METHODS

GENERAL

- $trace = $e->trace

  Get a snapshot of the *Test2::EventFacet::Trace* as it was when this event was generated

- $bool_or_undef = $e->related($e2)

  Check if 2 events are related. In this case related means their traces share a signature meaning they were created with the same context (or at the very least by contexts which share an id, which is the same thing unless someone is doing something very bad).

  This can be used to reliably link multiple events created by the same tool. For instance a failing test like `ok(0, "fail"` will generate 2 events, one being a *Test2::Event::Ok*, the other being a *Test2::Event::Diag*, both of these events are related having been created under the same context and by the same initial tool (though multiple tools may have been nested under the initial one).

  This will return `undef` if the relationship cannot be checked, which happens if either event has an incomplete or missing trace. This will return `0` if the traces are complete, but do not match. 1 will be returned if there is a match.

- $e->add_amnesty({tag => $TAG, details => $DETAILS});

  This can be used to add amnesty to this event. Amnesty only affects failing assertions in most cases, but some formatters may display them for passing assertions, or even non-assertions as well.

  Amnesty will prevent a failed assertion from causing the overall test to fail. In other words it marks a failure as expected and allowed.

  Note: This is how 'TODO' is implemented under the hood. TODO is essentially amnesty with the 'TODO' tag. The details are the reason for the TODO.

- $uuid = $e->uuid

  If UUID tagging is enabled (See Test::API) then any event that has made its way through a hub will be tagged with a UUID. A newly created event will not yet be tagged in most cases.

- $class = $e->load_facet($name)

  This method is used to load a facet by name (or key). It will attempt to load the facet class, if it succeeds it will return the class it loaded. If it fails it will return `undef`. This caches the result at the class level so that future calls will be faster.

  The `$name` variable should be the key used to access the facet in a facets hashref. For instance the assertion facet has the key 'assert', the information facet has the 'info' key, and the error facet has the key 'errors'. You may include or omit the 's' at the end of the name, the method is smart enough to try both the 's' and no-'s' forms, it will check what you provided first, and if that is not found it will add or strip the 's and try again.

- @classes = $e->FACET_TYPES()

- @classes = Test2::Event->FACET_TYPES()

  This returns a list of all facets that have been loaded using the `load_facet()` method. This will not return any classes that have not been loaded, or have been loaded directly without a call to `load_facet()`.

  Note: The core facet types are automatically loaded and populated in this list.

NEW API

- $hashref = $e->common_facet_data();

  This can be used by subclasses to generate a starting facet data hashref. This will populate the hashref with the trace, meta, amnesty, and about facets. These facets are nearly always produced the same way for all events.

- $hashref = $e->facet_data()

  If you do not override this then the default implementation will attempt to generate facets from the legacy API. This generation is limited only to what the legacy API can provide. It is recommended that you override this method and write out explicit facet data.

- $hashref = $e->facets()

This takes the hashref from `facet_data()` and blesses each facet into the proper `Test2::EventFacet::*` subclass. If no class can be found for any given facet it will be passed along unchanged.

- @errors = $e->validate_facet_data();
- @errors = $e->validate_facet_data(%params);
- @errors = $e->validate_facet_data(\%facets, %params);
- @errors = Test2::Event->validate_facet_data(%params);
- @errors = Test2::Event->validate_facet_data(\%facets, %params);

  This method will validate facet data and return a list of errors. If no errors are found this will return an empty list.

  This can be called as an object method with no arguments, in which case the `facet_data()` method will be called to get the facet data to be validated.

  When used as an object method the `\%facet_data` argument may be omitted.

  When used as a class method the `\%facet_data` argument is required.

  Remaining arguments will be slurped into a `%params` hash.

  Currently only 1 parameter is defined:

  – require_facet_class => $BOOL

    When set to true (default is false) this will reject any facets where a facet class cannot be found. Normally facets without classes are assumed to be custom and are ignored.

WHAT ARE FACETS?

Facets are how events convey their purpose to the Test2 internals and formatters. An event without facets will have no intentional effect on the overall test state, and will not be displayed at all by most formatters, except perhaps to say that an event of an unknown type was seen.

Facets are produced by the `facet_data()` subroutine, which you should nearly-always override. `facet_data()` is expected to return a hashref where each key is the facet type, and the value is either a hashref with the data for that facet, or an array of hashrefs. Some facets must be defined as single hashrefs, some must be defined as an array of hashrefs, No facets allow both.

`facet_data()` **MUST NOT** bless the data it returns, the main hashref, and nested facet hashrefs **MUST** be bare, though items contained within each facet may be blessed. The data returned by this method **should** also be copies of the internal data in order to prevent accidental state modification.

`facets()` takes the data from `facet_data()` and blesses it into the `Test2::EventFacet::*` packages. This is rarely used however, the EventFacet packages are primarily for convenience and documentation. The EventFacet classes are not used at all internally, instead the raw data is used.

Here is a list of facet types by package. The packages are not used internally, but are where the documentation for each type is kept.

Note: Every single facet type has the `'details'` field. This field is always intended for human consumption, and when provided, should explain the 'why' for the facet. All other fields are facet specific.

- about => {...}

  Test2::EventFacet::About

  This contains information about the event itself such as the event package name. The `details` field for this facet is an overall summary of the event.

- assert => {...}

  Test2::EventFacet::Assert

  This facet is used if an assertion was made. The `details` field of this facet is the description of the assertion.

- control => {...}

  Test2::EventFacet::Control

  This facet is used to tell the *Test2::Hub* about special actions the event causes. Things like halting all testing, terminating the current test, etc. In this facet the `details` field explains why any special action was taken.

  Note: This is how bail-out is implemented.

- meta => {...}

  Test2::EventFacet::Meta

  The meta facet contains all the meta-data attached to the event. In this case the `details` field has no special meaning, but may be present if something sets the 'details' meta-key on the event.

- parent => {...}

  Test2::EventFacet::Parent

  This facet contains nested events and similar details for subtests. In this facet the `details` field will typically be the name of the subtest.

- plan => {...}

  Test2::EventFacet::Plan

  This facet tells the system that a plan has been set. The `details` field of this is usually left empty, but when present explains why the plan is what it is, this is most useful if the plan is to skip-all.

- trace => {...}

  Test2::EventFacet::Trace

  This facet contains information related to when and where the event was generated. This is how the test file and line number of a failure is known. This facet can also help you to tell if tests are related.

  In this facet the `details` field overrides the "failed at test_file.t line 42." message provided on assertion failure.

- amnesty => [{...}, ...]

  Test2::EventFacet::Amnesty

  The amnesty facet is a list instead of a single item, this is important as amnesty can come from multiple places at once.

  For each instance of amnesty the `details` field explains why amnesty was granted.

  Note: Outside of formatters amnesty only acts to forgive a failing assertion.

- errors => [{...}, ...]

  Test2::EventFacet::Error

  The errors facet is a list instead of a single item, any number of errors can be listed. In this facet `details` describes the error, or may contain the raw error message itself (such as an exception). In perl exception may be blessed objects, as such the raw data for this facet may contain nested items which are blessed.

  Not all errors are considered fatal, there is a `fail` field that must be set for an error to cause the test to fail.

  Note: This facet is unique in that the field name is 'errors' while the package is 'Error'. This is because this is the only facet type that is both a list, and has a name where the plural is not the same as the singular. This may cause some confusion, but I feel it will be less confusing than the alternative.

- info => [{...}, ...]

  Test2::EventFacet::Info

  The 'info' facet is a list instead of a single item, any quantity of extra information can be attached to an event. Some information may be critical diagnostics, others may be simply commentary in nature, this is determined by the debug flag.

  For this facet the `details` flag is the info itself. This info may be a string, or it may be a data structure to display. This is one of the few facet types that may contain blessed items.

LEGACY API

- $bool = $e->causes_fail

  Returns true if this event should result in a test failure. In general this should be false.

- $bool = $e->increments_count

  Should be true if this event should result in a test count increment.

- $e->callback($hub)

  If your event needs to have extra effects on the *Test2::Hub* you can override this method.

  This is called **BEFORE** your event is passed to the formatter.

- $num = $e->nested

  If this event is nested inside of other events, this should be the depth of nesting. (This is mainly for subtests)

- $bool = $e->global

  Set this to true if your event is global, that is ALL threads and processes should see it no matter when or where it is generated. This is not a common thing to want, it is used by bail-out and skip_all to end testing.

- $code = $e->terminate

  This is called **AFTER** your event has been passed to the formatter. This should normally return undef, only change this if your event should cause the test to exit immediately.

  If you want this event to cause the test to exit you should return the exit code here. Exit code of 0 means exit success, any other integer means exit with failure.

  This is used by *Test2::Event::Plan* to exit 0 when the plan is 'skip_all'. This is also used by Test2::Event:Bail to force the test to exit with a failure.

  This is called after the event has been sent to the formatter in order to ensure the event is seen and understood.

- $msg = $e->summary

  This is intended to be a human readable summary of the event. This should ideally only be one line long, but you can use multiple lines if necessary. This is intended for human consumption. You do not need to make it easy for machines to understand.

  The default is to simply return the event package name.

- ($count, $directive, $reason) = $e->sets_plan()

  Check if this event sets the testing plan. It will return an empty list if it does not. If it does set the plan it will return a list of 1 to 3 items in order: Expected Test Count, Test Directive, Reason for directive.

- $bool = $e->diagnostics

  True if the event contains diagnostics info. This is useful because a non-verbose harness may choose to hide events that are not in this category. Some formatters may choose to send these to STDERR instead of STDOUT to ensure they are seen.

- $bool = $e->no_display

  False by default. This will return true on events that should not be displayed by formatters.

- $id = $e->in_subtest

  If the event is inside a subtest this should have the subtest ID.

- $id = $e->subtest_id

  If the event is a final subtest event, this should contain the subtest ID.

THIRD PARTY META-DATA

This object consumes *Test2::Util::ExternalMeta* which provides a consistent way for you to attach meta-data to instances of this class. This is useful for tools, plugins, and other extensions.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Modules

- `Test2::Event::Bail` - Bailout!
- `Test2::Event::Diag` - Diag event type
- `Test2::Event::Encoding` - Set the encoding for the output stream
- `Test2::Event::Exception` - Exception event
- `Test2::Event::Fail` - Event for a simple failed assertion
- `Test2::Event::Generic` - Generic event type.
- `Test2::Event::Note` - Note event type
- `Test2::Event::Ok` - Ok event type
- `Test2::Event::Pass` - Event for a simple passing assertion
- `Test2::Event::Plan` - The event of a plan
- `Test2::Event::Skip` - Skip event type
- `Test2::Event::Subtest` - Event for subtest types
- `Test2::Event::TAP`
- `Test2::Event::V2` - Second generation event.
- `Test2::Event::Waiting` - Tell all procs/threads it is time to be done

Test2::Event::Bail

Bailout!

DESCRIPTION

The bailout event is generated when things go horribly wrong and you need to halt all testing in the current file.

SYNOPSIS

```
use Test2::API qw/context/;
use Test2::Event::Bail;

my $ctx = context();
my $event = $ctx->bail('Stuff is broken');
```

METHODS

Inherits from *Test2::Event*. Also defines:

- $reason = $e->reason

  The reason for the bailout.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::Event::Diag

Diag event type

DESCRIPTION

Diagnostics messages, typically rendered to STDERR.

SYNOPSIS

```
use Test2::API qw/context/;
use Test2::Event::Diag;

my $ctx = context();
my $event = $ctx->diag($message);
```

ACCESSORS

- $diag->message

  The message for the diag.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::Event::Encoding

Set the encoding for the output stream

DESCRIPTION

The encoding event is generated when a test file wants to specify the encoding to be used when formatting its output. This event is intended to be produced by formatter classes and used for interpreting test names, message contents, etc.

SYNOPSIS

```
use Test2::API qw/context/;
use Test2::Event::Encoding;

my $ctx = context();
my $event = $ctx->send_event('Encoding', encoding => 'UTF-8');
```

METHODS

Inherits from *Test2::Event*. Also defines:

- $encoding = $e->encoding

  The encoding being specified.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::Event::Exception

Exception event

DESCRIPTION

An exception event will display to STDERR, and will prevent the overall test file from passing.

SYNOPSIS

```
use Test2::API qw/context/;
use Test2::Event::Exception;

my $ctx = context();
my $event = $ctx->send_event('Exception', error => 'Stuff is broken
 ↪');
```

METHODS

Inherits from *Test2::Event*. Also defines:

- $reason = $e->error

  The reason for the exception.

CAVEATS

Be aware that all exceptions are stringified during construction.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::Event::Fail

Event for a simple failed assertion

DESCRIPTION

This is an optimal representation of a failed assertion.

SYNOPSIS

```
use Test2::API qw/context/;

sub fail {
    my ($name) = @_;
    my $ctx = context();
    $ctx->fail($name);
    $ctx->release;
}
```

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::Event::Generic

Generic event type.

DESCRIPTION

This is a generic event that lets you customize all fields in the event API. This is useful if you have need for a custom event that does not make sense as a published reusable event subclass.

SYNOPSIS

```
use Test2::API qw/context/;

sub send_custom_fail {
    my $ctx = shift;

    $ctx->send_event('Generic', causes_fail => 1, summary => 'The
 ↪sky is falling');

    $ctx->release;
}

send_custom_fail();
```

METHODS

- $e->facet_data($data)
- $data = $e->facet_data

  Get or set the facet data (see *Test2::Event*). If no facet_data is set then `Test2::Event->facet_data` will be called to produce facets from the other data.

- $e->callback($hub)

  Call the custom callback if one is set, otherwise this does nothing.

- $e->set_callback(sub { ... })

  Set the custom callback. The custom callback must be a coderef. The first argument to your callback will be the event itself, the second will be the *Test2::Hub* that is using the callback.

- $bool = $e->causes_fail
- $e->set_causes_fail($bool)

  Get/Set the `causes_fail` attribute. This defaults to `0`.

- $bool = $e->diagnostics
- $e->set_diagnostics($bool)

  Get/Set the `diagnostics` attribute. This defaults to `0`.

- $bool_or_undef = $e->global
- @bool_or_empty = $e->global
- $e->set_global($bool_or_undef)

  Get/Set the `diagnostics` attribute. This defaults to an empty list which is undef in scalar context.

- $bool = $e->increments_count
- $e->set_increments_count($bool)

  Get/Set the `increments_count` attribute. This defaults to `0`.

- $bool = $e->no_display
- $e->set_no_display($bool)

  Get/Set the `no_display` attribute. This defaults to `0`.

- @plan = $e->sets_plan

  Get the plan if this event sets one. The plan is a list of up to 3 items: `($count, $directive, $reason)`. `$count` must be defined, the others may be undef, or may not exist at all.

- $e->set_sets_plan(\@plan)

  Set the plan. You must pass in an arrayref with up to 3 elements.

- $summary = $e->summary
- $e->set_summary($summary_or_undef)

  Get/Set the summary. This will default to the event package `'Test2::Event::Generic'`. You can set it to any value. Setting this to `undef` will reset it to the default.

- $int_or_undef = $e->terminate
- @int_or_empty = $e->terminate
- $e->set_terminate($int_or_undef)

  This will get/set the `terminate` attribute. This defaults to undef in scalar context, or an empty list in list context. Setting this to undef will clear it completely. This must be set to a positive integer (0 or larger).

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::Event::Note

Note event type

DESCRIPTION

Notes, typically rendered to STDOUT.

SYNOPSIS

```
use Test2::API qw/context/;
use Test2::Event::Note;

my $ctx = context();
my $event = $ctx->Note($message);
```

ACCESSORS

- $note->message

  The message for the note.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::Event::Ok

Ok event type

DESCRIPTION

Ok events are generated whenever you run a test that produces a result. Examples are `ok()`, and `is()`.

SYNOPSIS

```
use Test2::API qw/context/;
use Test2::Event::Ok;

my $ctx = context();
my $event = $ctx->ok($bool, $name, \@diag);
```

or:

```
my $ctx   = context();
my $event = $ctx->send_event(
    'Ok',
    pass => $bool,
    name => $name,
);
```

ACCESSORS

- $rb = $e->pass

  The original true/false value of whatever was passed into the event (but reduced down to 1 or 0).

- $name = $e->name

  Name of the test.

- $b = $e->effective_pass

  This is the true/false value of the test after TODO and similar modifiers are taken into account.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::Event::Pass

Event for a simple passing assertion

DESCRIPTION

This is an optimal representation of a passing assertion.

SYNOPSIS

```
use Test2::API qw/context/;

sub pass {
    my ($name) = @_;
    my $ctx = context();
    $ctx->pass($name);
    $ctx->release;
}
```

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::Event::Plan

The event of a plan

DESCRIPTION

Plan events are fired off whenever a plan is declared, done testing is called, or a subtext completes.

SYNOPSIS

```
use Test2::API qw/context/;
use Test2::Event::Plan;

my $ctx = context();

# Plan for 10 tests to run
my $event = $ctx->plan(10);

# Plan to skip all tests (will exit 0)
$ctx->plan(0, skip_all => "These tests need to be skipped");
```

ACCESSORS

- $num = $plan->max

  Get the number of expected tests

- $dir = $plan->directive

  Get the directive (such as TODO, skip_all, or no_plan).

- $reason = $plan->reason

  Get the reason for the directive.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::Event::Skip

Skip event type

DESCRIPTION

Skip events bump test counts just like *Test2::Event::Ok* events, but they can never fail.

SYNOPSIS

```
use Test2::API qw/context/;
use Test2::Event::Skip;

my $ctx = context();
my $event = $ctx->skip($name, $reason);
```

or:

```
my $ctx   = context();
my $event = $ctx->send_event(
    'Skip',
    name   => $name,
    reason => $reason,
);
```

ACCESSORS

- $reason = $e->reason

  The original true/false value of whatever was passed into the event (but reduced down to 1 or 0).

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::Event::Subtest

Event for subtest types

DESCRIPTION

This class represents a subtest. This class is a subclass of *Test2::Event::Ok*.

ACCESSORS

This class inherits from *Test2::Event::Ok*.

- $arrayref = $e->subevents

  Returns the arrayref containing all the events from the subtest

- $bool = $e->buffered

  True if the subtest is buffered, that is all subevents render at once. If this is false it means all subevents render as they are produced.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::Event::TAP

Namespace containing the following modules:

- `Test2::Event::TAP::Version` - Event for TAP version.

Test2::Event::TAP::Version

Event for TAP version.

DESCRIPTION

This event is used if a TAP formatter wishes to set a version.

SYNOPSIS

```
use Test2::API qw/context/;
use Test2::Event::Encoding;

my $ctx = context();
my $event = $ctx->send_event('TAP::Version', version => 42);
```

METHODS

Inherits from *Test2::Event*. Also defines:

- $version = $e->version

  The TAP version being parsed.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::Event::V2

Second generation event.

DESCRIPTION

This is the event type that should be used instead of *Test2::Event* or its legacy subclasses.

SYNOPSIS

USING A CONTEXT

```
use Test2::API qw/context/;

sub my_tool {
    my $ctx = context();

    my $event = $ctx->send_ev2(info => [{tag => 'NOTE', details =>
 ↪"This is a note"}]);

    $ctx->release;

    return $event;
}
```

USING THE CONSTRUCTOR

```
use Test2::Event::V2;

my $e = Test2::Event::V2->new(
    trace => {frame => [$PKG, $FILE, $LINE, $SUBNAME]},
    info  => [{tag => 'NOTE', details => "This is a note"}],
);
```

METHODS

This class inherits from *Test2::Event*.

- $fd = $e->facet_data()

  This will return a hashref of facet data. Each facet hash will be a shallow copy of the original.

- $about = $e->about()

  This will return the 'about' facet hashref.

  NOTE: This will return the internal hashref, not a copy.

- $trace = $e->trace()

  This will return the 'trace' facet, normally blessed (but this is not enforced when the trace is set using `set_trace()`.

  NOTE: This will return the internal trace, not a copy.

MUTATION

- $e->add_amnesty({...})

  Inherited from *Test2::Event*. This can be used to add 'amnesty' facets to an existing event. Each new item is added to the **END** of the list.

  NOTE: Items **ARE** blessed when added.

- $e->add_hub({...})

  Inherited from *Test2::Event*. This is used by hubs to stamp events as they pass through. New items are added to the **START** of the list.

  NOTE: Items **ARE NOT** blessed when added.

- $e->set_uuid($UUID)

  Inherited from *Test2::Event*, overridden to also vivify/mutate the 'about' facet.

- $e->set_trace($trace)

  Inherited from *Test2::Event* which allows you to change the trace.

  Note: This method does not bless/clone the trace for you. Many things will expect the trace to be blessed, so you should probably do that.

LEGACY SUPPORT METHODS

These are all imported from *Test2::Util::Facets2Legacy*, see that module or *Test2::Event* for documentation on what they do.

- causes_fail
- diagnostics
- global
- increments_count
- no_display
- sets_plan
- subtest_id
- summary
- terminate

THIRD PARTY META-DATA

This object consumes *Test2::Util::ExternalMeta* which provides a consistent way for you to attach meta-data to instances of this class. This is useful for tools, plugins, and other extensions.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::Event::Waiting

Tell all procs/threads it is time to be done

DESCRIPTION

This event has no data of its own. This event is sent out by the IPC system when the main process/thread is ready to end.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::EventFacet

Base class for all event facets.

DESCRIPTION

Base class for all event facets.

METHODS

- $key = $facet_class->facet_key()

  This will return the key for the facet in the facet data hash.

- $bool = $facet_class->is_list()

  This will return true if the facet should be in a list instead of a single item.

- $clone = $facet->clone()

- $clone = $facet->clone(%replace)

  This will make a shallow clone of the facet. You may specify fields to override as arguments.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Modules

- `Test2::EventFacet::About` - Facet with event details.
- `Test2::EventFacet::Amnesty` - Facet for assertion amnesty.
- `Test2::EventFacet::Assert` - Facet representing an assertion.
- `Test2::EventFacet::Control` - Facet for hub actions and behaviors.
- `Test2::EventFacet::Error` - Facet for errors that need to be shown.
- `Test2::EventFacet::Hub` - Facet for the hubs an event passes through.
- `Test2::EventFacet::Info` - Facet for information a developer might care about.
- `Test2::EventFacet::Meta` - Facet for meta-data
- `Test2::EventFacet::Parent` - Facet for events contains other events
- `Test2::EventFacet::Plan` - Facet for setting the plan
- `Test2::EventFacet::Render` - Facet that dictates how to render an event.
- `Test2::EventFacet::Trace` - Debug information for events

Test2::EventFacet::About

Facet with event details.

DESCRIPTION

This facet has information about the event, such as event package.

FIELDS

- $string = $about->{details}
- $string = $about->details()

  Summary about the event.
- $package = $about->{package}
- $package = $about->package()

  Event package name.
- $bool = $about->{no_display}

- $bool = $about->no_display()

  True if the event should be skipped by formatters.

- $uuid = $about->{uuid}
- $uuid = $about->uuid()

  Will be set to a uuid if uuid tagging was enabled.

- $uuid = $about->{eid}
- $uuid = $about->eid()

  A unique (for the test job) identifier for the event.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::EventFacet::Amnesty

Facet for assertion amnesty.

DESCRIPTION

This package represents what is expected in units of amnesty.

NOTES

This facet appears in a list instead of being a single item.

FIELDS

- $string = $amnesty->{details}
- $string = $amnesty->details()

  Human readable explanation of why amnesty was granted.

  Example: *Not implemented yet, will fix*

- $short_string = $amnesty->{tag}
- $short_string = $amnesty->tag()

  Short string (usually 10 characters or less, not enforced, but may be truncated by renderers) categorizing the amnesty.

- $bool = $amnesty->{inherited}
- $bool = $amnesty->inherited()

  This will be true if the amnesty was granted to a parent event and inherited by this event, which is a child, such as an assertion within a subtest that is marked todo.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::EventFacet::Assert

Facet representing an assertion.

DESCRIPTION

The assertion facet is provided by any event representing an assertion that was made.

FIELDS

- $string = $assert->{details}
- $string = $assert->details()

  Human readable description of the assertion.

- $bool = $assert->{pass}
- $bool = $assert->pass()

  True if the assertion passed.

- $bool = $assert->{no_debug}
- $bool = $assert->no_debug()

  Set this to true if you have provided custom diagnostics and do not want the defaults to be displayed.

- $int = $assert->{number}
- $int = $assert->number()

  (Optional) assertion number. This may be omitted or ignored. This is usually only useful when parsing/processing TAP.

  Note: This is not set by the Test2 system, assertion number is not known until AFTER the assertion has been processed. This attribute is part of the spec only for harnesses.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::EventFacet::Control

Facet for hub actions and behaviors.

DESCRIPTION

This facet is used when the event needs to give instructions to the Test2 internals.

FIELDS

- $string = $control->{details}
- $string = $control->details()

  Human readable explanation for the special behavior.
- $bool = $control->{global}
- $bool = $control->global()

  True if the event is global in nature and should be seen by all hubs.
- $exit = $control->{terminate}
- $exit = $control->terminate()

  Defined if the test should immediately exit, the value is the exit code and may be `0`.
- $bool = $control->{halt}
- $bool = $control->halt()

  True if all testing should be halted immediately.
- $bool = $control->{has_callback}
- $bool = $control->has_callback()

  True if the `callback($hub)` method on the event should be called.
- $encoding = $control->{encoding}
- $encoding = $control->encoding()

  This can be used to change the encoding from this event onward.
- $phase = $control->{phase}
- $phase = $control->phase()

  Used to signal that a phase change has occurred. Currently only the perl END phase is signaled.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::EventFacet::Error

Facet for errors that need to be shown.

DESCRIPTION

This facet is used when an event needs to convey errors.

NOTES

This facet has the hash key `'errors'`, and is a list of facets instead of a single item.

FIELDS

- $string = $error->{details}
- $string = $error->details()

  Explanation of the error, or the error itself (such as an exception). In perl exceptions may be blessed objects, so this field may contain a blessed object.

- $short_string = $error->{tag}
- $short_string = $error->tag()

  Short tag to categorize the error. This is usually 10 characters or less, formatters may truncate longer tags.

- $bool = $error->{fail}
- $bool = $error->fail()

  Not all errors are fatal, some are displayed having already been handled. Set this to true if you want the error to cause the test to fail. Without this the error is simply a diagnostics message that has no effect on the overall pass/fail result.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::EventFacet::Hub

Facet for the hubs an event passes through.

DESCRIPTION

These are a record of the hubs an event passes through. Most recent hub is the first one in the list.

FACET FIELDS

- $string = $trace->{details}
- $string = $trace->details()

  The hub class or subclass
- $int = $trace->{pid}
- $int = $trace->pid()

  PID of the hub this event was sent to.
- $int = $trace->{tid}
- $int = $trace->tid()

  The thread ID of the hub the event was sent to.
- $hid = $trace->{hid}
- $hid = $trace->hid()

  The ID of the hub that the event was send to.

- $huuid = $trace->{huuid}
- $huuid = $trace->huuid()

  The UUID of the hub that the event was sent to.
- $int = $trace->{nested}
- $int = $trace->nested()

  How deeply nested the hub was.
- $bool = $trace->{buffered}
- $bool = $trace->buffered()

  True if the event was buffered and not sent to the formatter independent of a parent (This should never be set when nested is `0` or `undef`).

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::EventFacet::Info

Facet for information a developer might care about.

DESCRIPTION

This facet represents messages intended for humans that will help them either understand a result, or diagnose a failure.

NOTES

This facet appears in a list instead of being a single item.

FIELDS

- $string_or_structure = $info->{details}
- $string_or_structure = $info->details()

  Human readable string or data structure, this is the information to display. Formatters are free to render the structures however they please. This may contain a blessed object.

  If the `table` attribute (see below) is set then a renderer may choose to display the table instead of the details.

- $structure = $info->{table}
- $structure = $info->table()

  If the data the `info` facet needs to convey can be represented as a table then the data may be placed in this attribute in a more raw form for better display. The data must also be represented in the `details` attribute for renderers which do not support rendering tables directly.

  The table structure:

```
my %table = {
    header => [ 'column 1 header', 'column 2 header', ... ], #␣
↪Optional

    rows => [
        ['row 1 column 1', 'row 1, column 2', ... ],
        ['row 2 column 1', 'row 2, column 2', ... ],
        ...
    ],

    # Allow the renderer to hide empty columns when true,␣
↪Optional
    collapse => $BOOL,

    # List by name or number columns that should never be␣
↪collapsed
    no_collapse => \@LIST,
}
```

- $short_string = $info->{tag}
- $short_string = $info->tag()

  Short tag to categorize the info. This is usually 10 characters or less, formatters may truncate longer tags.

- $bool = $info->{debug}
- $bool = $info->debug()

Set this to true if the message is critical, or explains a failure. This is info that should be displayed by formatters even in less-verbose modes.

When false the information is not considered critical and may not be rendered in less-verbose modes.

- $bool = $info->{important}
- $bool = $info->important

  This should be set for non debug messages that are still important enough to show when a formatter is in quiet mode. A formatter should send these to STDOUT not STDERR, but should show them even in non-verbose mode.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Modules

- `Test2::EventFacet::Info::Table` - Intermediary representation of a table.

Test2::EventFacet::Info::Table

Intermediary representation of a table.

DESCRIPTION

Intermediary representation of a table for use in specialized *Test2::API::Context* methods which generate *Test2::EventFacet::Info* facets.

SYNOPSIS

```
use Test2::EventFacet::Info::Table;
use Test2::API qw/context/;

sub my_tool {
    my $ctx = context();

    ...

    $ctx->fail(
        $name,
        "failure diag message",
        Test2::EventFacet::Info::Table->new(
            # Required
            rows => [['a', 'b'], ['c', 'd'], ...],

            # Strongly Recommended
            as_string => "... string to print when table cannot be
↪rendered ...",

            # Optional
            header => ['col1', 'col2'],
            collapse => $bool,
            no_collapse => ['col1', ...],
        ),
    );

    ...

    $ctx->release;
}

my_tool();
```

ATTRIBUTES

- $header_aref = $t->header()
- $rows_aref = $t->rows()
- $bool = $t->collapse()
- $aref = $t->no_collapse()

  The above are all directly tied to the table hashref structure described in *Test2::EventFacet::Info*.

- $str = $t->as_string()

  This returns the string form of the table if it was set, otherwise it returns the string `"<TABLE NOT DISPLAYED>"`.

- $href = $t->as_hash()

This returns the data structure used for tables by *Test2::EventFacet::Info*.

- This returns the arguments that should be used to construct the proper *Test2::EventFacet::Info* structure.

```
return (table => $t->as_hash(), details => $t->as_string());
```

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::EventFacet::Meta

Facet for meta-data

DESCRIPTION

This facet can contain any random meta-data that has been attached to the event.

METHODS AND FIELDS

Any/all fields and accessors are autovivified into existence. There is no way to know what metadata may be added, so any is allowed.

- $anything = $meta->{anything}
- $anything = $meta->anything()

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::EventFacet::Parent

Facet for events contains other events

DESCRIPTION

This facet is used when an event contains other events, such as a subtest.

FIELDS

- $string = $parent->{details}
- $string = $parent->details()

  Human readable description of the event.
- $hid = $parent->{hid}
- $hid = $parent->hid()

  Hub ID of the hub that is represented in the parent-child relationship.
- $arrayref = $parent->{children}
- $arrayref = $parent->children()

  Arrayref containing the facet-data hashes of events nested under this one.

  To get the actual events you need to get them from the parent event directly
- $bool = $parent->{buffered}
- $bool = $parent->buffered()

  True if the subtest is buffered (meaning the formatter has probably not seen them yet).

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::EventFacet::Plan

Facet for setting the plan

DESCRIPTION

Events use this facet when they need to set the plan.

FIELDS

- $string = $plan->{details}
- $string = $plan->details()

  Human readable explanation for the plan being set. This is normally not rendered by most formatters except when the `skip` field is also set.

- $positive_int = $plan->{count}
- $positive_int = $plan->count()

  Set the number of expected assertions. This should usually be set to `0` when `skip` or none are also set.

- $bool = $plan->{skip}
- $bool = $plan->skip()

  When true the entire test should be skipped. This is usually paired with an explanation in the `details` field, and a `control` facet that has `terminate` set to `0`.

- $bool = $plan->{none}

- $bool = $plan->none()

  This is mainly used by legacy *Test::Builder* tests which set the plan to `no plan`, a construct that predates the much better `done_testing()`.

  If you are using this in non-legacy code you may need to reconsider the course of your life, maybe a hermitage would suite you?

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::EventFacet::Render

Facet that dictates how to render an event.

DESCRIPTION

This facet is used to dictate how the event should be rendered by the standard test2 rendering tools. If this facet is present then ONLY what is specified by it will be rendered. It is assumed that anything important or note-worthy will be present here, no other facets will be considered for rendering/display.

This facet is a list type, you can add as many items as needed.

FIELDS

- $string = $render->[#]->{details}
- $string = $render->[#]->details()

  Human readable text for display.

- $string = $render->[#]->{tag}

- $string = $render->[#]->tag()

  Tag that should prefix/identify the main text.

- $string = $render->[#]->{facet}
- $string = $render->[#]->facet()

  Optional, if the display text was generated from another facet this should state what facet it was.

- $mode = $render->[#]->{mode}
- $mode = $render->[#]->mode()
  - calculated

    Calculated means the facet was generated from another facet. Calculated facets may be cleared and regenerated whenever the event state changes.

  - replace

    Replace means the facet is intended to replace the normal rendering of the event.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::EventFacet::Trace

Debug information for events

DESCRIPTION

The *Test2::API::Context* object, as well as all *Test2::Event* types need to have access to information about where they were created. This object represents that information.

SYNOPSIS

```
use Test2::EventFacet::Trace;

my $trace = Test2::EventFacet::Trace->new(
    frame => [$package, $file, $line, $subname],
);
```

FACET FIELDS

- $string = $trace->{details}
- $string = $trace->details()

  Used as a custom trace message that will be used INSTEAD of `at <FILE> line <LINE>` when calling `$trace->debug`.
- $frame = $trace->{frame}
- $frame = $trace->frame()

  Get the call frame arrayref.

  ```
  [$package, $file, $line, $subname]
  ```
- $int = $trace->{pid}
- $int = $trace->pid()

  The process ID in which the event was generated.
- $int = $trace->{tid}
- $int = $trace->tid()

  The thread ID in which the event was generated.
- $id = $trace->{cid}
- $id = $trace->cid()

  The ID of the context that was used to create the event.
- $uuid = $trace->{uuid}
- $uuid = $trace->uuid()

  The UUID of the context that was used to create the event. (If uuid tagging was enabled)
- ($pkg, $file, $line, $subname) = $trace->call

  Get the basic call info as a list.
- @caller = $trace->full_call

  Get the full caller(N) results.
- $warning_bits = $trace->warning_bits

  Get index 9 from the full caller info. This is the warnings_bits field.

The value of this is not portable across perl versions or even processes. However it can be used in the process that generated it to reproduce the warnings settings in a new scope.

```
eval <<EOT;
BEGIN { ${^WARNING_BITS} = $trace->warning_bits };
... context's warning settings apply here ...
EOT
```

DISCOURAGED HUB RELATED FIELDS

These fields were not always set properly by tools. These are **MOSTLY** deprecated by the *Test2::EventFacet::Hub* facets. These fields are not required, and may only reflect the hub that was current when the event was created, which is not necessarily the same as the hub the event was sent through.

Some tools did do a good job setting these to the correct hub, but you cannot always rely on that. Use the 'hubs' facet list instead.

- $hid = $trace->{hid}
- $hid = $trace->hid()

  The ID of the hub that was current when the event was created.

- $huuid = $trace->{huuid}
- $huuid = $trace->huuid()

  The UUID of the hub that was current when the event was created. (If uuid tagging was enabled).

- $int = $trace->{nested}
- $int = $trace->nested()

  How deeply nested the event is.

- $bool = $trace->{buffered}
- $bool = $trace->buffered()

  True if the event was buffered and not sent to the formatter independent of a parent (This should never be set when nested is `0` or `undef`).

METHODS

Note: All facet frames are also methods.

- $trace->set_detail($msg)
- $msg = $trace->detail

  Used to get/set a custom trace message that will be used INSTEAD of `at <FILE> line <LINE>` when calling `$trace->debug`.

  `detail()` is an alias to the `details` facet field for backwards compatibility.

- $str = $trace->debug

  Typically returns the string `at <FILE> line <LINE>`. If `detail` is set then its value will be returned instead.

- $trace->alert($MESSAGE)

  This issues a warning at the frame (filename and line number where errors should be reported).

- $trace->throw($MESSAGE)

  This throws an exception at the frame (filename and line number where errors should be reported).

- ($package, $file, $line, $subname) = $trace->call()

  Get the caller details for the debug-info. This is where errors should be reported.

- $pkg = $trace->package

  Get the debug-info package.

- $file = $trace->file

  Get the debug-info filename.

- $line = $trace->line

  Get the debug-info line number.

- $subname = $trace->subname

  Get the debug-info subroutine name.

- $sig = trace->signature

  Get a signature string that identifies this trace. This is used to check if multiple events are related. The signature includes pid, tid, file, line number, and the cid.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::Formatter

Namespace for formatters.

DESCRIPTION

This is the namespace for formatters. This is an empty package.

CREATING FORMATTERS

A formatter is any package or object with a `write($event, $num)` method.

```
package Test2::Formatter::Foo;
use strict;
use warnings;

sub write {
    my $self_or_class = shift;
    my ($event, $assert_num) = @_;
    ...
}

sub hide_buffered { 1 }

sub terminate { }

sub finalize { }

sub supports_tables { return $BOOL }

sub new_root {
    my $class = shift;
    ...
    $class->new(@_);
}

1;
```

The `write` method is a method, so it either gets a class or instance. The two arguments are the `$event` object it should record, and the `$assert_num` which is the number of the current assertion (ok), or the last assertion if this event is not itself an assertion. The assertion number may be any integer 0 or greater, and may be undefined in some cases.

The `hide_buffered()` method must return a boolean. This is used to tell buffered subtests whether or not to send it events as they are being buffered. See *"run_subtest(...)" in Test2::API* for more information.

The `terminate` and `finalize` methods are optional methods called that you can implement if the format you're generating needs to handle these cases, for example if you are generating XML and need close open tags.

The `terminate` method is called when an event's `terminate` method returns true, for example when a *Test2::Event::Plan* has a `'skip_all'` plan, or when a *Test2::Event::Bail* event is sent. The `terminate` method is passed a single argument, the *Test2::Event* object which triggered the terminate.

The `finalize` method is always the last thing called on the formatter, *except when* `terminate` *is called for a Bail event*. It is passed the following arguments:

The `supports_tables` method should be true if the formatter supports directly rendering table data from the `info` facets. This is a newer feature and many older formatters may not support it. When not supported the formatter falls back to rendering `detail` instead of the `table` data.

The `new_root` method is used when constructing a root formatter. The default is to just delegate to the regular `new()` method, most formatters can ignore this.

- The number of tests that were planned
- The number of tests actually seen
- The number of tests which failed
- A boolean indicating whether or not the test suite passed
- A boolean indicating whether or not this call is for a subtest

The `new_root` method is called when `Test2::API::Stack` Initializes the root hub for the first time. Most formatters will simply have this call `$class->new`, which is the default behavior. Some formatters however may want to take extra action during construction of the root formatter, this is where they can do that.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Modules

- `Test2::Formatter::TAP` - Standard TAP formatter

Test2::Formatter::TAP

Standard TAP formatter

DESCRIPTION

This is what takes events and turns them into TAP.

SYNOPSIS

```
use Test2::Formatter::TAP;
my $tap = Test2::Formatter::TAP->new();

# Switch to utf8
$tap->encoding('utf8');

$tap->write($event, $number); # Output an event
```

METHODS

- $bool = $tap->no_numbers
- $tap->set_no_numbers($bool)

  Use to turn numbers on and off.
- $arrayref = $tap->handles
- $tap->set_handles(\@handles);

  Can be used to get/set the filehandles. Indexes are identified by the `OUT_STD` and `OUT_ERR` constants.
- $encoding = $tap->encoding
- $tap->encoding($encoding)

  Get or set the encoding. By default no encoding is set, the original settings of STDOUT and STDERR are used.

  This directly modifies the stored filehandles, it does not create new ones.
- $tap->write($e, $num)

  Write an event to the console.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org
- Kent Fredric kentnl@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::Handle

Base class for Test2 handles used in V# bundles.

DESCRIPTION

This is what you interact with when you use the `T2()` function in a test that uses *Test2::V1*.

SYNOPSIS

RECOMMENDED

```
use Test2::V1;

my $handle = T2();

$handle->ok(1, "Passing Test");
```

WITHOUT SUGAR

```
use Test2::Handle();

my $handle = Test2::Handle->new(base => 'Test2::V1::Base');

$handle->ok(1, "Passing test");
```

METHODS

Most methods are delegated to the base class provided at construction. There are however a few methods that are defined by this package itself.

- $base = $class_or_inst->DEFAULT_HANDLE_BASE

  Get the default handle base. This throws an exception on the base handle class, you should override it in a subclass.

- $base = $inst->HANDLE_BASE

  In this base class this method always throws an exception. In a subclass it should return the default base class to use for that subclass.

- $namespace = $inst->HANDLE_NAMESPACE

  Get the namespace used to store function we wrap as methods.

- @sub_names = $inst->HANDLE_SUBS

  Get a list of all subs available in the handle namespace.

- $inst->HANDLE_INCLUDE($package, @subs)

  Import the specified subs from the specified package into our internal namespace.

- $inst = $class->import()

  Used to create a `T2()` sub in your namsepace at import.

- $inst->init()

  Internally used to intialize and validate the handle object.

- AUTOLOAD

  Internally used to wrap functions as methods.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Hub

The conduit through which all events flow.

SYNOPSIS

```
use Test2::Hub;

my $hub = Test2::Hub->new();
$hub->send(...);
```

DESCRIPTION

The hub is the place where all events get processed and handed off to the formatter. The hub also tracks test state, and provides several hooks into the event pipeline.

COMMON TASKS

SENDING EVENTS

```
$hub->send($event)
```

The `send()` method is used to issue an event to the hub. This method will handle thread/fork sync, filters, listeners, TAP output, etc.

ALTERING OR REMOVING EVENTS

You can use either `filter()` or `pre_filter()`, depending on your needs. Both have identical syntax, so only `filter()` is shown here.

```
$hub->filter(sub {
    my ($hub, $event) = @_;

    my $action = get_action($event);

    # No action should be taken
    return $event if $action eq 'none';

    # You want your filter to remove the event
    return undef if $action eq 'delete';

    if ($action eq 'do_it') {
        my $new_event = copy_event($event);
        ... Change your copy of the event ...
        return $new_event;
    }

    die "Should not happen";
});
```

By default, filters are not inherited by child hubs. That means if you start a subtest, the subtest will not inherit the filter. You can change this behavior with the `inherit` parameter:

```
$hub->filter(sub { ... }, inherit => 1);
```

LISTENING FOR EVENTS

```
$hub->listen(sub {
    my ($hub, $event, $number) = @_;

    ... do whatever you want with the event ...

    # return is ignored
});
```

By default listeners are not inherited by child hubs. That means if you start a subtest, the subtest will not inherit the listener. You can change this behavior with the `inherit` parameter:

```
$hub->listen(sub { ... }, inherit => 1);
```

POST-TEST BEHAVIORS

```
$hub->follow_up(sub {
    my ($trace, $hub) = @_;

    ... do whatever you need to ...

    # Return is ignored
});
```

follow_up subs are called only once, either when done_testing is called, or in an END block.

SETTING THE FORMATTER

By default an instance of *Test2::Formatter::TAP* is created and used.

```
my $old = $hub->format(My::Formatter->new);
```

Setting the formatter will REPLACE any existing formatter. You may set the formatter to undef to prevent output. The old formatter will be returned if one was already set. Only one formatter is allowed at a time.

METHODS

- $hub->send($event)

  This is where all events enter the hub for processing.

- $hub->process($event)

  This is called by send after it does any IPC handling. You can use this to bypass the IPC process, but in general you should avoid using this.

- $old = $hub->format($formatter)

  Replace the existing formatter instance with a new one. Formatters must be objects that implement a `$formatter->write($event)` method.

- $sub = $hub->listen(sub { ... }, %optional_params)

  You can use this to record all events AFTER they have been sent to the formatter. No changes made here will be meaningful, except possibly to other listeners.

```
$hub->listen(sub {
    my ($hub, $event, $number) = @_;

    ... do whatever you want with the event ...

    # return is ignored
});
```

  Normally listeners are not inherited by child hubs such as subtests. You can add the `inherit => 1` parameter to allow a listener to be inherited.

- $hub->unlisten($sub)

  You can use this to remove a listen callback. You must pass in the coderef returned by the `listen()` method.

- $sub = $hub->filter(sub { ... }, %optional_params)

- $sub = $hub->pre_filter(sub { ... }, %optional_params)

  These can be used to add filters. Filters can modify, replace, or remove events before anything else can see them.

```
$hub->filter(
    sub {
        my ($hub, $event) = @_;

        return $event;    # No Changes
        return;           # Remove the event

        # Or you can modify an event before returning it.
        $event->modify;
        return $event;
    }
);
```

If you are not using threads, forking, or IPC then the only difference between a `filter` and a `pre_filter` is that `pre_filter` subs run first. When you are using threads, forking, or IPC, pre_filters happen to events before they are sent to their destination proc/thread, ordinary filters happen only in the destination hub/thread.

You cannot add a regular filter to a hub if the hub was created in another process or thread. You can always add a pre_filter.

- $hub->unfilter($sub)
- $hub->pre_unfilter($sub)

  These can be used to remove filters and pre_filters. The `$sub` argument is the reference returned by `filter()` or `pre_filter()`.

- $hub->follow_up(sub { ... })

  Use this to add behaviors that are called just before the hub is finalized. The only argument to your codeblock will be a *Test2::EventFacet::Trace* instance.

```
$hub->follow_up(sub {
    my ($trace, $hub) = @_;

    ... do whatever you need to ...

    # Return is ignored
});
```

  follow_up subs are called only once, ether when done_testing is called, or in an END block.

- $sub = $hub->add_context_acquire(sub { ... });

  Add a callback that will be called every time someone tries to acquire a context. It gets a single argument, a reference of the hash of parameters being used the construct the context. This is your chance to change the parameters by directly altering the hash.

```
test2_add_callback_context_acquire(sub {
    my $params = shift;
    $params->{level}++;
});
```

  This is a very scary API function. Please do not use this unless you need to. This is here for *Test::Builder* and backwards compatibility. This has you directly manipulate the hash instead of returning a new one for performance reasons.

  Note Using this hook could have a huge performance impact.

  The coderef you provide is returned and can be used to remove the hook later.

- $hub->remove_context_acquire($sub);

  This can be used to remove a context acquire hook.

- $sub = $hub->add_context_init(sub { ... });

This allows you to add callbacks that will trigger every time a new context is created for the hub. The only argument to the sub will be the *Test2::API::Context* instance that was created.

Note Using this hook could have a huge performance impact.

The coderef you provide is returned and can be used to remove the hook later.

- $hub->remove_context_init($sub);

  This can be used to remove a context init hook.

- $sub = $hub->add_context_release(sub { ... });

  This allows you to add callbacks that will trigger every time a context for this hub is released. The only argument to the sub will be the *Test2::API::Context* instance that was released. These will run in reverse order.

  Note Using this hook could have a huge performance impact.

  The coderef you provide is returned and can be used to remove the hook later.

- $hub->remove_context_release($sub);

  This can be used to remove a context release hook.

- $hub->cull()

  Cull any IPC events (and process them).

- $pid = $hub->pid()

  Get the process id under which the hub was created.

- $tid = $hub->tid()

  Get the thread id under which the hub was created.

- $hud = $hub->hid()

  Get the identifier string of the hub.

- $uuid = $hub->uuid()

  If UUID tagging is enabled (see *Test2::API*) then the hub will have a UUID.

- $ipc = $hub->ipc()

  Get the IPC object used by the hub.

- $hub->set_no_ending($bool)
- $bool = $hub->no_ending

  This can be used to disable auto-ending behavior for a hub. The auto-ending behavior is triggered by an end block and is used to cull IPC events, and output the final plan if the plan was 'NO PLAN'.

- $bool = $hub->active
- $hub->set_active($bool)

  These are used to get/set the 'active' attribute. When true this attribute will force `hub->finalize()` to take action even if there is no plan, and no tests have been

run. This flag is useful for plugins that add follow-up behaviors that need to run even if no events are seen.

STATE METHODS

- $hub->reset_state()

  Reset all state to the start. This sets the test count to 0, clears the plan, removes the failures, etc.

- $num = $hub->count

  Get the number of tests that have been run.

- $num = $hub->failed

  Get the number of failures (Not all failures come from a test fail, so this number can be larger than the count).

- $bool = $hub->ended

  True if the testing has ended. This MAY return the stack frame of the tool that ended the test, but that is not guaranteed.

- $bool = $hub->is_passing

- $hub->is_passing($bool)

  Check if the overall test run is a failure. Can also be used to set the pass/fail status.

- $hub->plan($plan)

- $plan = $hub->plan

  Get or set the plan. The plan must be an integer larger than 0, the string 'NO PLAN', or the string 'SKIP'.

- $bool = $hub->check_plan

  Check if the plan and counts match, but only if the tests have ended. If tests have not ended this will return undef, otherwise it will be a true/false.

THIRD PARTY META-DATA

This object consumes *Test2::Util::ExternalMeta* which provides a consistent way for you to attach meta-data to instances of this class. This is useful for tools, plugins, and other extensions.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Modules

- `Test2::Hub::Interceptor` - Hub used by interceptor to grab results.
- `Test2::Hub::Subtest` - Hub used by subtests

Test2::Hub::Interceptor

Hub used by interceptor to grab results.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Modules

- `Test2::Hub::Interceptor::Terminator` - Exception class used by Test2::Hub::Interceptor

Test2::Hub::Interceptor::Terminator

Exception class used by Test2::Hub::Interceptor

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::Hub::Subtest

Hub used by subtests

DESCRIPTION

Subtests make use of this hub to route events.

TOGGLES

- $bool = $hub->manual_skip_all
- $hub->set_manual_skip_all($bool)

  The default is false.

  Normally a skip-all plan event will cause a subtest to stop executing. This is accomplished via `last LABEL` to a label inside the subtest code. Most of the time this is perfectly fine. There are times however where this flow control causes bad things to happen.

This toggle lets you turn off the abort logic for the hub. When this is toggled to true **you** are responsible for ensuring no additional events are generated.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::IPC

Turn on IPC for threading or forking support.

SYNOPSIS

You should `use Test2::IPC;` as early as possible in your test file. If you import this module after API initialization it will attempt to retrofit IPC onto the existing hubs.

DISABLING IT

You can use `no Test2::IPC;` to disable IPC for good. You can also use the T2_NO_IPC env var.

EXPORTS

All exports are optional.

- cull()

  Cull allows you to collect results from other processes or threads on demand.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Modules

- `Test2::IPC::Driver` - Base class for Test2 IPC drivers.

Test2::IPC::Driver

Base class for Test2 IPC drivers.

SYNOPSIS

```
package Test2::IPC::Driver::MyDriver;

use base 'Test2::IPC::Driver';

...
```

METHODS

- $self->abort($msg)

  If an IPC encounters a fatal error it should use this. This will print the message to STDERR with `'IPC Fatal Error: '` prefixed to it, then it will forcefully exit 255. IPC errors may occur in threads or processes other than the main one, this method provides the best chance of the harness noticing the error.

- $self->abort_trace($msg)

  This is the same as `$ipc->abort($msg)` except that it uses `Carp::longmess` to add a stack trace to the message.

LOADING DRIVERS

Test2::IPC::Driver has an `import()` method. All drivers inherit this import method. This import method registers the driver.

In most cases you just need to load the desired IPC driver to make it work. You should load this driver as early as possible. A warning will be issued if you load it too late for it to be effective.

```
use Test2::IPC::Driver::MyDriver;
...
```

WRITING DRIVERS

```
package Test2::IPC::Driver::MyDriver;
use strict;
use warnings;

use base 'Test2::IPC::Driver';

sub is_viable {
    return 0 if $^O eq 'win32'; # Will not work on windows.
    return 1;
}

sub add_hub {
    my $self = shift;
    my ($hid) = @_;

    ... # Make it possible to contact the hub
}

sub drop_hub {
    my $self = shift;
    my ($hid) = @_;

    ... # Nothing should try to reach the hub anymore.
}

sub send {
    my $self = shift;
    my ($hid, $e, $global) = @_;

    ... # Send the event to the proper hub.

    # This may notify other procs/threads that there is a pending␣
↪event.
    Test2::API::test2_ipc_set_pending($uniq_val);
}
```

(continues on next page)

(continued from previous page)

```
sub cull {
    my $self = shift;
    my ($hid) = @_;

    my @events = ...; # Here is where you get the events for the hub

    return @events;
}

sub waiting {
    my $self = shift;

    ... # Notify all listening procs and threads that the main
    ... # process/thread is waiting for them to finish.
}

1;
```

METHODS SUBCLASSES MUST IMPLEMENT

- $ipc->is_viable

  This should return true if the driver works in the current environment. This should return false if it does not. This is a CLASS method.

- $ipc->add_hub($hid)

  This is used to alert the driver that a new hub is expecting events. The driver should keep track of the process and thread ids, the hub should only be dropped by the proc+thread that started it.

  ```
  sub add_hub {
      my $self = shift;
      my ($hid) = @_;

      ... # Make it possible to contact the hub
  }
  ```

- $ipc->drop_hub($hid)

  This is used to alert the driver that a hub is no longer accepting events. The driver should keep track of the process and thread ids, the hub should only be dropped by the proc+thread that started it (This is the drivers responsibility to enforce).

  ```
  sub drop_hub {
      my $self = shift;
      my ($hid) = @_;

      ... # Nothing should try to reach the hub anymore.
  }
  ```

- $ipc->send($hid, $event);

- $ipc->send($hid, $event, $global);

  Used to send events from the current process/thread to the specified hub in its process+thread.

```
sub send {
    my $self = shift;
    my ($hid, $e) = @_;

    ... # Send the event to the proper hub.

    # This may notify other procs/threads that there is a
↪pending event.
    Test2::API::test2_ipc_set_pending($uniq_val);
}
```

  If `$global` is true then the driver should send the event to all hubs in all processes and threads.

- @events = $ipc->cull($hid)

  Used to collect events that have been sent to the specified hub.

```
sub cull {
    my $self = shift;
    my ($hid) = @_;

    my @events = ...; # Here is where you get the events for
↪the hub

    return @events;
}
```

- $ipc->waiting()

  This is called in the parent process when it is complete and waiting for all child processes and threads to complete.

```
sub waiting {
    my $self = shift;

    ... # Notify all listening procs and threads that the main
    ... # process/thread is waiting for them to finish.
}
```

METHODS SUBCLASSES MAY IMPLEMENT OR OVERRIDE

- $ipc->driver_abort($msg)

  This is a hook called by `Test2::IPC::Driver->abort()`. This is your chance to cleanup when an abort happens. You cannot prevent the abort, but you can gracefully except it.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Modules

- `Test2::IPC::Driver::Files` - Temp dir + Files concurrency model.

Test2::IPC::Driver::Files

Temp dir + Files concurrency model.

DESCRIPTION

This is the default, and fallback concurrency model for *Test2*. This sends events between processes and threads using serialized files in a temporary directory. This is not particularly fast, but it works everywhere.

SYNOPSIS

```
use Test2::IPC::Driver::Files;

# IPC is now enabled
```

ENVIRONMENT VARIABLES

- T2_KEEP_TEMPDIR=0

  When true, the tempdir used by the IPC driver will not be deleted when the test is done.

- T2_TEMPDIR_TEMPLATE='test2-XXXXXX'

  This can be used to set the template for the IPC temp dir. The template should follow template specifications from *File::Temp*.

SEE ALSO

See *Test2::IPC::Driver* for methods.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::Manual

Documentation hub for Test2 and Test2-Suite.

DESCRIPTION

This is the hub for *Test2* and *Test2::Suite* documentation.

WRITING TESTS

The *Test2::Manual::Testing* POD is the hub for documentation related to writing tests.

WRITING TOOLS

The *Test2::Manual::Tooling* POD is the hub for documentation related to writing new tools.

GUTS AND INNER WORKINGS

The *Test2::Manual::Anatomy* POD is the hub for documentation of the inner workings of Test2 components.

A NOTE ON CONCURRENCY (SUPPORT FOR FORKING AND THREADING)

The *Test2::Manual::Concurrency* POD documents the concurrency support policy for *Test2*.

CONTRIBUTING

The *Test2::Manual::Contributing* POD is for people who want to contribute to *Test2* or *Test2::Suite* directly.

SEE ALSO

Test2 - Test2 itself.

Test2::Suite - Initial tools built using *Test2*.

SOURCE

The source code repository for Test2-Manual can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Modules

- `Test2::Manual::Anatomy` - The hub for documentation of the inner workings of Test2 components.
- `Test2::Manual::Concurrency` - Documentation for Concurrency support.
- `Test2::Manual::Contributing` - How to contribute to the Test2 project.

- `Test2::Manual::Testing` - Hub for documentation about writing tests with Test2.
- `Test2::Manual::Tooling` - Manual page for tool authors.

Test2::Manual::Anatomy

The hub for documentation of the inner workings of Test2 components.

DESCRIPTION

This section covers internals of the Test2 architecture. This is useful information for toolbuilder, but is essential information for maintainers of Test2 itself.

END TO END

The *Test2::Manual::Anatomy::EndToEnd* document is an overview of Test2 from load to finish.

EVENTS

The *Test2::Manual::Anatomy::Event* document explains the internals of events.

THE CONTEXT

The *Test2::Manual::Anatomy::Context* document explains how the *Test2::API::Context* object works.

THE API AND THE API INSTANCE

The *Test2::Manual::Anatomy::API* document explains the inner workings of the Test2 API.

HUBS

The *Test2::Manual::Anatomy::Hubs* document explains the inner working of the Test2 hub stack, and the hubs therein.

THE IPC SYSTEM

The *Test2::Manual::Anatomy::IPC* document describes the IPC system.

INTERNAL UTILITIES

The *Test2::Manual::Anatomy::Utilities* document describes various utilities provided by the Test2 system.

SEE ALSO

Test2::Manual - Primary index of the manual.

SOURCE

The source code repository for Test2-Manual can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Modules

- `Test2::Manual::Anatomy::API` - Internals documentation for the API.
- `Test2::Manual::Anatomy::Context` - Internals documentation for the Context objects.
- `Test2::Manual::Anatomy::EndToEnd` - Overview of Test2 from load to finish.
- `Test2::Manual::Anatomy::Event` - The internals of events
- `Test2::Manual::Anatomy::Hubs` - Internals documentation for the hub stack, and hubs.
- `Test2::Manual::Anatomy::IPC` - Manual for the IPC system.
- `Test2::Manual::Anatomy::Utilities` - Overview of utilities for Test2.

Test2::Manual::Anatomy::API

Internals documentation for the API.

DESCRIPTION

This document covers some of the internals of *Test2::API*.

IMPLEMENTATION DETAILS

Test2::API

Test2::API provides a functional interface to any test2 global state. This API should be preserved regardless of internal details of how and where the global state is stored.

This module itself does not store any state (with a few minor exceptions) but instead relies on *Test2::API::Instance* to store state. This module is really intended to be the layer between the consumer and the implementation details. Ideally the implementation details can change any way they like, and this module can be updated to use the new details without breaking anything.

Test2::API::Instance

Test2::API::Instance is where the global state is actually managed. This is an implementation detail, and should not be relied upon. It is entirely possible that *Test2::API::Instance* could be removed completely, or changed in incompatible ways. Really these details are free to change so long as *Test2::API* is not broken.

Test2::API::Instance is fairly well documented, so no additionally documentation is needed for this manual page.

SEE ALSO

Test2::Manual - Primary index of the manual.

SOURCE

The source code repository for Test2-Manual can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Manual::Anatomy::Context

Internals documentation for the Context objects.

DESCRIPTION

This document explains how the *Test2::API::Context* object works.

WHAT IS THE CONTEXT OBJECT?

The context object is one of the key components of Test2, and makes many features possible that would otherwise be impossible. Every test tool starts by getting a context, and ends by releasing the context. A test tool does all its work between getting and releasing the context. The context instance is the primary interface for sending events to the Test2 stack. Finally the context system is responsible for tracking what file and line number a tool operates on, which is critical for debugging.

PRIMARY INTERFACE FOR TEST TOOLS

Nearly every Test2 based tool should start by calling `$ctx = Test2::API::context()` in order to get a context object, and should end by calling `$ctx->release()`. Once a tool has its context object it can call methods on the object to send events or have other effects. Nearly everything a test tool needs to do should be done through the context object.

TRACK FILE AND LINE NUMBERS FOR ERROR REPORTING

When you call `Test2::API::Context` a new context object will be returned. If there is already a context object in effect (from a different point in the stack) you will get a clone of the existing one. If there is not already a current context then a completely new one will be generated. When a new context is generated Test2 will determine the file name and line number for your test code, these will be used when reporting any failures.

Typically the file and line number will be determined using `caller()` to look at your tools caller. The `$Test::Builder::Level` will be respected if detected, but is discouraged in favor of just using context objects at every level.

When calling `Test2::API::Context()` you can specify the `level => $count` arguments if you need to look at a deeper caller.

PRESERVE $?, $!, $^E AND $@

When you call `Test2::API::context()` the current values of `$?`, `$!`, `$^E`, and `$@` are stored in the context object itself. Whenever the context is released the original values of these variables will be restored. This protects the variables from any side effects caused by testing tools.

FINALIZE THE API STATE

Test2::API works via a hidden singleton instance of *Test2::API::Instance*. The singleton has some state that is not set in stone until the last possible minute. The last possible minute happens to be the first time a context is acquired. State includes IPC instance, Formatter class, Root PID, etc.

FIND/CREATE THE CURRENT/ROOT HUB

Test2 has a stack of hubs, the stack can be accessed via *Test2::API::test2_stack*. When you get a context it will find the current hub, if there is no current hub then the root one will be initialized.

PROVIDE HOOKS

There are hooks that run when contexts are created, found, and released. See *Test2::API* for details on these hooks and how to use them.

SEE ALSO

Test2::Manual - Primary index of the manual.

SOURCE

The source code repository for Test2-Manual can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Manual::Anatomy::EndToEnd

Overview of Test2 from load to finish.

DESCRIPTION

This is a high level overview of everything from loading Test2 through the end of a test script.

WHAT HAPPENS WHEN I LOAD THE API?

```
use Test2::API qw/context/;
```

- A singleton instance of Test2::API::Instance is created.

  You have no access to this, it is an implementation detail.

- Several API functions are defined that use the singleton instance.

  You can import these functions, or use them directly.

- Then what?

  It waits...

  The API intentionally does as little as possible. At this point something can still change the formatter, load *Test2::IPC*, or have other global effects that need to be done before the first *Test2::API::Context* is created. Once the first *Test2::API::Context* is created the API will finish initialization.

  See "WHAT HAPPENS WHEN I ACQUIRE A CONTEXT?" for more information.

WHAT HAPPENS WHEN I USE A TOOL?

This section covers the basic workflow all tools such as `ok()` must follow.

```
sub ok($$) {
    my ($bool, $name) = @_;

    my $ctx = context();

    my $event = $ctx->send_event('Ok', pass => $bool, name =>
↪$name);

    ...

    $ctx->release;
    return $bool;
}

ok(1, "1 is true");
```

- A tool function is run.

  ```
  ok(1, "1 is true");
  ```

- The tool acquires a context object.

```
my $ctx = context();
```

See “WHAT HAPPENS WHEN I ACQUIRE A CONTEXT?” for more information.

- The tool uses the context object to create, send, and return events.

  See “WHAT HAPPENS WHEN I SEND AN EVENT?” for more information.

```
my $event = $ctx->send_event('Ok', pass => $bool, name =>
↪$name);
```

- When done the tool MUST release the context.

  See “WHAT HAPPENS WHEN I RELEASE A CONTEXT?” for more information.

```
$ctx->release();
```

- The tool returns.

```
return $bool;
```

WHAT HAPPENS WHEN I ACQUIRE A CONTEXT?

```
my $ctx = context();
```

These actions may not happen exactly in this order, but that is an implementation detail. For the purposes of this document this order is used to help the reader understand the flow.

- $!, $@, $? and $^E are captured and preserved.

  Test2 makes a point to preserve the values of $!, $@, $? and $^E such that the test tools do not modify these variables unexpectedly. They are captured first thing so that they can be restored later.

- The API state is changed to ‘loaded’.

  The ‘loaded’ state means that test tools have already started running. This is important as some plugins need to take effect before any tests are run. This state change only happens the first time a context is acquired, and may trigger some hooks defined by plugins to run.

- The current hub is found.

  A context attaches itself to the current *Test2::Hub*. If there is no current hub then the root hub will be initialized. This will also initialize the hub stack if necessary.

- Context acquire hooks fire.

  It is possible to create global, or hub-specific hooks that fire whenever a context is acquired, these hooks will fire now. These hooks fire even if there is an existing context.

- Any existing context is found.

If the current hub already has a context then a clone of it will be used instead of a completely new context. This is important because it allows nested tools to inherit the context used by parent tools.

- Stack depth is measured.

  Test2 makes a point to catch mistakes in how the context is used. The stack depth is used to accomplish this. If there is an existing context the depth will be checked against the one found here. If the old context has the same stack depth, or a shallower one, it means a tool is misbehaving and did not clean up the context when it was done, in which case the old context will be cleaned up, and a warning issued.

- A new context is created (if no existing context was found)

  If there is no existing context, a new one will be created using the data collected so far.

- Context init hooks fire (if no existing context was found)

  If a new context was created, context-creation hooks will fire.

- $!, $@, $?, and $^E are restored.

  We make sure $!, $@, $?, and $^E are unchanged at this point so that changes we made will not affect anything else. This is done in case something inside the context construction accidentally changed these vars.

- The context is returned.

  You have a shiny new context object, or a clone of the existing context.

WHAT HAPPENS WHEN I SEND AN EVENT?

```
my $event = $ctx->send_event('Ok', pass => $bool, name => $name);
```

- The Test2::Event::Ok module is loaded.

  The `send_event()` method will automatically load any Event package necessary. Normally `send_event()` will assume the first argument is an event class without the `Test2::Event::` prefix, which it will add for you. If you want to use an event class that is in a different namespace you can prefix the class name with a `+` to tell the tool that you are giving a fully qualified class name:

  ```
  my $event = $ctx->send_event('+Fully::Qualified::Event', pass =>
  ↪ $bool, name => $name);
  ```

- A new instance of Test2::Event::Ok is created.

  The event object is instantiated using the provided parameters.

- The event object is sent to the hub.

  The hub takes over from here.

- The hub runs the event through any filters.

  Filters are able to modify or remove events. Filters are run first, before the event can modify global test state.

- The global test state is updated to reflect the event.

  If the event affects test count then the count will be incremented. If the event causes failure then the failure count will be incremented. There are a few other ways the global state can be affected as well.

- The event is sent to the formatter

  After the state is changed the hub will send the event to the formatter for rendering. This is where TAP is normally produced.

- The event is sent to all listeners.

  There can be any number of listeners that take action when events are processed, this happens now.

WHAT HAPPENS WHEN I RELEASE A CONTEXT?

```
$ctx->release;
```

- The current context clone is released.

  If your tool is nested inside another, then releasing will simply destroy the copy of the context, nothing else will happen.

- If this was the canonical context, it will actually release

  When a context is created it is considered 'canon'. Any context obtained by a nested tool will be considered a child context linked to the canonical one. Releasing child contexts does not do anything of note (but is still required).

- Release hooks are called

  Release hooks are the main motivation behind making the `release()` method, and making it a required action on the part of test tools. These are hooks that we can have called when a tool is complete. This is how plugins like *Test2::Plugin::DieOnFail* are implemented. If we simply had a destructor call the hooks then we would be unable to write this plugin as a `die` inside of a destructor is useless.

- The context is cleared

  The main context data is cleared allowing the next tool to create a new context. This is important as the next tool very likely has a new line number.

- $!, $@, $?, and $^E are restored

  When a Test2 tool is complete it will restore $@, $!, $? and $^E to avoid action at a distance.

WHAT HAPPENS WHEN I USE done_testing()?

```
done_testing();
```

- Any pending IPC events will be culled.

  If IPC is turned on, a final culling will take place.

- Follow-up hooks are run

  The follow-up hooks are a way to run actions when a hub is complete. This is useful for adding cleanup tasks, or final tests to the end of a test.

- The final plan event is generated and processed.

  The final plan event will be produced using the current test count as the number of tests planned.

- The current hub is finalized.

  This will mark the hub is complete, and will not allow new events to be processed.

WHAT HAPPENS WHEN A TEST SCRIPT IS DONE?

Test2 has some behaviors it runs in an `END { ... }` block after tests are done running. This end block does some final checks to warn you if something went wrong. This end block also sets the exit value of the script.

- API Versions are checked.

  A warning will be produced if *Test::Builder* is loaded, but has a different version compared to *Test2::API*. This situation can happen if you downgrade to an older Test-Simple distribution, and is a bad situation.

- Any remaining context objects are cleaned up.

  If there are leftover context objects they will need to be cleaned up. A leftover context is never a good thing, and usually requires a warning. A leftover context could also be the result of an exception being thrown which terminates the script, *Test2* is fairly good at noticing this and not warning in these cases as the warning would simply be noise.

- Child processes are sent a 'waiting' event.

  If IPC is active, a waiting event is sent to all child processes.

- The script will wait for all child processes and/or threads to complete.

  This happens only when IPC is loaded, but Test::Builder is not. This behavior is useful, but would break compatibility for legacy tests.

- The hub stack is cleaned up.

  All hubs are finalized starting from the top. Leftover hubs are usually a bad thing, so a warning is produced if any are found.

- The root hub is finalized.

  This step is a no-op if `done_testing()` was used. If needed this will mark the root hub as finished.

- Exit callbacks are called.

  This is a chance for plugins to modify the final exit value of the script.

- The scripts exit value ($?) is set.

  If the test encountered any failures this will be set to a non-zero value. If possible this will be set to the number of failures, or 255 if the number is larger than 255 (the max value allowed).

- Broken module diagnostics

  Test2 is aware of many modules which were broken by Test2's release. At this point the script will check if any known-broken modules were loaded, and warn you if they were.

  Note: This only happens if there were test failures. No broken module warnings are produced on a success.

SEE ALSO

Test2::Manual - Primary index of the manual.

SOURCE

The source code repository for Test2-Manual can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Manual::Anatomy::Event

The internals of events

DESCRIPTION

Events are how tools affect global state, and pass information along to the harness, or the human running the tests.

HISTORY

Before proceeding it is important that you know some history of events. Initially there was an event API, and an event would implement the API to produce an effect. This API proved to be lossy and inflexible. Recently the 'facet' system was introduced, and makes up for the shortcoming and inflexibility of the old API.

All events must still implement the old API, but that can be largely automated if you use the facet system effectively. Likewise essential facets can often be deduced from events that only implement the old API, though their information maybe less complete.

THE EVENT OBJECT

All event objects must subclass *Test2::Event*. If you inherit from this base class, and implement the old API properly, facets will be generated for you for free. On the other hand you can inherit from this, and also import *Test2::Util::Facets2Legacy* which will instead rely on your facet data, and deduce the old API from them.

All new events MUST implement both APIs one way or the other. A common way to do this is to simply implement both APIs directly in your event.

Here is a good template for a new event:

```
package Test2::Event::Mine;
use strict;
use warnings;

use parent 'Test2::Event';
use Test2::Util::Facets2Legacy ':ALL';

sub facet_data {
    my $self = shift;

    # Adds 'about', 'amnesty', and 'trace' facets
    my $out = $self->common_facet_data;

    # Add any additional facets to the $out hashref
    ...

    return $out;
}

1;
```

THE FACET API

The new API is a single method: `facet_data()`. This method must return a hashref where each key is specific to a facet type, and the value is either a facet hashref, or an array of hashrefs. Some facets MUST be lone hashrefs, others MUST be hashrefs inside an arrayref.

The *standard* facet types are as follows:

- assert => {details => $name, pass => $bool, no_debug => $bool, number => $maybe_int}

  Documented in *Test2::EventFacet::Assert*. An event may only have one.

  The 'details' key is the name of the assertion.

  The 'pass' key denotes a passing or failing assertion.

The 'no_debug' key tells any harness or formatter that diagnostics should not be added automatically to a failing assertion (used when there are custom diagnostics instead).

The 'number' key is for harness use, never set it yourself.

- about => {details => $string, no_display => $bool, package => $pkg}

  Documented in *Test2::EventFacet::About*. An event may only have one.

  'details' is a human readable string describing the overall event.

  'no_display' means that a formatter/harness should hide the event.

  'package' is the package of the event the facet describes (IE: *Test2::Event::Ok*)

- amnesty => [{details => $string, tag => $short_string, inherited => $bool}]

  Documented in *Test2::EventFacet::Amnesty*. An event may have multiple.

  This event is how things like 'todo' are implemented. Amnesty prevents a failing assertion from causing a global test failure.

  'details' is a human readable description of why the failure is being granted amnesty (IE The 'todo' reason)

  'tag' is a short human readable string, or category for the amnesty. This is typically 'TODO' or 'SKIP'.

  'inherited' is true if the amnesty was applied in a parent context (true if this test is run in a subtest that is marked todo).

- control => {details => $string, global => $bool, terminate => $maybe_int, halt => $bool, has_callback => $bool, encoding => $enc}

  Documented in *Test2::EventFacet::Control*. An event may have one.

  This facet is used to apply extra behavior when the event is processed.

  'details' is a human readable explanation for the behavior.

  'global' true if this event should be forwarded to, and processed by, all hubs everywhere. (bail-out uses this)

  'terminate' this should either be undef, or an integer. When defined this will cause the test to exit with the specific exit code.

  'halt' is used to signal any harness that no further test files should be run (bail-out uses this).

  'has_callback' is set to true if the event has a callback sub defined.

  'encoding' used to tell the formatter what encoding to use.

- errors => [{details => $string, tag => $short_string, fail => $bool}]

  Documented in *Test2::EventFacet::Error*. An event may have multiple.

  'details' is a human readable explanation of the error.

  'tag' is a short human readable category for the error.

  'fail' is true if the error should cause test failure. If this is false the error is simply informative, but not fatal.

- info => [{details => $string, tag => $short_string, debug => $bool, important => $bool}]

  Documented in *Test2::EventFacet::Info*. An event may have multiple.

  This is how diag and note are implemented.

  'details' human readable message.

  'tag' short category for the message, such as 'diag' or 'note'.

  'debug' is true if the message is diagnostics in nature, this is the main difference between a note and a diag.

  'important' is true if the message is not diagnostics, but is important to have it shown anyway. This is primarily used to communicate with a harness.

- parent => {details => $string, hid => $hid, children => [...], buffered => 1}

  Documented in *Test2::EventFacet::Parent*. An event may have one.

  This is used by subtests.

  'details' human readable name of the subtest.

  'hid' subtest hub id.

  'children' an arrayref containing facet_data instances from all child events.

  'buffered' true if it was a buffered subtest.

- plan => {details => $string, count => $int, skip => $bool, none => $bool}

  Documented in *Test2::EventFacet::Plan*. An event may have one.

  'details' is a human readable string describing the plan (for instance, why a test is skipped)

  'count' is the number of expected assertions (0 for skip)

  'skip' is true if the plan is to skip the test.

  'none' used for Test::More's 'no_plan' plan.

- trace => {details => $string, frame => [$pkg, $file, $line, $sub], pid => $int, tid => $int, cid => $cid, hid => $hid, nested => $int, buffered => $bool}

  Documented in *Test2::EventFacet::Trace*. An event may have one.

  This is how debugging information is tracked. This is taken from the context object at event creation.

  'details' human readable debug message (otherwise generated from frame)

  'frame' first 4 fields returned by caller: `[$package, $file, $line, $subname]`.

  'pid' the process id in which the event was created.

  'tid' the thread is in which the event was created.

  'cid' the id of the context used to create the event.

  'hid' the id of the hub to which the event was sent.

  'nest' subtest nesting depth of the event.

'buffered' is true if the event was generated inside a buffered subtest.

Note that ALL facet types have a 'details' key that may have a string. This string should always be human readable, and should be an explanation for the facet. For an assertion this is the test name. For a plan this is the reason for the plan (such as skip reason). For info it is the human readable diagnostics message.

CUSTOM FACETS

You can write custom facet types as well, simply add a new key to the hash and populated it. The general rule is that any code looking at the facets should ignore any it does not understand.

Optionally you can also create a package to document your custom facet. The package should be proper object, and may have additional methods to help work with your facet.

```
package Test2::EventFacet::MyFacet;

use parent 'Test2::EventFacet';

sub facet_key { 'myfacet' }
sub is_list { 0 }

1;
```

Your facet package should always be under the Test2::EventFacet:: namespace if you want any tools to automatically find it. The last part of the namespace should be the non-plural name of your facet with only the first word capitalized.

- $string = $facet_class->facet_key

  The key for your facet should be the same as the last section of the namespace, but all lowercase. You *may* append 's' to the key if your facet is a list type.

- $bool = $facet_class->is_list

  True if an event should put these facets in a list:

  ```
  { myfacet => [{}, {}] }
  ```

  False if an event may only have one of this type of facet at a time:

  ```
  { myfacet => {} }
  ```

EXAMPLES

The assert facet is not a list type, so its implementation would look like this:

```
package Test2::EventFacet::Assert;
sub facet_key { 'assert' }
sub is_list { 0 }
```

The amnesty facet is a list type, but amnesty does not need 's' appended to make it plural:

```
package Test2::EventFacet::Amnesty;
sub facet_key { 'amnesty' }
sub is_list { 1 }
```

The error facet is a list type, and appending 's' makes error plural as errors. This means the package name is '::Error', but the key is 'errors'.

```
package Test2::EventFacet::Error;
sub facet_key { 'errors' }
sub is_list { 1 }
```

Note2: In practice most tools completely ignore the facet packages, and work with the facet data directly in its raw structure. This is by design and recommended. The facet data is intended to be serialized frequently and passed around. When facets are concerned, data is important, classes and methods are not.

THE OLD API

The old API was simply a set of methods you were required to implement:

- $bool = $e->causes_fail

  Returns true if this event should result in a test failure. In general this should be false.

- $bool = $e->increments_count

  Should be true if this event should result in a test count increment.

- $e->callback($hub)

  If your event needs to have extra effects on the *Test2::Hub* you can override this method.

  This is called **BEFORE** your event is passed to the formatter.

- $num = $e->nested

  If this event is nested inside of other events, this should be the depth of nesting. (This is mainly for subtests)

- $bool = $e->global

  Set this to true if your event is global, that is ALL threads and processes should see it no matter when or where it is generated. This is not a common thing to want, it is used by bail-out and skip_all to end testing.

- $code = $e->terminate

  This is called **AFTER** your event has been passed to the formatter. This should normally return undef, only change this if your event should cause the test to exit immediately.

  If you want this event to cause the test to exit you should return the exit code here. Exit code of 0 means exit success, any other integer means exit with failure.

  This is used by *Test2::Event::Plan* to exit 0 when the plan is 'skip_all'. This is also used by *Test2::Event::Bail* to force the test to exit with a failure.

This is called after the event has been sent to the formatter in order to ensure the event is seen and understood.

- $msg = $e->summary

  This is intended to be a human readable summary of the event. This should ideally only be one line long, but you can use multiple lines if necessary. This is intended for human consumption. You do not need to make it easy for machines to understand.

  The default is to simply return the event package name.

- ($count, $directive, $reason) = $e->sets_plan()

  Check if this event sets the testing plan. It will return an empty list if it does not. If it does set the plan it will return a list of 1 to 3 items in order: Expected Test Count, Test Directive, Reason for directive.

- $bool = $e->diagnostics

  True if the event contains diagnostics info. This is useful because a non-verbose harness may choose to hide events that are not in this category. Some formatters may choose to send these to STDERR instead of STDOUT to ensure they are seen.

- $bool = $e->no_display

  False by default. This will return true on events that should not be displayed by formatters.

SEE ALSO

Test2::Manual - Primary index of the manual.

SOURCE

The source code repository for Test2-Manual can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Manual::Anatomy::Hubs

Internals documentation for the hub stack, and hubs.

DESCRIPTION

This document describes the hub stack, and the hubs it contains. It explains why we have a stack, and when to add/remove hubs from it.

WHAT IS A HUB?

Test2 is an event system, tools generate events, those events are then processed to modify the testing state (number of tests, number of failures, etc). The hub is responsible for receiving and processing events to record the change in state. All events should eventually reach a destination hub.

The base hub is *Test2::Hub*. All hub classes should inherit from the base hub class. The base hub class provides several hooks that allow you to monitor or modify events. Hubs are also responsible for forwarding events to the output formatter.

WHY DO WE HAVE A HUB STACK?

There are cases where it makes sense to have more than one hub:

- subtests

  In Test2 subtests are implemented using the hub stack. When you start a subtest a new *Test2::Hub::Subtest* instance is created and pushed to the stack. Once this is done all calls to `Test2::API::context` will find the new hub and send all events to it. When the subtest tool is complete it will remove the new hub, and send a final subtest event to the parent hub.

- testing your test tools

  `Test2::API::intercept()` is implemented using the hub stack. The `Test2::API::intercept()` function will add an *Test2::Hub::Interceptor* instance to the stack, any calls to Test2::API::context() will find the new hub, and send it all events. The intercept hub is special in that is has no connection to the parent hub, and usually does not have a formatter.

WHEN SHOULD I ADD A HUB TO THE STACK?

Any time you want to intercept or block events from affecting the test state. Adding a new hub is essentially a way to create a sandbox where you have absolute control over what events do. Adding a new hub insures that the main test state will not be affected.

WHERE IS THE STACK?

The stack is an instance of *Test2::API::Stack*. You can access the global hub stack using `Test2::API::test2_stack`.

WHAT ABOUT THE ROOT HUB?

The root hub is created automatically as needed. A call to `Test2::API::test2_stack->top()` will create the root hub if it does not already exist.

HOW DO HUBS HANDLE IPC?

If the IPC system (*Test2::IPC*) was not loaded, then IPC is not handled at all. Forking or creating new threads without the IPC system can cause unexpected problems.

All hubs track the PID and Thread ID that was current when they were created. If an event is sent to a hub in a new process/thread the hub will detect this and try to forward the event along to the correct process/thread. This is accomplished using the IPC system.

SEE ALSO

Test2::Manual - Primary index of the manual.

SOURCE

The source code repository for Test2-Manual can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Manual::Anatomy::IPC

Manual for the IPC system.

DESCRIPTION

This document describes the IPC system.

WHAT IS THE IPC SYSTEM

The IPC system is activated by loading *Test2::IPC*. This makes hubs process/thread aware, and makes them forward events along to the parent process/thread as necessary.

HOW DOES THE IPC SYSTEM AFFECT EVERYTHING?

Test2::API and *Test2::API::Instance* have some behaviors that trigger if *Test2::IPC* is loaded before the global state is initialized. Mainly an IPC driver will be initiated and stored in the global state.

If an IPC driver is initialized then all hubs will be initialized with a reference to the driver instance. If a hub has an IPC driver instance it will use it to forward events to parent processes and threads.

WHAT DOES AN IPC DRIVER DO?

An *Test2::IPC::Driver* provides a way to send event data to a destination process+thread+hub (or to all globally). The driver must also provide a way for a process/thread/hub to read in any pending events that have been sent to it.

HOW DOES THE DEFAULT IPC DRIVER WORK?

The default IPC driver is *Test2::IPC::Driver::Files*. This default driver, when initialized, starts by creating a temporary directory. Any time an event needs to be sent to another process/thread/hub, the event will be written to a file using *Storable*. The file is written with the destination process, thread, and hub as part of the filename. All hubs will regularly check for pending IPC events and will process them.

This driver is further optimized using a small chunk of SHM. Any time a new event is sent via IPC the shm is updated to have a new value. Hubs will not bother checking for new IPC events unless the shm value has changed since their last poll. A result of this is that the IPC system is surprisingly fast, and does not waste time polling the hard drive when there are no pending events.

SEE ALSO

Test2::Manual - Primary index of the manual.

SOURCE

The source code repository for Test2-Manual can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Manual::Anatomy::Utilities

Overview of utilities for Test2.

DESCRIPTION

This is a brief overview of the utilities provided by Test2.

Test2::Util

Test2::Util provides functions to help you find out about the current system, or to run generic tasks that tend to be Test2 specific.

This utility provides things like an internal `try {...}` implementation, and constants for things like threading and forking support.

Test2::Util::ExternalMeta

Test2::Util::ExternalMeta allows you to quickly and easily attach meta-data to an object class.

Test2::Util::Facets2Legacy

Test2::Util::Facets2Legacy is a set of functions you can import into a more recent event class to provide the classic event API.

Test2::Util::HashBase

Test2::Util::HashBase is a local copy of Object::HashBase. All object classes provided by *Test2* use this to generate methods and accessors.

SEE ALSO

Test2::Manual - Primary index of the manual.

SOURCE

The source code repository for Test2-Manual can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Manual::Concurrency

Documentation for Concurrency support.

FORKING

Test2

Test2 supports forking. For forking to work you need to load *Test2::IPC*.

Test::Builder

Test::Builder Did not used to support forking, but now that it is based on *Test2* it does. *Test2::IPC* must be loaded just as with *Test2*.

Test2::Suite

Test2::Suite tools should all work fine with *true* forking unless otherwise noted. Pseudo-fork via threads (Windows and a few others) is not supported, but may work.

Patches will be accepted to repair any pseudo-fork issues, but for these to be used or tested they must be requested. Fork tests should not run on pseudo-fork systems unless they are requested with an environment var, or the AUTHOR_TESTING var. Pseudo-fork is fragile, and we do not want to block install due to a pseudo-fork flaw.

Test::SharedFork

Test::SharedFork is currently support and maintained, though it is no longer necessary thanks to *Test2::IPC*. If usage ever drops off then the module may be deprecated, but for now the policy is to not let it break. Currently it simply loads *Test2::IPC* if it can, and falls back to the old methods on legacy installs.

Others

Individual authors are free to support or not support forking as they see fit.

THREADING

Note This only applies to ithreads.

Test2

The core of Test2 supports threading so long as *Test2::IPC* is loaded. Basic threading support (making sure events make it to the parent thread) is fully supported, and must not be broken.

Some times perl installs have broken threads (Some 5.10 versions compiled on newer gcc's will segv by simply starting a thread). This is beyond Test2's control, and not solvable in Test2. That said we strive for basic threading support on perl 5.8.1+.

If Test2 fails for threads on any perl 5.8 or above, and it is reasonably possible for Test2 to work around the issue, it should. (Patches and bug reports welcome).

Test::Builder

Test::Builder has had thread support for a long time. With Test2 the mechanism for thread support was switched to *Test2::IPC*. *Test::Builder* should still support threads as much as it did before the switch to Test2. Support includes auto-enabling thread support if *threads* is loaded before Test::Builder.

If there is a deviation between the new and old threading behavior then it is a bug (unless the old behavior itself can be classified as a bug.) Please report (or patch!) any such threading issues.

Test2::Suite

Tools in *Test2::Suite* have minimal threading support. Most of these tools do not care/notice threading and simply work because *Test2::IPC* handles it. Feel free to report any thread related bugs in Test2::Suite. Be aware though that these tools are not legacy, and have no pre-existing thread support, we reserve the right to refuse adding thread support to them.

Test2::Workflow

Test2::Workflow has been merged into *Test2::Suite*, so it gets addressed by this policy.

Test2::Workflow has thread support, but you must ask for it. Thread tests for Test2::Workflow do not event run without setting either the AUTHOR_TESTING env var, or the T2_DO_THREAD_TESTS env var.

To use threads with Test2::Workflow you must set the T2_WORKFLOW_USE_THREADS env var.

If you do rely on threads with Test2::Workflow and find a bug please report it, but it will be given an ultra-low priority. Merging patches that fix threading issues will be given normal priority.

SEE ALSO

Test2 - Test2 itself.

Test2::Suite - Initial tools built using *Test2*.

Test2::Manual - Primary index of the manual.

SOURCE

The source code repository for Test2-Manual can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Manual::Contributing

How to contribute to the Test2 project.

DESCRIPTION

This is a short manual page dedicated to helping people who wish to contribute to the Test2 project.

WAYS TO HELP

REPORT BUGS

The easiest way to help is to report bugs when you find them. Bugs are a fact of life when writing or using software. If you use Test2 long enough you are likely to find a bug. When you find such a bug it would help us out if you would submit a ticket.

BUG TRACKERS

Always try to find the preferred bug tracker for the module that has the bug. Here are the big 3 for the main Test2 project:

- Test2/Test-Builder/Test-More/test-more

  https://github.com/Test-More/test-more/issues

- Test2-Harness

  https://github.com/Test-More/Test2-Harness/issues

SUBMIT PATCHES

You are welcome to fix bugs you find, or from the tracker. We also often accept patches that add new features or update documentation. The preferred method of submitting patches is a github pull request, that said we also accept patches via email.

ADD/UPDATE DOCUMENTATION

Documentation can be flawed just like code can be. Documentation can also become outdated. If you see some incorrect documentation, or documentation that is missing, we would love to get a patch to fix it!

ANSWER QUESTIONS ON IRC/SLACK

We are always hanging out on irc://irc.perl.org, the #perl-qa and #toolchain channels are a good place to find us.

There is also a Test2 slack channel: https://perl-test2.slack.com.

WRITE NEW TOOLS USING TEST2

Writing a new tool using Test2 is always a good way to contribute. When you write a tool that you think is useful, it is nice to share it by putting it on CPAN.

PORT OLD TOOLS TO TEST2

The `Test::*` namespace has been around for a long time, and has a LOT of tools. The `Test2::Tools::*` namespace is fairly young, and has less tools. Finding a useful old tool with no modern equivalent, and writing a port is a very good use of your time.

SEE ALSO

Test2::Manual - Primary index of the manual.

SOURCE

The source code repository for Test2-Manual can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Manual::Testing

Hub for documentation about writing tests with Test2.

DESCRIPTION

This document outlines all the tutorials and POD that cover writing tests. This section does not cover any Test2 internals, nor does it cover how to write new tools, for that see *Test2::Manual::Tooling*.

NAMESPACE MAP

When writing tests there are a couple namespaces to focus on:

- Test2::Tools::*

  This is where toolsets can be found. A toolset exports functions that help you make assertions about your code. Toolsets will only export functions, they should not ever have extra/global effects.

- Test2::Plugins::*

  This is where plugins live. Plugins should not export anything, but instead will introduce or alter behaviors for Test2 in general. These behaviors may be lexically scoped, or they may be global.

- Test2::Bundle::*

  Bundles combine toolsets and plugins together to reduce your boilerplate. First time test writers are encouraged to start with the *Test2::V1* bundle (which is an exception to the namespace rule as it does not live under `Test2::Bundle::`). If you find yourself loading several plugins and toolsets over and over again you could benefit from writing your own bundle.

- Test2::Require::*

  This namespace contains modules that will cause a test to skip if specific conditions are not met. Use this if you have tests that only run on specific perl versions, or require external libraries that may not always be available.

LISTING DEPENDENCIES

When you use *Test2*, specifically things included in *Test2::Suite* you need to list them in your modules test dependencies. It is important to note that you should list the tools/plugins/bundles you need, you should not simply list *Test2::Suite* as your dependency. *Test2::Suite* is a living distribution intended to represent the "current" best practices. As tools, plugins, and bundles evolve, old ones will become discouraged and potentially be moved from *Test2::Suite* into their own distributions.

One goal of *Test2::Suite* is to avoid breaking backwards compatibility. Another goal is to always improve by replacing bad designs with better ones. When necessary *Test2::Suite* will break old modules out into separate dists and define new ones, typically with a new bundle. In short, if we feel the need to break something we will do so by creating a new bundle, and discouraging the old one, but we will not break the old one.

So for example, if you use *Test2::V1*, and Dist::Zilla you should have this in your config:

```
[Prereqs / TestRequires]
Test2::V1 = 0.000060
```

You **SHOULD NOT** do this:

```
[Prereqs / TestRequires]
Test2::Suite = 0.000060
```

Because *Test2::V1* might not always be part of *Test2::Suite*.

When writing new tests you should often check *Test2::Suite* to see what the current recommended bundle is.

Dist::Zilla

```
[Prereqs / TestRequires]
Test2::V1 = 0.000060
```

ExtUtils::MakeMaker

```
my %WriteMakefileArgs = (
  ...,
  "TEST_REQUIRES" => {
    "Test2::V1" => "0.000060"
  },
  ...
);
```

Module::Install

```
test_requires 'Test2::V1' => '0.000060';
```

Module::Build

```
my $build = Module::Build->new(
    ...,
    test_requires => {
        "Test2::V1" => "0.000060",
    },
    ...
);
```

TUTORIALS

SIMPLE/INTRODUCTION TUTORIAL

Test2::Manual::Testing::Introduction is an introduction to writing tests using the *Test2* tools.

MIGRATING FROM TEST::BUILDER and TEST::MORE

Test2::Manual::Testing::Migrating Is a tutorial for converting old tests that use *Test::Builder* or *Test::More* to the newer *Test2* way of doing things.

ADVANCED PLANNING

Test2::Manual::Testing::Planning is a tutorial on the many ways to set a plan.

TODO TESTS

Test2::Manual::Testing::Todo is a tutorial for markings tests as TODO.

SUBTESTS

COMING SOON.

COMPARISONS

COMING SOON.

SIMPLE COMPARISONS

COMING SOON.

ADVANCED COMPARISONS

COMING SOON.

TESTING EXPORTERS

COMING SOON.

TESTING CLASSES

COMING SOON.

TRAPPING

COMING SOON.

TRAPPING EXCEPTIONS

COMING SOON.

TRAPPING WARNINGS

COMING SOON.

DEFERRED TESTING

COMING SOON.

MANAGING ENCODINGS

COMING SOON.

AUTO-ABORT ON FAILURE

COMING SOON.

CONTROLLING RANDOM BEHAVIOR

COMING SOON.

WRITING YOUR OWN BUNDLE

COMING SOON.

TOOLSET DOCUMENTATION

COMING SOON.

PLUGIN DOCUMENTATION

COMING SOON.

BUNDLE DOCUMENTATION

COMING SOON.

REQUIRE DOCUMENTATION

COMING SOON.

SEE ALSO

Test2::Manual - Primary index of the manual.

SOURCE

The source code repository for Test2-Manual can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Modules

- `Test2::Manual::Testing::Introduction` - Introduction to testing with Test2.
- `Test2::Manual::Testing::Migrating` - How to migrate existing tests from Test::More to Test2.
- `Test2::Manual::Testing::Planning` - The many ways to set a plan.
- `Test2::Manual::Testing::Todo` - Tutorial for marking tests as TODO.

Test2::Manual::Testing::Introduction

Introduction to testing with Test2.

DESCRIPTION

This tutorial is a beginners introduction to testing. This will take you through writing a test file, making assertions, and running your test.

BOILERPLATE

THE TEST FILE

Test files typically are placed inside the `t/` directory, and end with the `.t` file extension.

`t/example.t`:

```
use Test2::V1 -ipP;

# Assertions will go here

done_testing;
```

This is all the boilerplate you need.

- use Test2::V1 -ipP;

  This loads a collection of testing tools that will be described later in the tutorial. See *Test2::V1* for more details, but for starters '-ipP' is a good set of import flags.

  If you do not like importing a ton of symbols or enabling pragmas/plugins all in one swoop you can do `use Test2::V1;`. If you do this you will need to use the T2() function to access tools, and load any pragmas/plugins manually:

  ```
  use Test2::V1;
  use strict;
  use warnings;
  use Test2::Plugin::UTF8;
  use Test2::Plugin::SRand;

  T2->ok(1, "pass");
  T2->is(5, 5, "5 is 5");

  T2->done_testing;
  ```

- done_testing;

  This should always be at the end of your test files. This tells *Test2* that you are done making assertions. This is important as `test2` will assume the test did not complete successfully without this, or some other form of test "plan".

DIST CONFIG

You should always list bundles and tools directly. You should not simply list *Test2::Suite* and call it done, bundles and tools may be moved out of *Test2::Suite* to their own dists at any time.

Dist::Zilla

```
[Prereqs / TestRequires]
Test2::V1 = 0.000060
```

ExtUtils::MakeMaker

```
my %WriteMakefileArgs = (
  ...,
  "TEST_REQUIRES" => {
    "Test2::V1" => "0.000060"
  },
  ...
);
```

Module::Install

```
test_requires 'Test2::V1' => '0.000060';
```

Module::Build

```
my $build = Module::Build->new(
    ...,
    test_requires => {
        "Test2::V1" => "0.000060",
    },
    ...
);
```

MAKING ASSERTIONS

The most simple tool for making assertions is `ok()`. `ok()` lets you assert that a condition is true.

```
ok($CONDITION, "Description of the condition");
```

Here is a complete `t/example.t`:

```
use Test2::V1 -import;

ok(1, "1 is true, so this will pass");

done_testing;
```

If you are doing it without imports:

```
use Test2::V1;

T2->ok(1, "1 is true, so this will pass");

T2->done_testing;
```

RUNNING THE TEST

Test files are simply scripts. Just like any other script you can run the test directly with perl. Another option is to use a test "harness" which runs the test for you, and provides extra information and checks the scripts exit value for you.

RUN DIRECTLY

```
$ perl -Ilib t/example.t
```

Which should produce output like this:

```
# Seeded srand with seed '20161028' from local date.
ok 1 - 1 is true, so this will pass
1..1
```

If the test had failed (`ok(0, ...)`) it would look like this:

```
# Seeded srand with seed '20161028' from local date.
not ok 1 - 0 is false, so this will fail
1..1
```

Test2 will also set the exit value of the script, a successful run will have an exit value of 0, a failed run will have a non-zero exit value.

USING YATH

The `yath` command line tool is provided by Test2::Harness which you may need to install yourself from cpan. `yath` is the harness written specifically for *Test2*.

```
$ yath -Ilib t/example.t
```

This will produce output similar to this:

```
( PASSED )  job  1    t/example.t

================================================================================

Run ID: 1508027909

All tests were successful!
```

You can also request verbose output with the `-v` flag:

```
$ yath -Ilib -v t/example.t
```

Which produces:

```
( LAUNCH )  job  1    example.t
(  NOTE  )  job  1    Seeded srand with seed '20171014' from local␣
 ↪date.
[  PASS  ]  job  1  + 1 is true, so this will pass
[  PLAN  ]  job  1    Expected assertions: 1
( PASSED )  job  1    example.t

================================================================================

Run ID: 1508028002

All tests were successful!
```

USING PROVE

The `prove` command line tool is provided by the *Test::Harness* module which comes with most versions of perl. *Test::Harness* is dual-life, which means you can also install the latest version from cpan.

```
$ prove -Ilib t/example.t
```

This will produce output like this:

```
example.t .. ok
All tests successful.
Files=1, Tests=1,  0 wallclock secs ( 0.01 usr  0.00 sys +  0.05␣
 ↪cusr  0.00 csys =  0.06 CPU)
Result: PASS
```

You can also request verbose output with the `-v` flag:

```
$ prove -Ilib -v t/example.t
```

The verbose output looks like this:

```
example.t ..
# Seeded srand with seed '20161028' from local date.
```

(continues on next page)

(continued from previous page)

```
ok 1 - 1 is true, so this will pass
1..1
ok
All tests successful.
Files=1, Tests=1,  0 wallclock secs ( 0.02 usr  0.00 sys +  0.06
↪cusr  0.00 csys =  0.08 CPU)
Result: PASS
```

THE "PLAN"

All tests need a "plan". The job of a plan is to make sure you ran all the tests you expected. The plan prevents a passing result from a test that exits before all the tests are run.

There are 2 primary ways to set the plan:

- done_testing()
- T2->done_testing()

  The most common, and recommended way to set a plan is to add `done_testing` at the end of your test file. This will automatically calculate the plan for you at the end of the test. If the test were to exit early then `done_testing` would not run and no plan would be found, forcing a failure.

- plan($COUNT)
- T2->plan($COUNT)

  The `plan()` function allows you to specify an exact number of assertions you want to run. If you run too many or too few assertions then the plan will not match and it will be counted as a failure. The primary problem with this way of planning is that you need to add up the number of assertions, and adjust the count whenever you update the test file.

  `plan()` must be used before all assertions, or after all assertions, it cannot be done in the middle of making assertions.

ADDITIONAL ASSERTION TOOLS

The *Test2::V1* bundle provides a lot more than `ok()`, `plan()`, and `done_testing()`. The biggest tools to note are:

- is($a, $b, $description)
- T2->is($a, $b, $description)

  `is()` allows you to compare 2 structures and insure they are identical. You can use it for simple string comparisons, or even deep data structure comparisons.

  ```
  is("foo", "foo", "Both strings are identical");

  is(["foo", 1], ["foo", 1], "Both arrays contain the same
  ↪elements");
  ```

- like($a, $b, $description)

- T2->like($a, $b, $description)

  `like()` is similar to `is()` except that it only checks items listed on the right, it ignores any extra values found on the left.

```
like([1, 2, 3, 4], [1, 2, 3], "Passes, the extra element on the␣
↪left is ignored");
```

  You can also used regular expressions on the right hand side:

```
like("foo bar baz", qr/bar/, "The string matches the regex,␣
↪this passes");
```

  You can also nest the regexes:

```
like([1, 2, 'foo bar baz', 3], [1, 2, qr/bar/], "This passes");
```

SEE ALSO

Test2::Manual - Primary index of the manual.

SOURCE

The source code repository for Test2-Manual can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Manual::Testing::Migrating

How to migrate existing tests from Test::More to Test2.

DESCRIPTION

This tutorial covers the conversion of an existing test. This tutorial assumes you have a test written using *Test::More*.

LEGACY TEST

This tutorial will be converting this example test one section at a time:

`t/example.t`:

```
#####################
# Boilerplate

use strict;
use warnings;

use Test::More tests => 14;

use_ok 'Scalar::Util';
require_ok 'Exporter';

####################
# Simple assertions (no changes)

ok(1, "pass");

is("apple", "apple", "Simple string compare");

like("foo bar baz", qr/bar/, "Regex match");

####################
# Todo

{
    local $TODO = "These are todo";

    ok(0, "oops");
}

####################
# Deep comparisons

is_deeply([1, 2, 3], [1, 2, 3], "Deep comparison");

####################
# Comparing references

my $ref = [1];
is($ref, $ref, "Check that we have the same ref both times");
```

(continues on next page)

(continued from previous page)

```
####################
# Things that are gone

ok(eq_array([1], [1]), "array comparison");
ok(eq_hash({a => 1}, {a => 1}), "hash comparison");
ok(eq_set([1, 3, 2], [1, 2, 3]), "set comparison");

note explain([1, 2, 3]);

{
    package THING;
    sub new { bless({}, shift) }
}

my $thing = new_ok('THING');

####################
# Tools that changed

isa_ok($thing, 'THING', '$thing');

can_ok(__PACKAGE__, qw/ok is/);
```

BOILERPLATE

BEFORE:

```
use strict;
use warnings;

use Test::More tests => 14;

use_ok 'Scalar::Util';
require_ok 'Exporter';
```

AFTER:

```
use strict;
use warnings;

use Test2::V1 '-import';
plan(11);

use Scalar::Util;
require Exporter;
```

- Replace Test::More with Test2::V1

  Test2::V1 is the recommended bundle. In a full migration you will want to replace *Test::More* with the *Test2::V1* bundle.

Note: You should always double check the latest *Test2* to see if there is a new recommended bundle. When writing a new test you should always use the newest Test::V# module. Higher numbers are newer version.

You probably want the `-import` argument when using *Test2::V0* as it will populate your namespace with all the tools you expect from a test helper module. However if you want your namespace left clean you can omit the argument, in which case `T2()` is the only thing added to your namespace, and it can be used to access the tools:

```
use Test2::V1;

T2->ok(1, "pass");

T2->done_testing;
```

- NOTE: srand

  When srand is on (not default in V1, but Default in older V0) it can cause problems with things like *File::Temp* which will end up attempting the same "random" filenames for every test process started on a given day (or sharing the same seed).

  If this is a problem for you then please disable srand when loading

  For *Test2::V0*:

  ```
  use Test2::V0 -no_srand => 1;
  ```

  For *Test2::V1* simply do not use the `-P`, or `-Plugins` import option and it will not be loaded.

- Stop using use_ok()

  `use_ok()` has been removed. a `use MODULE` statement will throw an exception on failure anyway preventing the test from passing.

  If you *REALLY* want/need to assert that the file loaded you can use the *ok* module:

  ```
  use ok 'Scalar::Util';
  ```

  The main difference here is that there is a space instead of an underscore.

- Stop using require_ok()

  `require_ok` has been removed just like `use_ok`. There is no *ok* module equivalent here. Just use `require`.

- (optional) remove strict/warnings

  In the *Test2::V0* bundle turns strict and warnings on for you.

  In the *Test2::V1* bundle you must ask for strict and warnings with one of the following import args: `-p`, `-pragmas`, `-strict`, `-warnings`.

- Change where the plan is set

  Test2 does not allow you to set the plan at import. In the old code you would pass `tests => 11` as an import argument. In *Test2* you either need to use the `plan()` function to set the plan, or use `done_testing()` at the end of the test.

If your test already uses `done_testing()` you can keep that and no plan changes are necessary.

Note: We are also changing the plan from 14 to 11, that is because we dropped `use_ok`, `require_ok`, and we will be dropping one more later on. This is why `done_testing()` is recommended over a set plan.

SIMPLE ASSERTIONS

The vast majority of assertions will not need any changes:

```
####################
# Simple assertions (no changes)

ok(1, "pass");

is("apple", "apple", "Simple string compare");

like("foo bar baz", qr/bar/, "Regex match");
```

TODO

```
{
    local $TODO = "These are todo";

    ok(0, "oops");
}
```

The `$TODO` package variable is gone. You now have a `todo()` function.

There are 2 ways this can be used:

- todo $reason => sub { ... }

  ```
  todo "These are todo" => sub {
      ok(0, "oops");
  };
  ```

  This is the cleanest way to do a todo. This will make all assertions inside the code-block into TODO assertions.

- { my $TODO = todo $reason; ... }

  ```
  {
      my $TODO = todo "These are todo";

      ok(0, "oops");
  }
  ```

  This is a system that emulates the old way. Instead of modifying a global `$TODO` variable you create a todo object with the `todo()` function and assign it to a lexical variable. Once the todo object falls out of scope the TODO ends.

DEEP COMPARISONS

```
is_deeply([1, 2, 3], [1, 2, 3], "Deep comparison");
```

Deep comparisons are easy, simply replace `is_deeply()` with `is()`.

```
is([1, 2, 3], [1, 2, 3], "Deep comparison");
```

COMPARING REFERENCES

```
my $ref = [1];
is($ref, $ref, "Check that we have the same ref both times");
```

The `is()` function provided by *Test::More* forces both arguments into strings, which makes this a comparison of the reference addresses. *Test2*'s `is()` function is a deep comparison, so this will still pass, but fails to actually test what we want (that both references are the same exact ref, not just identical structures.)

We now have the `ref_is()` function that does what we really want, it ensures both references are the same reference. This function does the job better than the original, which could be thrown off by string overloading.

```
my $ref = [1];
ref_is($ref, $ref, "Check that we have the same ref both times");
```

TOOLS THAT ARE GONE

```
ok(eq_array([1], [1]), "array comparison");
ok(eq_hash({a => 1}, {a => 1}), "hash comparison");
ok(eq_set([1, 3, 2], [1, 2, 3]), "set comparison");

note explain([1, 2, 3]);

{
    package THING;
    sub new { bless({}, shift) }
}

my $thing = new_ok('THING');
```

`eq_array`, `eq_hash` and `eq_set` have been considered deprecated for a very long time, *Test2* does not provide them at all. Instead you can just use `is()`:

```
is([1], [1], "array comparison");
is({a => 1}, {a => 1}, "hash comparison");
```

`eq_set` is a tad more complicated, see *Test2::Tools::Compare* for an explanation:

```
is([1, 3, 2], bag { item 1; item 2; item 3; end }, "set comparison
↪");
```

`explain()` has a rocky history. There have been arguments about how it should work. *Test2* decided to simply not include `explain()` to avoid the arguments. You can instead directly use Data::Dumper:

```
use Data::Dumper;
note Dumper([1, 2, 3]);
```

`new_ok()` is gone. The implementation was complicated, and did not add much value:

```
{
    package THING;
    sub new { bless({}, shift) }
}

my $thing = THING->new;
ok($thing, "made a new thing");
```

The complete section after the conversion is:

```
is([1], [1], "array comparison");
is({a => 1}, {a => 1}, "hash comparison");
is([1, 3, 2], bag { item 1; item 2; item 3; end }, "set comparison
 ↪");

use Data::Dumper;
note Dumper([1, 2, 3]);

{
    package THING;
    sub new { bless({}, shift) }
}

my $thing = THING->new;
ok($thing, "made a new thing");
```

TOOLS THAT HAVE CHANGED

```
isa_ok($thing, 'THING', '$thing');

can_ok(__PACKAGE__, qw/ok is/);
```

In *Test::More* these functions are very confusing, and most people use them wrong!

`isa_ok()` from *Test::More* takes a thing, a class/reftype to check, and then uses the third argument as an alternative display name for the first argument (NOT a test name!).

`can_ok()` from *Test::More* is not consistent with `isa_ok` as all arguments after the first are subroutine names.

Test2 fixes this by making both functions consistent and obvious:

```
isa_ok($thing, ['THING'], 'got a THING');

can_ok(__PACKAGE__, [qw/ok is/], "have expected subs");
```

You will note that both functions take a thing, an arrayref as the second argument, then a test name as the third argument.

FINAL VERSION

IMPORTS

```
######################
# Boilerplate

use strict;
use warnings;
use Test2::V1 '-import';
plan(11);

use Scalar::Util;
require Exporter;

######################
# Simple assertions (no changes)

ok(1, "pass");

is("apple", "apple", "Simple string compare");

like("foo bar baz", qr/bar/, "Regex match");

######################
# Todo

todo "These are todo" => sub {
    ok(0, "oops");
};

######################
# Deep comparisons

is([1, 2, 3], [1, 2, 3], "Deep comparison");

######################
# Comparing references

my $ref = [1];
ref_is($ref, $ref, "Check that we have the same ref both times");

######################
```

(continues on next page)

(continued from previous page)

```
# Things that are gone

is([1], [1], "array comparison");
is({a => 1}, {a => 1}, "hash comparison");

is([1, 3, 2], bag { item 1; item 2; item 3; end }, "set comparison
 ↪");

use Data::Dumper;
note Dumper([1, 2, 3]);

{
    package THING;
    sub new { bless({}, shift) }
}

my $thing = THING->new;

######################
# Tools that changed

isa_ok($thing, ['THING'], 'got a THING');

can_ok(__PACKAGE__, [qw/ok is/], "have expected subs");
```

REDUCED BOILERPLATE

```
######################
# Boilerplate

use Test2::V1 '-ipP';
plan(11);

use Scalar::Util;
require Exporter;

######################
# Simple assertions (no changes)

ok(1, "pass");

is("apple", "apple", "Simple string compare");

like("foo bar baz", qr/bar/, "Regex match");

######################
# Todo
```

(continues on next page)

(continued from previous page)

```
todo "These are todo" => sub {
    ok(0, "oops");
};

#####################
# Deep comparisons

is([1, 2, 3], [1, 2, 3], "Deep comparison");

#####################
# Comparing references

my $ref = [1];
ref_is($ref, $ref, "Check that we have the same ref both times");

#####################
# Things that are gone

is([1], [1], "array comparison");
is({a => 1}, {a => 1}, "hash comparison");

is([1, 3, 2], bag { item 1; item 2; item 3; end }, "set comparison
 ↪");

use Data::Dumper;
note Dumper([1, 2, 3]);

{
    package THING;
    sub new { bless({}, shift) }
}

my $thing = THING->new;

#####################
# Tools that changed

isa_ok($thing, ['THING'], 'got a THING');

can_ok(__PACKAGE__, [qw/ok is/], "have expected subs");
```

CLEAN NAMESPACE

```
use Test2::V1;
T2->plan(11);

use Scalar::Util;
require Exporter;
```

(continues on next page)

(continued from previous page)

```
####################
# Simple assertions (no changes)

T2->ok(1, "pass");

T2->is("apple", "apple", "Simple string compare");

T2->like("foo bar baz", qr/bar/, "Regex match");

####################
# Todo

T2->todo("These are todo" => sub {
    ok(0, "oops");
});

####################
# Deep comparisons

T2->is([1, 2, 3], [1, 2, 3], "Deep comparison");

####################
# Comparing references

my $ref = [1];
T2->ref_is($ref, $ref, "Check that we have the same ref both times
 ↪");

####################
# Things that are gone

T2->is([1], [1], "array comparison");
T2->is({a => 1}, {a => 1}, "hash comparison");

T2->is([1, 3, 2], bag { item 1; item 2; item 3; end }, "set␣
 ↪comparison");

use Data::Dumper;
T2->note(Dumper([1, 2, 3]));

{
    package THING;
    sub new { bless({}, shift) }
}

my $thing = THING->new;

####################
```

(continues on next page)

(continued from previous page)

```
# Tools that changed

T2->isa_ok($thing, ['THING'], 'got a THING');

T2->can_ok(__PACKAGE__, [qw/ok is/], "have expected subs");
```

SEE ALSO

Test2::Manual - Primary index of the manual.

SOURCE

The source code repository for Test2-Manual can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Manual::Testing::Planning

The many ways to set a plan.

DESCRIPTION

This tutorial covers the many ways of setting a plan.

TEST COUNT

The `plan()` function is provided by *Test2::Tools::Basic*. This function lets you specify an exact number of tests to run. This can be done at the start of testing, or at the end. This cannot be done partway through testing.

```
use Test2::Tools::Basic;
plan(10); # 10 tests expected

...
```

DONE TESTING

The done_testing() function is provided by *Test2::Tools::Basic*. This function will automatically set the plan to the number of tests that were run. This must be used at the very end of testing.

```
use Test2::Tools::Basic;

...

done_testing();
```

SKIP ALL

The skip_all() function is provided by *Test2::Tools::Basic*. This function will set the plan to 0, and exit the test immediately. You may provide a skip reason that explains why the test should be skipped.

```
use Test2::Tools::Basic;
skip_all("This test will not run here") if ...;

...
```

CUSTOM PLAN EVENT

A plan is simply an *Test2::Event::Plan* event that gets sent to the current hub. You could always write your own tool to set the plan.

```
use Test2::API qw/context/;

sub set_plan {
    my $count = @_;

    my $ctx = context();
    $ctx->send_event('Plan', max => $count);
    $ctx->release;

    return $count;
}
```

SEE ALSO

Test2::Manual - Primary index of the manual.

SOURCE

The source code repository for Test2-Manual can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Manual::Testing::Todo

Tutorial for marking tests as TODO.

DESCRIPTION

This tutorial covers the process of marking tests as TODO. It also describes how TODO works under the hood.

THE TOOL

```
use Test2::Tools::Basic qw/todo/;
```

TODO BLOCK

This form is low-magic. All tests inside the block are marked as todo, tests outside the block are not todo. You do not need to do any variable management. The flaw with this form is that it adds a couple levels to the stack, which can break some high-magic tests.

Overall this is the preferred form unless you have a special case that requires the variable form.

```
todo "Reason for the todo" => sub {
    ok(0, "fail but todo");
    ...
};
```

TODO VARIABLE

This form maintains the todo scope for the life of the variable. This is useful for tests that are sensitive to scope changes. This closely emulates the *Test::More* style which localized the `$TODO` package variable. Once the variable is destroyed (set it to undef, scope end, etc) the TODO state ends.

```
my $todo = todo "Reason for the todo";
ok(0, "fail but todo");
...
$todo = undef;
```

MANUAL TODO EVENTS

```
use Test2::API qw/context/;

sub todo_ok {
    my ($bool, $name, $todo) = @_;

    my $ctx = context();
    $ctx->send_event('Ok', pass => $bool, effective_pass => 1, todo
 ↪=> $todo);
    $ctx->release;

    return $bool;
}
```

The *Test2::Event::Ok* event has a `todo` field which should have the todo reason. The event also has the `pass` and `effective_pass` fields. The `pass` field is the actual pass/fail value. The `effective_pass` is used to determine if the event is an actual failure (should always be set tot true with todo).

HOW THE TODO TOOLS WORK UNDER THE HOOD

The *Test2::Todo* library gets the current *Test2::Hub* instance and adds a filter. The filter that is added will set the `todo` and `effective_pass` fields on any *Test2::Event::Ok* events that pass through the hub. The filter also converts *Test2::Event::Diag* events into *Test2::Event::Note* events.

SEE ALSO

Test2::Manual - Primary index of the manual.

SOURCE

The source code repository for Test2-Manual can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Manual::Tooling

Manual page for tool authors.

DESCRIPTION

This section covers writing new tools, plugins, and other Test2 components.

TOOL TUTORIALS

FIRST TOOL

Test2::Manual::Tooling::FirstTool - Introduction to writing tools by cloning ok().

MOVING FROM Test::Builder

Test2::Manual::Tooling::TestBuilder - This section maps Test::Builder methods to Test2 concepts.

NESTING TOOLS

Test2::Manual::Tooling::Nesting - How to call other tools from your tool.

TOOLS WITH SUBTESTS

Test2::Manual::Tooling::Subtest - How write tools that make use of subtests.

TESTING YOUR TEST TOOLS

Test2::Manual::Tooling::Testing - How to write tests for your test tools.

PLUGIN TUTORIALS

TAKING ACTION WHEN A NEW TOOL STARTS

Test2::Manual::Tooling::Plugin::ToolStarts - How to add behaviors that occur when a tool starts work.

TAKING ACTION AFTER A TOOL IS DONE

Test2::Manual::Tooling::Plugin::ToolCompletes - How to add behaviors that occur when a tool completes work.

TAKING ACTION AT THE END OF TESTING

Test2::Manual::Tooling::Plugin::TestingDone - How to add behaviors that occur when testing is complete (IE done_testing, or end of test).

TAKING ACTION JUST BEFORE EXIT

Test2::Manual::Tooling::Plugin::TestExit - How to safely add pre-exit behaviors.

WRITING A SIMPLE JSONL FORMATTER

Test2::Manual::Tooling::Formatter - How to write a custom formatter, in our case a JSONL formatter.

WHERE TO FIND HOOKS AND APIS

- global API

  Test2::API is the global API. This is primarily used by plugins that provide global behavior.

- In hubs

  Test2::Hub is the base class for all hubs. This is where hooks for manipulating events, or running things at the end of testing live.

SEE ALSO

Test2::Manual - Primary index of the manual.

SOURCE

The source code repository for Test2-Manual can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Modules

- `Test2::Manual::Tooling::FirstTool` - Write your first tool with Test2.
- `Test2::Manual::Tooling::Formatter` - How to write a custom formatter, in our case a JSONL formatter.
- `Test2::Manual::Tooling::Nesting` - Tutorial for using other tools within your own.
- `Test2::Manual::Tooling::Plugin`
- `Test2::Manual::Tooling::Subtest` - How to implement a tool that makes use of subtests.
- `Test2::Manual::Tooling::TestBuilder` - This section maps Test::Builder methods to Test2 concepts.
- `Test2::Manual::Tooling::Testing` - Tutorial on how to test your testing tools.

Test2::Manual::Tooling::FirstTool

Write your first tool with Test2.

DESCRIPTION

This tutorial will help you write your very first tool by cloning the `ok()` tool.

COMPLETE CODE UP FRONT

```
package Test2::Tools::MyOk;
use strict;
use warnings;

use Test2::API qw/context/;

```

(continues on next page)

(continued from previous page)

```
use base 'Exporter';
our @EXPORT = qw/ok/;

sub ok($;$@) {
    my ($bool, $name, @diag) = @_;

    my $ctx = context();

    return $ctx->pass_and_release($name) if $bool;
    return $ctx->fail_and_release($name, @diag);
}

1;
```

LINE BY LINE

- sub ok($;$@) {

  In this case we are emulating the `ok()` function exported by *Test2::Tools::Basic*.

  `ok()` and similar test tools use prototypes to enforce argument parsing. Your test tools do not necessarily need prototypes, like any perl function you need to make the decision based on how it is used.

  The prototype requires at least 1 argument, which will be forced into a scalar context. The second argument is optional, and is also forced to be scalar, it is the name of the test. Any remaining arguments are treated as diagnostics messages that will only be used if the test failed.

- my ($bool, $name, @diag) = @_;

  This line does not need much explanation, we are simply grabbing the args.

- my $ctx = context();

  This is a vital line in **ALL** tools. The context object is the primary API for test tools. You **MUST** get a context if you want to issue any events, such as making assertions. Further, the context is responsible for making sure failures are attributed to the correct file and line number.

  Note: A test function **MUST** always release the context when it is done, you cannot simply let it fall out of scope and be garbage collected. Test2 does a pretty good job of yelling at you if you make this mistake.

  Note: You **MUST NOT** ever store or pass around a *real* context object. If you wish to hold on to a context for any reason you must use clone to make a copy `my $copy = $ctx->clone`. The copy may be passed around or stored, but the original **MUST** be released when you are done with it.

- return $ctx->pass_and_release($name) if $bool;

  When `$bool` is true, this line uses the context object to issue a *Test2::Event::Pass* event. Along with issuing the event this will also release the context object and return true.

This is short form for:

```
if($bool) {
    $ctx->pass($name);
    $ctx->release;
    return 1;
}
```

- return $ctx->fail_and_release($name, @diag);

  This line issues a *Test2::Event::Fail* event, releases the context object, and returns false. The fail event will include any diagnostics messages from the `@diag` array.

  This is short form for:

```
$ctx->fail($name, @diag);
$ctx->release;
return 0;
```

CONTEXT OBJECT DOCUMENTATION

Test2::API::Context is the place to read up on what methods the context provides.

SEE ALSO

Test2::Manual - Primary index of the manual.

SOURCE

The source code repository for Test2-Manual can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Manual::Tooling::Formatter

How to write a custom formatter, in our case a JSONL formatter.

DESCRIPTION

This tutorial explains a minimal formatter that outputs each event as a json string on its own line. A true formatter will probably be significantly more complicated, but this will give you the basics needed to get started.

COMPLETE CODE UP FRONT

```
package Test2::Formatter::MyFormatter;
use strict;
use warnings;

use JSON::MaybeXS qw/encode_json/;

use base qw/Test2::Formatter/;

sub new { bless {}, shift }

sub encoding {};

sub write {
    my ($self, $e, $num, $f) = @_;
    $f ||= $e->facet_data;

    print encode_json($f), "\n";
}

1;
```

LINE BY LINE

- use base qw/Test2::Formatter/;

  All formatters should inherit from *Test2::Formatter*.

- sub new { bless {}, shift }

  Formatters need to be instantiable objects, this is a minimal `new()` method.

- sub encoding {};

  For this example we leave this sub empty. In general you should implement this sub to make sure you honor situations where the encoding is set. *Test2::V0* itself will try to set the encoding to UTF8.

- sub write { ... }

  The `write()` method is the most important, each event is sent here.

- my ($self, $e, $num, $f) = @_;

The `write()` method receives 3 or 4 arguments, the fourth is optional.

- $self

  The formatter itself.

- $e

  The event being written

- $num

  The most recent assertion number. If the event being processed is an assertion then this will have been bumped by 1 since the last call to write. For non assertions this number is set to the most recent assertion.

- $f

  This MAY be a hashref containing all the facet data from the event. More often then not this will be undefined. This is only set if the facet data was needed by the hub, and it usually is not.

- $f ||= $e->facet_data;

  We want to dump the event facet data. This will set `$f` to the facet data unless we already have the facet data.

- print encode_json($f), "\n";

  This line prints the JSON encoded facet data, and a newline.

SEE ALSO

Test2::Manual - Primary index of the manual.

SOURCE

The source code repository for Test2-Manual can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Manual::Tooling::Nesting

Tutorial for using other tools within your own.

DESCRIPTION

Sometimes you find yourself writing the same test pattern over and over, in such cases you may want to encapsulate the logic in a new test function that calls several tools together. This sounds easy enough, but can cause headaches if not done correctly.

NAIVE WAY

Lets say you find yourself writing the same test pattern over and over for multiple objects:

```
my $obj1 = $class1->new;
is($obj1->foo, 'foo', "got foo");
is($obj1->bar, 'bar', "got bar");

my $obj2 = $class1->new;
is($obj2->foo, 'foo', "got foo");
is($obj2->bar, 'bar', "got bar");

... 10x more times for classes 2-12
```

The naive way to do this is to write a `check_class()` function like this:

```
sub check_class {
    my $class = shift;
    my $obj = $class->new;
    is($obj->foo, 'foo', "got foo");
    is($obj->bar, 'bar', "got bar");
}

check_class($class1);
check_class($class2);
check_class($class3);
...
```

This will appear to work fine, and you might not notice any problems, *so long as the tests are passing.*

WHAT'S WRONG WITH IT?

The problems with the naive approach become obvious if things start to fail. The diagnostics that tell you what file and line the failure occurred on will be wrong. The failure will be reported to the line *inside* `check_class`, not to the line where `check_class()` was called. This is problem because it leaves you with no idea which class is failing.

HOW TO FIX IT

Luckily this is extremely easy to fix. You need to acquire a context object at the start of your function, and release it at the end... yes it is that simple.

```
use Test2::API qw/context/;

sub check_class {
    my $class = shift;

    my $ctx = context();

    my $obj = $class->new;
    is($obj->foo, 'foo', "got foo");
    is($obj->bar, 'bar', "got bar");

    $ctx->release;
}
```

See, that was easy. With these 2 additional lines we know have proper file+line reporting. The nested tools will find the context we acquired here, and know to use its file and line numbers.

THE OLD WAY (DO NOT DO THIS ANYMORE)

With *Test::Builder* there was a global variables called `$Test::Builder::Level` which helped solve this problem:

```
sub check_class {
    my $class = shift;

    local $Test::Builder::Level = $Test::Builder::Level + 1;

    my $obj = $class->new;
    is($obj->foo, 'foo', "got foo");
    is($obj->bar, 'bar', "got bar");
}
```

This variable worked well enough (and will still work) but was not very discoverable. Another problem with this variable is that it becomes cumbersome if you have a more deeply nested code structure called the nested tools, you might need to count stack frames, and hope they never change due to a third party module. The context solution has no such caveats.

SEE ALSO

Test2::Manual - Primary index of the manual.

SOURCE

The source code repository for Test2-Manual can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Manual::Tooling::Plugin

Namespace containing the following modules:

- `Test2::Manual::Tooling::Plugin::TestExit` - How to safely add pre-exit behaviors.
- `Test2::Manual::Tooling::Plugin::TestingDone` - Run code when the test file is finished, or when done_testing is called.
- `Test2::Manual::Tooling::Plugin::ToolCompletes` - How to add behaviors that occur when a tool completes work.
- `Test2::Manual::Tooling::Plugin::ToolStarts` - How to add behaviors that occur when a tool starts work.

Test2::Manual::Tooling::Plugin::TestExit

How to safely add pre-exit behaviors.

DESCRIPTION

This describes the correct/safe way to add pre-exit behaviors to tests via a custom plugin.

The naive way to attempt this would be to add an `END { ... }` block. That can work, and may not cause problems.... On the other hand there are a lot of ways that can bite you. Describing all the potential problems of an END block, and how it might conflict with Test2 (Which has its own END block) is beyond the scope of this document.

COMPLETE CODE UP FRONT

```
package Test2::Plugin::MyPlugin;

use Test2::API qw{test2_add_callback_exit};

sub import {
    my $class = shift;

    test2_add_callback_exit(sub {
        my ($ctx, $orig_code, $new_exit_code_ref) = @_;

        return if $orig_code == 42;

        $$new_exit_code_ref = 42;
    });
}

1;
```

LINE BY LINE

- use Test2::API qw{test2_add_callback_exit};

  This imports the `(test2_add_callback_exit)` callback.

- test2_add_callback_exit(sub { ... });

  This adds our callback to be called before exiting.

- my ($ctx, $orig_code, $new_exit_code_ref) = @_

  The callback gets 3 arguments. First is a context object you may use. The second is the original exit code of the END block Test2 is using. The third argument is a scalar reference which you may use to get the current exit code, or set a new one.

- return if $orig_code == 42

  This is a short-cut to do nothing if the original exit code was already 42.

- $$new_exit_code_ref = 42

  This changes the exit code to 42.

SEE ALSO

Test2::Manual - Primary index of the manual.

SOURCE

The source code repository for Test2-Manual can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Manual::Tooling::Plugin::TestingDone

Run code when the test file is finished, or when done_testing is called.

DESCRIPTION

This is a way to add behavior to the end of a test file. This code is run either when done_testing() is called, or when the test file has no more run-time code to run.

When triggered by done_testing() this will be run BEFORE the plan is calculated and sent. This means it IS safe to make test assertions in this callback.

COMPLETE CODE UP FRONT

```
package Test2::Plugin::MyPlugin;

use Test2::API qw{test2_add_callback_testing_done};

sub import {
    my $class = shift;

    test2_add_callback_testing_done(sub {
        ok(!$some_global, '$some_global was not set');
        print "The test file is done, or done_testing was just
 ↪called\n"
    });
}

1;
```

LINE BY LINE

- use Test2::API qw{test2_add_callback_testing_done};

  This imports the `test2_add_callback_testing_done()` callback.

- test2_add_callback_testing_done(sub { ... });

  This adds our callback to be called when testing is done.

- ok(!$some_global, '$some_global was not set')

  It is safe to make assertions in this type of callback. This code simply asserts that some global was never set over the course of the test.

- print "The test file is done, or done_testing was just called\n"

  This prints a message when the callback is run.

UNDER THE HOOD

Before test2_add_callback_testing_done() this kind of thing was still possible, but it was hard to get right, here is the code to do it:

```
test2_add_callback_post_load(sub {
    my $stack = test2_stack();

    # Insure we have at least one hub, but we do not necessarily␣
↪want the
    # one this returns.
    $stack->top;

    # We want the root hub, not the top one.
    my ($root) = Test2::API::test2_stack->all;

    # Make sure the hub does not believe nothing has happened.
    $root->set_active(1);

    # Now we can add our follow-up code
    $root->follow_up(sub {
        # Your callback code here
    });
});
```

SEE ALSO

Test2::Manual - Primary index of the manual.

SOURCE

The source code repository for Test2-Manual can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Manual::Tooling::Plugin::ToolCompletes

How to add behaviors that occur when a tool completes work.

DESCRIPTION

This tutorial helps you understand how to add behaviors that occur when a tool is done with its work. All tools need to acquire and then release a context, for this tutorial we make use of the release hooks that are called every time a tool releases the context object.

COMPLETE CODE UP FRONT

```
package Test2::Plugin::MyPlugin;

use Test2::API qw{test2_add_callback_context_release};

sub import {
    my $class = shift;

    test2_add_callback_context_release(sub {
        my $ctx_ref = shift;

        print "Context was released\n";
    });
}

1;
```

LINE BY LINE

- use Test2::API qw{test2_add_callback_context_release};

  This imports the `test2_add_callback_context_release()` callback.

- test2_add_callback_context_release(sub { ... })

- my $ctx_ref = shift

  The coderefs for test2_add_callback_context_release() will receive exactly 1 argument, the context being released.

- print "Context was released\n"

  Print a notification whenever the context is released.

SEE ALSO

Test2::Manual - Primary index of the manual.

SOURCE

The source code repository for Test2-Manual can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Manual::Tooling::Plugin::ToolStarts

How to add behaviors that occur when a tool starts work.

DESCRIPTION

This tutorial will help you write plugins that have behavior when a tool starts. All tools should start by acquiring a context object. This tutorial shows you the hooks you can use to take advantage of the context acquisition.

COMPLETE CODE UP FRONT

```
package Test2::Plugin::MyPlugin;

use Test2::API qw{
    test2_add_callback_context_init
```

(continues on next page)

(continued from previous page)

```
    test2_add_callback_context_acquire
};

sub import {
    my $class = shift;

    # Let us know every time a tool requests a context, and give us␣
↪a
    # chance to modify the parameters before we find it.
    test2_add_callback_context_acquire(sub {
        my $params_ref = shift;

        print "A tool has requested the context\n";
    });

    # Callback every time a new context is created, not called if an
    # existing context is found.
    test2_add_callback_context_init(sub {
        my $ctx_ref = shift;

        print "A new context was created\n";
    });
}

1;
```

LINE BY LINE

- use Test2::API qw{test2_add_callback_context_init test2_add_callback_context_acquire};

  This imports the `test2_add_callback_context_init()` and `test2_add_callback_context_acquire()` callbacks.

- test2_add_callback_context_acquire(sub { ... })

  This is where we add our callback for context acquisition. Every time `Test2::API::context()` is called the callback will be run.

- my $params_ref = shift

  In the test2_add_callback_context_acquire() callbacks we get exactly 1 argument, a reference to the parameters that `context()` will use to find the context.

- print “A tool has requested the context\n”

  Print a notification whenever a tool asks for a context.

- test2_add_callback_context_init(sub { ... })

  Add our context init callback. These callbacks are triggered whenever a completely new context is created. This is not called if an existing context is found. In short this only fires off for the top level tool, not nested tools.

- my $ctx_ref = shift

  The coderefs for test2_add_callback_context_init() will receive exactly 1 argument, the newly created context.

- print “A new context was created\n”

  Print a notification whenever a new context is created.

SEE ALSO

Test2::Manual - Primary index of the manual.

SOURCE

The source code repository for Test2-Manual can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Manual::Tooling::Subtest

How to implement a tool that makes use of subtests.

DESCRIPTION

Subtests are a nice way of making related events visually, and architecturally distinct.

WHICH TYPE OF SUBTEST DO I NEED?

There are 2 types of subtest. The first type is subtests with user-supplied coderefs, such as the `subtest()` function itself. The second type is subtest that do not have any user supplied coderefs.

So which type do you need? The answer to that is simple, if you are going to let the user define the subtest with their own codeblock, you have the first type, otherwise you have the second.

In either case, you will still need use the same API function: `Test2::API::run_subtest`.

SUBTEST WITH USER SUPPLIED CODEREF

This example will emulate the `subtest` function.

```
use Test2::API qw/context run_subtest/;

sub my_subtest {
    my ($name, $code) = @_;

    # Like any other tool, you need to acquire a context, if you do not then
    # things will not report the correct file and line number.
    my $ctx = context();

    my $bool = run_subtest($name, $code);

    $ctx->release;

    return $bool;
}
```

This looks incredibly simple... and it is. `run_subtest()` does all the hard work for you. This will issue an *Test2::Event::Subtest* event with the results of the subtest. The subtest event itself will report to the proper file and line number due to the context you acquired (even though it does not *look* like you used the context.

`run_subtest()` can take additional arguments:

```
run_subtest($name, $code, \%params, @args);
```

- @args

  This allows you to pass arguments into the codeblock that gets run.

- \%params

  This is a hashref of parameters. Currently there are 3 possible parameters:

  - buffered => $bool

    This will turn the subtest into the new style buffered subtest. This type of subtest is recommended, but not default.

  - inherit_trace => $bool

    This is used for tool-side coderefs.

  - no_fork => $bool

    react to forking/threading inside the subtest itself. In general you are unlikely to need/want this parameter.

SUBTEST WITH TOOL-SIDE CODEREF

This is particularly useful if you want to turn a tool that wraps other tools into a subtest. For this we will be using the tool we created in *Test2::Manual::Tooling::Nesting*.

```
use Test2::API qw/context run_subtest/;

sub check_class {
    my $class = shift;

    my $ctx = context();

    my $code = sub {
        my $obj = $class->new;
        is($obj->foo, 'foo', "got foo");
        is($obj->bar, 'bar', "got bar");
    };

    my $bool = run_subtest($class, $code, {buffered => 1, inherit_
↪trace => 1});

    $ctx->release;

    return $bool;
}
```

The `run_subtest()` function does all the heavy lifting for us. All we need to do is give the function a name, a coderef to run, and the `inherit_trace => 1` parameter. The `buffered => 1` parameter is optional, but recommended.

The `inherit_trace` parameter tells the subtest tool that the contexts acquired inside the nested tools should use the same trace as the subtest itself. For user-supplied codeblocks you do not use inherit_trace because you want errors to report to the user-supplied file+line.

SEE ALSO

Test2::Manual - Primary index of the manual.

SOURCE

The source code repository for Test2-Manual can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Manual::Tooling::TestBuilder

This section maps Test::Builder methods to Test2 concepts.

DESCRIPTION

With Test::Builder tools were encouraged to use methods on the Test::Builder singleton object. Test2 has a different approach, every tool should get a new *Test2::API::Context* object, and call methods on that. This document maps several concepts from Test::Builder to Test2.

CONTEXT

First thing to do, stop using the Test::Builder singleton, in fact stop using or even loading Test::Builder. Instead of Test::Builder each tool you write should follow this template:

```
use Test2::API qw/context/;

sub my_tool {
    my $ctx  = context();

    ... do work ...

    $ctx->ok(1, "a passing assertion");

    $ctx->release;

    return $whatever;
}
```

The original Test::Builder style was this:

```
use Test::Builder;
my $tb = Test::Builder->new; # gets the singleton

sub my_tool {
    ... do work ...
```

(continues on next page)

(continued from previous page)

```
    $tb->ok(1, "a passing assertion");

    return $whatever;
}
```

TEST BUILDER METHODS

- $tb->BAIL_OUT($reason)

  The context object has a 'bail' method:

  ```
  $ctx->bail($reason)
  ```

- $tb->diag($string)
- $tb->note($string)

  The context object has diag and note methods:

  ```
  $ctx->diag($string);
  $ctx->note($string);
  ```

- $tb->done_testing

  The context object has a done_testing method:

  ```
  $ctx->done_testing;
  ```

  Unlike the Test::Builder version, no arguments are allowed.

- $tb->like
- $tb->unlike

  These are not part of context, instead look at *Test2::Compare* and *Test2::Tools::Compare*.

- $tb->ok($bool, $name)

  ```
  # Preferred
  $ctx->pass($name);
  $ctx->fail($name, @diag);

  # Discouraged, but supported:
  $ctx->ok($bool, $name, \@failure_diags)
  ```

- $tb->subtest

  use the `Test2::API::run_subtest()` function instead. See *Test2::API* for documentation.

- $tb->todo_start
- $tb->todo_end

  See *"skip($why)" in Test2::Tools::Basic*, and *Test2::Todo* instead.

- $tb->output, $tb->failure_output, and $tb->todo_output

  These are handled via formatters now. See *Test2::Formatter* and *Test2::Formatter::TAP*.

LEVEL

Test::Builder had the `$Test::Builder::Level` variable that you could modify in order to set the stack depth. This was useful if you needed to nest tools and wanted to make sure your file and line number were correct. It was also frustrating and prone to errors. Some people never even discovered the level variable and always had incorrect line numbers when their tools would fail.

Test2 uses the context system, which solves the problem a better way. The top-most tool get a context, and holds on to it until it is done. Any tool nested under the first will find and use the original context instead of generating a new one. This means the level problem is solved for free, no variables to mess with.

Test2 is also smart enough to honor `$Test::Builder::Level` if it is set.

TODO

Test::Builder used the `$TODO` package variable to set the TODO state. This was confusing, and easy to get wrong. See Test2::Tools::Todo for the modern way to accomplish a TODO state.

SEE ALSO

Test2::Manual - Primary index of the manual.

SOURCE

The source code repository for Test2-Manual can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Manual::Tooling::Testing

Tutorial on how to test your testing tools.

DESCRIPTION

Testing your test tools used to be a complex and difficult prospect. The old tools such as *Test::Tester* and *Test::Builder::Tester* were limited, and fragile. Test2 on the other hand was designed from the very start to be easily tested! This tutorial shows you how.

THE HOLY GRAIL OF TESTING YOUR TOOLS

The key to making Test2 easily testable (specially when compared to Test::Builder) is the `intercept` function.

```
use Test2::API qw/intercept/;

my $events = intercept {
    ok(1, "pass");
    ok(0, "fail");

    diag("A diag");
};
```

The intercept function lets you use any test tools you want inside a codeblock. No events or contexts generated within the intercept codeblock will have any effect on the outside testing state. The `intercept` function completely isolates the tools called within.

Note: Plugins and things that affect global API state may not be fully isolated. `intercept` is intended specifically for event isolation.

The `intercept` function will return an arrayref containing all the events that were generated within the codeblock. You can now make any assertions you want about the events you expected your tools to generate.

```
[
    bless({...}, 'Test2::Event::Ok'),   # pass
    bless({...}, 'Test2::Event::Ok'),   # fail
    bless({...}, 'Test2::Event::Diag'), # Failure diagnostics (not␣
↪always a second event)
    bless({...}, 'Test2::Event::Diag'), # custom 'A diag' message
]
```

Most test tools eventually produce one or more events. To effectively verify the events you get from intercept you really should read up on how events work *Test2::Manual::Anatomy::Event*. Once you know about events you can move on to the next section which points you at some helpers.

ADDITIONAL HELPERS

Test2::Tools::Tester

This is the most recent set of tools to help you test your events. To really understand these you should familiarize yourself with *Test2::Manual::Anatomy::Event*. If you are going to be writing anything more than the most simple of tools you should know how events work.

The *Test2::Tools::Tester* documentation is a good place for further reading.

Test2::Tools::HarnessTester

The Test2::Tools::HarnessTester can export the `summarize_events()` tool. This tool lets you run your event arrayref through Test2::Harness so that you can get a pass/fail summary.

```
my $summary = summarize_events($events);
```

The summary looks like this:

```
{
    plan       => $plan_facet,          # the plan event facet
    pass       => $bool,                # true if the events result␣
↪in a pass
    fail       => $bool,                # true if the events result␣
↪in a fail
    errors     => $error_count,         # Number of error facets seen
    failures   => $failure_count,       # Number of failing␣
↪assertions seen
    assertions => $assertion_count,     # Total number of assertions␣
↪seen
}
```

Test2::Tools::Compare

DEPRECATED These tools were written before the switch to faceted events. These will still work, but are no longer the recommended way to test your tools.

The *Test2::Tools::Compare* library exports a handful of extras to help test events.

- event $TYPE => ...

  Use in an array check against $events to check for a specific type of event with the properties you specify.

- fail_events $TYPE => ...

  Use when you expect a failing assertion of $TYPE. This will automatically check that the next event following it is a diagnostics message with the default failure text.

  Note: This is outdated as a single event may now possess both the failing assertion AND the failing text, such events will fail this test.

SEE ALSO

Test2::Manual - Primary index of the manual.

SOURCE

The source code repository for Test2-Manual can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Mock

Module for managing mocked classes and instances.

DESCRIPTION

This module lets you add and override methods for any package temporarily. When the instance is destroyed it will restore the package to its original state.

SYNOPSIS

```
use Test2::Mock;
use MyClass;

my $mock = Test2::Mock->new(
    track => $BOOL, # enable call tracking if desired
    class => 'MyClass',
    override => [
        name => sub { 'fred' },
        ...
    ],
    add => [
        is_mocked => sub { 1 }
        ...
```

(continues on next page)

(continued from previous page)

```
    ],
    ...
);

# Unmock the 'name' sub
$mock->restore('name');

...

$mock = undef; # Will remove all the mocking
```

CONSTRUCTION

METHODS

- $mock = Test2::Mock->new(class => $CLASS, ...)

  This will create a new instance of *Test2::Mock* that manages mocking for the specified `$CLASS`.

  Any `Test2::Mock` method can be used as a constructor argument, each should be followed by an arrayref of arguments to be used within the method. For instance the `add()` method:

  ```
  my $mock = Test2::Mock->new(
      class => 'AClass',
      add => [foo => sub { 'foo' }],
  );
  ```

  is identical to this:

  ```
  my $mock = Test2::Mock->new(
      class => 'AClass',
  );
  $mock->add(foo => sub { 'foo' });
  ```

- $mock->track($bool)

  Turn tracking on or off. Any sub added/overridden/set when tracking is on will log every call in a hash retrievable via `$mock->tracking`. Changing the tracking toggle will not affect subs already altered, but will affect any additional alterations.

- $hashref = $mock->sub_tracking

  The tracking data looks like this:

  ```
  {
      sub_name => [
          {sub_name => $sub_name, sub_ref => $mock_subref, args =>
  ↪ [... copy of @_ from the call ... ]},
          ...,
          ...,
  ```

(continues on next page)

(continued from previous page)

```
    ],
}
```

Unlike call_tracking, this lists all calls by sub, so you can choose to only look at the sub specific calls.

Please note: The hashref items with the subname and args are shared with call_tracking, modifying one modifies the other, so copy first!

- $arrayref = $mock->call_tracking

  The tracking data looks like this:

```
[
    {sub_name => $sub_name, sub_ref => $mock_subref, args => [..
↪. copy of @_ from the call ... ]},
    ...,
    ...,
]
```

  Unlike sub_tracking this lists all calls to any mocked sub, in the order they were called. To filter by sub use sub_tracking.

  Please note: The hashref items with the subname and args are shared with sub_tracking, modifying one modifies the other, so copy first!

- $mock->clear_sub_tracking()
- $mock->clear_sub_tracking(\@subnames)

  Clear tracking data. With no arguments ALL tracking data is cleared. When arguments are provided then only those specific keys will be cleared.

- $mock->clear_call_tracking()

  Clear all items from call_tracking.

- $mock->add('symbol' => ..., 'symbol2' => ...)
- $mock->override('symbol1' => ..., 'symbol2' => ...)
- $mock->set('symbol1' => ..., 'symbol2' => ...)

  `add()` and `override()` are the primary ways to add/modify methods for a class. Both accept the exact same type of arguments. The difference is that `override` will fail unless the symbol you are overriding already exists, `add` on the other hand will fail if the symbol does already exist.

  `set()` was more recently added for cases where you may not know if the sub already exists. These cases are rare, and set should be avoided (think of it like 'no strict'). However there are valid use cases, so it was added.

  Note: Think of override as a push operation. If you call override on the same symbol multiple times it will track that. You can use `restore()` as a pop operation to go back to the previous mock. `reset` can be used to remove all the mocking for a symbol.

Arguments must be a symbol name, with optional sigil, followed by a new specification of the symbol. If no sigil is specified then '&' (sub) is assumed. A simple example of overriding a sub:

```
$mock->override(foo => sub { 'overridden foo' });
my $val = $class->foo; # Runs our override
# $val is now set to 'overridden foo'
```

You can also simply provide a value and it will be wrapped in a sub for you:

```
$mock->override( foo => 'foo' );
```

The example above will generate a sub that always returns the string 'foo'.

There are three *special* values that can be used to generate accessors:

```
$mock->add(
    name => 'rw',   # Generates a read/write accessor
    age  => 'ro',   # Generates a read only accessor
    size => 'wo',   # Generates a write only accessor
);
```

If you want to have a sub that actually returns one of the three special strings, or that returns a coderef, you can use a hashref as the spec:

```
my $ref = sub { 'my sub' };
$mock->add(
    rw_string => { val => 'rw' },
    ro_string => { val => 'ro' },
    wo_string => { val => 'wo' },
    coderef   => { val => $ref }, # the coderef method returns
 ↪$ref each time
);
```

You can also override/add other symbol types, such as hash:

```
package Foo;
...

$mock->add('%foo' => {a => 1});

print $Foo::foo{a}; # prints '1'
```

You can also tell mock to deduce the symbol type for the add/override from the reference, rules are similar to glob assignments:

```
$mock->add(
    -foo => sub { 'foo' },     # Adds the &foo sub to the␣
 ↪package
    -foo => { foo => 1 },      # Adds the %foo hash to the␣
 ↪package
    -foo => [ 'f', 'o', 'o' ], # Adds the @foo array to the␣
```

(continues on next page)

(continued from previous page)

```
↪package
    -foo => \"foo",            # Adds the $foo scalar to the␣
↪package
);
```

- $mock->restore($SYMBOL)

  Restore the symbol to what it was before the last override. If the symbol was recently added this will remove it. If the symbol has been overridden multiple times this will ONLY restore it to the previous state. Think of `override` as a push operation, and `restore` as the pop operation.

- $mock->reset($SYMBOL)

  Remove all mocking of the symbol and restore the original symbol. If the symbol was initially added then it will be completely removed.

- $mock->orig($SYMBOL)

  This will return the original symbol, before any mocking. For symbols that were added this will return undef.

- $mock->current($SYMBOL)

  This will return the current symbol.

- $mock->reset_all

  Remove all added symbols, and restore all overridden symbols to their originals.

- $mock->add_constructor($NAME => $TYPE)

- $mock->override_constructor($NAME => $TYPE)

  This can be used to inject constructors. The first argument should be the name of the constructor. The second argument specifies the constructor type.

  The `hash` type is the most common, all arguments are used to create a new hash that is blessed.

```
hash => sub  {
    my ($class, %params) = @_;
    return bless \%params, $class;
};
```

  The `array` type is similar to the hash type, but accepts a list instead of key/value pairs:

```
array => sub {
    my ($class, @params) = @_;
    return bless \@params, $class;
};
```

  The `ref` type takes a reference and blesses it. This will modify your original input argument.

```
ref => sub {
    my ($class, $params) = @_;
    return bless $params, $class;
};
```

The `ref_copy` type will copy your reference and bless the copy:

```
ref_copy => sub {
    my ($class, $params) = @_;
    my $type = reftype($params);

    return bless {%$params}, $class
        if $type eq 'HASH';

    return bless [@$params], $class
        if $type eq 'ARRAY';

    croak "Not sure how to construct a '$class' from '$params'";
};
```

- $mock->before($NAME, sub { ... })

  This will replace the original sub `$NAME` with a new sub that calls your custom code just before calling the original method. The return from your custom sub is ignored. Your sub and the original both get the unmodified arguments.

- $mock->after($NAME, sub { ... })

  This is similar to before, except your callback runs after the original code. The return from your callback is ignored.

- $mock->around($NAME, sub { ... })

  This gives you the chance to wrap the original sub:

```
$mock->around(foo => sub {
    my $orig = shift;
    my $self = shift;
    my (@args) = @_;

    ...
    $self->$orig(@args);
    ...

    return ...;
});
```

  The original sub is passed in as the first argument, even before `$self`. You are responsible for making sure your wrapper sub returns the correct thing.

- $mock->autoload

  This will inject an `AUTOLOAD` sub into the class. This autoload will automatically generate read-write accessors for any sub called that does not already exist.

- $mock->block_load

  This will prevent the real class from loading until the mock is destroyed. This will fail if the class is already loaded. This will let you mock a class completely without loading the original module.

- $pm_file = $mock->file

  This returns the relative path to the file for the module. This corresponds to the `%INC` entry.

- $bool = $mock->purge_on_destroy($bool)

  When true, this will cause the package stash to be completely obliterated when the mock object falls out of scope or is otherwise destroyed. You do not normally want this.

- $stash = $mock->stash

  This returns the stash for the class being mocked. This is the equivalent of:

  ```
  my $stash = \%{"${class}\::"};
  ```

  This saves you from needing to turn off strict.

- $class = $mock->class

  The class being mocked by this instance.

- $p = $mock->parent

  If you mock a class twice the first instance is the parent, the second is the child. This prevents the parent from being destroyed before the child, which would lead to a very unpleasant situation.

- $c = $mock->child

  Returns the child mock, if any.

SOURCE

The source code repository for Test2-Suite can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::Plugin

Documentation for plugins

DESCRIPTION

Plugins are packages that cause behavior changes, or other side-effects for the test file that loads them. They should not export any functions, or provide any tools. Plugins should be distinct units of functionality.

If you wish to combine behavior changes with tools then you should write a Plugin, a Tools module, and a bundle that loads them both.

FAQ

- Should I subclass Test2::Plugin?

  No. Currently this class is empty. Eventually we may want to add behavior, in which case we do not want anyone to already be subclassing it.

HOW DO I WRITE A PLUGIN?

Writing a plugin is not as simple as writing an *Test2::Bundle*, or writing *Test2::Tools*. Plugins alter behavior, or cause desirable side-effects. To accomplish this you typically need a custom `import()` method that calls one or more functions provided by the *Test2::API* package.

If you want to write a plugin you should look at existing plugins, as well as the *Test2::API* and *Test2::Hub* documentation. There is no formula for a Plugin, they are generally unique, however consistent rules are that they should not load other plugins, or export any functions.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Modules

- `Test2::Plugin::BailOnFail` - Automatically bail out of testing on the first test failure.
- `Test2::Plugin::DieOnFail` - Automatically die on the first test failure.
- `Test2::Plugin::ExitSummary` - Add extra diagnostics on failure at the end of the test.
- `Test2::Plugin::SRand` - Control the random seed for more controlled test environments.
- `Test2::Plugin::Times` - Output timing data at the end of the test.
- `Test2::Plugin::UTF8` - Test2 plugin to test with utf8.

Test2::Plugin::BailOnFail

Automatically bail out of testing on the first test failure.

DESCRIPTION

This module will issue a bailout event after the first test failure. This will prevent your tests from continuing. The bailout runs when the context is released; that is, it will run when the test function you are using, such as `ok()`, returns. This gives the tools the ability to output any extra diagnostics they may need.

SYNOPSIS

```
use Test2::V1;
use Test2::Plugin::BailOnFail;

T2->ok(1, "pass");
T2->ok(0, "fail");
T2->ok(1, "Will not run");
```

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Plugin::DieOnFail

Automatically die on the first test failure.

DESCRIPTION

This module will die after the first test failure. This will prevent your tests from continuing. The exception is thrown when the context is released, that is it will run when the test function you are using, such as `ok()`, returns. This gives the tools the ability to output any extra diagnostics they may need.

SYNOPSIS

```
use Test2::V1;
use Test2::Plugin::DieOnFail;

T2->ok(1, "pass");
T2->ok(0, "fail");
T2->ok(1, "Will not run");
```

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Plugin::ExitSummary

Add extra diagnostics on failure at the end of the test.

DESCRIPTION

This will provide some diagnostics after a failed test. These diagnostics can range from telling you how you deviated from your plan, warning you if there was no plan, etc. People used to *Test::More* generally expect these diagnostics.

SYNOPSIS

```
use Test2::Plugin::ExitSummary;
```

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Plugin::SRand

Control the random seed for more controlled test environments.

DESCRIPTION

This module gives you control over the random seed used for your unit tests. In some testing environments the random seed can play a major role in results.

The default configuration for this module will seed srand with the local date. Using the date as the seed means that on any given day the random seed will always be the same, this means behavior will not change from run to run on a given day. However the seed is different on different days allowing you to be sure the code still works with actual randomness.

The seed is printed for you on failure, or when the harness is verbose. You can use the `T2_RAND_SEED` environment variable to specify the seed. You can also provide a specific seed as a load-time argument to the plugin.

SYNOPSIS

Loading the plugin is easy, and the defaults are sane:

```
use Test2::Plugin::SRand;
```

Custom seed:

```
use Test2::Plugin::SRand seed => 42;
```

NOTE ON LOAD ORDER

If you use this plugin you probably want to use it as the first, or near-first plugin. `srand` is not called until the plugin is loaded, so other plugins loaded first may already be making use of random numbers before your seed takes effect.

CAVEATS

When srand is on (default) it can cause problems with things like *File::Temp* which will end up attempting the same "random" filenames for every test process started on a given day (or sharing the same seed).

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Plugin::Times

Output timing data at the end of the test.

CAVEAT

It is important to note that this timing data does not include global destruction. This data is only collected up until the point done_testing() is called. If your program takes time for END blocks, garbage collection, and similar, then this timing data will fall short of reality.

DESCRIPTION

This plugin will output a diagnostics message at the end of testing that tells you how much time elapsed, and how hard the system worked on the test.

This will produce a string like one of these (Note these numbers are completely made up). *Which string is used depends on the time elapsed.*

```
0.12345s on wallclock (0.05 usr 0.00 sys + 0.00 cusr 0.00 csys = 0.
↪05 CPU)

11.1234s on wallclock (0.05 usr 0.00 sys + 0.00 cusr 0.00 csys = 0.
↪05 CPU)

01m:54.45s on wallclock (0.05 usr 0.00 sys + 0.00 cusr 0.00 csys =␣
↪0.05 CPU)

18h:22m:54.45s on wallclock (0.05 usr 0.00 sys + 0.00 cusr 0.00 csys␣
↪= 0.05 CPU)

04d:18h:22m:54.45s on wallclock (0.05 usr 0.00 sys + 0.00 cusr 0.00␣
↪csys = 0.05 CPU)
```

SYNOPSIS

```
use Test2::Plugin::Times;
```

This is also useful at the command line for 1-time use:

```
$ perl -MTest2::Plugin::Times path/to/test.t
```

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Plugin::UTF8

Test2 plugin to test with utf8.

DESCRIPTION

When used, this plugin will make tests work with utf8. This includes turning on the utf8 pragma and updating the Test2 output formatter to use utf8.

SYNOPSIS

```
use Test2::Plugin::UTF8;
```

This is similar to:

```
use utf8;
BEGIN {
    require Test2::Tools::Encoding;
    Test2::Tools::Encoding::set_encoding('utf8');
}
```

You can also disable the utf8 import by using 'encoding_only' to only enable utf8 encoding on the output format.

```
use Test2::Plugin::UTF8 qw(encoding_only);
```

import options

encoding_only

Does not import utf8 in your test and only enables the encoding mode on the output.

NOTES

This module currently sets output handles to have the ':utf8' output layer. Some might prefer ':encoding(utf-8)' which is more strict about verifying characters. There is a debate about whether or not encoding to utf8 from perl internals can ever fail, so it may not matter. This was also chosen because the alternative causes threads to segfault, see perlbug 31923.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Require

Base class and documentation for skip-unless type test packages.

DESCRIPTION

Test2::Require::* packages are packages you load to ensure your test file is skipped unless a specific requirement is met. Modules in this namespace may subclass *Test2::Require* if they wish, but it is not strictly necessary to do so.

HOW DO I WRITE A 'REQUIRE' MODULE?

AS A SUBCLASS

```
package Test2::Require::Widget;
use strict;
use warnings;

use base 'Test2::Require';

sub HAVE_WIDGETS { ... };

sub skip {
    my $class = shift;
    my @import_args = @_;

    if (HAVE_WIDGETS()) {
        # We have widgets, do not skip
        return undef;
    }
    else {
        # No widgets, skip the test
        return "Skipped because there are no widgets" unless HAVE_
↪WIDGETS();
    }
}

1;
```

A subclass of *Test2::Require* simply needs to implement a `skip()` method. This method will receive all import arguments. This method should return undef if the test should run, and should return a reason for skipping if the test should be skipped.

STAND-ALONE

If you do not wish to subclass *Test2::Require* then you should write an `import()` method:

```
package Test2::Require::Widget;
use strict;
use warnings;

use Test2::API qw/context/;

sub HAVE_WIDGETS { ... };

sub import {
    my $class = shift;

    # Have widgets, should run.
    return if HAVE_WIDGETS();
```

(continues on next page)

(continued from previous page)

```
    # Use the context object to create the event
    my $ctx = context();
    $ctx->plan(0, SKIP => "Skipped because there are no widgets");
    $ctx->release;
}

1;
```

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Modules

- `Test2::Require::AuthorTesting` - Only run a test when the AUTHOR_TESTING environment variable is set.
- `Test2::Require::AutomatedTesting` - Only run a test when the AUTOMATED_TESTING environment variable is set.
- `Test2::Require::EnvVar` - Only run a test when a specific environment variable is set.
- `Test2::Require::ExtendedTesting` - Only run a test when the EXTENDED_TESTING environment variable is set.
- `Test2::Require::Fork` - Skip a test file unless the system supports forking
- `Test2::Require::Module` - Skip tests if certain packages are not installed, or insufficient versions.
- `Test2::Require::NonInteractiveTesting` - Only run a test when the NONINTERACTIVE_TESTING environment variable is set.

- `Test2::Require::Perl` - Skip the test unless the necessary version of Perl is installed.
- `Test2::Require::RealFork` - Skip a test file unless the system supports true forking
- `Test2::Require::ReleaseTesting` - Only run a test when the RELEASE_TESTING environment variable is set.
- `Test2::Require::Threads` - Skip a test file unless the system supports threading

Test2::Require::AuthorTesting

Only run a test when the AUTHOR_TESTING environment variable is set.

DESCRIPTION

It is common practice to write tests that are only run when the AUTHOR_TESTING environment variable is set. This module automates the (admittedly trivial) work of creating such a test.

SYNOPSIS

```
use Test2::Require::AuthorTesting;

...

done_testing;
```

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Require::AutomatedTesting

Only run a test when the AUTOMATED_TESTING environment variable is set.

DESCRIPTION

It is common practice to write tests that are only run when the AUTOMATED_TESTING environment variable is set. This module automates the (admittedly trivial) work of creating such a test.

SYNOPSIS

```
use Test2::Require::AutomatedTesting;

...

done_testing;
```

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Require::EnvVar

Only run a test when a specific environment variable is set.

DESCRIPTION

It is common practice to write tests that are only run when an environment variable is set. This module automates the (admittedly trivial) work of creating such a test.

SYNOPSIS

```
use Test2::Require::EnvVar 'SOME_VAR';

...

done_testing;
```

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Require::ExtendedTesting

Only run a test when the EXTENDED_TESTING environment variable is set.

DESCRIPTION

It is common practice to write tests that are only run when the EXTENDED_TESTING environment variable is set. This module automates the (admittedly trivial) work of creating such a test.

SYNOPSIS

```
use Test2::Require::ExtendedTesting;

...

done_testing;
```

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Require::Fork

Skip a test file unless the system supports forking

DESCRIPTION

It is fairly common to write tests that need to fork. Not all systems support forking. This library does the hard work of checking if forking is supported on the current system. If forking is not supported then this will skip all tests and exit true.

SYNOPSIS

```
use Test2::Require::Fork;

... Code that forks ...
```

EXPLANATION

Checking if the current system supports forking is not simple. Here is an example of how to do it:

```
use Config;

sub CAN_FORK {
    return 1 if $Config{d_fork};

    # Some platforms use ithreads to mimic forking
    return 0 unless $^O eq 'MSWin32' || $^O eq 'NetWare';
```

(continues on next page)

(continued from previous page)

```
    return 0 unless $Config{useithreads};
    return 0 unless $Config{ccflags} =~ /-DPERL_IMPLICIT_SYS/;

    # Threads are not reliable before 5.008001
    return 0 unless $] >= 5.008001;

    # Devel::Cover currently breaks with threads
    return 0 if $INC{'Devel/Cover.pm'};
    return 1;
}
```

Duplicating this non-trivial code in all tests that need to fork is error-prone. It is easy to forget bits, or get it wrong. On top of these checks, you also need to tell the harness that no tests should run and why.

SEE ALSO

- *Test2::Require::RealFork*

  Similar to this module, but will skip on any perl that only has fork emulation.

- *Test2::Require::Threads*

  Skip the test file if the system does not support threads.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Require::Module

Skip tests if certain packages are not installed, or insufficient versions.

DESCRIPTION

Sometimes you have tests that are nice to run, but depend on tools that may not be available. Instead of adding the tool as a dep, or making the test always skip, it is common to make the test run conditionally. This package helps make that possible.

This module is modeled after Test::Requires. The difference is that this module is based on *Test2* directly, and does not go through *Test::Builder*. Another difference is that the packages you check for are not imported into your namespace for you. This is intentional.

SYNOPSIS

```
# The test will be skipped unless Some::Module is installed, any␣
 ↪version.
use Test2::Require::Module 'Some::Module';

# The test will be skipped unless 'Other::Module' is installed and␣
 ↪at
# version '5.555' or greater.
use Test2::Require::Module 'Other::Module' => '5.555';

# We now need to use them directly, Test2::Require::Module does not␣
 ↪import
# them for us.
use Some::Module;
use Other::Module;
```

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See http://dev.perl.org/licenses/

Test2::Require::NonInteractiveTesting

Only run a test when the NONINTERACTIVE_TESTING environment variable is set.

DESCRIPTION

It is common practice to write tests that are only run when the NONINTERACTIVE_TESTING environment variable is set. This module automates the (admittedly trivial) work of creating such a test.

SYNOPSIS

```
use Test2::Require::NonInteractiveTesting;

...

done_testing;
```

SOURCE

The source code repository for Test2-Suite can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See http://dev.perl.org/licenses/

Test2::Require::Perl

Skip the test unless the necessary version of Perl is installed.

DESCRIPTION

Sometimes you have tests that are nice to run, but depend on a certain version of Perl. This package lets you run the test conditionally, depending on if the correct version of Perl is available.

SYNOPSIS

```
# Skip the test unless perl 5.10 or greater is installed.
use Test2::Require::Perl 'v5.10';

# Enable 5.10 features.
use v5.10;
```

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Require::RealFork

Skip a test file unless the system supports true forking

DESCRIPTION

It is fairly common to write tests that need to fork. Not all systems support forking. This library does the hard work of checking if forking is supported on the current system. If forking is not supported then this will skip all tests and exit true.

SYNOPSIS

```
use Test2::Require::RealFork;

... Code that forks ...
```

SEE ALSO

- *Test2::Require::Fork*

  Similar to this module, but will allow fork emulation.

- *Test2::Require::Threads*

  Skip the test file if the system does not support threads.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Require::ReleaseTesting

Only run a test when the RELEASE_TESTING environment variable is set.

DESCRIPTION

It is common practice to write tests that are only run when the RELEASE_TESTING environment variable is set. This module automates the (admittedly trivial) work of creating such a test.

SYNOPSIS

```
use Test2::Require::ReleaseTesting;

...

done_testing;
```

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Require::Threads

Skip a test file unless the system supports threading

DESCRIPTION

It is fairly common to write tests that need to use threads. Not all systems support threads. This library does the hard work of checking if threading is supported on the current system. If threading is not supported then this will skip all tests and exit true.

SYNOPSIS

```
use Test2::Require::Threads;

... Code that uses threads ...
```

EXPLANATION

Checking if the current system supports threading is not simple, here is an example of how to do it:

```
use Config;

sub CAN_THREAD {
    # Threads are not reliable before 5.008001
    return 0 unless $] >= 5.008001;
    return 0 unless $Config{'useithreads'};

    # Devel::Cover currently breaks with threads
    return 0 if $INC{'Devel/Cover.pm'};
    return 1;
}
```

Duplicating this non-trivial code in all tests that need to use threads is error-prone. It is easy to forget bits, or get it wrong. On top of these checks you also need to tell the harness that no tests should run and why.

SEE ALSO

- *Test2::Require::Fork*

  Skip the test file if the system does not support forking.

- *Test2*

  Test2::Require::Threads uses *Test2* under the hood.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Suite

Distribution with a rich set of tools built upon the Test2 framework.

DESCRIPTION

Rich set of tools, plugins, bundles, etc built upon the *Test2* testing library. If you are interested in writing tests, this is the distribution for you.

WHAT ARE TOOLS, PLUGINS, AND BUNDLES?

- TOOLS

  Tools are packages that export functions for use in test files. These functions typically generate events. Tools **SHOULD NEVER** alter behavior of other tools, or the system in general.

- PLUGINS

  Plugins are packages that produce effects, or alter behavior of tools. An example would be a plugin that causes the test to bail out after the first failure. Plugins **SHOULD NOT** export anything.

- BUNDLES

  Bundles are collections of tools and plugins. A bundle should load and re-export functions from Tool packages. A bundle may also load and configure any number of plugins.

If you want to write something that both exports new functions, and affects behavior, you should write both a Tools distribution, and a Plugin distribution, then a Bundle that loads them both. This is important as it helps avoid the problem where a package exports much-desired tools, but also produces undesirable side-effects.

INCLUDED BUNDLES

- Test2::V#

  These do not live in the bundle namespace as they are the primary ways to use Test2::Suite.

  The current latest is *Test2::V1*.

  ```
  use Test2::V1;

  T2->ok(...);

  # Note: is does deep checking, unlike the 'is' from Test::More.
  $T2->is(...);

  ...

  T2->done_testing;
  ```

  See *Test2::V1* for complete documentation.

- Extended

  ** **Deprecated** ** See *Test2::V0* or *Test2::V1*.

```
use Test2::Bundle::Extended;
# strict and warnings are on for you now.

ok(...);

# Note: is does deep checking, unlike the 'is' from Test::More.
is(...);

...

done_testing;
```

  This bundle includes every tool listed in the "INCLUDED TOOLS" section below, except for *Test2::Tools::ClassicCompare*. This bundle provides most of what anyone writing tests could need. This is also the preferred bundle/toolset of the *Test2* author.

  See *Test2::Bundle::Extended* for complete documentation.

- More

```
use Test2::Bundle::More;
use strict;
use warnings;

plan 3; # Or you can use done_testing at the end

ok(...);

is(...); # Note: String compare

is_deeply(...);

...

done_testing; # Use instead of plan
```

  This bundle is meant to be a *mostly* drop-in replacement for *Test::More*. There are some notable differences to be aware of however. Some exports are missing: `eq_array`, `eq_hash`, `eq_set`, `$TODO`, `explain`, `use_ok`, `require_ok`. As well it is no longer possible to set the plan at import: `use .. tests => 5`. `$TODO` has been replaced by the `todo()` function. Planning is done using `plan`, `skip_all`, or `done_testing`.

  See *Test2::Bundle::More* for complete documentation.

- Simple

```
use Test2::Bundle::Simple;
use strict;
```

(continues on next page)

(continued from previous page)

```
use warnings;

plan 1;

ok(...);
```

This bundle is meant to be a *mostly* drop-in replacement for *Test::Simple*. See *Test2::Bundle::Simple* for complete documentation.

INCLUDED TOOLS

- Basic

  Basic provides most of the essential tools previously found in *Test::More*. However it does not export any tools used for comparison. The basic `pass`, `fail`, `ok` functions are present, as are functions for planning.

  See *Test2::Tools::Basic* for complete documentation.

- Compare

  This provides `is`, `like`, `isnt`, `unlike`, and several additional helpers. **Note:** These are all *deep* comparison tools and work like a combination of *Test::More*'s `is` and `is_deeply`.

  See *Test2::Tools::Compare* for complete documentation.

- ClassicCompare

  This provides *Test::More* flavored `is`, `like`, `isnt`, `unlike`, and `is_deeply`. It also provides `cmp_ok`.

  See *Test2::Tools::ClassicCompare* for complete documentation.

- Class

  This provides functions for testing objects and classes, things like `isa_ok`.

  See *Test2::Tools::Class* for complete documentation.

- Defer

  This provides functions for writing test functions in one place, but running them later. This is useful for testing things that run in an altered state.

  See *Test2::Tools::Defer* for complete documentation.

- Encoding

  This exports a single function that can be used to change the encoding of all your test output.

  See *Test2::Tools::Encoding* for complete documentation.

- Exports

  This provides tools for verifying exports. You can verify that functions have been imported, or that they have not been imported.

  See *Test2::Tools::Exports* for complete documentation.

- Mock

  This provides tools for mocking objects and classes. This is based largely on Mock::Quick, but several interface improvements have been added that cannot be added to Mock::Quick itself without breaking backwards compatibility.

  See *Test2::Tools::Mock* for complete documentation.

- Ref

  This exports tools for validating and comparing references.

  See *Test2::Tools::Ref* for complete documentation.

- Spec

  This is an RSPEC implementation with concurrency support.

  See *Test2::Tools::Spec* for more details.

- Subtest

  This exports tools for running subtests.

  See *Test2::Tools::Subtest* for complete documentation.

- Target

  This lets you load the package(s) you intend to test, and alias them into constants/package variables.

  See *Test2::Tools::Target* for complete documentation.

INCLUDED PLUGINS

- BailOnFail

  The much requested “bail-out on first failure” plugin. When this plugin is loaded, any failure will cause the test to bail out immediately.

  See *Test2::Plugin::BailOnFail* for complete documentation.

- DieOnFail

  The much requested “die on first failure” plugin. When this plugin is loaded, any failure will cause the test to die immediately.

  See *Test2::Plugin::DieOnFail* for complete documentation.

- ExitSummary

  This plugin gives you statistics and diagnostics at the end of your test in the event of a failure.

  See *Test2::Plugin::ExitSummary* for complete documentation.

- SRand

  Use this to set the random seed to a specific seed, or to the current date.

  See *Test2::Plugin::SRand* for complete documentation.

- UTF8

  Turn on utf8 for your testing. This sets the current file to be utf8, it also sets STDERR, STDOUT, and your formatter to all output utf8.

  See *Test2::Plugin::UTF8* for complete documentation.

INCLUDED REQUIREMENT CHECKERS

- AuthorTesting

  Using this package will cause the test file to be skipped unless the AUTHOR_TESTING environment variable is set.

  See *Test2::Require::AuthorTesting* for complete documentation.

- EnvVar

  Using this package will cause the test file to be skipped unless a custom environment variable is set.

  See *Test2::Require::EnvVar* for complete documentation.

- Fork

  Using this package will cause the test file to be skipped unless the system is capable of forking (including emulated forking).

  See *Test2::Require::Fork* for complete documentation.

- RealFork

  Using this package will cause the test file to be skipped unless the system is capable of true forking.

  See *Test2::Require::RealFork* for complete documentation.

- Module

  Using this package will cause the test file to be skipped unless the specified module is installed (and optionally at a minimum version).

  See *Test2::Require::Module* for complete documentation.

- Perl

  Using this package will cause the test file to be skipped unless the specified minimum perl version is met.

  See *Test2::Require::Perl* for complete documentation.

- Threads

  Using this package will cause the test file to be skipped unless the system has threading enabled.

  Note: This will not turn threading on for you.

  See *Test2::Require::Threads* for complete documentation.

SEE ALSO

See the *Test2* documentation for a namespace map. Everything in this distribution uses *Test2*.

Test2::Manual is the Test2 Manual.

CONTACTING US

Many Test2 developers and users lurk on irc://irc.perl.org/#perl. We also have a slack team that can be joined by anyone with an `@cpan.org` email address https://perl-test2.slack.com/ If you do not have an `@cpan.org` email you can ask for a slack invite by emailing Chad Granum exodist@cpan.org.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Todo

TODO extension for Test2.

DESCRIPTION

This is an object that lets you create and manage TODO states for tests. This is an extension, not a plugin or a tool. This library can be used by plugins and tools to manage todo states.

If you simply want to write a todo test then you should look at the `todo` function provided by *Test2::Tools::Basic*.

SYNOPSIS

```
use Test2::Todo;

# Start the todo
my $todo = Test2::Todo->new(reason => 'Fix later');

# Will be considered todo, so suite still passes
ok(0, "oops");

# End the todo
$todo->end;

# TODO has ended, this test will actually fail.
ok(0, "oops");
```

CONSTRUCTION OPTIONS

- reason (required)

  The reason for the todo, this can be any defined value.

- hub (optional)

  The hub to which the TODO state should be applied. If none is provided then the current global hub is used.

INSTANCE METHODS

- $todo->end

  End the todo state.

CLASS METHODS

- $count = Test2::Todo->hub_in_todo($hub)

  If the hub has any todo objects this will return the total number of them. If the hub has no todo objects it will return 0.

OTHER NOTES

How it works

When an instance is created a filter sub is added to the *Test2::Hub*. This filter will set the `todo` and `diag_todo` attributes on all events as they come in. When the instance is destroyed, or `end()` is called, the filter is removed.

When a new hub is pushed (such as when a subtest is started) the new hub will inherit the filter, but it will only set `diag_todo`, it will not set `todo` on events in child hubs.

$todo->end is called at destruction

If your `$todo` object falls out of scope and gets garbage collected, the todo will end.

Can I use multiple instances?

Yes. The most recently created one that is still active will win.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Tools

Documentation for Tools.

DESCRIPTION

Tools are packages that export test functions, typically all related to a specific aspect of testing. If you have a couple different categories of exports then you may want to break them into separate modules.

Tools should export testing functions. Loading tools **should not** have side- effects, or alter the behavior of other tools. If you want to alter behaviors or create side-effects then you probably want to write a *Test2::Plugin*.

FAQ

- Why is it called Test2::Tools, and not Test2::Tool?

  This question arises since Tools is the only namespace in the plural. This is because each Plugin should be a distinct unit of functionality, but a Tools dist can (and usually should) export several tools. A bundle is also typically described as a single unit. Nobody would like Test2::Bundles::Foo.

- Should my tools subclass Test2::Tools?

  No. Currently this class is empty. Eventually we may want to add behavior, in which case we do not want anyone to already be subclassing it.

HOW DO I WRITE A 'TOOLS' MODULE?

It is very easy to write tools:

```
package Test2::Tools::Mine
use strict;
use warnings;

# All tools should use the context() function.
use Test2::API qw/context/;

our @EXPORTS = qw/ok plan/;
use base 'Exporter';

sub ok($;$) {
    my ($bool, $name) = @_;

    # All tool functions should start by grabbing a context
    my $ctx = context();

    # The context is the primary interface for generating events
    $ctx->ok($bool, $name);

    # When you are done you release the context
    $ctx->release;

    return $bool ? 1 : 0;
}

sub plan {
    my ($max) = @_;
    my $ctx = context();
    $ctx->plan($max);
    $ctx->release;
}

1;
```

See *Test2::API::Context* for documentation on what the `$ctx` object can do.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Modules

- `Test2::Tools::AsyncSubtest` - Tools for writing async subtests.
- `Test2::Tools::Basic` - Test2 implementation of the basic testing tools.
- `Test2::Tools::Class` - Test2 implementation of the tools for testing classes.
- `Test2::Tools::ClassicCompare` - Classic (Test::More style) comparison tools.
- `Test2::Tools::Compare` - Tools for comparing deep data structures.
- `Test2::Tools::Defer` - Write tests that get executed at a later time
- `Test2::Tools::Encoding` - Tools for managing the encoding of *Test2* based tests.
- `Test2::Tools::Event` - Tools for generating test events.
- `Test2::Tools::Exception` - Test2 based tools for checking exceptions
- `Test2::Tools::Exports` - Tools for validating exporters.
- `Test2::Tools::GenTemp` - Tool for generating a populated temp directory.
- `Test2::Tools::Grab` - Temporarily intercept all events without adding a scope level.
- `Test2::Tools::Mock` - Class/Instance mocking for Test2.
- `Test2::Tools::Ref` - Tools for validating references.
- `Test2::Tools::Refcount` - assert reference counts on objects
- `Test2::Tools::Spec` - RSPEC implementation on top of Test2::Workflow
- `Test2::Tools::Subtest` - Tools for writing subtests
- `Test2::Tools::Target` - Alias the testing target package.
- `Test2::Tools::Tester` - Tools to help you test other testing tools.
- `Test2::Tools::Tiny` - Tiny set of tools for unfortunate souls who cannot use *Test2::Suite*.
- `Test2::Tools::Warnings` - Tools to verify warnings.

Test2::Tools::AsyncSubtest

Tools for writing async subtests.

DESCRIPTION

These are tools for writing async subtests. Async subtests are subtests which can be started and stashed so that they can continue to receive events while other events are also being generated.

SYNOPSIS

```
use Test2::Bundle::Extended;
use Test2::Tools::AsyncSubtest;

my $ast1 = async_subtest local => sub {
    ok(1, "Inside subtest");
};

my $ast2 = fork_subtest child => sub {
    ok(1, "Inside subtest in another process");
};

# You must call finish on the subtests you create. Finish will wait/
 ↪join on
# any child processes and threads.
$ast1->finish;
$ast2->finish;

done_testing;
```

EXPORTS

Everything is exported by default.

- $ast = async_subtest $name
- $ast = async_subtest $name => sub { ... }
- $ast = async_subtest $name => \%hub_params, sub { ... }

  Create an async subtest. Run the codeblock if it is provided.
- $ast = fork_subtest $name => sub { ... }
- $ast = fork_subtest $name => \%hub_params, sub { ... }

  Create an async subtest. Run the codeblock in a forked process.
- $ast = thread_subtest $name => sub { ... }
- $ast = thread_subtest $name => \%hub_params, sub { ... }

  ** **DISCOURAGED** ** Threads are fragile. Thread tests are not even run unless the AUTHOR_TESTING or T2_DO_THREAD_TESTS env vars are enabled.

Create an async subtest. Run the codeblock in a thread.

NOTES

- Async Subtests are always buffered.

  Always buffered.

- Do not use done_testing() yourself.

  using done_testing() inside an async subtest will not work properly, the async subtest must be finalized by calling `$st->finish`.

SOURCE

The source code repository for Test2-AsyncSubtest can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist7@gmail.com.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Tools::Basic

Test2 implementation of the basic testing tools.

DESCRIPTION

This is a *Test2* based implementation of the more basic tools originally provided by *Test::More*. Not all *Test::More* tools are provided by this package, only the basic/simple ones. Some tools have been modified for better diagnostics capabilities.

SYNOPSIS

```
use Test2::Tools::Basic;

ok($x, "simple test");
```

(continues on next page)

(continued from previous page)

```
if ($passing) {
    pass('a passing test');
}
else {
    fail('a failing test');
}

diag "This is a diagnostics message on STDERR";
note "This is a diagnostics message on STDOUT";

{
    my $todo = todo "Reason for todo";
    ok(0, "this test is todo");
}

ok(1, "this test is not todo");

todo "reason" => sub {
    ok(0, "this test is todo");
};

ok(1, "this test is not todo");

SKIP: {
    skip "This will wipe your drive";

    # This never gets run:
    ok(!system('sudo rm -rf /'), "Wipe drive");
}

done_testing;
```

EXPORTS

All subs are exported by default.

PLANNING

- plan($num)
- plan('tests' => $num)
- plan('skip_all' => $reason)

  Set the number of tests that are expected. This must be done first or last, never in the middle of testing.

  For legacy compatibility you can specify 'tests' as the first argument before the number. You can also use this to skip all with the 'skip_all' prefix, followed by a reason for skipping.

- skip_all($reason)

Set the plan to 0 with a reason, then exit true. This should be used before any tests are run.

- done_testing

  Used to mark the end of testing. This is a safe way to have a dynamic or unknown number of tests.

- bail_out($reason)

  Invoked when something has gone horribly wrong: stop everything, kill all threads and processes, end the process with a false exit status.

ASSERTIONS

- ok($bool)
- ok($bool, $name)
- ok($bool, $name, @diag)

  Simple assertion. If `$bool` is true the test passes, and if it is false the test fails. The test name is optional, and all arguments after the name are added as diagnostics message if and only if the test fails. If the test passes all the diagnostics arguments will be ignored.

- pass()
- pass($name)

  Fire off a passing test (a single Ok event). The name is optional

- fail()
- fail($name)
- fail($name, @diag)

  Fire off a failing test (a single Ok event). The name and diagnostics are optional.

DIAGNOSTICS

- diag(@messages)

  Write diagnostics messages. All items in `@messages` will be joined into a single string with no separator. When using TAP, diagnostics are sent to STDERR.

  Returns false, so as to preserve failure.

- note(@messages)

  Write note-diagnostics messages. All items in `@messages` will be joined into a single string with no separator. When using TAP, notes are sent to STDOUT.

META

- $todo = todo($reason)

- todo $reason => sub { ... }

  This is used to mark some results as TODO. TODO means that the test may fail, but will not cause the overall test suite to fail.

  There are two ways to use this. The first is to use a codeblock, and the TODO will only apply to the codeblock.

```
ok(1, "before"); # Not TODO

todo 'this will fail' => sub {
    # This is TODO, as is any other test in this block.
    ok(0, "blah");
};

ok(1, "after"); # Not TODO
```

  The other way is to use a scoped variable. TODO will end when the variable is destroyed or set to undef.

```
ok(1, "before"); # Not TODO

{
    my $todo = todo 'this will fail';

    # This is TODO, as is any other test in this block.
    ok(0, "blah");
};

ok(1, "after"); # Not TODO
```

  This is the same thing, but without the `{...}` scope.

```
ok(1, "before"); # Not TODO

my $todo = todo 'this will fail';

ok(0, "blah"); # TODO

$todo = undef;

ok(1, "after"); # Not TODO
```

- skip($why)
- skip($why, $count)

  This is used to skip some tests. This requires you to wrap your tests in a block labeled `SKIP:`. This is somewhat magical. If no `$count` is specified then it will issue a single result. If you specify `$count` it will issue that many results.

```
SKIP: {
    skip "This will wipe your drive";
```

(continues on next page)

(continued from previous page)

```
    # This never gets run:
    ok(!system('sudo rm -rf /'), "Wipe drive");
}
```

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Tools::Class

Test2 implementation of the tools for testing classes.

DESCRIPTION

Test2 based tools for validating classes and objects. These are similar to some tools from *Test::More*, but they have a more consistent interface.

SYNOPSIS

```
use Test2::Tools::Class;

isa_ok($CLASS_OR_INSTANCE, $PARENT_CLASS1, $PARENT_CLASS2, ...);
isa_ok($CLASS_OR_INSTANCE, [$PARENT_CLASS1, $PARENT_CLASS2, ...],
 ↪"Test Name");

can_ok($CLASS_OR_INSTANCE, $METHOD1, $METHOD2, ...);
can_ok($CLASS_OR_INSTANCE, [$METHOD1, $METHOD2, ...], "Test Name");

DOES_ok($CLASS_OR_INSTANCE, $ROLE1, $ROLE2, ...);
DOES_ok($CLASS_OR_INSTANCE, [$ROLE1, $ROLE2, ...], "Test Name");
```

EXPORTS

All subs are exported by default.

- can_ok($thing, @methods)
- can_ok($thing, \@methods, $test_name)

  This checks that `$thing` (either a class name, or a blessed instance) has the specified methods.

  If the second argument is an arrayref then it will be used as the list of methods leaving the third argument to be the test name.

- isa_ok($thing, @classes)
- isa_ok($thing, \@classes, $test_name)

  This checks that `$thing` (either a class name, or a blessed instance) is or subclasses the specified classes.

  If the second argument is an arrayref then it will be used as the list of classes leaving the third argument to be the test name.

- DOES_ok($thing, @roles)
- DOES_ok($thing, \@roles, $test_name)

  This checks that `$thing` (either a class name, or a blessed instance) does the specified roles.

  If the second argument is an arrayref then it will be used as the list of roles leaving the third argument to be the test name.

  Note 1: This uses the `$class->DOES(...)` method, not the `does()` method Moose provides.

  Note 2: Not all perls have the `DOES()` method, if you use this on those perls the test will be skipped.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Tools::ClassicCompare

Classic (Test::More style) comparison tools.

DESCRIPTION

This provides comparison functions that behave like they did in *Test::More*, unlike the *Test2::Tools::Compare* plugin which has modified them.

SYNOPSIS

```
use Test2::Tools::ClassicCompare qw/is is_deeply isnt like unlike␣
↪cmp_ok/;

is($got, $expect, "These are the same when stringified");
isnt($got, $unexpected, "These are not the same when stringified");

like($got, qr/.../, "'got' matches the pattern");
unlike($got, qr/.../, "'got' does not match the pattern");

is_deeply($got, $expect, "These structures are same when checked␣
↪deeply");

cmp_ok($GOT, $OP, $WANT, 'Compare these items using the specified␣
↪operatr');
```

EXPORTS

- $bool = is($got, $expect)
- $bool = is($got, $expect, $name)
- $bool = is($got, $expect, $name, @diag)

  This does a string comparison of the two arguments. If the two arguments are the same after stringification the test passes. The test will also pass if both arguments are undef.

  The test `$name` is optional.

  The test `@diag` is optional, it is extra diagnostics messages that will be displayed if the test fails. The diagnostics are ignored if the test passes.

  It is important to note that this tool considers `"1"` and `"1.0"` to not be equal as it uses a string comparison.

See *Test2::Tools::Compare* if you want an `is()` function that tries to be smarter for you.

- $bool = isnt($got, $dont_expect)
- $bool = isnt($got, $dont_expect, $name)
- $bool = isnt($got, $dont_expect, $name, @diag)

  This is the inverse of `is()`, it passes when the strings are not the same.

- $bool = like($got, $pattern)
- $bool = like($got, $pattern, $name)
- $bool = like($got, $pattern, $name, @diag)

  Check if `$got` matches the specified pattern. Will fail if it does not match.

  The test `$name` is optional.

  The test `@diag` is optional. It contains extra diagnostics messages that will be displayed if the test fails. The diagnostics are ignored if the test passes.

- $bool = unlike($got, $pattern)
- $bool = unlike($got, $pattern, $name)
- $bool = unlike($got, $pattern, $name, @diag)

  This is the inverse of `like()`. This will fail if `$got` matches `$pattern`.

- $bool = is_deeply($got, $expect)
- $bool = is_deeply($got, $expect, $name)
- $bool = is_deeply($got, $expect, $name, @diag)

  This does a deep check, comparing the structures in `$got` with those in `$expect`. It will recurse into hashrefs, arrayrefs, and scalar refs. All other values will be stringified and compared as strings. It is important to note that this tool considers `"1"` and `"1.0"` to not be equal as it uses a string comparison.

  This is the same as `Test2::Tools::Compare::is()`.

- cmp_ok($got, $op, $expect)
- cmp_ok($got, $op, $expect, $name)
- cmp_ok($got, $op, $expect, $name, @diag)

  Compare `$got` to `$expect` using the operator specified in `$op`. This is effectively an `eval "\$got $op \$expect"` with some other stuff to make it more sane. This is useful for comparing numbers, overloaded objects, etc.

  Overloading Note: Your input is passed as-is to the comparison. If the comparison fails between two overloaded objects, the diagnostics will try to show you the overload form that was used in comparisons. It is possible that the diagnostics will be wrong, though attempts have been made to improve them since *Test::More*.

  Exceptions: If the comparison results in an exception then the test will fail and the exception will be shown.

`cmp_ok()` has an internal list of operators it supports. If you provide an unsupported operator it will issue a warning. You can add operators to the `%Test2::Tools::ClassicCompare::OPS` hash, the key should be the operator, and the value should either be 'str' for string comparison operators, 'num' for numeric operators, or any other true value for other operators.

Supported operators:

- == (num)
- != (num)
- = (num)
- <= (num)
- (num)
- < (num)
- <=> (num)
- eq (str)
- ne (str)
- gt (str)
- lt (str)
- ge (str)
- le (str)
- cmp (str)
- !~ (str)
- =~ (str)
- &&
- ||
- xor
- or
- and
- //
- &
- |
- ~~

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Tools::Compare

Tools for comparing deep data structures.

DESCRIPTION

Test::More had `is_deeply()`. This library is the *Test2* version that can be used to compare data structures, but goes a step further in that it provides tools for building a data structure specification against which you can verify your data. There are both 'strict' and 'relaxed' versions of the tools.

SYNOPSIS

```
use Test2::Tools::Compare;

# Hash for demonstration purposes
my $some_hash = {a => 1, b => 2, c => 3};

# Strict checking, everything must match
is(
    $some_hash,
    {a => 1, b => 2, c => 3},
    "The hash we got matches our expectations"
);

# Relaxed Checking, only fields we care about are checked, and we
 ↪can use a
# regex to approximate a field.
like(
    $some_hash,
    {a => 1, b => qr/\A[0-9]+\z/},
    "'a' is 1, 'b' is an integer, we don't care about 'c'."
);
```

ADVANCED

Declarative hash, array, and objects builders are available that allow you to generate specifications. These are more verbose than simply providing a hash, but have the advantage that every component you specify has a line number associated. This is helpful for debugging as the failure output will tell you not only which fields was incorrect, but also the line on which you declared the field.

```
use Test2::Tools::Compare qw{
    is like isnt unlike
    match mismatch validator
    hash array bag object meta number float rounded within string
 ↪subset bool
    in_set not_in_set check_set
    item field call call_list call_hash prop check all_items all_
 ↪keys all_vals all_values
    etc end filter_items
    T F D DF E DNE FDNE U L
    event fail_events
    exact_ref
};

is(
    $some_hash,
    hash {
        field a => 1;
        field b => 2;
        field c => 3;
    },
    "Hash matches spec"
);
```

COMPARISON TOOLS

- $bool = is($got, $expect)
- $bool = is($got, $expect, $name)
- $bool = is($got, $expect, $name, @diag)

  `$got` is the data structure you want to check. `$expect` is what you want `$got` to look like. `$name` is an optional name for the test. `@diag` is optional diagnostics messages that will be printed to STDERR in event of failure, they will not be displayed when the comparison is successful. The boolean true/false result of the comparison is returned.

  This is the strict checker. The strict checker requires a perfect match between `$got` and `$expect`. All hash fields must be specified, all array items must be present, etc. All non-scalar/hash/array/regex references must be identical (same memory address). Scalar, hash and array references will be traversed and compared. Regex references will be compared to see if they have the same pattern.

```
is(
    $some_hash,
    {a => 1, b => 2, c => 3},
    "The hash we got matches our expectations"
);
```

The only exception to strictness is when it is given an `$expect` object that was built from a specification, in which case the specification determines the strictness. Strictness only applies to literal values/references that are provided and converted to a specification for you.

```
is(
    $some_hash,
    hash {    # Note: the hash function is not exported by␣
↪default
        field a => 1;
        field b => match(qr/\A[0-9]+\z/);    # Note: The match␣
↪function is not exported by default
        # Don't care about other fields.
    },
    "The hash comparison is not strict"
);
```

This works for both deep and shallow structures. For instance you can use this to compare two strings:

```
is('foo', 'foo', "strings match");
```

Note: This is not the tool to use if you want to check if two references are the same exact reference, use `ref_is()` from the *Test2::Tools::Ref* plugin instead. *Most* of the time this will work as well, however there are problems if your reference contains a cycle and refers back to itself at some point. If this happens, an exception will be thrown to break an otherwise infinite recursion.

Note: Non-reference values will be compared as strings using eq, so that means strings '2.0' and '2' will not match, but numeric 2.0 and 2 will, since they are both stringified to '2'.

- $bool = isnt($got, $expect)
- $bool = isnt($got, $expect, $name)
- $bool = isnt($got, $expect, $name, @diag)

  Opposite of `is()`. Does all the same checks, but passes when there is a mismatch.

- $bool = like($got, $expect)
- $bool = like($got, $expect, $name)
- $bool = like($got, $expect, $name, @diag)

  `$got` is the data structure you want to check. `$expect` is what you want `$got` to look like. `$name` is an optional name for the test. `@diag` is optional diagnostics messages that will be printed to STDERR in event of failure, they will not be displayed

when the comparison is successful. The boolean true/false result of the comparison is returned.

This is the relaxed checker. This will ignore hash keys or array indexes that you do not actually specify in your `$expect` structure. In addition regex and sub references will be used as validators. If you provide a regex using `qr/.../`, the regex itself will be used to validate the corresponding value in the `$got` structure. The same is true for coderefs, the value is passed in as the first argument (and in `$_`) and the sub should return a boolean value. In this tool regexes will stringify the thing they are checking.

```
like(
    $some_hash,
    {a => 1, b => qr/\A[0-9]+\z/},
    "'a' is 1, 'b' is an integer, we don't care about other␣
↪fields"
);
```

This works for both deep and shallow structures. For instance you can use this to compare two strings:

```
like('foo bar', qr/^foo/, "string matches the pattern");
```

- $bool = unlike($got, $expect)
- $bool = unlike($got, $expect, $name)
- $bool = unlike($got, $expect, $name, @diag)

  Opposite of `like()`. Does all the same checks, but passes when there is a mismatch.

The `is()`, `isnt()`, `like()`, and `unlike()` functions can be made to dump `$got` using *Data::Dumper* when tests fail by setting the `T2_AUTO_DUMP` environment variable to "1". (Alternatively, `T2_AUTO_DUMP` can be set to the name of a Perl module providing a compatible `Dump()` method.) The `T2_AUTO_DEPARSE` environment variable can be used to enable Data::Dumper's deparsing of coderefs.

QUICK CHECKS

Note: None of these are exported by default. You need to request them.

Quick checks are a way to quickly generate a common value specification. These can be used in structures passed into `is` and `like` through the `$expect` argument.

Example:

```
is($foo, T(), '$foo has a true value');
```

- $check = T()

  This verifies that the value in the corresponding `$got` structure is true, any true value will do.

```
is($foo, T(), '$foo has a true value');
```

(continues on next page)

(continued from previous page)

```
is(
    { a => 'xxx' },
    { a => T() },
    "The 'a' key is true"
);
```

- $check = F()

  This verifies that the value in the corresponding `$got` structure is false, any false value will do, **but the value must exist**.

```
is($foo, F(), '$foo has a false value');

is(
    { a => 0 },
    { a => F() },
    "The 'a' key is false"
);
```

  It is important to note that a nonexistent value does not count as false. This check will generate a failing test result:

```
is(
    { a => 1 },
    { a => 1, b => F() },
    "The 'b' key is false"
);
```

  This will produce the following output:

```
not ok 1 - The b key is false
# Failed test "The 'b' key is false"
# at some_file.t line 10.
# +------+------------------+-------+---------+
# | PATH | GOT              | OP    | CHECK   |
# +------+------------------+-------+---------+
# | {b}  | <DOES NOT EXIST> | FALSE | FALSE() |
# +------+------------------+-------+---------+
```

  In Perl, you can have behavior that is different for a missing key vs. a false key, so it was decided not to count a completely absent value as false. See the `DNE()` shortcut below for checking that a field is missing.

  If you want to check for false and/or DNE use the `FDNE()` check.

- $check = D()

  This is to verify that the value in the `$got` structure is defined. Any value other than `undef` will pass.

  This will pass:

```
is('foo', D(), 'foo is defined');
```

This will fail:

```
is(undef, D(), 'foo is defined');
```

- $check = U()

  This is to verify that the value in the `$got` structure is undefined.

  This will pass:

```
is(undef, U(), 'not defined');
```

  This will fail:

```
is('foo', U(), 'not defined');
```

- $check = DF()

  This is to verify that the value in the `$got` structure is defined but false. Any false value other than `undef` will pass.

  This will pass:

```
is(0, DF(), 'foo is defined but false');
```

  These will fail:

```
is(undef, DF(), 'foo is defined but false');
is(1, DF(), 'foo is defined but false');
```

- $check = E()

  This can be used to check that a value exists. This is useful to check that an array has more values, or to check that a key exists in a hash, even if the value is undefined.

  These pass:

```
is(['a', 'b', undef], ['a', 'b', E()], "There is a third item␣
↪in the array");
is({a => 1, b => 2}, {a => 1, b => E()}, "The 'b' key exists in␣
↪the hash");
```

  These will fail:

```
is(['a', 'b'], ['a', 'b', E()], "Third item exists");
is({a => 1}, {a => 1, b => E()}, "'b' key exists");
```

- $check = DNE()

  This can be used to check that no value exists. This is useful to check the end bound of an array, or to check that a key does not exist in a hash.

  These pass:

```
is(['a', 'b'], ['a', 'b', DNE()], "There is no third item in␣
↪the array");
is({a => 1}, {a => 1, b => DNE()}, "The 'b' key does not exist␣
↪in the hash");
```

These will fail:

```
is(['a', 'b', 'c'], ['a', 'b', DNE()], "No third item");
is({a => 1, b => 2}, {a => 1, b => DNE()}, "No 'b' key");
```

- $check = FDNE()

  This is a combination of `F()` and `DNE()`. This will pass for a false value, or a nonexistent value.

- $check = L()

  This is to verify that the value in the `$got` structure is defined and has length. Any value other than `undef` or the empty string will pass (including references).

  These will pass:

```
is('foo', L(), 'value is defined and has length');
is([],    L(), 'value is defined and has length');
```

  These will fail:

```
is(undef, L(), 'value is defined and has length');
is('',    L(), 'value is defined and has length');
```

VALUE SPECIFICATIONS

Note: None of these are exported by default. You need to request them.

- $check = string "..."

  Verify that the value matches the given string using the eq operator.

- $check = !string "..."

  Verify that the value does not match the given string using the ne operator.

- $check = number ...;

  Verify that the value matches the given number using the == operator.

- $check = !number ...;

  Verify that the value does not match the given number using the != operator.

- $check = number_lt ...;
- $check = number_le ...;
- $check = number_ge ...;
- $check = number_gt ...;

Verify that the value is less than, less than or equal to, greater than or equal to, or greater than the given number.

- $check = float ...;

  Verify that the value is approximately equal to the given number.

  If a 'precision' parameter is specified, both operands will be rounded to 'precision' number of fractional decimal digits and compared with eq.

```
is($near_val, float($val, precision => 4), "Near 4 decimal␣
↪digits");
```

  Otherwise, the check will be made within a range of +/- 'tolerance', with a default 'tolerance' of 1e-08.

```
is( $near_val, float($val, tolerance => 0.01), "Almost there...
↪");
```

  See also `within` and `rounded`.

- $check = !float ...;

  Verify that the value is not approximately equal to the given number.

  If a 'precision' parameter is specified, both operands will be rounded to 'precision' number of fractional decimal digits and compared with eq.

  Otherwise, the check will be made within a range of +/- 'tolerance', with a default 'tolerance' of 1e-08.

  See also `!within` and `!rounded`.

- $check = within($num, $tolerance);

  Verify that the value approximately matches the given number, within a range of +/- `$tolerance`. Compared using the == operator.

  `$tolerance` is optional and defaults to 1e-08.

- $check = !within($num, $tolerance);

  Verify that the value does not approximately match the given number within a range of +/- `$tolerance`. Compared using the != operator.

  `$tolerance` is optional and defaults to 1e-08.

- $check = rounded($num, $precision);

  Verify that the value approximately matches the given number, when both are rounded to `$precision` number of fractional digits. Compared using the eq operator.

- $check = !rounded($num, $precision);

  Verify that the value does not approximately match the given number, when both are rounded to `$precision` number of fractional digits. Compared using the ne operator.

- $check = bool ...;

  Verify the value has the same boolean value as the given argument (XNOR).

- $check = !bool ...;

  Verify the value has a different boolean value from the given argument (XOR).

- $check = check_isa ...;

  Verify the value is an instance of the given class name.

- $check = !check_isa ...;

  Verify the value is not an instance of the given class name.

- $check = match qr/.../

- $check = !mismatch qr/.../

  Verify that the value matches the regex pattern. This form of pattern check will **NOT** stringify references being checked.

  Note: `!mismatch()` is documented for completion, please do not use it.

- $check = !match qr/.../

- $check = mismatch qr/.../

  Verify that the value does not match the regex pattern. This form of pattern check will **NOT** stringify references being checked.

  Note: `mismatch()` was created before overloading of `!` for `match()` was a thing.

- $check = validator(sub{ ... })

- $check = validator($NAME => sub{ ... })

- $check = validator($OP, $NAME, sub{ ... })

  The coderef is the only required argument. The coderef should check that the value is what you expect and return a boolean true or false. Optionally, you can specify a name and operator that are used in diagnostics. They are also provided to the sub itself as named parameters.

  Check the value using this sub. The sub gets the value in $_, and it receives the value and several other items as named parameters.

```
my $check = validator(sub {
    my %params = @_;

    # These both work:
    my $got = $_;
    my $got = $params{got};

    # Check if a value exists at all
    my $exists = $params{exists}

    # What $OP (if any) did we specify when creating the␣
↪validator
    my $operator = $params{operator};

    # What name (if any) did we specify when creating the␣
↪validator
```

(continues on next page)

(continued from previous page)

```
    my $name = $params{name};

    ...

    return $bool;
}
```

- $check = exact_ref($ref)

  Check that the value is exactly the same reference as the one provided.

SET BUILDERS

Note: None of these are exported by default. You need to request them.

- my $check = check_set($check1, $check2, ...)

  Check that the value matches ALL of the specified checks.

- my $check = in_set($check1, $check2, ...)

  Check that the value matches ONE OR MORE of the specified checks.

- not_in_set($check1, $check2, ...)

  Check that the value DOES NOT match ANY of the specified checks.

- check $thing

  Check that the value matches the specified thing.

HASH BUILDER

Note: None of these are exported by default. You need to request them.

```
$check = hash {
    field foo => 1;
    field bar => 2;

    # Ensure the 'baz' keys does not even exist in the hash.
    field baz => DNE();

    # Ensure the key exists, but is set to undef
    field bat => undef;

    # Any check can be used
    field boo => $check;

    # Set checks that apply to all keys or values. Can be done␣
↪multiple
    # times, and each call can define multiple checks, all will be␣
↪run.
    all_vals match qr/a/, match qr/b/;    # All values must have an
```

(continues on next page)

(continued from previous page)

```
↪'a' and a 'b'
    all_keys match qr/x/;                  # All keys must have an 'x
↪'

    ...

    end(); # optional, enforces that no other keys are present.
};
```

- $check = hash { ... }

  This is used to define a hash check.

- field $NAME => $VAL
- field $NAME => $CHECK

  Specify a field check. This will check the hash key specified by `$NAME` and ensure it matches the value in `$VAL`. You can put any valid check in `$VAL`, such as the result of another call to `array { ... }`, `DNE()`, etc.

  Note: This function can only be used inside a hash builder sub, and must be called in void context.

- all_keys($CHECK1, $CHECK2, ...)

  Add checks that apply to all keys. You can put this anywhere in the hash block, and can call it any number of times with any number of arguments.

- all_vals($CHECK1, $CHECK2, ...)
- all_values($CHECK1, $CHECK2, ...)

  Add checks that apply to all values. You can put this anywhere in the hash block, and can call it any number of times with any number of arguments.

- end()

  Enforce that no keys are found in the hash other than those specified. This is essentially the `use strict` of a hash check. This can be used anywhere in the hash builder, though typically it is placed at the end.

- etc()

  Ignore any extra keys found in the hash. This is the opposite of `end()`. This can be used anywhere in the hash builder, though typically it is placed at the end.

- DNE()

  This is a handy check that can be used with `field()` to ensure that a field (D)oes (N)ot (E)xist.

  ```
  field foo => DNE();
  ```

ARRAY BUILDER

Note: None of these are exported by default. You need to request them.

```
$check = array {
    # Uses the next index, in this case index 0;
    item 'a';

    # Gets index 1 automatically
    item 'b';

    # Specify the index
    item 2 => 'c';

    # We skipped index 3, which means we don't care what it is.
    item 4 => 'e';

    # Gets index 5.
    item 'f';

    # Remove any REMAINING items that contain 0-9.
    filter_items { grep {!m/[0-9]/} @_ };

    # Set checks that apply to all items. Can be done multiple
↪times, and
    # each call can define multiple checks, all will be run.
    all_items match qr/a/, match qr/b/;
    all_items match qr/x/;

    # Of the remaining items (after the filter is applied) the next
↪one
    # (which is now index 6) should be 'g'.
    item 6 => 'g';

    item 7 => DNE; # Ensure index 7 does not exist.

    end(); # Ensure no other indexes exist.
};
```

- $check = array { ... }
- item $VAL
- item $CHECK
- item $IDX, $VAL
- item $IDX, $CHECK

  Add an expected item to the array. If `$IDX` is not specified it will automatically calculate it based on the last item added. You can skip indexes, which means you do not want them to be checked.

  You can provide any value to check in `$VAL`, or you can provide any valid check object.

Note: Items MUST be added in order.

Note: This function can only be used inside an array, bag or subset builder sub, and must be called in void context.

- filter_items { my @remaining = @_; ...; return @filtered }

  This function adds a filter, all items remaining in the array from the point the filter is reached will be passed into the filter sub as arguments, the sub should return only the items that should be checked.

  Note: This function can only be used inside an array builder sub, and must be called in void context.

- all_items($CHECK1, $CHECK2, ...)

  Add checks that apply to all items. You can put this anywhere in the array block, and can call it any number of times with any number of arguments.

- end()

  Enforce that there are no indexes after the last one specified. This will not force checking of skipped indexes.

- etc()

  Ignore any extra items found in the array. This is the opposite of `end()`. This can be used anywhere in the array builder, though typically it is placed at the end.

- DNE()

  This is a handy check that can be used with `item()` to ensure that an index (D)oes (N)ot (E)xist.

```
item 5 => DNE();
```

BAG BUILDER

Note: None of these are exported by default. You need to request them.

```
$check = bag {
    item 'a';
    item 'b';

    end(); # Ensure no other elements exist.
};
```

A bag is like an array, but we don't care about the order of the items. In the example, `$check` would match both `['a','b']` and `['b','a']`.

- $check = bag { ... }
- item $VAL
- item $CHECK

  Add an expected item to the bag.

  You can provide any value to check in `$VAL`, or you can provide any valid check object.

Note: This function can only be used inside an array, bag or subset builder sub, and must be called in void context.

- all_items($CHECK1, $CHECK2, ...)

  Add checks that apply to all items. You can put this anywhere in the bag block, and can call it any number of times with any number of arguments.

- end()

  Enforce that there are no more items after the last one specified.

- etc()

  Ignore any extra items found in the array. This is the opposite of `end()`. This can be used anywhere in the bag builder, though typically it is placed at the end.

ORDERED SUBSET BUILDER

Note: None of these are exported by default. You need to request them.

```
$check = subset {
    item 'a';
    item 'b';
    item 'c';

    # Doesn't matter if the array has 'd', the check will skip past␣
↪any
    # unknown items until it finds the next one in our subset.

    item 'e';
    item 'f';
};
```

- $check = subset { ... }
- item $VAL
- item $CHECK

  Add an expected item to the subset.

  You can provide any value to check in `$VAL`, or you can provide any valid check object.

  Note: Items MUST be added in order.

  Note: This function can only be used inside an array, bag or subset builder sub, and must be called in void context.

META BUILDER

Note: None of these are exported by default. You need to request them.

```
my $check = meta {
    prop blessed => 'My::Module'; # Ensure value is blessed as our␣
```

(continues on next page)

(continued from previous page)

```
↪package
    prop reftype => 'HASH';       # Ensure value is a blessed hash
    prop isa     => 'My::Base';   # Ensure value is an instance of␣
↪our class
    prop size    => 4;            # Check the number of hash keys
    prop this    => ...;          # Check the item itself
};
```

- meta { ... }
- meta_check { ... }

  Build a meta check. If you are using *Moose* then the `meta()` function would conflict with the one exported by *Moose*, in such cases `meta_check()` is available. Neither is exported by default.

- prop $NAME => $VAL
- prop $NAME => $CHECK

  Check the property specified by `$name` against the value or check.

  Valid properties are:

  – 'blessed'

    What package (if any) the thing is blessed as.

  – 'reftype'

    Reference type (if any) the thing is.

  – 'isa'

    What class the thing is an instance of.

  – 'this'

    The thing itself.

  – 'size'

    For array references this returns the number of elements. For hashes this returns the number of keys. For everything else this returns undef.

OBJECT BUILDER

Note: None of these are exported by default. You need to request them.

```
my $check = object {
    call foo => 1; # Call the 'foo' method, check the result.

    # Call the specified sub-ref as a method on the object, check␣
↪the
    # result. This is useful for wrapping methods that return␣
↪multiple
    # values.
```

(continues on next page)

(continued from previous page)

```
    call sub { [ shift->get_list ] } => [...];

    # This can be used to ensure a method does not exist.
    call nope => DNE();

    # Check the hash key 'foo' of the underlying reference, this
↪only works
    # on blessed hashes.
    field foo => 1;

    # Check the value of index 4 on the underlying reference, this
↪only
    # works on blessed arrays.
    item 4 => 'foo';

    # Check the meta-property 'blessed' of the object.
    prop blessed => 'My::Module';

    # Check if the object is an instance of the specified class.
    prop isa => 'My::Base';

    # Ensure only the specified hash keys or array indexes are
↪present in
    # the underlying hash. Has no effect on meta-property checks or
↪method
    # checks.
    end();
};
```

- $check = object { ... }

  Specify an object check for use in comparisons.

- call $METHOD_NAME => $RESULT
- call $METHOD_NAME => $CHECK
- call [$METHOD_NAME, @METHOD_ARGS] => $RESULT
- call [$METHOD_NAME, @METHOD_ARGS] => $CHECK
- call sub { ... }, $RESULT
- call sub { ... }, $CHECK

  Call the specified method (or coderef) and verify the result. If you pass an arrayref, the first element must be the method name, the others are the arguments it will be called with.

  The coderef form is useful if you need to do something more complex.

```
my $ref = sub {
  local $SOME::GLOBAL::THING = 3;
  return [shift->get_values_for('thing')];
```

(continues on next page)

(continued from previous page)

```
};

call $ref => ...;
```

- call_list $METHOD_NAME => $RESULT
- call_list $METHOD_NAME => $CHECK
- call_list [$METHOD_NAME, @METHOD_ARGS] => $RESULT
- call_list [$METHOD_NAME, @METHOD_ARGS] => $CHECK
- call_list sub { ... }, $RESULT
- call_list sub { ... }, $CHECK

  Same as `call`, but the method is invoked in list context, and the result is always an arrayref.

```
call_list get_items => [ ... ];
```

- call_hash $METHOD_NAME => $RESULT
- call_hash $METHOD_NAME => $CHECK
- call_hash [$METHOD_NAME, @METHOD_ARGS] => $RESULT
- call_hash [$METHOD_NAME, @METHOD_ARGS] => $CHECK
- call_hash sub { ... }, $RESULT
- call_hash sub { ... }, $CHECK

  Same as `call`, but the method is invoked in list context, and the result is always a hashref. This will warn if the method returns an odd number of values.

```
call_hash get_items => { ... };
```

- field $NAME => $VAL

  Works just like it does for hash checks.

- item $VAL
- item $IDX, $VAL

  Works just like it does for array checks.

- prop $NAME => $VAL
- prop $NAME => $CHECK

  Check the property specified by `$name` against the value or check.

  Valid properties are:

  – 'blessed'

    What package (if any) the thing is blessed as.

  - 'reftype'

    Reference type (if any) the thing is.

  - 'isa'

    What class the thing is an instance of.

  - 'this'

    The thing itself.

  - 'size'

    For array references this returns the number of elements. For hashes this returns the number of keys. For everything else this returns undef.

- DNE()

  Can be used with `item`, or `field` to ensure the hash field or array index does not exist. Can also be used with `call` to ensure a method does not exist.

- end()

  Turn on strict array/hash checking, ensuring that no extra keys/indexes are present.

- etc()

  Ignore any extra items found in the hash/array. This is the opposite of `end()`. This can be used anywhere in the builder, though typically it is placed at the end.

EVENT BUILDERS

Note: None of these are exported by default. You need to request them.

Check that we got an event of a specified type:

```
my $check = event 'Ok';
```

Check for details about the event:

```
my $check = event Ok => sub {
    # Check for a failure
    call pass => 0;

    # Effective pass after TODO/SKIP are accounted for.
    call effective_pass => 1;

    # Check the diagnostics
    call diag => [ match qr/Failed test foo/ ];

    # Check the file the event reports to
    prop file => 'foo.t';

    # Check the line number the event reports to
    prop line => '42';
```

(continues on next page)

(continued from previous page)

```
    # You can check the todo/skip values as well:
    prop skip => 'broken';
    prop todo => 'fixme';

    # Thread-id and process-id where event was generated
    prop tid => 123;
    prop pid => 123;
};
```

You can also provide a fully qualified event package with the '+' prefix:

```
my $check = event '+My::Event' => sub { ... }
```

You can also provide a hashref instead of a sub to directly check hash values of the event:

```
my $check = event Ok => { pass => 1, ... };
```

USE IN OTHER BUILDERS

You can use these all in other builders, simply use them in void context to have their value(s) appended to the build.

```
my $check = array {
    event Ok => { ... };
    event Note => { ... };

    fail_events Ok => { pass => 0 };
    # Get a Diag for free.
};
```

SPECIFICS

- $check = event $TYPE;
- $check = event $TYPE => sub { ... };
- $check = event $TYPE => { ... };

  This works just like an object builder. In addition to supporting everything the object check supports, you also have to specify the event type, and many extra meta-properties are available.

  Extra properties are:

  - 'file'

    File name to which the event reports (for use in diagnostics).

  - 'line'

    Line number to which the event reports (for use in diagnostics).

  - 'package'

    Package to which the event reports (for use in diagnostics).

- 'subname'

  Sub that was called to generate the event (example: `ok()`).

- 'skip'

  Set to the skip value if the result was generated by skipping tests.

- 'todo'

  Set to the todo value if TODO was set when the event was generated.

- 'trace'

  The `at file foo.t line 42` string that will be used in diagnostics.

- 'tid'

  Thread ID in which the event was generated.

- 'pid'

  Process ID in which the event was generated.

NOTE: Event checks have an implicit `etc()` added. This means you need to use `end()` if you want to fail on unexpected hash keys or array indexes. This implicit `etc()` extends to all forms, including builder, hashref, and no argument.

- @checks = fail_events $TYPE;
- @checks = fail_events $TYPE => sub { ... };
- @checks = fail_events $TYPE => { ... };

  Just like `event()` documented above. The difference is that this produces two events, the one you specify, and a `Diag` after it. There are no extra checks in the Diag.

  Use this to validate a simple failure where you do not want to be bothered with the default diagnostics. It only adds a single Diag check, so if your failure has custom diagnostics you will need to add checks for them.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Tools::Defer

Write tests that get executed at a later time

DESCRIPTION

Sometimes you need to test things BEFORE loading the necessary functions. This module lets you do that. You can write tests, and then have them run later, after `Test2` is loaded. You tell it what test function to run, and what arguments to give it. The function name and arguments will be stored to be executed later. When ready, run `do_def()` to kick them off once the functions are defined.

SYNOPSIS

```
use strict;
use warnings;

use Test2::Tools::Defer;

BEGIN {
    def ok => (1, 'pass');
    def is => ('foo', 'foo', 'runs is');
    ...
}

use Test2::Tools::Basic;

do_def(); # Run the tests

# Declare some more tests to run later:
def ok => (1, "another pass");
...

do_def(); # run the new tests

done_testing;
```

EXPORTS

- def function => @args;

  This will store the function name, and the arguments to be run later. Note that each package has a separate store of tests to run.

- do_def()

  This will run all the stored tests. It will also reset the list to be empty so you can add more tests to run even later.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Tools::Encoding

Tools for managing the encoding of *Test2* based tests.

DESCRIPTION

This module exports a function that lets you dynamically change the output encoding at will.

SYNOPSIS

```
use Test2::Tools::Encoding;

set_encoding('utf8');
```

EXPORTS

All subs are exported by default.

- set_encoding($encoding)

  This will set the encoding to whatever you specify. This will only affect the output of the current formatter, which is usually your TAP output formatter.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Tools::Event

Tools for generating test events.

DESCRIPTION

This module provides tools for generating events quickly by bypassing the context/hub. This is particularly useful when testing other *Test2* packages.

EXPORTS

- $e = gen_event($TYPE)
- $e = gen_event($TYPE, %FIELDS)
- $e = gen_event 'Ok';
- $e = gen_event Ok => (...)
- $e = gen_event '+Test2::Event::Ok' => (...)

  This will produce an event of the specified type. `$TYPE` is assumed to be shorthand for `Test2::Event::$TYPE`, you can prefix `$TYPE` with a '+' to drop the assumption.

  An *Test2::Util::Trace* will be generated using `caller(0)` and will be put in the 'trace' field of your new event, unless you specified your own 'trace' field.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Tools::Exception

Test2 based tools for checking exceptions

DESCRIPTION

This is the *Test2* implementation of code used to test exceptions. This is similar to Test::Fatal, but it intentionally does much less.

SYNOPSIS

```
use Test2::Tools::Exception qw/dies lives/;

like(
    dies { die 'xxx' },
    qr/xxx/,
    "Got exception"
);

ok(lives { ... }, "did not die") or note($@);
```

EXPORTS

All subs are exported by default.

- $e = dies { ... }

  This will trap any exception the codeblock throws. If no exception is thrown the sub will return undef. If an exception is thrown it will be returned. This function preserves $@, it will not be altered from its value before the sub is called.

- $bool = lives { ... }

  This will trap any exception thrown in the codeblock. It will return true when there is no exception, and false when there is. $@ is preserved from before the sub

is called when there is no exception. When an exception is trapped `$@` will have the exception so that you can look at it.

- $bool = try_ok { ... }
- $bool = try_ok { ... } "Test Description"

  This will run the code block trapping any exception. If there is no exception a passing event will be issued. If the test fails a failing event will be issued, and the exception will be reported as diagnostics.

  Note: This function does not preserve `$@` on failure, it will be set to the exception the codeblock throws, this is by design so that you can obtain the exception if desired.

DIFFERENCES FROM TEST::FATAL

Test::Fatal sets `$Test::Builder::Level` such that failing tests inside the exception block will report to the line where `exception()` is called. I disagree with this, and think the actual line of the failing test is more important. Ultimately, though Test::Fatal cannot be changed, people probably already depend on that behavior.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Tools::Exports

Tools for validating exporters.

DESCRIPTION

These are tools for checking that symbols have been imported into your namespace.

SYNOPSIS

```
use Test2::Tools::Exports

use Data::Dumper;
imported_ok qw/Dumper/;
not_imported_ok qw/dumper/;
```

EXPORTS

All subs are exported by default.

- imported_ok(@SYMBOLS)

  Check that the specified symbols exist in the current package. This will not find inherited subs. This will only find symbols in the current package's symbol table. This **WILL NOT** confirm that the symbols were defined outside of the package itself.

  ```
  imported_ok( '$scalar', '@array', '%hash', '&sub', 'also_a_sub'␣
  ↪);
  ```

  `@SYMBOLS` can contain any number of symbols. Each item in the array must be a string. The string should be the name of a symbol. If a sigil is present then it will search for that specified type, if no sigil is specified it will be used as a sub name.

- not_imported_ok(@SYMBOLS)

  Check that the specified symbols do not exist in the current package. This will not find inherited subs. This will only look at symbols in the current package's symbol table.

  ```
  not_imported_ok( '$scalar', '@array', '%hash', '&sub', 'also_a_
  ↪sub' );
  ```

  `@SYMBOLS` can contain any number of symbols. Each item in the array must be a string. The string should be the name of a symbol. If a sigil is present, then it will search for that specified type. If no sigil is specified, it will be used as a sub name.

CAVEATS

Before Perl 5.10, it is very difficult to distinguish between a package scalar that is undeclared vs declared and undefined. Currently `imported_ok` and `not_imported_ok` cannot see package scalars declared using `our $var` unless the variable has been assigned a defined value.

This will pass on recent perls, but fail on perls older than 5.10:

```
use Test2::Tools::Exports;

our $foo;

# Fails on perl onlder than 5.10
imported_ok(qw/$foo/);
```

If `$foo` is imported from another module, or imported using `use vars qw/$foo/;` then it will work on all supported perl versions.

```
use Test2::Tools::Exports;

use vars qw/$foo/;
use Some::Module qw/$bar/;

# Always works
imported_ok(qw/$foo $bar/);
```

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Tools::GenTemp

Tool for generating a populated temp directory.

DESCRIPTION

This exports a tool that helps you make a temporary directory, nested directories and text files within.

SYNOPSIS

```
use Test2::Tools::GenTemp qw/gen_temp/;

my $dir = gen_temp(
    a_file => "Contents of a_file",
    a_dir  => {
        'a_file' => 'Contents of a_dir/afile',
        a_nested_dir => { ... },
    },
    ...
);

done_testing;
```

EXPORTS

All subs are exported by default.

- gen_temp(file => 'content', subdir => [sub_dir_file => 'content', ...], ...)
- gen_temp(-tempdir => \@TEMPDIR_ARGS, file => 'content', subdir => [sub_dir_file => 'content', ...], ...)

  This will generate a new temporary directory with all the files and subdirs you specify, recursively. The initial temp directory is created using `File::Temp::tempdir()`, you may pass arguments to tempdir using the `-tempdir => [...]` argument.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Tools::Grab

Temporarily intercept all events without adding a scope level.

DESCRIPTION

This package provides a function that returns an object that grabs all events. Once the object is destroyed events will once again be sent to the main hub.

SYNOPSIS

```
use Test2::Tools::Grab;

my $grab = grab();

# Generate some events, they are intercepted.
ok(1, "pass");
ok(0, "fail");

my $events_a = $grab->flush;

# Generate some more events, they are intercepted.
ok(1, "pass");
ok(0, "fail");

my $events_b = $grab->finish;
```

EXPORTS

- $grab = grab()

  This lets you intercept all events for a section of code without adding anything to your call stack. This is useful for things that are sensitive to changes in the stack depth.

  ```
  my $grab = grab();
      ok(1, 'foo');
      ok(0, 'bar');

  my $events = $grab->finish;

  is(@$events, 2, "grabbed 2 events.");
  ```

  If the `$grab` object is destroyed without calling `finish()`, it will automatically clean up after itself and restore the parent hub.

  ```
  {
      my $grab = grab();
      # Things are grabbed
  }
  # Things are back to normal
  ```

By default the hub used has no_ending set to true. This will prevent the hub from enforcing that you issued a plan and ran at least 1 test. You can turn enforcement back one like this:

```
$grab->hub->set_no_ending(0);
```

With no_ending turned off, finish will run the post-test checks to enforce the plan and that tests were run. In many cases this will result in additional events in your events array.

SEE ALSO

Test2::Util::Grabber - The object constructed and returned by this tool.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See http://dev.perl.org/licenses/

Test2::Tools::Mock

Class/Instance mocking for Test2.

DESCRIPTION

Mocking is often an essential part of testing. This library covers some of the most common mocking needs. This plugin is heavily influenced by Mock::Quick, but with an improved API. This plugin is also intended to play well with other plugins in ways Mock::Quick would be unable to.

SYNOPSIS

```
my $mock = mock 'Some::Class' => (
    track => $BOOL, # Enable/Disable tracking on subs defined below

    add => [
        new_method => sub { ... },
    ],
    override => [
        replace_method => sub { ... },
    ],
    set => [
        replace_or_inject => sub { ... },
    ],

    track => $bool, # enable/disable tracking again to affect mocks
 ↪made after this point
    ..., # Argument keys may be repeated
);

Some::Class->new_method();        # Calls the newly injected method
Some::Class->replace_method();    # Calls our replacement method.

$mock->override(...) # Override some more

$mock = undef; # Undoes all the mocking, restoring all original
 ↪methods.

my $simple_mock = mock {} => (
    add => [
        is_active => sub { ... }
    ]
);

$simple_mock->is_active();        # Calls our newly mocked method.
```

EXPORTS

DEFAULT

- mock

  This is a one-stop shop function that delegates to one of the other methods depending on how it is used. If you are not comfortable with a function that has a lot of potential behaviors, you can use one of the other functions directly.

- @mocks = mocked($object)

- @mocks = mocked($class)

  Check if an object or class is mocked. If it is mocked the `$mock` object(s) (*Test2::Mock*) will be returned.

- $mock = mock $class => (...);
- $mock = mock $instance => (...)
- $mock = mock 'class', $class => (...)

  These forms delegate to `mock_class()` to mock a package. The third form is to be explicit about what type of mocking you want.

- $obj = mock()
- $obj = mock { ... }
- $obj = mock 'obj', ...;

  These forms delegate to `mock_obj()` to create instances of anonymous packages where methods are vivified into existence as needed.

- mock $mock => sub { ... }
- mock $method => (...)

  These forms go together, the first form will set `$mock` as the current mock build, then run the sub. Within the sub you can declare mock specifications using the second form. The first form delegates to `mock_build()`.

  The second form calls the specified method on the current build. This second form delegates to `mock_do()`.

BY REQUEST

DEFINING MOCKS

- $obj = mock_obj(...)
- $obj = mock_obj { ... } => (...)
- $obj = mock_obj sub { ... }
- $obj = mock_obj { ... } => sub { ... }

  This method lets you quickly generate a blessed object. The object will be an instance of a randomly generated package name. Methods will vivify as read/write accessors as needed.

  Arguments can be any method available to *Test2::Mock* followed by an argument. If the very first argument is a hashref then it will be blessed as your new object.

  If you provide a coderef instead of key/value pairs, the coderef will be run to build the mock. (See the "BUILDING MOCKS" section).

- $mock = mock_class $class => (...)
- $mock = mock_class $instance => (...)
- $mock = mock_class ... => sub { ... }

  This will create a new instance of *Test2::Mock* to control the package specified. If you give it a blessed reference it will use the class of the instance.

  Arguments can be any method available to *Test2::Mock* followed by an argument. If the very first argument is a hashref then it will be blessed as your new object.

If you provide a coderef instead of key/value pairs, the coderef will be run to build the mock. (See the "BUILDING MOCKS" section).

BUILDING MOCKS

- mock_build $mock => sub { ... }

  Set `$mock` as the current build, then run the specified code. `$mock` will no longer be the current build when the sub is complete.

- $mock = mock_building()

  Get the current building `$mock` object.

- mock_do $method => $args

  Run the specified method on the currently building object.

METHOD GENERATORS

- $sub = mock_accessor $field

  Generate a read/write accessor for the specified field. This will generate a sub like the following:

```
$sub = sub {
    my $self = shift;
    ($self->{$field}) = @_ if @_;
    return $self->{$field};
};
```

- $sub = mock_getter $field

  Generate a read only accessor for the specified field. This will generate a sub like the following:

```
$sub = sub {
    my $self = shift;
    return $self->{$field};
};
```

- $sub = mock_setter $field

  Generate a write accessor for the specified field. This will generate a sub like the following:

```
$sub = sub {
    my $self = shift;
    ($self->{$field}) = @_;
};
```

- Generates several read/write accessors at once, returns key/value pairs where the key is the field name, and the value is the coderef.
- Generates several read only accessors at once, returns key/value pairs where the key is the field name, and the value is the coderef.

- Generates several write accessors at once, returns key/value pairs where the key is the field name, and the value is the coderef.

MOCK CONTROL OBJECTS

```
my $mock = mock(...);
```

Mock objects are instances of *Test2::Mock*. See it for their methods.

SOURCE

The source code repository for Test2-Suite can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::Tools::Ref

Tools for validating references.

DESCRIPTION

This module contains tools that allow you to verify that something is a ref. It also has tools to check if two refs are the same exact ref, or different. None of the functions in this module do deep comparisons.

SYNOPSIS

```
use Test2::Tools::Ref;

# Ensure something is a ref.
ref_ok($ref);

# Check that $ref is a HASH reference
```

(continues on next page)

(continued from previous page)

```
ref_ok($ref, 'HASH', 'Must be a hash')

ref_is($refa, $refb, "Same exact reference");

ref_is_not($refa, $refb, "Not the same exact reference");
```

EXPORTS

All subs are exported by default.

- ref_ok($thing)
- ref_ok($thing, $type)
- ref_ok($thing, $type, $name)

  This checks that `$thing` is a reference. If `$type` is specified then it will check that `$thing` is that type of reference.

- ref_is($ref1, $ref2, $name)

  Verify that two references are the exact same reference.

- ref_is_not($ref1, $ref2, $name)

  Verify that two references are not the exact same reference.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Tools::Refcount

assert reference counts on objects

SYNOPSIS

```
use Test2::Tools::Refcount;

use Some::Class;
my $object = Some::Class->new();

is_oneref( $object, '$object has a refcount of 1' );

my $otherref = $object;

is_refcount( $object, 2, '$object now has 2 references' );
```

DESCRIPTION

The Perl garbage collector uses simple reference counting during the normal execution of a program. This means that cycles or unweakened references in other parts of code can keep an object around for longer than intended. To help avoid this problem, the reference count of a new object from its class constructor ought to be 1. This way, the caller can know the object will be properly DESTROYed when it drops all of its references to it.

This module provides two test functions to help ensure this property holds for an object class, so as to be polite to its callers.

If the assertion fails; that is, if the actual reference count is different to what was expected, either of the following two modules may be used to assist the developer in finding where the references are.

- If Devel::MAT is installed, this test module will use it to dump the state of the memory after a failure. It will create a `.pmat` file named the same as the unit test, but with the trailing `.t` suffix replaced with `-TEST.pmat` where `TEST` is the number of the test that failed (in case there was more than one).

See the examples below for more information.

FUNCTIONS

is_refcount

```
is_refcount( $object, $count, $name )
```

Test that $object has $count references to it.

is_oneref

```
is_oneref( $object, $name )
```

Assert that the $object has only 1 reference to it.

refcount

```
$count = refcount( $object )
```

Returns the reference count of the given object as used by the test functions. This is useful for making tests that don't care what the count is before they start, but simply assert that the count hasn't changed by the end.

```
use Test2::Tools::Refcount import => [qw( is_refcount refcount )];
{
   my $count = refcount( $object );

   do_something( $object );

   is_refcount( $object, $count, 'do_something() preserves refcount'
↪);
}
```

EXAMPLE

Suppose, having written a new class `MyBall`, you now want to check that its constructor and methods are well-behaved, and don't leak references. Consider the following test script:

```
use Test::More tests => 2;
use Test2::Tools::Refcount;

use MyBall;

my $ball = MyBall->new();
is_oneref( $ball, 'One reference after construct' );

$ball->bounce;

# Any other code here that might be part of the test script

is_oneref( $ball, 'One reference just before EOF' );
```

The first assertion is just after the constructor, to check that the reference returned by it is the only reference to that object. This fact is important if we ever want `DESTROY` to behave properly. The second call is right at the end of the file, just before the main scope closes. At this stage we expect the reference count also to be one, so that the object is properly cleaned up.

Suppose, when run, this produces the following output (presuming Devel::MAT::Dumper is available):

```
1..2
ok 1 - One reference after construct
not ok 2 - One reference just before EOF
#   Failed test 'One reference just before EOF'
#   at ex.pl line 26.
#   expected 1 references, found 2
# SV address is 0x55e14c310278
# Writing heap dump to ex-2.pmat
# Looks like you failed 1 test of 2.
```

This has written a `ex-2.pmat` file we can load using the `pmat` shell and use the `identify` command on the given address to find where it went:

```
$ pmat ex-2.pmat
Perl memory dumpfile from perl 5.28.1 threaded
Heap contains 25233 objects
pmat> identify 0x55e14c310278
HASH(0)=MyBall at 0x55e14c310278 is:
├─(via RV) the lexical $ball at depth 1 of CODE() at␣
 ↪0x55e14c3104a0=main_cv, which is:
│  └─the main code
└─(via RV) value {self} of HASH(2) at 0x55e14cacb860, which is (*A):
   └─(via RV) value {cycle} of HASH(2) at 0x55e14cacb860, which is:
     itself
```

(This document isn't intended to be a full tutorial on Devel::MAT and the `pmat` shell; for that see Devel::MAT::UserGuide).

From this output, we can see that the constructor was well-behaved, but that a reference was leaked by the end of the script - the reference count was 2, when we expected just 1. Reading the trace output, we can see that there were 2 references that could be found - one stored in the $ball lexical in the main program, and one stored in a HASH. Since we expected to find the $ball lexical variable, we know we are now looking for a leak in a hash somewhere in the code. From reading the test script, we can guess this leak is likely to be in the bounce() method. Furthermore, we know that the reference to the object will be stored in a HASH in a member called `self`.

By reading the code which implements the bounce() method, we can see this is indeed the case:

```
sub bounce
{
   my $self = shift;
   my $cycle = { self => $self };
   $cycle->{cycle} = $cycle;
}
```

From reading the tracing output, we find that the HASH this object is referenced in also contains a reference to itself, in a member called `cycle`. This comes from the last line in this function, a line that purposely created a cycle, to demonstrate the point. While a

real program probably wouldn't do anything quite this obvious, the trace would still be useful in finding the likely cause of the leak.

If `Devel::MAT::Dumper` is not available, then these detailed traces will not be produced. The basic reference count testing will still take place, but a smaller message will be produced:

```
1..2
ok 1 - One reference after construct
not ok 2 - One reference just before EOF
#   Failed test 'One reference just before EOF'
#   at demo.pl line 16.
#   expected 1 references, found 2
# Looks like you failed 1 test of 2.
```

BUGS

- Temporaries created on the stack

  Code which creates temporaries on the stack, to be released again when the called function returns does not work correctly on perl 5.8 (and probably before). Examples such as

  ```
  is_oneref( [] );
  ```

  may fail and claim a reference count of 2 instead.

  Passing a variable such as

  ```
  my $array = [];
  is_oneref( $array );
  ```

  works fine. Because of the intention of this test module; that is, to assert reference counts on some object stored in a variable during the lifetime of the test script, this is unlikely to cause any problems.

ACKNOWLEDGEMENTS

Peter Rabbitson ribasushi@cpan.org - for suggesting using core's `B` instead of `Devel::Refcount` to obtain refcounts

AUTHOR

Paul Evans leonerd@leonerd.org.uk

Test2::Tools::Spec

RSPEC implementation on top of Test2::Workflow

DESCRIPTION

This uses *Test2::Workflow* to implement an RSPEC variant. This variant supports isolation and/or concurrency via forking or threads.

SYNOPSIS

```
use Test2::Bundle::Extended;
use Test2::Tools::Spec;

describe foo => sub {
    before_all  once => sub { ... };
    before_each many => sub { ... };

    after_all  once => sub { ... };
    after_each many => sub { ... };

    case condition_a => sub { ... };
    case condition_b => sub { ... };

    tests foo => sub { ... };
    tests bar => sub { ... };
};

done_testing;
```

EXPORTS

All of these use the same argument pattern. The first argument must always be a name for the block. The last argument must always be a code reference. Optionally a configuration hash can be inserted between the name and the code reference.

```
FUNCTION "name" => sub { ... };

FUNCTION "name" => {...}, sub { ... };
```

- NAME

  The first argument to a Test2::Tools::Spec function MUST be a name. The name does not need to be unique.

- PARAMS

  This argument is optional. If present this should be a hashref.

  Here are the valid keys for the hashref:

  - flat => $bool

    If this is set to true then the block will not render as a subtest, instead the events will be inline with the parent subtest (or main test).

  - async => $bool

Set this to true to mark a block as being capable of running concurrently with other test blocks. This does not mean the block WILL be run concurrently, just that it can be.

- iso => $bool

  Set this to true if the block MUST be run in isolation. If this is true then the block will run in its own forked process.

  These tests will be skipped on any platform that does not have true forking, or working/enabled threads.

  Threads will ONLY be used if the T2_WORKFLOW_USE_THREADS env var is set. Thread tests are only run if the T2_DO_THREAD_TESTS env var is set.

- todo => $reason

  Use this to mark an entire block as TODO.

- skip => $reason

  Use this to prevent a block from running at all.

- CODEREF

  This argument is required. This should be a code reference that will run some assertions.

ESSENTIALS

- tests NAME => sub { ... }
- tests NAME => \%params, sub { ... }
- tests($NAME, \%PARAMS, \&CODE)
- it NAME => sub { ... }
- it NAME => \%params, sub { ... }
- it($NAME, \%PARAMS, \&CODE)

  This defines a test block. Test blocks are essentially subtests. All test blocks will be run, and are expected to produce events. Test blocks can run multiple times if the `case()` function is also used.

  `it()` is an alias to `tests()`.

  These ARE NOT inherited by nested describe blocks.

- case NAME => sub { ... }
- case NAME => \%params, sub { ... }
- case($NAME, \%PARAMS, \&CODE)

  This lets you specify multiple conditions in which the test blocks should be run. Every test block within the same group (`describe`) will be run once per case.

  These ARE NOT inherited by nested describe blocks, but nested describe blocks will be executed once per case.

- before_each NAME => sub { ... }

- before_each NAME => \%params, sub { ... }
- before_each($NAME, \%PARAMS, \&CODE)

  Specify a codeblock that should be run multiple times, once before each `tests()` block is run. These will run AFTER `case()` blocks but before `tests()` blocks.

  These ARE inherited by nested describe blocks.
- before_case NAME => sub { ... }
- before_case NAME => \%params, sub { ... }
- before_case($NAME, \%PARAMS, \&CODE)

  Same as `before_each()`, except these blocks run BEFORE `case()` blocks.

  These ARE NOT inherited by nested describe blocks.
- before_all NAME => sub { ... }
- before_all NAME => \%params, sub { ... }
- before_all($NAME, \%PARAMS, \&CODE)

  Specify a codeblock that should be run once, before all the test blocks run.

  These ARE NOT inherited by nested describe blocks.
- around_each NAME => sub { ... }
- around_each NAME => \%params, sub { ... }
- around_each($NAME, \%PARAMS, \&CODE)

  Specify a codeblock that should wrap around each test block. These blocks are run AFTER case blocks, but before test blocks.

```
around_each wrapit => sub {
    my $cont = shift;

    local %ENV = ( ... );

    $cont->();

    ...
};
```

  The first argument to the codeblock will be a callback that MUST be called somewhere inside the sub in order for nested items to run.

  These ARE inherited by nested describe blocks.
- around_case NAME => sub { ... }
- around_case NAME => \%params, sub { ... }
- around_case($NAME, \%PARAMS, \&CODE)

  Same as `around_each` except these run BEFORE case blocks.

  These ARE NOT inherited by nested describe blocks.

- around_all NAME => sub { ... }
- around_all NAME => \%params, sub { ... }
- around_all($NAME, \%PARAMS, \&CODE)

  Same as `around_each` except that it only runs once to wrap ALL test blocks.

  These ARE NOT inherited by nested describe blocks.
- after_each NAME => sub { ... }
- after_each NAME => \%params, sub { ... }
- after_each($NAME, \%PARAMS, \&CODE)

  Same as `before_each` except it runs right after each test block.

  These ARE inherited by nested describe blocks.
- after_case NAME => sub { ... }
- after_case NAME => \%params, sub { ... }
- after_case($NAME, \%PARAMS, \&CODE)

  Same as `after_each` except it runs right after the case block, and before the test block.

  These ARE NOT inherited by nested describe blocks.
- after_all NAME => sub { ... }
- after_all NAME => \%params, sub { ... }
- after_all($NAME, \%PARAMS, \&CODE)

  Same as `before_all` except it runs after all test blocks have been run.

  These ARE NOT inherited by nested describe blocks.

SHORTCUTS

These are shortcuts. Each of these is the same as `tests()` except some parameters are added for you.

These are NOT exported by default/.

- mini NAME => sub { ... }

  Same as:

  ```
  tests NAME => { flat => 1 }, sub { ... }
  ```

- iso NAME => sub { ... }

  Same as:

  ```
  tests NAME => { iso => 1 }, sub { ... }
  ```

- miso NAME => sub { ... }

  Same as:

```
tests NAME => { mini => 1, iso => 1 }, sub { ... }
```

- async NAME => sub { ... }

  Same as:

```
tests NAME => { async => 1 }, sub { ... }
```

  Note: This conflicts with the `async()` exported from *threads*. Don't import both.

- masync NAME => sub { ... }

  Same as:

```
tests NAME => { minit => 1, async => 1 }, sub { ... }
```

CUSTOM ATTRIBUTE DEFAULTS

Sometimes you want to apply default attributes to all `tests()` or `case()` blocks. This can be done, and is lexical to your describe or package root!

```
use Test2::Bundle::Extended;
use Test2::Tools::Spec ':ALL';

# All 'tests' blocks after this declaration will have C<<iso => 1>>␣
 ↪by default
spec_defaults tests => (iso => 1);

tests foo => sub { ... }; # isolated

tests foo, {iso => 0}, sub { ... }; # Not isolated

spec_defaults tests => (iso => 0); # Turn it off again
```

Defaults are inherited by nested describe blocks. You can also override the defaults for the scope of the describe:

```
spec_defaults tests => (iso => 1);

describe foo => sub {
    spec_defaults tests => (async => 1); # Scoped to this describe␣
 ↪and any child describes

    tests bar => sub { ... }; # both iso and async
};

tests baz => sub { ... }; # Just iso, no async.
```

You can apply defaults to any type of blocks:

```
spec_defaults case => (iso => 1); # All cases are 'iso';
```

Defaults are not inherited when a builder's return is captured.

```
spec_defaults tests => (iso => 1);

# Note we are not calling this in void context, that is the key␣
 ↪here.
my $d = describe foo => {
    tests bar => sub { ... }; # Not iso
};
```

EXECUTION ORDER

As each function is encountered it executes, just like any other function. The `describe()` function will immediately execute the codeblock it is given. All other functions will stash their codeblocks to be run later. When `done_testing()` is run the workflow will be compiled, at which point all other blocks will run.

Here is an overview of the order in which blocks get called once compiled (at `done_testing()`).

```
before_all
    for-each-case {
        before_case
            case
        after_case

        # AND/OR nested describes
        before_each
            tests
        after_each
    }
after_all
```

SOURCE

The source code repository for Test2-Workflow can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist7@gmail.com.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Tools::Subtest

Tools for writing subtests

DESCRIPTION

This package exports subs that let you write subtests.

There are two types of subtests, buffered and streamed. Streamed subtests mimic subtests from *Test::More* in that they render all events as soon as they are produced. Buffered subtests wait until the subtest completes before rendering any results.

The main difference is that streamed subtests are unreadable when combined with concurrency. Buffered subtests look fine with any number of concurrent threads and processes.

SYNOPSIS

BUFFERED

```
use Test2::Tools::Subtest qw/subtest_buffered/;

subtest_buffered my_test => sub {
    ok(1, "subtest event A");
    ok(1, "subtest event B");
};
```

This will produce output like this:

```
ok 1 - my_test {
    ok 1 - subtest event A
    ok 2 - subtest event B
    1..2
}
```

STREAMED

The default option is 'buffered'. If you want streamed subtests, the way *Test::Builder* does it, use this:

```
use Test2::Tools::Subtest qw/subtest_streamed/;

subtest_streamed my_test => sub {
    ok(1, "subtest event A");
    ok(1, "subtest event B");
};
```

This will produce output like this:

```
# Subtest: my_test
    ok 1 - subtest event A
    ok 2 - subtest event B
    1..2
ok 1 - Subtest: my_test
```

IMPORTANT NOTE

You can use `bail_out` or `skip_all` in a subtest, but not in a BEGIN block or `use` statement. This is due to the way flow control works within a BEGIN block. This is not normally an issue, but can happen in rare conditions using eval, or script files as subtests.

EXPORTS

- subtest_streamed $name => $sub
- subtest_streamed($name, $sub, @args)
- subtest_streamed $name => \%params, $sub
- subtest_streamed($name, \%params, $sub, @args)

  Run subtest coderef, stream events as they happen.

  `\%params` is a hashref with any arguments you wish to pass into hub construction.
- subtest_buffered $name => $sub
- subtest_buffered($name, $sub, @args)
- subtest_buffered $name => \%params, $sub
- subtest_buffered($name, \%params, $sub, @args)

  Run subtest coderef, render events all at once when subtest is complete.

  `\%params` is a hashref with any arguments you wish to pass into hub construction.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See http://dev.perl.org/licenses/

Test2::Tools::Target

Alias the testing target package.

DESCRIPTION

This lets you alias the package you are testing into a constant and a package variable.

SYNOPSIS

```
use Test2::Tools::Target 'Some::Package';

CLASS()->xxx; # Call 'xxx' on Some::Package
$CLASS->xxx;  # Same
```

Or you can specify names:

```
use Test2::Tools::Target pkg => 'Some::Package';

pkg()->xxx; # Call 'xxx' on Some::Package
$pkg->xxx;  # Same
```

SOURCE

The source code repository for Test2-Suite can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See http://dev.perl.org/licenses/

Test2::Tools::Tester

Tools to help you test other testing tools.

DESCRIPTION

This is a collection of tools that are useful when testing other test tools.

SYNOPSIS

```
use Test2::Tools::Tester qw/event_groups filter_events facets/;

use Test2::Tools::Basic qw/plan pass ok/;
use Test2::Tools::Compare qw/is like/;

my $events = intercept {
    plan 11;

    pass('pass');
    ok(1, 'pass');

    is(1, 1, "pass");
    like(1, 1, "pass");
};

# Grab events generated by tools in Test2::Tools::Basic
my $basic = filter $events => 'Test2::Tools::Basic';

# Grab events generated by Test2::Tools::Basic;
my $compare = filter $events => 'Test2::Tools::Compare';

# Grab events generated by tools named 'ok'.
my $oks = filter $events => qr/.*::ok$/;

my $grouped = group_events $events;
# Breaks events into this structure:
{
    '__NA__' => [ ... ],
    'Test2::Tools::Basic' => {
        '__ALL__' => [ $events->[0], $events->[1], $events->[2] ],
        plan => [ $events->[0] ],
        pass => [ $events->[1] ],
        ok => [ $events->[2] ],
    },
    Test2::Tools::Compare => { ... },
}

# Get an arrayref of all the assert facets from the list of events.
my $assert_facets = facets assert => $events;
# [
```

(continues on next page)

(continued from previous page)

```
#   bless({ details => 'pass', pass => 1},
 ↪'Test2::EventFacet::Assert'),
#   bless({ details => 'pass', pass => 1},
 ↪'Test2::EventFacet::Assert'),
# ]

# Same, but for info facets
my $info_facets = facets info => $events;
```

EXPORTS

No subs are exported by default.

- $array_ref = filter $events => $PACKAGE
- $array_ref = filter $events => $PACKAGE1, $PACKAGE2
- $array_ref = filter $events => qr/match/
- $array_ref = filter $events => qr/match/, $PACKAGE

  This function takes an arrayref of events as the first argument. All additional arguments must either be a package name, or a regex. Any event that is generated by a tool in any of the package, or by a tool that matches any of the regexes, will be returned in an arrayref.

- $grouped = group_events($events)

  This function iterates all the events in the argument arrayref and splits them into groups. The resulting data structure is:

  ```
  { PACKAGE => { SUBNAME => [ $EVENT1, $EVENT2, ... }}
  ```

  If the package of an event is not known it will be put into and arrayref under the '__NA__' key at the root of the structure. If a sub name is not known it will typically go under the '__ANON__' key in under the package name.

  In addition there is an '__ALL__' key under each package which stores all of the events sorted into that group.

  A more complete example:

  ```
  {
      '__NA__' => [ $event->[3] ],
      'Test2::Tools::Basic' => {
          '__ALL__' => [ $events->[0], $events->[1], $events->[2],
   ↪],
          plan => [ $events->[0] ],
          pass => [ $events->[1] ],
          ok => [ $events->[2] ],
      },
  }
  ```

- $arrayref = facets TYPE => $events

This function will compile a list of all facets of the specified type that are found in the arrayref of events. If the facet has a `Test2::EventFacet::TYPE` package available then the facet will be constructed into an instance of the class, otherwise it is left as a hashref. Facet Order is preserved.

```
my $assert_facets = facets assert => $events;
# [
#   bless({ details => 'pass', pass => 1},
 ↪'Test2::EventFacet::Assert'),
#   bless({ details => 'pass', pass => 1},
 ↪'Test2::EventFacet::Assert'),
# ]
```

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Tools::Tiny

Tiny set of tools for unfortunate souls who cannot use *Test2::Suite*.

DESCRIPTION

You should really look at *Test2::Suite*. This package is some very basic essential tools implemented using *Test2*. This exists only so that *Test2* and other tools required by *Test2::Suite* can be tested. This is the package *Test2* uses to test itself.

USE Test2::Suite INSTEAD

Use *Test2::Suite* if at all possible.

EXPORTS

- ok($bool, $name)
- ok($bool, $name, @diag)

  Run a simple assertion.
- is($got, $want, $name)
- is($got, $want, $name, @diag)

  Assert that 2 strings are the same.
- isnt($got, $do_not_want, $name)
- isnt($got, $do_not_want, $name, @diag)

  Assert that 2 strings are not the same.
- like($got, $regex, $name)
- like($got, $regex, $name, @diag)

  Check that the input string matches the regex.
- unlike($got, $regex, $name)
- unlike($got, $regex, $name, @diag)

  Check that the input string does not match the regex.
- is_deeply($got, $want, $name)
- is_deeply($got, $want, $name, @diag)

  Check 2 data structures. Please note that this is a *DUMB* implementation that compares the output of *Data::Dumper* against both structures.
- diag($msg)

  Issue a diagnostics message to STDERR.
- note($msg)

  Issue a diagnostics message to STDOUT.
- skip_all($reason)

  Skip all tests.
- todo $reason => sub { ... }

  Run a block in TODO mode.
- plan($count)

  Set the plan.
- done_testing()

  Set the plan to the current test count.
- $warnings = warnings { ... }

  Capture an arrayref of warnings from the block.

- $exception = exception { ... }

  Capture an exception.

- tests $name => sub { ... }

  Run a subtest.

- $output = capture { ... }

  Capture STDOUT and STDERR output.

  Result looks like this:

```
{
    STDOUT => "...",
    STDERR => "...",
}
```

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::Tools::Warnings

Tools to verify warnings.

DESCRIPTION

This is a collection of tools that can be used to test code that issues warnings.

SYNOPSIS

```
use Test2::Tools::Warnings qw/warns warning warnings no_warnings/;

ok(warns { warn 'a' }, "the code warns");
ok(!warns { 1 }, "The code does not warn");
is(warns { warn 'a'; warn 'b' }, 2, "got 2 warnings");

ok(no_warnings { ... }, "code did not warn");

like(
    warning { warn 'xxx' },
    qr/xxx/,
    "Got expected warning"
);

is(
    warnings { warn "a\n"; warn "b\n" },
    [
        "a\n",
        "b\n",
    ],
    "Got 2 specific warnings"
);
```

EXPORTS

All subs are exported by default.

- $count = warns { ... }

  Returns the count of warnings produced by the block. This will always return 0, or a positive integer.

- $warning = warning { ... }

  Returns the first warning generated by the block. If the block produces more than one warning, they will all be shown as notes, and an actual warning will tell you about it.

- $warnings_ref = warnings { ... }

  Returns an arrayref with all the warnings produced by the block. This will always return an array reference. If there are no warnings, this will return an empty array reference.

- $bool = no_warnings { ... }

  Return true if the block has no warnings. Returns false if there are warnings.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Util

Tools used by Test2 and friends.

DESCRIPTION

Collection of tools used by *Test2* and friends.

EXPORTS

All exports are optional. You must specify subs to import.

- ($success, $error) = try { ... }

  Eval the codeblock, return success or failure, and the error message. This code protects $@ and $!, they will be restored by the end of the run. This code also temporarily blocks $SIG{DIE} handlers.

- protect { ... }

  Similar to try, except that it does not catch exceptions. The idea here is to protect $@ and $! from changes. $@ and $! will be restored to whatever they were before the run so long as it is successful. If the run fails $! will still be restored, but $@ will contain the exception being thrown.

- CAN_FORK

  True if this system is capable of true or pseudo-fork.

- CAN_REALLY_FORK

  True if the system can really fork. This will be false for systems where fork is emulated.

- CAN_THREAD

  True if this system is capable of using threads.

- USE_THREADS

  Returns true if threads are enabled, false if they are not.

- get_tid

  This will return the id of the current thread when threads are enabled, otherwise it returns 0.

- my $file = pkg_to_file($package)

  Convert a package name to a filename.

- $string = ipc_separator()

  Get the IPC separator. Currently this is always the string `'~'`.

- $string = gen_uid()

  Generate a unique id (NOT A UUID). This will typically be the process id, the thread id, the time, and an incrementing integer all joined with the `ipc_separator()`.

  These ID's are unique enough for most purposes. For identical ids to be generated you must have 2 processes with the same PID generate IDs at the same time with the same current state of the incrementing integer. This is a perfectly reasonable thing to expect to happen across multiple machines, but is quite unlikely to happen on one machine.

  This can fail to be unique if a process generates an id, calls exec, and does it again after the exec and it all happens in less than a second. It can also happen if the systems process id's cycle in less than a second allowing 2 different programs that use this generator to run with the same PID in less than a second. Both these cases are sufficiently unlikely. If you need universally unique ids, or ids that are unique in these conditions, look at *Data::UUID*.

- ($ok, $err) = do_rename($old_name, $new_name)

  Rename a file, this wraps `rename()` in a way that makes it more reliable cross-platform when trying to rename files you recently altered.

- ($ok, $err) = do_unlink($filename)

  Unlink a file, this wraps `unlink()` in a way that makes it more reliable cross-platform when trying to unlink files you recently altered.

NOTES && CAVEATS

- 5.10.0

  Perl 5.10.0 has a bug when compiled with newer gcc versions. This bug causes a segfault whenever a new thread is launched. Test2 will attempt to detect this, and note that the system is not capable of forking when it is detected.

- Devel::Cover

  Devel::Cover does not support threads. CAN_THREAD will return false if Devel::Cover is loaded before the check is first run.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org
- Kent Fredric kentnl@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Modules

- `Test2::Util::ExternalMeta` - Allow third party tools to safely attach meta-data to your instances.
- `Test2::Util::Facets2Legacy` - Convert facet data to the legacy event API.
- `Test2::Util::Grabber` - Object used to temporarily intercept all events.
- `Test2::Util::Guard` - Inline copy of Scope::Guard
- `Test2::Util::HashBase` - Build hash based classes.
- `Test2::Util::Importer` - Inline copy of Importer.
- `Test2::Util::Ref` - Tools for inspecting or manipulating references.
- `Test2::Util::Sig` - Signal tools used by Test2 and friends.
- `Test2::Util::Stash` - Utilities for manipulating stashes and globs.
- `Test2::Util::Sub` - Tools for inspecting and manipulating subs.
- `Test2::Util::Table` - Format a header and rows into a table
- `Test2::Util::Times` - Format timing/benchmark information.
- `Test2::Util::Trace` - Legacy wrapper for *Test2::EventFacet::Trace*.

Test2::Util::ExternalMeta

Allow third party tools to safely attach meta-data to your instances.

DESCRIPTION

This package lets you define a clear, and consistent way to allow third party tools to attach meta-data to your instances. If your object consumes this package, and imports its methods, then third party meta-data has a safe place to live.

SYNOPSIS

```
package My::Object;
use strict;
use warnings;

use Test2::Util::ExternalMeta qw/meta get_meta set_meta delete_meta/
 ↪;

...
```

Now to use it:

```
my $inst = My::Object->new;

$inst->set_meta(foo => 'bar');
my $val = $inst->get_meta('foo');
```

WHERE IS THE DATA STORED?

This package assumes your instances are blessed hashrefs, it will not work if that is not true. It will store all meta-data in the `_meta` key on your objects hash. If your object makes use of the `_meta` key in its underlying hash, then there is a conflict and you cannot use this package.

EXPORTS

- $val = $obj->meta($key)
- $val = $obj->meta($key, $default)

  This will get the value for a specified meta `$key`. Normally this will return `undef` when there is no value for the `$key`, however you can specify a `$default` value to set when no value is already set.
- $val = $obj->get_meta($key)

  This will get the value for a specified meta `$key`. This does not have the `$default` overhead that `meta()` does.
- $val = $obj->delete_meta($key)

  This will remove the value of a specified meta `$key`. The old `$val` will be returned.

- $obj->set_meta($key, $val)

  Set the value of a specified meta `$key`.

META-KEY RESTRICTIONS

Meta keys must be defined, and must be true when used as a boolean. Keys may not be references. You are free to stringify a reference `"$ref"` for use as a key, but this package will not stringify it for you.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::Util::Facets2Legacy

Convert facet data to the legacy event API.

DESCRIPTION

This module exports several subroutines from the older event API (see *Test2::Event*). These subroutines can be used as methods on any object that provides a custom `facet_data()` method. These subroutines can also be used as functions that take a facet data hashref as arguments.

SYNOPSIS

AS METHODS

```
package My::Event;

use Test2::Util::Facets2Legacy ':ALL';
```

(continues on next page)

(continued from previous page)

```
sub facet_data { return { ... } }
```

Then to use it:

```
my $e = My::Event->new(...);

my $causes_fail = $e->causes_fail;
my $summary     = $e->summary;
....
```

AS FUNCTIONS

```
use Test2::Util::Facets2Legacy ':ALL';

my $f = {
    assert => { ... },
    info => [{...}, ...],
    control => {...},
    ...
};

my $causes_fail = causes_fail($f);
my $summary     = summary($f);
```

NOTE ON CYCLES

When used as methods, all these subroutines call `$e->facet_data()`. The default `facet_data()` method in *Test2::Event* relies on the legacy methods this module emulates in order to work. As a result of this it is very easy to create infinite recursion bugs.

These methods have cycle detection and will throw an exception early if a cycle is detected. `uuid()` is currently the only subroutine in this library that has a fallback behavior when cycles are detected.

EXPORTS

Nothing is exported by default. You must specify which methods to import, or use the ':ALL' tag.

- $bool = $e->causes_fail()
- $bool = causes_fail($f)

  Check if the event or facets result in a failing state.

- $bool = $e->diagnostics()
- $bool = diagnostics($f)

  Check if the event or facets contain any diagnostics information.

- $bool = $e->global()
- $bool = global($f)

  Check if the event or facets need to be globally processed.
- $bool = $e->increments_count()
- $bool = increments_count($f)

  Check if the event or facets make an assertion.
- $bool = $e->no_display()
- $bool = no_display($f)

  Check if the event or facets should be rendered or hidden.
- ($max, $directive, $reason) = $e->sets_plan()
- ($max, $directive, $reason) = sets_plan($f)

  Check if the event or facets set a plan, and return the plan details.
- $id = $e->subtest_id()
- $id = subtest_id($f)

  Get the subtest id, if any.
- $string = $e->summary()
- $string = summary($f)

  Get the summary of the event or facets hash, if any.
- $undef_or_int = $e->terminate()
- $undef_or_int = terminate($f)

  Check if the event or facets should result in process termination, if so the exit code is returned (which could be 0). undef is returned if no termination is requested.
- $uuid = $e->uuid()
- $uuid = uuid($f)

  Get the UUID of the facets or event.

  Note: This will fall back to `$e->SUPER::uuid()` if a cycle is detected and an event is used as the argument.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::Util::Grabber

Object used to temporarily intercept all events.

DESCRIPTION

Once created this object will intercept and stash all events sent to the shared *Test2::Hub* object. Once the object is destroyed, events will once again be sent to the shared hub.

SYNOPSIS

```
use Test2 qw/Core Grab/;

my $grab = grab();

# Generate some events, they are intercepted.
ok(1, "pass");
ok(0, "fail");

my $events_a = $grab->flush;

# Generate some more events, they are intercepted.
ok(1, "pass");
ok(0, "fail");

# Same as flush, except it destroys the grab object.
my $events_b = $grab->finish;
```

After calling `finish()` the grab object is destroyed and `$grab` is set to undef. `$events_a` is an arrayref with the first two events. `$events_b` is an arrayref with the second two events.

EXPORTS

- $grab = grab()

  This lets you intercept all events for a section of code without adding anything to your call stack. This is useful for things that are sensitive to changes in the stack depth.

```
my $grab = grab();
    ok(1, 'foo');
    ok(0, 'bar');

# $grab is magically undef after this.
my $events = $grab->finish;

is(@$events, 2, "grabbed two events.");
```

When you call `finish()` the `$grab` object will automagically undef itself, but only for the reference used in the method call. If you have other references to the `$grab` object they will not be set to undef.

If the `$grab` object is destroyed without calling `finish()`, it will automatically clean up after itself and restore the parent hub.

```
{
    my $grab = grab();
    # Things are grabbed
}
# Things are back to normal
```

By default the hub used has `no_ending` set to true. This will prevent the hub from enforcing that you issued a plan and ran at least one test. You can turn enforcement back one like this:

```
$grab->hub->set_no_ending(0);
```

With `no_ending` turned off, `finish` will run the post-test checks to enforce the plan and that tests were run. In many cases this will result in additional events in your events array.

METHODS

- $grab = $class->new()

  Create a new grab object, immediately starts intercepting events.

- $ar = $grab->flush()

  Get an arrayref of all the events so far, clearing the grab objects internal list.

- $ar = $grab->events()

  Get an arrayref of all events so far. Does not clear the internal list.

- $ar = $grab->finish()

  Get an arrayref of all the events, then destroy the grab object.

- $hub = $grab->hub()

  Get the hub that is used by the grab event.

ENDING BEHAVIOR

By default the hub used has no_ending set to true. This will prevent the hub from enforcing that you issued a plan and ran at least one test. You can turn enforcement back one like this:

```
$grab->hub->set_no_ending(0);
```

With no_ending turned off, `finish` will run the post-test checks to enforce the plan and that tests were run. In many cases this will result in additional events in your events array.

SEE ALSO

Test2::Tools::Intercept - Accomplish the same thing, but using blocks instead.

SOURCE

The source code repository for Test2 can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Util::Guard

Inline copy of Scope::Guard

SEE ALSO

See Scope::Guard

ORIGINAL AUTHOR

- chocolateboy chocolate@cpan.org

INLINE AND MODIFICATION AUTHOR

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright (c) 2005-2015, chocolateboy.

Modified copy is Copyright Chad Granum exodist7@gmail.com.

This module is free software. It may be used, redistributed and/or modified under the same terms as Perl itself.

Test2::Util::HashBase

Build hash based classes.

SYNOPSIS

A class:

```
package My::Class;
use strict;
use warnings;

# Generate 3 accessors
use Test2::Util::HashBase qw/foo -bar ^baz <bat >ban +boo/;

# Chance to initialize defaults
sub init {
    my $self = shift;    # No other args
    $self->{+FOO} ||= "foo";
    $self->{+BAR} ||= "bar";
    $self->{+BAZ} ||= "baz";
    $self->{+BAT} ||= "bat";
    $self->{+BAN} ||= "ban";
    $self->{+BOO} ||= "boo";
}

sub print {
    print join ", " => map { $self->{$_} } FOO, BAR, BAZ, BAT, BAN,␣
 ↪BOO;
}
```

Subclass it

```
package My::Subclass;
use strict;
use warnings;

# Note, you should subclass before loading HashBase.
use base 'My::Class';
```

(continues on next page)

(continued from previous page)

```
use Test2::Util::HashBase qw/bub/;

sub init {
    my $self = shift;

    # We get the constants from the base class for free.
    $self->{+FOO} ||= 'SubFoo';
    $self->{+BUB} ||= 'bub';

    $self->SUPER::init();
}
```

use it:

```
package main;
use strict;
use warnings;
use My::Class;

# These are all functionally identical
my $one   = My::Class->new(foo => 'MyFoo', bar => 'MyBar');
my $two   = My::Class->new({foo => 'MyFoo', bar => 'MyBar'});
my $three = My::Class->new(['MyFoo', 'MyBar']);

# Readers!
my $foo = $one->foo;    # 'MyFoo'
my $bar = $one->bar;    # 'MyBar'
my $baz = $one->baz;    # Defaulted to: 'baz'
my $bat = $one->bat;    # Defaulted to: 'bat'
# '>ban' means setter only, no reader
# '+boo' means no setter or reader, just the BOO constant

# Setters!
$one->set_foo('A Foo');

#'-bar' means read-only, so the setter will throw an exception (but
 ↪is defined).
$one->set_bar('A bar');

# '^baz' means deprecated setter, this will warn about the setter
 ↪being
# deprecated.
$one->set_baz('A Baz');

# '<bat' means no setter defined at all
# '+boo' means no setter or reader, just the BOO constant

$one->{+FOO} = 'xxx';
```

DESCRIPTION

This package is used to generate classes based on hashrefs. Using this class will give you a `new()` method, as well as generating accessors you request. Generated accessors will be getters, `set_ACCESSOR` setters will also be generated for you. You also get constants for each accessor (all caps) which return the key into the hash for that accessor. Single inheritance is also supported.

THIS IS A BUNDLED COPY OF HASHBASE

This is a bundled copy of Object::HashBase. This file was generated using the `/home/exodist/perl5/perlbrew/perls/main/bin/hashbase_inc.pl` script.

METHODS

PROVIDED BY HASH BASE

- $it = $class->new(%PAIRS)
- $it = $class->new(\%PAIRS)
- $it = $class->new(\@ORDERED_VALUES)

  Create a new instance.

  HashBase will not export `new()` if there is already a `new()` method in your packages inheritance chain.

  If you do not want this method you can define your own you just have to declare it before loading *Test2::Util::HashBase*.

```
package My::Package;

# predeclare new() so that HashBase does not give us one.
sub new;

use Test2::Util::HashBase qw/foo bar baz/;

# Now we define our own new method.
sub new { ... }
```

  This makes it so that HashBase sees that you have your own `new()` method. Alternatively you can define the method before loading HashBase instead of just declaring it, but that scatters your use statements.

  The most common way to create an object is to pass in key/value pairs where each key is an attribute and each value is what you want assigned to that attribute. No checking is done to verify the attributes or values are valid, you may do that in `init()` if desired.

  If you would like, you can pass in a hashref instead of pairs. When you do so the hashref will be copied, and the copy will be returned blessed as an object. There is no way to ask HashBase to bless a specific hashref.

  In some cases an object may only have 1 or 2 attributes, in which case a hashref may be too verbose for your liking. In these cases you can pass in an arrayref with

only values. The values will be assigned to attributes in the order the attributes were listed. When there is inheritance involved the attributes from parent classes will come before subclasses.

HOOKS

- $self->init()

  This gives you the chance to set some default values to your fields. The only argument is `$self` with its indexes already set from the constructor.

  Note: Test2::Util::HashBase checks for an init using `$class->can('init')` during construction. It DOES NOT call `can()` on the created object. Also note that the result of the check is cached, it is only ever checked once, the first time an instance of your class is created. This means that adding an `init()` method AFTER the first construction will result in it being ignored.

ACCESSORS

READ/WRITE

To generate accessors you list them when using the module:

```
use Test2::Util::HashBase qw/foo/;
```

This will generate the following subs in your namespace:

- foo()

  Getter, used to get the value of the `foo` field.

- set_foo()

  Setter, used to set the value of the `foo` field.

- FOO()

  Constant, returns the field `foo`'s key into the class hashref. Subclasses will also get this function as a constant, not simply a method, that means it is copied into the subclass namespace.

  The main reason for using these constants is to help avoid spelling mistakes and similar typos. It will not help you if you forget to prefix the '+' though.

READ ONLY

```
use Test2::Util::HashBase qw/-foo/;
```

- set_foo()

  Throws an exception telling you the attribute is read-only. This is exported to override any active setters for the attribute in a parent class.

DEPRECATED SETTER

```
use Test2::Util::HashBase qw/^foo/;
```

- set_foo()

  This will set the value, but it will also warn you that the method is deprecated.

NO SETTER

```
use Test2::Util::HashBase qw/<foo/;
```

Only gives you a reader, no `set_foo` method is defined at all.

NO READER

```
use Test2::Util::HashBase qw/>foo/;
```

Only gives you a write (`set_foo`), no `foo` method is defined at all.

CONSTANT ONLY

```
use Test2::Util::HashBase qw/+foo/;
```

This does not create any methods for you, it just adds the `FOO` constant.

SUBCLASSING

You can subclass an existing HashBase class.

```
use base 'Another::HashBase::Class';
use Test2::Util::HashBase qw/foo bar baz/;
```

The base class is added to `@ISA` for you, and all constants from base classes are added to subclasses automatically.

GETTING A LIST OF ATTRIBUTES FOR A CLASS

Test2::Util::HashBase provides a function for retrieving a list of attributes for an Test2::Util::HashBase class.

- @list = Test2::Util::HashBase::attr_list($class)
- @list = $class->Test2::Util::HashBase::attr_list()

  Either form above will work. This will return a list of attributes defined on the object. This list is returned in the attribute definition order, parent class attributes are listed before subclass attributes. Duplicate attributes will be removed before the list is returned.

  Note: This list is used in the `$class->new(\@ARRAY)` constructor to determine the attribute to which each value will be paired.

SOURCE

The source code repository for HashBase can be found at https://github.com/Test-More/HashBase/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::Util::Importer

Inline copy of Importer.

DESCRIPTION

See Importer.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist7@gmail.com.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See http://dev.perl.org/licenses/

Test2::Util::Ref

Tools for inspecting or manipulating references.

DESCRIPTION

These are used by *Test2::Tools* to inspect, render, or manipulate references.

EXPORTS

All exports are optional. You must specify subs to import.

- $type = rtype($ref)

  A normalization between `Scalar::Util::reftype()` and `ref()`.

  Always returns a string.

  Returns `'REGEXP'` for regex types

  Returns `''` for non-refs

  Otherwise returns what `Scalar::Util::reftype()` returns.

- $addr_str = render_ref($ref)

  Always returns a string. For unblessed references this returns something like `"SCALAR(0x...)"`. For blessed references it returns `"My::Thing=SCALAR(0x...)"`. The only difference between this and `$add_str = "$thing"` is that it ignores any overloading to ensure it is always the ref address.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org
- Kent Fredric kentnl@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Util::Sig

Signal tools used by Test2 and friends.

DESCRIPTION

Collection of signal tools used by *Test2* and friends.

EXPORTS

All exports are optional. You must specify subs to import.

- ($ok, $err) = try_sig_mask { ... }

  Complete an action with several signals masked, they will be unmasked at the end allowing any signals that were intercepted to get handled.

  This is primarily used when you need to make several actions atomic (against some signals anyway).

  Signals that are intercepted:

  - SIGINT
  - SIGALRM
  - SIGHUP
  - SIGTERM
  - SIGUSR1
  - SIGUSR2

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org
- Kent Fredric kentnl@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::Util::Stash

Utilities for manipulating stashes and globs.

DESCRIPTION

This is a collection of utilities for manipulating and inspecting package stashes and globs.

EXPORTS

- $stash = get_stash($package)

  Gets the package stash. This is the same as `$stash = \%Package::Name::`.

- $sym_spec = parse_symbol($symbol)
- $sym_spec = parse_symbol($symbol, $package)

  Parse a symbol name, and return a hashref with info about the symbol.

  `$symbol` can be a simple name, or a fully qualified symbol name. The sigil is optional, and & is assumed if none is provided. If `$symbol` is fully qualified, and `$package` is also provided, then the package of the symbol must match the `$package`.

  Returns a structure like this:

```
return {
    name    => 'BAZ',
    sigil   => '$',
    type    => 'SCALAR',
    symbol  => '&Foo::Bar::BAZ',
    package => 'Foo::Bar',
};
```

- $glob_ref = get_glob($symbol)
- $glob_ref = get_glob($symbol, $package)

  Get a glob ref. Arguments are the same as for `parse_symbol`.

- $ref = get_symbol($symbol)
- $ref = get_symbol($symbol, $package)

  Get a reference to the symbol. Arguments are the same as for `parse_symbol`.

- $ref = purge_symbol($symbol)
- $ref = purge_symbol($symbol, $package)

  Completely remove the symbol from the package symbol table. Arguments are the same as for `parse_symbol`. A reference to the removed symbol is returned.

- $sig = slot_to_sig($slot)

  Convert a slot (like 'SCALAR') to a sigil (like '$').

- $slot = sig_to_slot($sig)

  Convert a sigil (like '$') to a slot (like 'SCALAR').

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Util::Sub

Tools for inspecting and manipulating subs.

DESCRIPTION

Utilities used by Test2::Tools to inspect and manipulate subroutines.

EXPORTS

All exports are optional, you must specify subs to import.

- $name = sub_name(\&sub)

  Get the name of the sub.

- my $hr = sub_info(\&code)

  This returns a hashref with information about the sub:

```
{
    ref        => \&code,
    cobj       => $cobj,
    name       => "Some::Mod::code",
    file       => "Some/Mod.pm",
    package    => "Some::Mod",
```

(continues on next page)

(continued from previous page)

```
    # Note: These have been adjusted based on guesswork.
    start_line => 22,
    end_line   => 42,
    lines      => [22, 42],

    # Not a bug, these lines are different!
    all_lines  => [23, 25, ..., 39, 41],
};
```

- $info->{ref} => \&code

  This is the original sub passed to `sub_info()`.

- $info->{cobj} => $cobj

  This is the c-object representation of the coderef.

- $info->{name} => "Some::Mod::code"

  This is the name of the coderef. For anonymous coderefs this may end with `'__ANON__'`. Also note that the package 'main' is special, and 'main::' may be omitted.

- $info->{file} => "Some/Mod.pm"

  The file in which the sub was defined.

- $info->{package} => "Some::Mod"

  The package in which the sub was defined.

- $info->{start_line} => 22

- $info->{end_line} => 42

- $info->{lines} => [22, 42]

  These three fields are the *adjusted* start line, end line, and array with both. It is important to note that these lines have been adjusted and may not be accurate.

  The lines are obtained by walking the ops. As such, the first line is the line of the first statement, and the last line is the line of the last statement. This means that in multi-line subs the lines are usually off by 1. The lines in these keys will be adjusted for you if it detects a multi-line sub.

- $info->{all_lines} => [23, 25, ..., 39, 41]

  This is an array with the lines of every statement in the sub. Unlike the other line fields, these have not been adjusted for you.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org
- Kent Fredric kentnl@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Util::Table

Format a header and rows into a table

DESCRIPTION

This is used by some failing tests to provide diagnostics about what has gone wrong. This module is able to generic format rows of data into tables.

SYNOPSIS

```
use Test2::Util::Table qw/table/;

my @table = table(
    max_width => 80,
    collapse => 1, # Do not show empty columns
    header => [ 'name', 'age', 'hair color' ],
    rows => [
        [ 'Fred Flinstone',  2000000, 'black' ],
        [ 'Wilma Flinstone', 1999995, 'red' ],
        ...,
    ],
);

# The @table array contains each line of the table, no newlines␣
↪added.
say $_ for @table;
```

This prints a table like this:

```
+-----------------+---------+------------+
| name            | age     | hair color |
```

(continues on next page)

(continued from previous page)

```
+-----------------+---------+------------+
| Fred Flinstone  | 2000000 | black      |
| Wilma Flinstone | 1999995 | red        |
| ...             | ...     | ...        |
+-----------------+---------+------------+
```

EXPORTS

@rows = table(...)

The function returns a list of lines, lines do not have the newline \n character appended.

Options:

- header => [...]

  If you want a header specify it here. This takes an arrayref with each columns heading.

- rows => [[...], [...], ...]

  This should be an arrayref containing an arrayref per row.

- collapse => $bool

  Use this if you want to hide empty columns, that is any column that has no data in any row. Having a header for the column will not effect collapse.

- max_width => $num

  Set the maximum width of the table, the table may not be this big, but it will be no bigger. If none is specified it will attempt to find the width of your terminal and use that, otherwise it falls back to 80.

- sanitize => $bool

  This will sanitize all the data in the table such that newlines, control characters, and all whitespace except for ASCII 20 ' ' are replaced with escape sequences. This prevents newlines, tabs, and similar whitespace from disrupting the table.

  Note: newlines are marked as '\n', but a newline is also inserted into the data so that it typically displays in a way that is useful to humans.

  Example:

```
my $field = "foo\nbar\nbaz\n";

print join "\n" => table(
    sanitize => 1,
    rows => [
        [$field,      'col2'     ],
        ['row2 col1', 'row2 col2']
    ]
);
```

  Prints:

```
+------------------+-----------+
| foo\n            | col2      |
| bar\n            |           |
| baz\n            |           |
|                  |           |
| row2 col1        | row2 col2 |
+------------------+-----------+
```

So it marks the newlines by inserting the escape sequence, but it also shows the data across as many lines as it would normally display.

- mark_tail => $bool

  This will replace the last whitespace character of any trailing whitespace with its escape sequence. This makes it easier to notice trailing whitespace when comparing values.

my $cols = term_size()

Attempts to find the width in columns (characters) of the current terminal. Returns 80 as a safe bet if it cannot find it another way.

NOTE ON UNICODE/WIDE CHARACTERS

Some unicode characters, such as 婧 (U+5A67) are wider than others. These will render just fine if you `use utf8;` as necessary, and *Unicode::GCString* is installed, however if the module is not installed there will be anomalies in the table:

```
+-----+-----+---+
| a   | b   | c |
+-----+-----+---+
| 婧 | x   | y |
| x   | y   | z |
| x   | 婧 | z |
+-----+-----+---+
```

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See http://dev.perl.org/licenses/

Modules

- Test2::Util::Table::LineBreak - Break up lines for use in tables.

Test2::Util::Table::LineBreak

Break up lines for use in tables.

DESCRIPTION

This is meant for internal use. This package takes long lines of text and splits them so that they fit in table rows.

SYNOPSIS

```
use Test2::Util::Table::LineBreak;

my $lb = Test2::Util::Table::LineBreak->new(string => $STRING);

$lb->break($SIZE);
while (my $part = $lb->next) {
    ...
}
```

SOURCE

The source code repository for Test2-Suite can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Util::Times

Format timing/benchmark information.

DESCRIPTION

This modules exports tools for rendering timing data at the end of tests.

EXPORTS

All exports are optional. You must specify subs to import.

- $str = render_bench($start, $end, $user, $system, $cuser, $csystem)
- $str = render_bench($start, time(), times())

  This will produce a string like one of these (Note these numbers are completely made up). *Which string is used depends on the time elapsed.*

```
0.12345s on wallclock (0.05 usr 0.00 sys + 0.00 cusr 0.00 csys␣
↪= 0.05 CPU)

11.1234s on wallclock (0.05 usr 0.00 sys + 0.00 cusr 0.00 csys␣
↪= 0.05 CPU)

01m:54.45s on wallclock (0.05 usr 0.00 sys + 0.00 cusr 0.00␣
↪csys = 0.05 CPU)

18h:22m:54.45s on wallclock (0.05 usr 0.00 sys + 0.00 cusr 0.00␣
↪csys = 0.05 CPU)

04d:18h:22m:54.45s on wallclock (0.05 usr 0.00 sys + 0.00 cusr␣
↪0.00 csys = 0.05 CPU)
```

  The first 2 arguments are the `$start` and `$end` times in seconds (as returned by `time()` or `Time::HiRes::time()`).

  The last 4 arguments are timing information as returned by the `times()` function.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Util::Trace

Legacy wrapper for *Test2::EventFacet::Trace*.

DESCRIPTION

All the functionality for this class has been moved to *Test2::EventFacet::Trace*.

SOURCE

The source code repository for Test2 can be found at https://github.com/Test-More/test-more/.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/

Test2::V0

0Th edition of the Test2 recommended bundle.

V1 IS OUT NOW!

See *Test2::V1* for the most recent "Recommended" bundle.

DESCRIPTION

This is the big-daddy bundle. This bundle includes nearly every tool, and several plugins, that the Test2 author uses. This bundle is used extensively to test *Test2::Suite* itself.

NAMING, USING, DEPENDING

This bundle should not change in a *severely* incompatible way. Some minor breaking changes, specially bugfixes, may be allowed. If breaking changes are needed then a new `Test2::V#` module should be released instead.

As new V# modules are released old ones *may* be moved to different cpan distributions. You should always use a specific bundle version and list that version in your distributions testing requirements. You should never simply list *Test2::Suite* as your modules dep, instead list the specific bundle, or tools and plugins you use directly in your metadata.

SYNOPSIS

```
use Test2::V0;

ok(1, "pass");

...

done_testing;
```

RESOLVING CONFLICTS WITH MOOSE

```
use Test2::V0 '!meta';
```

Moose and *Test2::V0* both export very different `meta()` subs. Adding `'!meta'` to the import args will prevent the sub from being imported. This bundle also exports the sub under the name `meta_check()` so you can use that spelling as an alternative.

TAGS

- :DEFAULT

  The following are both identical:

```
use Test2::V0;

use Test2::V0 ':DEFAULT';
```

RENAMING ON IMPORT

```
use Test2::V0 ':DEFAULT', '!ok', ok => {-as => 'my_ok'};
```

This bundle uses *Test2::Util::Importer* for exporting, as such you can use any arguments it accepts.

Explanation:

- '!ok'

  Do not export `ok()`

- ok => {-as => 'my_ok'}

  Actually, go ahead and import `ok()` but under the name `my_ok()`.

If you did not add the `'!ok'` argument then you would have both `ok()` and `my_ok()`

PRAGMAS

All of these can be disabled via individual import arguments, or by the `-no_pragmas` argument.

```
use Test2::V0 -no_pragmas => 1;
```

STRICT

strict is turned on for you. You can disable this with the `-no_strict` or `-no_pragmas` import arguments:

```
use Test2::V0 -no_strict => 1;
```

WARNINGS

warnings are turned on for you. You can disable this with the `-no_warnings` or `-no_pragmas` import arguments:

```
use Test2::V0 -no_warnings => 1;
```

UTF8

This is actually done via the *Test2::Plugin::UTF8* plugin, see the "PLUGINS" section for details.

Note: `-no_pragmas => 1` will turn off the entire plugin.

PLUGINS

SRAND

See *Test2::Plugin::SRand*.

This will set the random seed to today's date. You can provide an alternate seed with the `-srand` import option:

```
use Test2::V0 -srand => 1234;
```

You can also disable this behavior:

```
use Test2::V0 -no_srand => 1;
```

Note When srand is on (default) it can cause problems with things like *File::Temp* which will end up attempting the same "random" filenames for every test process started on a given day (or sharing the same seed).

UTF8

See *Test2::Plugin::UTF8*.

This will set the file, and all output handles (including formatter handles), to utf8. This will turn on the utf8 pragma for the current scope.

This can be disabled using the `-no_utf8 => 1` or `-no_pragmas => 1` import arguments.

```
use Test2::V0 -no_utf8 => 1;
```

EXIT SUMMARY

See *Test2::Plugin::ExitSummary*.

This plugin has no configuration.

ENVIRONMENT VARIABLES

See *Test2::Env* for a list of meaningful environment variables.

API FUNCTIONS

See *Test2::API* for these

- $ctx = context()
- $events = intercept { ... }

TOOLS

TARGET

See *Test2::Tools::Target*.

You can specify a target class with the `-target` import argument. If you do not provide a target then `$CLASS` and `CLASS()` will not be imported.

```
use Test2::V0 -target => 'My::Class';

print $CLASS;  # My::Class
print CLASS(); # My::Class
```

Or you can specify names:

```
use Test2::V0 -target => { pkg => 'Some::Package' };

pkg()->xxx; # Call 'xxx' on Some::Package
$pkg->xxx;  # Same
```

- $CLASS

  Package variable that contains the target class name.

- $class = CLASS()

  Constant function that returns the target class name.

DEFER

See *Test2::Tools::Defer*.

- def $func => @args;
- do_def()

BASIC

See *Test2::Tools::Basic*.

- ok($bool, $name)
- ok($bool, $name, @diag)
- pass($name)
- pass($name, @diag)
- fail($name)
- fail($name, @diag)
- diag($message)
- note($message)
- $todo = todo($reason)
- todo $reason => sub { ... }

- skip($reason, $count)
- plan($count)
- skip_all($reason)
- done_testing()
- bail_out($reason)

COMPARE

See *Test2::Tools::Compare*.

- is($got, $want, $name)
- isnt($got, $do_not_want, $name)
- like($got, qr/match/, $name)
- unlike($got, qr/mismatch/, $name)
- $check = match(qr/pattern/)
- $check = mismatch(qr/pattern/)
- $check = validator(sub { return $bool })
- $check = hash { ... }
- $check = array { ... }
- $check = bag { ... }
- $check = object { ... }
- $check = meta { ... }
- $check = number($num)
- $check = string($str)
- $check = bool($bool)
- $check = check_isa($class_name)
- $check = in_set(@things)
- $check = not_in_set(@things)
- $check = check_set(@things)
- $check = item($thing)
- $check = item($idx => $thing)
- $check = field($name => $val)
- $check = call($method => $expect)
- $check = call_list($method => $expect)
- $check = call_hash($method => $expect)
- $check = prop($name => $expect)
- $check = check($thing)

- $check = T()
- $check = F()
- $check = D()
- $check = DF()
- $check = E()
- $check = DNE()
- $check = FDNE()
- $check = U()
- $check = L()
- $check = exact_ref($ref)
- end()
- etc()
- filter_items { grep { ... } @_ }
- $check = event $type => ...
- @checks = fail_events $type => ...

CLASSIC COMPARE

See *Test2::Tools::ClassicCompare*.

- cmp_ok($got, $op, $want, $name)

SUBTEST

See *Test2::Tools::Subtest*.

- subtest $name => sub { ... };

  (Note: This is called `subtest_buffered()` in the Tools module.)

CLASS

See *Test2::Tools::Class*.

- can_ok($thing, @methods)
- isa_ok($thing, @classes)
- DOES_ok($thing, @roles)

ENCODING

See *Test2::Tools::Encoding*.

- set_encoding($encoding)

EXPORTS

See *Test2::Tools::Exports*.

- imported_ok('function', '$scalar', ...)
- not_imported_ok('function', '$scalar', ...)

REF

See *Test2::Tools::Ref* .

- ref_ok($ref, $type)
- ref_is($got, $want)
- ref_is_not($got, $do_not_want)

See *Test2::Tools::Refcount*.

- is_refcount($ref, $count, $description)
- is_oneref($ref, $description)
- $count = refcount($ref)

MOCK

See *Test2::Tools::Mock*.

- $control = mock ...
- $bool = mocked($thing)

EXCEPTION

See *Test2::Tools::Exception*.

- $exception = dies { ... }
- $bool = lives { ... }
- $bool = try_ok { ... }

WARNINGS

See *Test2::Tools::Warnings*.

- $count = warns { ... }
- $warning = warning { ... }
- $warnings_ref = warnings { ... }
- $bool = no_warnings { ... }

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::V1

V1 edition of the Test2 recommended bundle.

DESCRIPTION

This is the first sequel to *Test2::V0*. This module is recommended over *Test2::V0* for new tests.

Key differences from *Test2::V0*

- Only 1 export by default: T2()
- No pragmas by default
- srand and utf8 are not enabled by default
- Easy to still import everything
- East to still enable pragmas

NAMING, USING, DEPENDING

This bundle should not change in a *severely* incompatible way. Some minor breaking changes, specially bugfixes, may be allowed. If breaking changes are needed then a new `Test2::V#` module should be released instead.

Adding new optional exports, and new methods on the T2() handle are not considered breaking changes, and are allowed without bumping the V# number. Adding new plugin shortcuts is also allowed, but they cannot be added to the `-P` or `-plugins` shortcuts without a bump in V# number.

As new V# modules are released old ones *may* be moved to different cpan distributions. You should always use a specific bundle version and list that version in your distributions testing requirements. You should never simply list *Test2::Suite* as your modules dep, instead list the specific bundle, or tools and plugins you use directly in your metadata.

See the "JUSTIFICATION" section for an explanation of why *Test2::V1* was created.

SYNOPSIS

RECOMMENDED

```
use Test2::V1 -utf8;

T2->ok(1, "pass");

T2->is({1 => 1}, {1 => 1}, "Structures Match");

# Note that prototypes do not work in method form:
my @foo = (1, 2, 3);
T2->is(scalar(@foo), 3, "Needed to force scalar context");

T2->done_testing;
```

WORK LIKE V0 DID

```
use Test2::V1 -ipP;

ok(1, "pass");

is({1 => 1}, {1 => 1}, "Structures Match");

my @foo = (1, 2, 3);
is(@foo, 3, "Prototype forces @foo into scalar context");

# You still have access to T2
T2->ok(1, "Another Pass");

done_testing;
```

The `-ipP` argument is short for `-include, -pragmas, -plugins` which together enable all pragmas, plugins, and import all symbols.

Note: The order in which `i`, `p`, and `P` appear is not important; `-Ppi` and `-piP` and any other order are all perfectly valid.

IMPORT ARGUMENT GUIDE

- `-P` or `-plugins`

  Shortcut to include the following plugins: *Test2::Plugin::UTF8*, *Test2::Plugin::SRand*, *Test2::Plugin::ExitSummary*.

- `-p` or `-pragmas`

  Shortcut to enable the following pragmas: `strict`, `warnings`.

- `-i` or `-import`

  Shortcut to import all possible exports.

- `-x`

  Shortcut to import any sub that has '&' in its prototype, things like `dies { ... }`, `warns { ... }`, etc.

  While these can be used in method form: `T2->dies(sub { ... })` it is a little less convenient than having them imported. '-x' will import all of these, and any added in the future or included via an `-include => ...` import argument.

- `-ipP`, `-pPi`, `-pP`, `-Pix`, etc..

  The `i`, `p`, `P`, and `x` short options may all be grouped in any order following a single dash.

- `@EXPORT_LIST`

  Any arguments provided that are not prefixed with a `-` will be assumed to be export requests. If there is an exported sub by the given name it will be imported into your namespace. If there is no such sub an exception will be thrown.

- `!EXPORT_NAME`

  You can prefix an export name with `!` to exclude it at import time. This is really only usedul when combined with `-import` or `-i`.

- `EXPORT_NAME => { -as => "ALT_NAME" }`
- `EXPORT_NAME => { -prefix => "PREFIX_" }`
- `EXPORT_NAME => { -postfix => "_POSTFIX" }`

  You may specify a hashref after an export name to rename it, or add a prefix/postfix to the name.

RENAMING IMPORTS

```
use Test2::V1 '-import', '!ok', ok => {-as => 'my_ok'};
```

Explanation:

- '-import'

  Bring in ALL imports, no need to list them all by hand.

- '!ok'

  Do not import `ok()` (remove it from the list added by '-import')

- ok => {-as => 'my_ok'}

  Actually, go ahead and import `ok()` but under the name `my_ok()`.

If you did not add the `'!ok'` argument then you would have both `ok()` and `my_ok()`

PRAGMAS AND PLUGINS

NO PRAGMAS ARE ENABLED BY DEFAULT ONLY THE EXIT SUMMARY PLUGIN IS STILL ENABLED BY DEFAULT

This is a significant departure from *Test2::V0*.

You can enable all of these with the `-pP` argument, which is short for `-plugins, -pragmas`. `P` is short for plugins, and `p` is short for pragmas. When using the single-letter form they may both be together following a single dash, and can be in any order. They may also be combined with `i` to bring in all imports. `-p` or `-P` ont heir own are also perfectly valid.

- strict

  You can enable this with any of these arguments: `-strict`, `-p`, `-pragmas`.

  This enables strict for you.

- warnings

  You can enable this with any of these arguments: `-warnings`, `-p`, `-pragmas`.

  This enables warnings for you.

- srand

  You can enable this in multiple ways:

  ```
  use Test2::V1 -srand
  use Test2::V1 -P
  use Test2::V1 -plugins
  ```

  See *Test2::Plugin::SRand*.

  This will set the random seed to today's date.

  You can also set a random seed:

  ```
  use Test2::V1 -srand => { seed => 'my seed' };
  ```

- utf8

  You can enable this in multiple ways:

  ```
  use Test2::V1 -utf8
  use Test2::V1 -P
  use Test2::V1 -plugins
  ```

  See *Test2::Plugin::UTF8*.

  This will set the file, and all output handles (including formatter handles), to utf8. This will turn on the utf8 pragma for the current scope.

- summary

  This is turned on by default.

  You can avoid enabling it at import this way:

  ```
  use Test2::V1 -summary => 0;
  ```

  See *Test2::Plugin::ExitSummary*.

  This plugin has no configuration.

ENVIRONMENT VARIABLES

See *Test2::Env* for a list of meaningful environment variables.

API FUNCTIONS

See *Test2::API* for these

- $ctx = T2->context()
- $events = T2->intercept(sub { ... });

THE T2() HANDLE

The `T2()` subroutine imported into your namespace returns an instance of *Test2::V1::Handle*. This gives you a handle on all the tools included by default. It also creates a completely new namespace for use by your test that can have additional tools added to it.

ADDING/OVERRIDING TOOLS IN YOUR T2 HANDLE

```
# Method 1
use Test2::V1 T2 => {
    include => [
        ['Test2::Tools::MyTool', 'my_tool', 'my_other_tool'],
        ['Data::Dumper', 'Dumper'],
    ],
};

# Method 2
use Test2::V1 T2 => {
    include => {
        'Test2::Tools::MyTool' => ['my_tool', 'my_other_tool'],
        'Data::Dumper'         => 'Dumper',
    },
};

# Method 3 (This also works with a hashref instead of an arrayref)
use Test2::V1 -include => [
    ['Test2::Tools::MyTool', 'my_tool', 'my_other_tool'],
```

(continues on next page)

(continued from previous page)

```
    ['Data::Dumper', 'Dumper'],
];

# Method 4
T2->include('Test2::Tools::MyTool', 'my_tool', 'my_other_tool');
T2->include('Data::Dumper', 'Dumper');

# Using them:

T2->my_tool(...);

T2->Dumper({hi => 'there'});
```

Note that you MAY override original tools such as ok(), note(), etc. by importing different copies this way. The first time you do this there should be no warnings or errors. If you pull in multiple tools of the same name an redefine warning is likely.

This also effects exports:

```
use Test2::V1 -import, -include => ['Data::Dumper'];

print Dumper("Dumper can be imported from your include!");
```

OTHER HANDLE OPTIONS

```
use Test2::V1 T2 => {
    include   => $ARRAYREF_OR_HASHREF,
    namespace => $NAMESPACE,
    base      => $BASE_PACKAGE // 'Test2::V1::Base',
    stomp     => $BOOL,
};
```

- include => $ARRAYREF_OR_HASHREF

  See “ADDING TOOLS TO YOUR T2 HANDLE”.

- namespace => $NAMESPACE

  Normally a new namespace will be generated for you. You **CANNOT** rely on the package name being anything specific unless you provide your own.

  The namespace here will be where any tools you ‘include’ will be imported into. It will also have its base class set to the base class you specify, or the *Test2::V1::Base* module if you do not provide any.

  If this namespace already has any symbols defined in it an exception will be thrown unless the `stomp` argument is set to true (not recommended).

- stomp => $BOOL

  Used to allow the handle to stomp on an existing namespace (NOT RECOMMENDED).

- base => $BASE

Set the base class from which functions should be inherited. Normally this is set to *Test2::V1::Base*.

Another interesting use case is to have multiple handles that use eachothers namespaces as base classes:

```
use Test2::V1;

use Test2::V1::Handle(
    'T3',
    base    => T2->HANDLE_NAMESPACE,
    include => {'Alt::Ok' => 'ok'};
);

T3->ok(1, "This uses ok() from Alt::Ok, but all other ->
↪methods are the original");
T3->done_testing(); # Uses the original done_testing
```

EXAMPLE USE CASES

OVERRIDING INCLUDED TOOLS WITH ALTERNATES

Lets say you want to use the Test2::Warnings version of `warning()`, `warnings()` instead of the *Test2::Tools::Warnings* versions, and also wanted to import everything else Test2::Warnings provides.

```
use Test2::V1 -import, -include => ['Test2::Warnings'];
```

The `-include => ['Test2::Warnings']` option means we want to import the default set of imports from Test2::Warnings into our `T2()` handle's private namespace. This will override any methods that were also previously defined by default.

The `-import` option means we want to import all subs into the current namespace. This includes anything we got from Test2::Warnings, and we will get the Test2::Warnings version of those subs.

```
like(
    warning { warn 'xxx' }, # This is the Test2::Warnings version of
↪'warning'
    qr/xxx/,
    "Got expected warning"
);
```

TOOLS

TARGET

Added to Test::V1 in 1.302217.

See *Test2::Tools::Target*.

You can specify a target class with the `-target` import argument. If you do not provide a target then `$CLASS` and `CLASS()` will not be imported.

```
use Test2::V1 -target => 'My::Class';

print $CLASS;  # My::Class
print CLASS(); # My::Class
```

Or you can specify names:

```
use Test2::V1 -target => { pkg => 'Some::Package' };

pkg()->xxx; # Call 'xxx' on Some::Package
$pkg->xxx;  # Same
```

- $CLASS

  Package variable that contains the target class name.

- $class = CLASS()

  Constant function that returns the target class name.

DEFER

See *Test2::Tools::Defer*.

- def $func => @args;

  Added to Test::V1 in 1.302217.

- do_def()

  Added to Test::V1 in 1.302217.

BASIC

See *Test2::Tools::Basic*.

- ok($bool, $name)
- ok($bool, $name, @diag)

  Added to Test::V1 in 1.302217.

- pass($name)
- pass($name, @diag)

  Added to Test::V1 in 1.302217.

- fail($name)
- fail($name, @diag)

  Added to Test::V1 in 1.302217.

- diag($message)

  Added to Test::V1 in 1.302217.

- note($message)

  Added to Test::V1 in 1.302217.

- $todo = todo($reason)
- todo $reason => sub { ... }

  Added to Test::V1 in 1.302217.
- skip($reason, $count)

  Added to Test::V1 in 1.302217.
- plan($count)

  Added to Test::V1 in 1.302217.
- skip_all($reason)

  Added to Test::V1 in 1.302217.
- done_testing()

  Added to Test::V1 in 1.302217.
- bail_out($reason)

  Added to Test::V1 in 1.302217.

COMPARE

See *Test2::Tools::Compare*.

- is($got, $want, $name)

  Added to Test::V1 in 1.302217.
- isnt($got, $do_not_want, $name)

  Added to Test::V1 in 1.302217.
- like($got, qr/match/, $name)

  Added to Test::V1 in 1.302217.
- unlike($got, qr/mismatch/, $name)

  Added to Test::V1 in 1.302217.
- $check = match(qr/pattern/)

  Added to Test::V1 in 1.302217.
- $check = mismatch(qr/pattern/)

  Added to Test::V1 in 1.302217.
- $check = validator(sub { return $bool })

  Added to Test::V1 in 1.302217.
- $check = hash { ... }

  Added to Test::V1 in 1.302217.
- $check = array { ... }

  Added to Test::V1 in 1.302217.

- $check = bag { ... }

  Added to Test::V1 in 1.302217.
- $check = object { ... }

  Added to Test::V1 in 1.302217.
- $check = meta { ... }

  Added to Test::V1 in 1.302217.
- $check = number($num)

  Added to Test::V1 in 1.302217.
- $check = string($str)

  Added to Test::V1 in 1.302217.
- $check = bool($bool)

  Added to Test::V1 in 1.302217.
- $check = check_isa($class_name)

  Added to Test::V1 in 1.302217.
- $check = in_set(@things)

  Added to Test::V1 in 1.302217.
- $check = not_in_set(@things)

  Added to Test::V1 in 1.302217.
- $check = check_set(@things)

  Added to Test::V1 in 1.302217.
- $check = item($thing)

  Added to Test::V1 in 1.302217.
- $check = item($idx => $thing)

  Added to Test::V1 in 1.302217.
- $check = field($name => $val)

  Added to Test::V1 in 1.302217.
- $check = call($method => $expect)

  Added to Test::V1 in 1.302217.
- $check = call_list($method => $expect)

  Added to Test::V1 in 1.302217.
- $check = call_hash($method => $expect)

  Added to Test::V1 in 1.302217.
- $check = prop($name => $expect)

  Added to Test::V1 in 1.302217.

- $check = check($thing)

  Added to Test::V1 in 1.302217.
- $check = T()

  Added to Test::V1 in 1.302217.
- $check = F()

  Added to Test::V1 in 1.302217.
- $check = D()

  Added to Test::V1 in 1.302217.
- $check = DF()

  Added to Test::V1 in 1.302217.
- $check = E()

  Added to Test::V1 in 1.302217.
- $check = DNE()

  Added to Test::V1 in 1.302217.
- $check = FDNE()

  Added to Test::V1 in 1.302217.
- $check = U()

  Added to Test::V1 in 1.302217.
- $check = L()

  Added to Test::V1 in 1.302217.
- $check = exact_ref($ref)

  Added to Test::V1 in 1.302217.
- end()

  Added to Test::V1 in 1.302217.
- etc()

  Added to Test::V1 in 1.302217.
- filter_items { grep { ... } @_ }

  Added to Test::V1 in 1.302217.
- $check = event $type => ...

  Added to Test::V1 in 1.302217.
- @checks = fail_events $type => ...

  Added to Test::V1 in 1.302217.

CLASSIC COMPARE

See *Test2::Tools::ClassicCompare*.

- cmp_ok($got, $op, $want, $name)

  Added to Test::V1 in 1.302217.

SUBTEST

See *Test2::Tools::Subtest*.

- subtest $name => sub { ... };

  Added to Test::V1 in 1.302217.

  (Note: This is called `subtest_buffered()` in the Tools module.)

CLASS

See *Test2::Tools::Class*.

- can_ok($thing, @methods)

  Added to Test::V1 in 1.302217.

- isa_ok($thing, @classes)

  Added to Test::V1 in 1.302217.

- DOES_ok($thing, @roles)

  Added to Test::V1 in 1.302217.

ENCODING

See *Test2::Tools::Encoding*.

- set_encoding($encoding)

  Added to Test::V1 in 1.302217.

EXPORTS

See *Test2::Tools::Exports*.

- imported_ok('function', '$scalar', ...)

  Added to Test::V1 in 1.302217.

- not_imported_ok('function', '$scalar', ...)

  Added to Test::V1 in 1.302217.

REF

See *Test2::Tools::Ref*.

- ref_ok($ref, $type)

  Added to Test::V1 in 1.302217.

- ref_is($got, $want)

  Added to Test::V1 in 1.302217.

- ref_is_not($got, $do_not_want)

  Added to Test::V1 in 1.302217.

See *Test2::Tools::Refcount*.

- is_refcount($ref, $count, $description)

  Added to Test::V1 in 1.302217.

- is_oneref($ref, $description)

  Added to Test::V1 in 1.302217.

- $count = refcount($ref)

  Added to Test::V1 in 1.302217.

MOCK

See *Test2::Tools::Mock*.

- $control = mock ...

  Added to Test::V1 in 1.302217.

- $bool = mocked($thing)

  Added to Test::V1 in 1.302217.

EXCEPTION

See *Test2::Tools::Exception*.

- $exception = dies { ... }

  Added to Test::V1 in 1.302217.

- $bool = lives { ... }

  Added to Test::V1 in 1.302217.

- $bool = try_ok { ... }

  Added to Test::V1 in 1.302217.

WARNINGS

See *Test2::Tools::Warnings*.

- $count = warns { ... }

  Added to Test::V1 in 1.302217.

- $warning = warning { ... }

  Added to Test::V1 in 1.302217.

- $warnings_ref = warnings { ... }

  Added to Test::V1 in 1.302217.

- $bool = no_warnings { ... }

  Added to Test::V1 in 1.302217.

JUSTIFICATION

Test2::V0 is a rich set of tools. But it made several assumptions about how it would be used. The assumptions are fairly good for new users writing simple scripts, but they can get in the way in many cases.

PROBLEMS WITH V0

- Assumptions of strict/warnings

  Many people would put custom strict/warnings settings at the top of their tests, only to have them wiped out when they use *Test2::V0*.

- Assumptions of UTF8

  Occasionally you do not want this assumption. The way it impacts all your regular and test handles, as well as how your source is read, can be a problem if you are not working with UTF8, or have other plans entirly.

- Huge default set of exports, which can grow

  Sometimes you want to keep your namespace clean.

  Sometimes you import a tool that does not conflict with anything in *Test2::V0*, then we go and add a new tool which conflicts with yours! We make a point not to break/remove exports, but there is no such commitment about adding new ones.

  Now the only default export is `T2()` which gives you a handle where all the tools we expose are provided as methods. You can also use the T2 module (Not bundled with Test-Simple) for use with an identical number of keystrokes, which allow you to leverage the prototypes on the original tool subroutines.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Modules

- `Test2::V1::Base` - Base namespace used for *Test2::Handle* objects created via *Test2::V1*.
- `Test2::V1::Handle` - V1 subclass of *Test2::Handle*.

Test2::V1::Base

Base namespace used for *Test2::Handle* objects created via *Test2::V1*.

DESCRIPTION

This is the default set of functions/methods available in *Test2::V1*.

SYNOPSIS

See *Test2::V1*. This module is not typically used directly.

INCLUDED FUNCTIONALITY

See *"TOOLS" in Test2::V1* for documentation about the tools included here, and when they were added.

Documentation is not duplicated here as that would mean maintaining 2 locations for every change.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::V1::Handle

V1 subclass of *Test2::Handle*.

DESCRIPTION

The *Test2::V1* subclass of the *Test2::Handle* object. This is what you interact with when you use the `T2()` function in a test.

SYNOPSIS

```
use Test2::V1::Handle;

my $t2 = Test2::V1::Handle->new();

$t2->ok(1, "Passing test");
```

SUBCLASS OVERRIDES

The default base class used is *Test2::V1::Base*.

SEE ALSO

See *Test2::Handle* for more information.

SOURCE

The source code repository for Test2-Suite can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist@cpan.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Workflow

A test workflow is a way of structuring tests using composable units.

DESCRIPTION

A test workflow is a way of structuring tests using composable units. A well known example of a test workflow is RSPEC. RSPEC is implemented using Test2::Workflow in *Test2::Tools::Spec* along with several extensions.

IMPORTANT CONCEPTS

BUILD

Test2::Workflow::Build

A Build is used to compose tasks. Usually a build object is pushed to the stack before running code that adds tasks to the build. Once the build sub is complete the build is popped and returned. Usually a build is converted into a root task or task group.

RUNNER

Test2::Workflow::Runner

A runner takes the composed tasks and executes them in the proper order.

TASK

Test2::Workflow::Task

A task is a unit of work to accomplish. There are 2 main types of task.

ACTION

An action is the most simple unit used in composition. An action is essentially a name and a codeblock to run.

GROUP

A group is a task that is composed of other tasks.

EXPORTS

All exports are optional, you must request the ones you want.

- $parsed = parse_args(args => \@args)
- $parsed = parse_args(args => \@args, level => $L)
- $parsed = parse_args(args => \@args, caller => [caller($L)])

  This will parse a "typical" task builders arguments. The `@args` array MUST contain a name (plain scalar containing text) and also a single CODE reference. The `@args` array MAY also contain any quantity of line numbers or hashrefs. The resulting data structure will be a single hashref with all the provided hashrefs squashed together, and the 'name', 'code', 'lines' and 'frame' keys set from other arguments.

```
{
    # All hashrefs from @args get squashed together:
    %squashed_input_hashref_data,

    # @args must have exactly 1 plaintext scalar that is not a␣
↪number, it
    # is considered the name:
    name => 'name from input args'

    # Integer values are treated as line numbers
    lines => [ 35, 44 ],

    # Exactly 1 coderef must be provided in @args:
    code => \&some_code,

    # 'frame' contains the 'caller' data. This may be passed in␣
↪directly,
    # obtained from the 'level' parameter, or automatically␣
↪deduced.
    frame => ['A::Package', 'a_file.pm', 42, ...],
}
```

- $build = init_root($pkg, %args)

  This will initialize (or return the existing) a build for the specified package. `%args` get passed into the *Test2::Workflow::Build* constructor. This uses the following defaults (which can be overridden using `%args`):

```
name    => $pkg,
flat    => 1,
iso     => 0,
async   => 0,
is_root => 1,
```

Note that `%args` is completely ignored if the package build has already been initialized.

- $build = root_build($pkg)

  This will return the root build for the specified package.

- $build = current_build()

  This will return the build currently at the top of the build stack (or undef).

- $build = build($name, \%params, sub { ... })

  This will push a new build object onto the build stash then run the provided codeblock. Once the codeblock has finished running the build will be popped off the stack and returned.

  See `parse_args()` for details about argument processing.

SEE ALSO

- Test2::Tools::Spec

  Test2::Tools::Spec is an implementation of RSPEC using this library.

SOURCE

The source code repository for Test2-Workflow can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist7@gmail.com.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Modules

- `Test2::Workflow::BlockBase` - Base class for all workflow blocks.
- `Test2::Workflow::Build` - Represents a build in progress.
- `Test2::Workflow::Runner` - Runs the workflows.
- `Test2::Workflow::Task` - Encapsulation of a Task

Test2::Workflow::BlockBase

Base class for all workflow blocks.

SOURCE

The source code repository for Test2-Workflow can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist7@gmail.com.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Workflow::Build

Represents a build in progress.

SOURCE

The source code repository for Test2-Workflow can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist7@gmail.com.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Workflow::Runner

Runs the workflows.

SOURCE

The source code repository for Test2-Workflow can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist7@gmail.com.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Workflow::Task

Encapsulation of a Task

SOURCE

The source code repository for Test2-Workflow can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist7@gmail.com.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Modules

- `Test2::Workflow::Task::Action` - Encapsulation of an action.
- `Test2::Workflow::Task::Group` - Encapsulation of a group (describe).

Test2::Workflow::Task::Action

Encapsulation of an action.

SOURCE

The source code repository for Test2-Workflow can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist7@gmail.com.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

Test2::Workflow::Task::Group

Encapsulation of a group (describe).

SOURCE

The source code repository for Test2-Workflow can be found at `https://github.com/Test-More/test-more/`.

MAINTAINERS

- Chad Granum exodist@cpan.org

AUTHORS

- Chad Granum exodist@cpan.org

COPYRIGHT

Copyright Chad Granum exodist7@gmail.com.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See `http://dev.perl.org/licenses/`

5.2.103 Text

Namespace containing the following modules:

- `Text::Abbrev` - abbrev - create an abbreviation table from a list
- `Text::Balanced` - Extract delimited text sequences from strings.
- `Text::CSV` - This section is taken from Text::CSV_XS.
- `Text::CSV_PP` - Text::CSV_XS compatible pure-Perl module
- `Text::LevenshteinXS` - The Levenshtein edit distance of two strings: the fewest insertions, deletions and substitutions that turn one into the other.
- `Text::LineFold` - Line Folding for Plain Text
- `Text::ParseWords` - parse text into an array of tokens or array of arrays
- `Text::Tabs` - expand and unexpand tabs like unix expand(1) and unexpand(1)
- `Text::Wrap` - line wrapping to form simple paragraphs

Text::Abbrev

abbrev - create an abbreviation table from a list

SYNOPSIS

```
use Text::Abbrev;
abbrev $hashref, LIST
```

DESCRIPTION

Stores all unambiguous truncations of each element of LIST as keys in the associative array referenced by `$hashref`. The values are the original list elements.

EXAMPLE

```
$hashref = abbrev qw(list edit send abort gripe);

%hash = abbrev qw(list edit send abort gripe);

abbrev $hashref, qw(list edit send abort gripe);

abbrev(*hash, qw(list edit send abort gripe));
```

Text::Balanced

Extract delimited text sequences from strings.

SYNOPSIS

```
use Text::Balanced qw (
    extract_delimited
    extract_bracketed
    extract_quotelike
    extract_codeblock
    extract_variable
    extract_tagged
    extract_multiple
    gen_delimited_pat
    gen_extract_tagged
);

# Extract the initial substring of $text that is delimited by
# two (unescaped) instances of the first character in $delim.

($extracted, $remainder) = extract_delimited($text,$delim);

# Extract the initial substring of $text that is bracketed
# with a delimiter(s) specified by $delim (where the string
# in $delim contains one or more of '(){}[]<>').

($extracted, $remainder) = extract_bracketed($text,$delim);

# Extract the initial substring of $text that is bounded by
# an XML tag.

($extracted, $remainder) = extract_tagged($text);

# Extract the initial substring of $text that is bounded by
# a C<BEGIN>...C<END> pair. Don't allow nested C<BEGIN> tags

($extracted, $remainder) =
    extract_tagged($text,"BEGIN","END",undef,{bad=>["BEGIN"]});

# Extract the initial substring of $text that represents a
# Perl "quote or quote-like operation"

($extracted, $remainder) = extract_quotelike($text);

# Extract the initial substring of $text that represents a block
# of Perl code, bracketed by any of character(s) specified by $delim
# (where the string $delim contains one or more of '(){}[]<>').

($extracted, $remainder) = extract_codeblock($text,$delim);
```

(continues on next page)

(continued from previous page)

```
# Extract the initial substrings of $text that would be extracted by
# one or more sequential applications of the specified functions
# or regular expressions

@extracted = extract_multiple($text,
                              [ \&extract_bracketed,
                                \&extract_quotelike,
                                \&some_other_extractor_sub,
                                qr/[xyz]*/,
                                'literal',
                              ]);

# Create a string representing an optimized pattern (a la Friedl)
# that matches a substring delimited by any of the specified␣
 ↪characters
# (in this case: any type of quote or a slash)

$patstring = gen_delimited_pat(q{'"`/});

# Generate a reference to an anonymous sub that is just like␣
 ↪extract_tagged
# but pre-compiled and optimized for a specific pair of tags, and
# consequently much faster (i.e. 3 times faster). It uses qr// for␣
 ↪better
# performance on repeated calls.

$extract_head = gen_extract_tagged('<HEAD>','</HEAD>');
($extracted, $remainder) = $extract_head->($text);
```

DESCRIPTION

The various `extract_...` subroutines may be used to extract a delimited substring, possibly after skipping a specified prefix string. By default, that prefix is optional whitespace (`/\s*/`), but you can change it to whatever you wish (see below).

The substring to be extracted must appear at the current `pos` location of the string's variable (or at index zero, if no `pos` position is defined). In other words, the `extract_...` subroutines *don't* extract the first occurrence of a substring anywhere in a string (like an unanchored regex would). Rather, they extract an occurrence of the substring appearing immediately at the current matching position in the string (like a `\G`-anchored regex would).

General Behaviour in List Contexts

In a list context, all the subroutines return a list, the first three elements of which are always:

- [0]

The extracted string, including the specified delimiters. If the extraction fails `undef` is returned.

- [1]

  The remainder of the input string (i.e. the characters after the extracted string). On failure, the entire string is returned.

- [2]

  The skipped prefix (i.e. the characters before the extracted string). On failure, `undef` is returned.

Note that in a list context, the contents of the original input text (the first argument) are not modified in any way.

However, if the input text was passed in a variable, that variable's `pos` value is updated to point at the first character after the extracted text. That means that in a list context the various subroutines can be used much like regular expressions. For example:

```
while ( $next = (extract_quotelike($text))[0] )
{
    # process next quote-like (in $next)
}
```

General Behaviour in Scalar and Void Contexts

In a scalar context, the extracted string is returned, having first been removed from the input text. Thus, the following code also processes each quote-like operation, but actually removes them from $text:

```
while ( $next = extract_quotelike($text) )
{
    # process next quote-like (in $next)
}
```

Note that if the input text is a read-only string (i.e. a literal), no attempt is made to remove the extracted text.

In a void context the behaviour of the extraction subroutines is exactly the same as in a scalar context, except (of course) that the extracted substring is not returned.

A Note About Prefixes

Prefix patterns are matched without any trailing modifiers (`/gimsox` etc.) This can bite you if you're expecting a prefix specification like '.*?(?=)' to skip everything up to the first tag. Such a prefix pattern will only succeed if the tag is on the current line, since . normally doesn't match newlines.

To overcome this limitation, you need to turn on /s matching within the prefix pattern, using the `(?s)` directive: '(?s).*?(?=)'

Functions

- `extract_delimited`

  The `extract_delimited` function formalizes the common idiom of extracting a single-character-delimited substring from the start of a string. For example, to extract a single-quote delimited string, the following code is typically used:

```
($remainder = $text) =~ s/\A('(\\.|[^'])*')//s;
$extracted = $1;
```

  but with `extract_delimited` it can be simplified to:

```
($extracted,$remainder) = extract_delimited($text, "'");
```

  `extract_delimited` takes up to four scalars (the input text, the delimiters, a prefix pattern to be skipped, and any escape characters) and extracts the initial substring of the text that is appropriately delimited. If the delimiter string has multiple characters, the first one encountered in the text is taken to delimit the substring. The third argument specifies a prefix pattern that is to be skipped (but must be present!) before the substring is extracted. The final argument specifies the escape character to be used for each delimiter.

  All arguments are optional. If the escape characters are not specified, every delimiter is escaped with a backslash (`\`). If the prefix is not specified, the pattern `'\s*'` - optional whitespace - is used. If the delimiter set is also not specified, the set `/["'`]/` is used. If the text to be processed is not specified either, `$_` is used.

  In list context, `extract_delimited` returns a array of three elements, the extracted substring (*including the surrounding delimiters*), the remainder of the text, and the skipped prefix (if any). If a suitable delimited substring is not found, the first element of the array is the empty string, the second is the complete original text, and the prefix returned in the third element is an empty string.

  In a scalar context, just the extracted substring is returned. In a void context, the extracted substring (and any prefix) are simply removed from the beginning of the first argument.

  Examples:

```
# Remove a single-quoted substring from the very beginning of
 ↪$text:

    $substring = extract_delimited($text, "'", '');

# Remove a single-quoted Pascalish substring (i.e. one in which
# doubling the quote character escapes it) from the very
# beginning of $text:

    $substring = extract_delimited($text, "'", '', "'");

# Extract a single- or double- quoted substring from the
# beginning of $text, optionally after some whitespace
# (note the list context to protect $text from modification):
```

(continues on next page)

(continued from previous page)

```
    ($substring) = extract_delimited $text, q{"'};

# Delete the substring delimited by the first '/' in $text:

    $text = join '', (extract_delimited($text,'/','[^/]*')[2,1];
```

Note that this last example is *not* the same as deleting the first quote-like pattern. For instance, if `$text` contained the string:

```
"if ('./cmd' =~ m/$UNIXCMD/s) { $cmd = $1; }"
```

then after the deletion it would contain:

```
"if ('.$UNIXCMD/s) { $cmd = $1; }"
```

not:

```
"if ('./cmd' =~ ms) { $cmd = $1; }"
```

See "extract_quotelike" for a (partial) solution to this problem.

- `extract_bracketed`

  Like `"extract_delimited"`, the `extract_bracketed` function takes up to three optional scalar arguments: a string to extract from, a delimiter specifier, and a prefix pattern. As before, a missing prefix defaults to optional whitespace and a missing text defaults to `$_`. However, a missing delimiter specifier defaults to `'{}()[]<>'` (see below).

  `extract_bracketed` extracts a balanced-bracket-delimited substring (using any one (or more) of the user-specified delimiter brackets: '(..)', '{..}', '[..]', or '<..>'). Optionally it will also respect quoted unbalanced brackets (see below).

  A "delimiter bracket" is a bracket in list of delimiters passed as `extract_bracketed`'s second argument. Delimiter brackets are specified by giving either the left or right (or both!) versions of the required bracket(s). Note that the order in which two or more delimiter brackets are specified is not significant.

  A "balanced-bracket-delimited substring" is a substring bounded by matched brackets, such that any other (left or right) delimiter bracket *within* the substring is also matched by an opposite (right or left) delimiter bracket *at the same level of nesting*. Any type of bracket not in the delimiter list is treated as an ordinary character.

  In other words, each type of bracket specified as a delimiter must be balanced and correctly nested within the substring, and any other kind of ("non-delimiter") bracket in the substring is ignored.

  For example, given the string:

```
$text = "{ an '[irregularly :-(] {} parenthesized >:-)' string }
 ↪";
```

  then a call to `extract_bracketed` in a list context:

```
@result = extract_bracketed( $text, '{}' );
```

would return:

```
( "{ an '[irregularly :-(] {} parenthesized >:-)' string }" , "
↪" , "" )
```

since both sets of `'{..}'` brackets are properly nested and evenly balanced. (In a scalar context just the first element of the array would be returned. In a void context, `$text` would be replaced by an empty string.)

Likewise the call in:

```
@result = extract_bracketed( $text, '{[' );
```

would return the same result, since all sets of both types of specified delimiter brackets are correctly nested and balanced.

However, the call in:

```
@result = extract_bracketed( $text, '{([<' );
```

would fail, returning:

```
( undef , "{ an '[irregularly :-(] {} parenthesized >:-)'␣
↪string }"  );
```

because the embedded pairs of `'(..)'`s and `'[..]'`s are "cross-nested" and the embedded `'>'` is unbalanced. (In a scalar context, this call would return an empty string. In a void context, `$text` would be unchanged.)

Note that the embedded single-quotes in the string don't help in this case, since they have not been specified as acceptable delimiters and are therefore treated as non-delimiter characters (and ignored).

However, if a particular species of quote character is included in the delimiter specification, then that type of quote will be correctly handled. for example, if `$text` is:

```
$text = '<A HREF=">>>>">link</A>';
```

then

```
@result = extract_bracketed( $text, '<">' );
```

returns:

```
( '<A HREF=">>>>">', 'link</A>', "" )
```

as expected. Without the specification of " as an embedded quoter:

```
@result = extract_bracketed( $text, '<>' );
```

the result would be:

```
( '<A HREF=">', '>>>">link</A>', "" )
```

In addition to the quote delimiters ', ", and `, full Perl quote-like quoting (i.e. q{string}, qq{string}, etc) can be specified by including the letter 'q' as a delimiter. Hence:

```
@result = extract_bracketed( $text, '<q>' );
```

would correctly match something like this:

```
$text = '<leftop: conj /and/ conj>';
```

See also: "extract_quotelike" and "extract_codeblock".

- extract_variable

  extract_variable extracts any valid Perl variable or variable-involved expression, including scalars, arrays, hashes, array accesses, hash look-ups, method calls through objects, subroutine calls through subroutine references, etc.

  The subroutine takes up to two optional arguments:

  1. A string to be processed ($_ if the string is omitted or undef)
  2. A string specifying a pattern to be matched as a prefix (which is to be skipped). If omitted, optional whitespace is skipped.

  On success in a list context, an array of 3 elements is returned. The elements are:

  – [0]

    the extracted variable, or variablish expression

  – [1]

    the remainder of the input text,

  – [2]

    the prefix substring (if any),

  On failure, all of these values (except the remaining text) are undef.

  In a scalar context, extract_variable returns just the complete substring that matched a variablish expression. undef is returned on failure. In addition, the original input text has the returned substring (and any prefix) removed from it.

  In a void context, the input text just has the matched substring (and any specified prefix) removed.

- extract_tagged

  extract_tagged extracts and segments text between (balanced) specified tags.

  The subroutine takes up to five optional arguments:

  1. A string to be processed ($_ if the string is omitted or undef)
  2. A string specifying a pattern (i.e. regex) to be matched as the opening tag. If the pattern string is omitted (or undef) then a pattern that matches any standard XML tag is used.

3. A string specifying a pattern to be matched at the closing tag. If the pattern string is omitted (or `undef`) then the closing tag is constructed by inserting a `/` after any leading bracket characters in the actual opening tag that was matched (*not* the pattern that matched the tag). For example, if the opening tag pattern is specified as `'{{\w+}}'` and actually matched the opening tag `"{{DATA}}"`, then the constructed closing tag would be `"{{/DATA}}"`.
4. A string specifying a pattern to be matched as a prefix (which is to be skipped). If omitted, optional whitespace is skipped.
5. A hash reference containing various parsing options (see below)

The various options that can be specified are:

- `reject => $listref`

  The list reference contains one or more strings specifying patterns that must *not* appear within the tagged text.

  For example, to extract an HTML link (which should not contain nested links) use:

```
extract_tagged($text, '<A>', '</A>', undef, {reject => ['<A>
↪']} );
```

- `ignore => $listref`

  The list reference contains one or more strings specifying patterns that are *not* to be treated as nested tags within the tagged text (even if they would match the start tag pattern).

  For example, to extract an arbitrary XML tag, but ignore "empty" elements:

```
extract_tagged($text, undef, undef, undef, {ignore => ['<[^>
↪]*/>']} );
```

  (also see "gen_delimited_pat" below).

- `fail => $str`

  The `fail` option indicates the action to be taken if a matching end tag is not encountered (i.e. before the end of the string or some `reject` pattern matches). By default, a failure to match a closing tag causes `extract_tagged` to immediately fail.

  However, if the string value associated with <reject> is "MAX", then `extract_tagged` returns the complete text up to the point of failure. If the string is "PARA", `extract_tagged` returns only the first paragraph after the tag (up to the first line that is either empty or contains only whitespace characters). If the string is "", the default behaviour (i.e. failure) is reinstated.

  For example, suppose the start tag "/para" introduces a paragraph, which then continues until the next "/endpara" tag or until another "/para" tag is encountered:

```
$text = "/para line 1\n\nline 3\n/para line 4";
```

(continues on next page)

(continued from previous page)

```
extract_tagged($text, '/para', '/endpara', undef,
                        {reject => '/para', fail => MAX );

# EXTRACTED: "/para line 1\n\nline 3\n"
```

Suppose instead, that if no matching "/endpara" tag is found, the "/para" tag refers only to the immediately following paragraph:

```
$text = "/para line 1\n\nline 3\n/para line 4";

extract_tagged($text, '/para', '/endpara', undef,
                 {reject => '/para', fail => MAX );

# EXTRACTED: "/para line 1\n"
```

Note that the specified `fail` behaviour applies to nested tags as well.

On success in a list context, an array of 6 elements is returned. The elements are:

– [0]

the extracted tagged substring (including the outermost tags),

– [1]

the remainder of the input text,

– [2]

the prefix substring (if any),

– [3]

the opening tag

– [4]

the text between the opening and closing tags

– [5]

the closing tag (or "" if no closing tag was found)

On failure, all of these values (except the remaining text) are `undef`.

In a scalar context, `extract_tagged` returns just the complete substring that matched a tagged text (including the start and end tags). `undef` is returned on failure. In addition, the original input text has the returned substring (and any prefix) removed from it.

In a void context, the input text just has the matched substring (and any specified prefix) removed.

- `gen_extract_tagged`

  `gen_extract_tagged` generates a new anonymous subroutine which extracts text between (balanced) specified tags. In other words, it generates a function identical in function to `extract_tagged`.

The difference between `extract_tagged` and the anonymous subroutines generated by `gen_extract_tagged`, is that those generated subroutines:

- do not have to reparse tag specification or parsing options every time they are called (whereas `extract_tagged` has to effectively rebuild its tag parser on every call);
- make use of the new qr// construct to pre-compile the regexes they use (whereas `extract_tagged` uses standard string variable interpolation to create tag-matching patterns).

The subroutine takes up to four optional arguments (the same set as `extract_tagged` except for the string to be processed). It returns a reference to a subroutine which in turn takes a single argument (the text to be extracted from).

In other words, the implementation of `extract_tagged` is exactly equivalent to:

```
sub extract_tagged
{
        my $text = shift;
        $extractor = gen_extract_tagged(@_);
        return $extractor->($text);
}
```

(although `extract_tagged` is not currently implemented that way).

Using `gen_extract_tagged` to create extraction functions for specific tags is a good idea if those functions are going to be called more than once, since their performance is typically twice as good as the more general-purpose `extract_tagged`.

- `extract_quotelike`

`extract_quotelike` attempts to recognize, extract, and segment any one of the various Perl quotes and quotelike operators (see perlop(3)) Nested backslashed delimiters, embedded balanced bracket delimiters (for the quotelike operators), and trailing modifiers are all caught. For example, in:

```
extract_quotelike 'q # an octothorpe: \# (not the end of the q!
↪) #'

extract_quotelike '  "You said, \"Use sed\"."  '

extract_quotelike ' s{([A-Z]{1,8}\.[A-Z]{3})} /\L$1\E/; '

extract_quotelike ' tr/\\\/\\\\/\\\\//ds; '
```

the full Perl quotelike operations are all extracted correctly.

Note too that, when using the /x modifier on a regex, any comment containing the current pattern delimiter will cause the regex to be immediately terminated. In other words:

```
'm /
        (?i)            # CASE INSENSITIVE
        [a-z_]          # LEADING ALPHABETIC/UNDERSCORE
```

(continues on next page)

(continued from previous page)

```
        [a-z0-9]*       # FOLLOWED BY ANY NUMBER OF␣
↪ALPHANUMERICS
  /x'
```

will be extracted as if it were:

```
'm /
        (?i)            # CASE INSENSITIVE
        [a-z_]          # LEADING ALPHABETIC/'
```

This behaviour is identical to that of the actual compiler.

`extract_quotelike` takes two arguments: the text to be processed and a prefix to be matched at the very beginning of the text. If no prefix is specified, optional whitespace is the default. If no text is given, `$_` is used.

In a list context, an array of 11 elements is returned. The elements are:

- [0]

  the extracted quotelike substring (including trailing modifiers),

- [1]

  the remainder of the input text,

- [2]

  the prefix substring (if any),

- [3]

  the name of the quotelike operator (if any),

- [4]

  the left delimiter of the first block of the operation,

- [5]

  the text of the first block of the operation (that is, the contents of a quote, the regex of a match or substitution or the target list of a translation),

- [6]

  the right delimiter of the first block of the operation,

- [7]

  the left delimiter of the second block of the operation (that is, if it is a `s`, `tr`, or `y`),

- [8]

  the text of the second block of the operation (that is, the replacement of a substitution or the translation list of a translation),

- [9]

  the right delimiter of the second block of the operation (if any),

– [10]

the trailing modifiers on the operation (if any).

For each of the fields marked "(if any)" the default value on success is an empty string. On failure, all of these values (except the remaining text) are undef.

In a scalar context, extract_quotelike returns just the complete substring that matched a quotelike operation (or undef on failure). In a scalar or void context, the input text has the same substring (and any specified prefix) removed.

Examples:

```
# Remove the first quotelike literal that appears in text

        $quotelike = extract_quotelike($text,'.*?');

# Replace one or more leading whitespace-separated quotelike
# literals in $_ with "<QLL>"

        do { $_ = join '<QLL>', (extract_quotelike)[2,1] }␣
↪until $@;


# Isolate the search pattern in a quotelike operation from $text

        ($op,$pat) = (extract_quotelike $text)[3,5];
        if ($op =~ /[ms]/)
        {
                print "search pattern: $pat\n";
        }
        else
        {
                print "$op is not a pattern matching operation\n
↪";
        }
```

- extract_quotelike

  extract_quotelike can successfully extract "here documents" from an input string, but with an important caveat in list contexts.

  Unlike other types of quote-like literals, a here document is rarely a contiguous substring. For example, a typical piece of code using here document might look like this:

```
<<'EOMSG' || die;
This is the message.
EOMSG
exit;
```

  Given this as an input string in a scalar context, extract_quotelike would correctly return the string "<<'EOMSG'\nThis is the message.\nEOMSG", leaving the string " || die;\nexit;" in the original variable. In other words, the two separate pieces of the here document are successfully extracted and concatenated.

In a list context, `extract_quotelike` would return the list

- [0]

  "<<'EOMSG'\nThis is the message.\nEOMSG\n" (i.e. the full extracted here document, including fore and aft delimiters),

- [1]

  " || die;\nexit;" (i.e. the remainder of the input text, concatenated),

- [2]

  "" (i.e. the prefix substring – trivial in this case),

- [3]

  "<<" (i.e. the "name" of the quotelike operator)

- [4]

  "'EOMSG'" (i.e. the left delimiter of the here document, including any quotes),

- [5]

  "This is the message.\n" (i.e. the text of the here document),

- [6]

  "EOMSG" (i.e. the right delimiter of the here document),

- [7..10]

  "" (a here document has no second left delimiter, second text, second right delimiter, or trailing modifiers).

However, the matching position of the input variable would be set to "exit;" (i.e. *after* the closing delimiter of the here document), which would cause the earlier " || die;\nexit;" to be skipped in any sequence of code fragment extractions.

To avoid this problem, when it encounters a here document whilst extracting from a modifiable string, `extract_quotelike` silently rearranges the string to an equivalent piece of Perl:

```
<<'EOMSG'
This is the message.
EOMSG
|| die;
exit;
```

in which the here document *is* contiguous. It still leaves the matching position after the here document, but now the rest of the line on which the here document starts is not skipped.

To prevent <extract_quotelike> from mucking about with the input in this way (this is the only case where a list-context `extract_quotelike` does so), you can pass the input variable as an interpolated literal:

```
$quotelike = extract_quotelike("$var");
```

- `extract_codeblock`

  `extract_codeblock` attempts to recognize and extract a balanced bracket delimited substring that may contain unbalanced brackets inside Perl quotes or quotelike operations. That is, `extract_codeblock` is like a combination of "`extract_bracketed`" and "`extract_quotelike`".

  `extract_codeblock` takes the same initial three parameters as `extract_bracketed`: a text to process, a set of delimiter brackets to look for, and a prefix to match first. It also takes an optional fourth parameter, which allows the outermost delimiter brackets to be specified separately (see below), and a fifth parameter used only by Parse::RecDescent.

  Omitting the first argument (input text) means process `$_` instead. Omitting the second argument (delimiter brackets) indicates that only '`{`' is to be used. Omitting the third argument (prefix argument) implies optional whitespace at the start. Omitting the fourth argument (outermost delimiter brackets) indicates that the value of the second argument is to be used for the outermost delimiters.

  Once the prefix and the outermost opening delimiter bracket have been recognized, code blocks are extracted by stepping through the input text and trying the following alternatives in sequence:

  1. Try and match a closing delimiter bracket. If the bracket was the same species as the last opening bracket, return the substring to that point. If the bracket was mismatched, return an error.
  2. Try to match a quote or quotelike operator. If found, call `extract_quotelike` to eat it. If `extract_quotelike` fails, return the error it returned. Otherwise go back to step 1.
  3. Try to match an opening delimiter bracket. If found, call `extract_codeblock` recursively to eat the embedded block. If the recursive call fails, return an error. Otherwise, go back to step 1.
  4. Unconditionally match a bareword or any other single character, and then go back to step 1.

  Examples:

```
# Find a while loop in the text

        if ($text =~ s/.*?while\s*\{/{/)
        {
                $loop = "while " . extract_codeblock($text);
        }

# Remove the first round-bracketed list (which may include
# round- or curly-bracketed code blocks or quotelike operators)

        extract_codeblock $text, "(){}", '[^(]*';
```

  The ability to specify a different outermost delimiter bracket is useful in some circumstances. For example, in the Parse::RecDescent module, parser actions which are to be performed only on a successful parse are specified using a `<defer:...>` directive. For example:

```
sentence: subject verb object
                <defer: {$::theVerb = $item{verb}} >
```

Parse::RecDescent uses `extract_codeblock($text, '{}<>')` to extract the code within the `<defer:...>` directive, but there's a problem.

A deferred action like this:

```
<defer: {if ($count>10) {$count--}} >
```

will be incorrectly parsed as:

```
<defer: {if ($count>
```

because the "less than" operator is interpreted as a closing delimiter.

But, by extracting the directive using `extract_codeblock($text, '{}', undef, '<>')` the '>' character is only treated as a delimited at the outermost level of the code block, so the directive is parsed correctly.

- `extract_multiple`

  The `extract_multiple` subroutine takes a string to be processed and a list of extractors (subroutines or regular expressions) to apply to that string.

  In an array context `extract_multiple` returns an array of substrings of the original string, as extracted by the specified extractors. In a scalar context, `extract_multiple` returns the first substring successfully extracted from the original string. In both scalar and void contexts the original string has the first successfully extracted substring removed from it. In all contexts `extract_multiple` starts at the current `pos` of the string, and sets that `pos` appropriately after it matches.

  Hence, the aim of a call to `extract_multiple` in a list context is to split the processed string into as many non-overlapping fields as possible, by repeatedly applying each of the specified extractors to the remainder of the string. Thus `extract_multiple` is a generalized form of Perl's `split` subroutine.

  The subroutine takes up to four optional arguments:

  1. A string to be processed (`$_` if the string is omitted or `undef`)
  2. A reference to a list of subroutine references and/or qr// objects and/or literal strings and/or hash references, specifying the extractors to be used to split the string. If this argument is omitted (or `undef`) the list:

     ```
     [
             sub { extract_variable($_[0], '') },
             sub { extract_quotelike($_[0],'') },
             sub { extract_codeblock($_[0],'{}','') },
     ]
     ```

     is used.
  3. An number specifying the maximum number of fields to return. If this argument is omitted (or `undef`), split continues as long as possible.

If the third argument is *N*, then extraction continues until *N* fields have been successfully extracted, or until the string has been completely processed.

Note that in scalar and void contexts the value of this argument is automatically reset to 1 (under `-w`, a warning is issued if the argument has to be reset).

4. A value indicating whether unmatched substrings (see below) within the text should be skipped or returned as fields. If the value is true, such substrings are skipped. Otherwise, they are returned.

The extraction process works by applying each extractor in sequence to the text string.

If the extractor is a subroutine it is called in a list context and is expected to return a list of a single element, namely the extracted text. It may optionally also return two further arguments: a string representing the text left after extraction (like $' for a pattern match), and a string representing any prefix skipped before the extraction (like $` in a pattern match). Note that this is designed to facilitate the use of other Text::Balanced subroutines with `extract_multiple`. Note too that the value returned by an extractor subroutine need not bear any relationship to the corresponding substring of the original text (see examples below).

If the extractor is a precompiled regular expression or a string, it is matched against the text in a scalar context with a leading '\G' and the gc modifiers enabled. The extracted value is either $1 if that variable is defined after the match, or else the complete match (i.e. $&).

If the extractor is a hash reference, it must contain exactly one element. The value of that element is one of the above extractor types (subroutine reference, regular expression, or string). The key of that element is the name of a class into which the successful return value of the extractor will be blessed.

If an extractor returns a defined value, that value is immediately treated as the next extracted field and pushed onto the list of fields. If the extractor was specified in a hash reference, the field is also blessed into the appropriate class,

If the extractor fails to match (in the case of a regex extractor), or returns an empty list or an undefined value (in the case of a subroutine extractor), it is assumed to have failed to extract. If none of the extractor subroutines succeeds, then one character is extracted from the start of the text and the extraction subroutines reapplied. Characters which are thus removed are accumulated and eventually become the next field (unless the fourth argument is true, in which case they are discarded).

For example, the following extracts substrings that are valid Perl variables:

```
@fields = extract_multiple($text,
                           [ sub { extract_variable($_[0]) } ],
                           undef, 1);
```

This example separates a text into fields which are quote delimited, curly bracketed, and anything else. The delimited and bracketed parts are also blessed to identify them (the "anything else" is unblessed):

```
@fields = extract_multiple($text,
          [
```

(continues on next page)

(continued from previous page)

```
                { Delim => sub { extract_delimited($_[0],q{'"}) }
↪,
                { Brack => sub { extract_bracketed($_[0],'{}') }
↪ },
            ]);
```

This call extracts the next single substring that is a valid Perl quotelike operator (and removes it from $text):

```
$quotelike = extract_multiple($text,
                              [
                                sub { extract_quotelike($_[0]) }
↪,
                              ], undef, 1);
```

Finally, here is yet another way to do comma-separated value parsing:

```
$csv_text = "a,'x b',c";
@fields = extract_multiple($csv_text,
                           [
                                 sub { extract_delimited($_[0],q{
↪'"}) },
                                 qr/([^,]+)/,
                           ],
                           undef,1);
# @fields is now ('a', "'x b'", 'c')
```

The list in the second argument means: *"Try and extract a ' or " delimited string, otherwise extract anything up to a comma..."*. The undef third argument means: *"...as many times as possible..."*, and the true value in the fourth argument means *"...discarding anything else that appears (i.e. the commas)"*.

If you wanted the commas preserved as separate fields (i.e. like split does if your split pattern has capturing parentheses), you would just make the last parameter undefined (or remove it).

- `gen_delimited_pat`

  The `gen_delimited_pat` subroutine takes a single (string) argument and builds a Friedl-style optimized regex that matches a string delimited by any one of the characters in the single argument. For example:

  ```
  gen_delimited_pat(q{'"})
  ```

  returns the regex:

  ```
  (?:\"(?:\\\"|(?!\").)*\"|\'(?:\\\'|(?!\').)*\')
  ```

  Note that the specified delimiters are automatically quotemeta'd.

  A typical use of `gen_delimited_pat` would be to build special purpose tags for `extract_tagged`. For example, to properly ignore "empty" XML elements (which might contain quoted strings):

```
my $empty_tag = '<(' . gen_delimited_pat(q{'"}) . '|.)+/>';

extract_tagged($text, undef, undef, undef, {ignore => [$empty_
↪tag]} );
```

gen_delimited_pat may also be called with an optional second argument, which specifies the "escape" character(s) to be used for each delimiter. For example to match a Pascal-style string (where ' is the delimiter and '' is a literal ' within the string):

```
gen_delimited_pat(q{'},q{'});
```

Different escape characters can be specified for different delimiters. For example, to specify that '/' is the escape for single quotes and '%' is the escape for double quotes:

```
gen_delimited_pat(q{'"},q{/%});
```

If more delimiters than escape chars are specified, the last escape char is used for the remaining delimiters. If no escape char is specified for a given specified delimiter, '' is used.

- `delimited_pat`

  Note that `gen_delimited_pat` was previously called `delimited_pat`. That name may still be used, but is now deprecated.

DIAGNOSTICS

In a list context, all the functions return `(undef,$original_text)` on failure. In a scalar context, failure is indicated by returning `undef` (in this case the input text is not modified in any way).

In addition, on failure in *any* context, the `$@` variable is set. Accessing `$@->{error}` returns one of the error diagnostics listed below. Accessing `$@->{pos}` returns the offset into the original string at which the error was detected (although not necessarily where it occurred!) Printing `$@` directly produces the error message, with the offset appended. On success, the `$@` variable is guaranteed to be `undef`.

The available diagnostics are:

- `Did not find a suitable bracket: "%s"`

  The delimiter provided to `extract_bracketed` was not one of `'()[]<>{}'`.

- `Did not find prefix: /%s/`

  A non-optional prefix was specified but wasn't found at the start of the text.

- `Did not find opening bracket after prefix: "%s"`

  `extract_bracketed` or `extract_codeblock` was expecting a particular kind of bracket at the start of the text, and didn't find it.

- `No quotelike operator found after prefix: "%s"`

`extract_quotelike` didn't find one of the quotelike operators `q`, `qq`, `qw`, `qx`, `s`, `tr` or `y` at the start of the substring it was extracting.

- `Unmatched closing bracket: "%c"`

  `extract_bracketed`, `extract_quotelike` or `extract_codeblock` encountered a closing bracket where none was expected.

- `Unmatched opening bracket(s): "%s"`

  `extract_bracketed`, `extract_quotelike` or `extract_codeblock` ran out of characters in the text before closing one or more levels of nested brackets.

- `Unmatched embedded quote (%s)`

  `extract_bracketed` attempted to match an embedded quoted substring, but failed to find a closing quote to match it.

- `Did not find closing delimiter to match '%s'`

  `extract_quotelike` was unable to find a closing delimiter to match the one that opened the quote-like operation.

- `Mismatched closing bracket: expected "%c" but found "%s"`

  `extract_bracketed`, `extract_quotelike` or `extract_codeblock` found a valid bracket delimiter, but it was the wrong species. This usually indicates a nesting error, but may indicate incorrect quoting or escaping.

- `No block delimiter found after quotelike "%s"`

  `extract_quotelike` or `extract_codeblock` found one of the quotelike operators `q`, `qq`, `qw`, `qx`, `s`, `tr` or `y` without a suitable block after it.

- `Did not find leading dereferencer`

  `extract_variable` was expecting one of '$', '@', or '%' at the start of a variable, but didn't find any of them.

- `Bad identifier after dereferencer`

  `extract_variable` found a '$', '@', or '%' indicating a variable, but that character was not followed by a legal Perl identifier.

- `Did not find expected opening bracket at %s`

  `extract_codeblock` failed to find any of the outermost opening brackets that were specified.

- `Improperly nested codeblock at %s`

  A nested code block was found that started with a delimiter that was specified as being only to be used as an outermost bracket.

- `Missing second block for quotelike "%s"`

  `extract_codeblock` or `extract_quotelike` found one of the quotelike operators `s`, `tr` or `y` followed by only one block.

- `No match found for opening bracket`

  `extract_codeblock` failed to find a closing bracket to match the outermost opening bracket.

- `Did not find opening tag: /%s/`

  `extract_tagged` did not find a suitable opening tag (after any specified prefix was removed).

- `Unable to construct closing tag to match: /%s/`

  `extract_tagged` matched the specified opening tag and tried to modify the matched text to produce a matching closing tag (because none was specified). It failed to generate the closing tag, almost certainly because the opening tag did not start with a bracket of some kind.

- `Found invalid nested tag: %s`

  `extract_tagged` found a nested tag that appeared in the "reject" list (and the failure mode was not "MAX" or "PARA").

- `Found unbalanced nested tag: %s`

  `extract_tagged` found a nested opening tag that was not matched by a corresponding nested closing tag (and the failure mode was not "MAX" or "PARA").

- `Did not find closing tag`

  `extract_tagged` reached the end of the text without finding a closing tag to match the original opening tag (and the failure mode was not "MAX" or "PARA").

EXPORTS

The following symbols are, or can be, exported by this module:

- Default Exports

  None.

- Optional Exports

  `extract_delimited`, `extract_bracketed`, `extract_quotelike`, `extract_codeblock`, `extract_variable`, `extract_tagged`, `extract_multiple`, `gen_delimited_pat`, `gen_extract_tagged`, `delimited_pat`.

- Export Tags

  - `:ALL`

    `extract_delimited`, `extract_bracketed`, `extract_quotelike`, `extract_codeblock`, `extract_variable`, `extract_tagged`, `extract_multiple`, `gen_delimited_pat`, `gen_extract_tagged`, `delimited_pat`.

KNOWN BUGS

See https://rt.cpan.org/Dist/Display.html?Status=Active&Queue=Text-Balanced.

FEEDBACK

Patches, bug reports, suggestions or any other feedback is welcome.

Patches can be sent as GitHub pull requests at https://github.com/steve-m-hay/Text-Balanced/pulls.

Bug reports and suggestions can be made on the CPAN Request Tracker at https://rt.cpan.org/Public/Bug/Report.html?Queue=Text-Balanced.

Currently active requests on the CPAN Request Tracker can be viewed at https://rt.cpan.org/Public/Dist/Display.html?Queue=Text-Balanced.

Please test this distribution. See CPAN Testers at https://www.cpantesters.org/ for details of how to get involved.

Previous test results on CPAN Testers can be viewed at https://www.cpantesters.org/distro/T/Text-Balanced.html and https://fast2-matrix.cpantesters.org/?dist=Text-Balanced.

AVAILABILITY

The latest version of this module is available from CPAN (see "CPAN" in perlmodlib for details) at

https://metacpan.org/release/Text-Balanced or

https://www.cpan.org/authors/id/S/SH/SHAY/ or

https://www.cpan.org/modules/by-module/Text/.

The latest source code is available from GitHub at https://github.com/steve-m-hay/Text-Balanced.

INSTALLATION

See the `INSTALL` file.

AUTHOR

Damian Conway <damian@conway.org>.

Steve Hay <shay@cpan.org> is now maintaining Text::Balanced as of version 2.03.

COPYRIGHT

Copyright (C) 1997-2001 Damian Conway. All rights reserved.

Copyright (C) 2009 Adam Kennedy.

Copyright (C) 2015, 2020, 2022 Steve Hay and other contributors. All rights reserved.

LICENCE

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself, i.e. under the terms of either the GNU General Public License or the Artistic License, as specified in the `LICENCE` file.

VERSION

Version 2.07

DATE

08 Aug 2025

HISTORY

See the Changes file.

Text::CSV

SYNOPSIS

This section is taken from Text::CSV_XS.

```
# Functional interface
use Text::CSV qw( csv );

# Read whole file in memory
my $aoa = csv (in => "data.csv");    # as array of array
my $aoh = csv (in => "data.csv",
               headers => "auto");   # as array of hash

# Write array of arrays as csv file
csv (in => $aoa, out => "file.csv", sep_char => ";");

# Only show lines where "code" is odd
csv (in => "data.csv", filter => { code => sub { $_ % 2 }});

# Object interface
use Text::CSV;

my @rows;
# Read/parse CSV
my $csv = Text::CSV->new ({ binary => 1, auto_diag => 1 });
open my $fh, "<:encoding(utf8)", "test.csv" or die "test.csv: $!";
while (my $row = $csv->getline ($fh)) {
    $row->[2] =~ m/pattern/ or next; # 3rd field should match
    push @rows, $row;
    }
close $fh;

# and write as CSV
open $fh, ">:encoding(utf8)", "new.csv" or die "new.csv: $!";
$csv->say ($fh, $_) for @rows;
close $fh or die "new.csv: $!";
```

DESCRIPTION

Text::CSV is a thin wrapper for Text::CSV_XS-compatible modules now. All the backend modules provide facilities for the composition and decomposition of comma-separated values. Text::CSV uses Text::CSV_XS by default, and when Text::CSV_XS is not available, falls back on *Text::CSV_PP*, which is bundled in the same distribution as this module.

CHOOSING BACKEND

This module respects an environmental variable called `PERL_TEXT_CSV` when it decides a backend module to use. If this environmental variable is not set, it tries to load Text::CSV_XS, and if Text::CSV_XS is not available, falls back on Text::CSV_PP;

If you always don't want it to fall back on Text::CSV_PP, set the variable like this (`export` may be `setenv`, `set` and the likes, depending on your environment):

```
> export PERL_TEXT_CSV=Text::CSV_XS
```

If you prefer Text::CSV_XS to Text::CSV_PP (default), then:

```
> export PERL_TEXT_CSV=Text::CSV_XS,Text::CSV_PP
```

You may also want to set this variable at the top of your test files, in order not to be bothered with incompatibilities between backends (you need to wrap this in `BEGIN`, and set before actually `use`-ing Text::CSV module, as it decides its backend as soon as it's loaded):

```
BEGIN { $ENV{PERL_TEXT_CSV}='Text::CSV_PP'; }
use Text::CSV;
```

NOTES

This section is also taken from Text::CSV_XS.

Embedded newlines

Important Note: The default behavior is to accept only ASCII characters in the range from `0x20` (space) to `0x7E` (tilde). This means that the fields can not contain newlines. If your data contains newlines embedded in fields, or characters above `0x7E` (tilde), or binary data, you ***must*** set `binary => 1` in the call to "new". To cover the widest range of parsing options, you will always want to set binary.

But you still have the problem that you have to pass a correct line to the "parse" method, which is more complicated from the usual point of usage:

```
my $csv = Text::CSV->new ({ binary => 1, eol => $/ });
while (<>) {           #  WRONG!
    $csv->parse ($_);
    my @fields = $csv->fields ();
    }
```

this will break, as the `while` might read broken lines: it does not care about the quoting. If you need to support embedded newlines, the way to go is to **not** pass `eol` in the parser (it accepts `\n`, `\r`, **and** `\r\n` by default) and then

```
my $csv = Text::CSV->new ({ binary => 1 });
open my $fh, "<", $file or die "$file: $!";
while (my $row = $csv->getline ($fh)) {
    my @fields = @$row;
    }
```

The old(er) way of using global file handles is still supported

```
while (my $row = $csv->getline (*ARGV)) { ... }
```

Unicode

Unicode is only tested to work with perl-5.8.2 and up.

See also "BOM".

The simplest way to ensure the correct encoding is used for in- and output is by either setting layers on the filehandles, or setting the "encoding" argument for "csv".

```
 open my $fh, "<:encoding(UTF-8)", "in.csv"  or die "in.csv: $!";
or
 my $aoa = csv (in => "in.csv",     encoding => "UTF-8");

 open my $fh, ">:encoding(UTF-8)", "out.csv" or die "out.csv: $!";
or
 csv (in => $aoa, out => "out.csv", encoding => "UTF-8");
```

On parsing (both for "getline" and "parse"), if the source is marked being UTF8, then all fields that are marked binary will also be marked UTF8.

On combining ("print" and "combine"): if any of the combining fields was marked UTF8, the resulting string will be marked as UTF8. Note however that all fields *before* the first field marked UTF8 and contained 8-bit characters that were not upgraded to UTF8, these will be `bytes` in the resulting string too, possibly causing unexpected errors. If you pass data of different encoding, or you don't know if there is different encoding, force it to be upgraded before you pass them on:

```
$csv->print ($fh, [ map { utf8::upgrade (my $x = $_); $x } @data ]);
```

For complete control over encoding, please use Text::CSV::Encoded:

```
use Text::CSV::Encoded;
my $csv = Text::CSV::Encoded->new ({
    encoding_in  => "iso-8859-1", # the encoding comes into   Perl
    encoding_out => "cp1252",     # the encoding comes out of Perl
    });

$csv = Text::CSV::Encoded->new ({ encoding  => "utf8" });
```

(continues on next page)

(continued from previous page)

```
# combine () and print () accept *literally* utf8 encoded data
# parse () and getline () return *literally* utf8 encoded data

$csv = Text::CSV::Encoded->new ({ encoding  => undef }); # default
# combine () and print () accept UTF8 marked data
# parse () and getline () return UTF8 marked data
```

BOM

BOM (or Byte Order Mark) handling is available only inside the "header" method. This method supports the following encodings: `utf-8`, `utf-1`, `utf-32be`, `utf-32le`, `utf-16be`, `utf-16le`, `utf-ebcdic`, `scsu`, `bocu-1`, and `gb-18030`. See Wikipedia.

If a file has a BOM, the easiest way to deal with that is

```
my $aoh = csv (in => $file, detect_bom => 1);
```

All records will be encoded based on the detected BOM.

This implies a call to the "header" method, which defaults to also set the "column_names". So this is **not** the same as

```
my $aoh = csv (in => $file, headers => "auto");
```

which only reads the first record to set "column_names" but ignores any meaning of possible present BOM.

METHODS

This section is also taken from Text::CSV_XS.

version

(Class method) Returns the current module version.

new

(Class method) Returns a new instance of class Text::CSV. The attributes are described by the (optional) hash ref `\%attr`.

```
my $csv = Text::CSV->new ({ attributes ... });
```

The following attributes are available:

eol

```
my $csv = Text::CSV->new ({ eol => $/ });
          $csv->eol (undef);
my $eol = $csv->eol;
```

The end-of-line string to add to rows for "print" or the record separator for "getline".

When not passed in a **parser** instance, the default behavior is to accept `\n`, `\r`, and `\r\n`, so it is probably safer to not specify `eol` at all. Passing `undef` or the empty string behave the same.

When not passed in a **generating** instance, records are not terminated at all, so it is probably wise to pass something you expect. A safe choice for `eol` on output is either `$/` or `\r\n`.

Common values for `eol` are `"\012"` (`\n` or Line Feed), `"\015\012"` (`\r\n` or Carriage Return, Line Feed), and `"\015"` (`\r` or Carriage Return). The `eol` attribute cannot exceed 7 (ASCII) characters.

If both `$/` and `eol` equal `"\015"`, parsing lines that end on only a Carriage Return without Line Feed, will be "parse"d correct.

eol_type

```
my $eol = $csv->eol_type;
```

This read-only method returns the internal state of what is considered the valid EOL for parsing.

sep_char

```
my $csv = Text::CSV->new ({ sep_char => ";" });
          $csv->sep_char (";");
my $c = $csv->sep_char;
```

The char used to separate fields, by default a comma. (`,`). Limited to a single-byte character, usually in the range from `0x20` (space) to `0x7E` (tilde). When longer sequences are required, use `sep`.

The separation character can not be equal to the quote character or to the escape character.

sep

```
my $csv = Text::CSV->new ({ sep => "\N{FULLWIDTH COMMA}" });
          $csv->sep (";");
my $sep = $csv->sep;
```

The chars used to separate fields, by default undefined. Limited to 8 bytes.

When set, overrules `sep_char`. If its length is one byte it acts as an alias to `sep_char`.

quote_char

```
my $csv = Text::CSV->new ({ quote_char => "'" });
          $csv->quote_char (undef);
my $c = $csv->quote_char;
```

The character to quote fields containing blanks or binary data, by default the double quote character ("). A value of undef suppresses quote chars (for simple cases only). Limited to a single-byte character, usually in the range from 0x20 (space) to 0x7E (tilde). When longer sequences are required, use quote.

quote_char can not be equal to sep_char.

quote

```
my $csv = Text::CSV->new ({ quote => "\N{FULLWIDTH QUOTATION MARK}"␣
 ↪});
            $csv->quote ("'");
my $quote = $csv->quote;
```

The chars used to quote fields, by default undefined. Limited to 8 bytes.

When set, overrules quote_char. If its length is one byte it acts as an alias to quote_char.

This method does not support undef. Use quote_char to disable quotation.

escape_char

```
my $csv = Text::CSV->new ({ escape_char => "\\" });
        $csv->escape_char (":");
my $c = $csv->escape_char;
```

The character to escape certain characters inside quoted fields. This is limited to a single-byte character, usually in the range from 0x20 (space) to 0x7E (tilde).

The escape_char defaults to being the double-quote mark ("). In other words the same as the default quote_char. This means that doubling the quote mark in a field escapes it:

```
"foo","bar","Escape ""quote mark"" with two ""quote marks""","baz"
```

If you change the quote_char without changing the escape_char, the escape_char will still be the double-quote ("). If instead you want to escape the quote_char by doubling it you will need to also change the escape_char to be the same as what you have changed the quote_char to.

Setting escape_char to undef or "" will completely disable escapes and is greatly discouraged. This will also disable escape_null.

The escape character can not be equal to the separation character.

binary

```
my $csv = Text::CSV->new ({ binary => 1 });
        $csv->binary (0);
my $f = $csv->binary;
```

If this attribute is 1, you may use binary characters in quoted fields, including line feeds, carriage returns and NULL bytes. (The latter could be escaped as "0.) By default this feature is off.

If a string is marked UTF8, binary will be turned on automatically when binary characters other than CR and NL are encountered. Note that a simple string like "\x{00a0}" might still be binary, but not marked UTF8, so setting { binary => 1 } is still a wise option.

strict

```
my $csv = Text::CSV->new ({ strict => 1 });
        $csv->strict (0);
my $f = $csv->strict;
```

If this attribute is set to 1, any row that parses to a different number of fields than the previous row will cause the parser to throw error 2014.

Empty rows or rows that result in no fields (like comment lines) are exempt from these checks.

strict_eol

```
my $csv = Text::CSV->new ({ strict_eol => 1 });
        $csv->strict_eol (0);
my $f = $csv->strict_eol;
```

If this attribute is set to 0, no EOL consistency checks are done.

If this attribute is set to 1, any row that parses with a EOL other than the EOL from the first row will cause a warning. The error will be ignored and parsing continues. This warning is only thrown once. Note that in data with various different line endings, \r\r will still throw an error that cannot be ignored.

If this attribute is set to 2 or higher, any row that parses with a EOL other than the EOL from the first row will cause error 2016 to be thrown. The line being parsed to this error might not be stored in the result.

skip_empty_rows

```
my $csv = Text::CSV->new ({ skip_empty_rows => 1 });
        $csv->skip_empty_rows ("eof");
my $f = $csv->skip_empty_rows;
```

This attribute defines the behavior for empty rows: an “eol” immediately following the start of line. Default behavior is to return one single empty field.

This attribute is only used in parsing. This attribute is ineffective when using “parse” and “fields”.

Possible values for this attribute are

- 0 | undef

```
my $csv = Text::CSV->new ({ skip_empty_rows => 0 });
$csv->skip_empty_rows (undef);
```

No special action is taken. The result will be one single empty field.

- 1 | "skip"

```
my $csv = Text::CSV->new ({ skip_empty_rows => 1 });
$csv->skip_empty_rows ("skip");
```

The row will be skipped.

- 2 | "eof" | "stop"

```
my $csv = Text::CSV->new ({ skip_empty_rows => 2 });
$csv->skip_empty_rows ("eof");
```

The parsing will stop as if an "eof" was detected.

- 3 | "die"

```
my $csv = Text::CSV->new ({ skip_empty_rows => 3 });
$csv->skip_empty_rows ("die");
```

The parsing will stop. The internal error code will be set to 2015 and the parser will `die`.

- 4 | "croak"

```
my $csv = Text::CSV->new ({ skip_empty_rows => 4 });
$csv->skip_empty_rows ("croak");
```

The parsing will stop. The internal error code will be set to 2015 and the parser will `croak`.

- 5 | "error"

```
my $csv = Text::CSV->new ({ skip_empty_rows => 5 });
$csv->skip_empty_rows ("error");
```

The parsing will fail. The internal error code will be set to 2015.

- callback

```
my $csv = Text::CSV->new ({ skip_empty_rows => sub { [] } });
$csv->skip_empty_rows (sub { [ 42, $., undef, "empty" ] });
```

The callback is invoked and its result used instead. If you want the parse to stop after the callback, make sure to return a false value.

The returned value from the callback should be an array-ref. Any other type will cause the parse to stop, so these are equivalent in behavior:

```
csv (in => $fh, skip_empty_rows => "stop");
csv (in => $fh. skip_empty_rows => sub { 0; });
```

Without arguments, the current value is returned: `0`, `1`, `eof`, `die`, `croak` or the callback.

formula_handling

Alias for “formula”

formula

```
my $csv = Text::CSV->new ({ formula => "none" });
          $csv->formula ("none");
my $f = $csv->formula;
```

This defines the behavior of fields containing *formulas*. As formulas are considered dangerous in spreadsheets, this attribute can define an optional action to be taken if a field starts with an equal sign (=).

For purpose of code-readability, this can also be written as

```
my $csv = Text::CSV->new ({ formula_handling => "none" });
          $csv->formula_handling ("none");
my $f = $csv->formula_handling;
```

Possible values for this attribute are

- none

  Take no specific action. This is the default.

  ```
  $csv->formula ("none");
  ```

- die

  Cause the process to `die` whenever a leading = is encountered.

  ```
  $csv->formula ("die");
  ```

- croak

  Cause the process to `croak` whenever a leading = is encountered. (See *Carp*)

  ```
  $csv->formula ("croak");
  ```

- diag

  Report position and content of the field whenever a leading = is found. The value of the field is unchanged.

  ```
  $csv->formula ("diag");
  ```

- empty

  Replace the content of fields that start with a = with the empty string.

  ```
  $csv->formula ("empty");
  $csv->formula ("");
  ```

- undef

  Replace the content of fields that start with a = with undef.

```
$csv->formula ("undef");
$csv->formula (undef);
```

- a callback

  Modify the content of fields that start with a = with the return-value of the callback. The original content of the field is available inside the callback as $_;

```
# Replace all formula's with 42
$csv->formula (sub { 42; });

# same as $csv->formula ("empty") but slower
$csv->formula (sub { "" });

# Allow =4+12
$csv->formula (sub { s/^=(\d+\+\d+)$/$1/eer });

# Allow more complex calculations
$csv->formula (sub { eval { s{^=([-+*/0-9()]+)$}{$1}ee }; $_ });
```

All other values will give a warning and then fallback to diag.

decode_utf8

```
my $csv = Text::CSV->new ({ decode_utf8 => 1 });
        $csv->decode_utf8 (0);
my $f = $csv->decode_utf8;
```

This attributes defaults to TRUE.

While *parsing*, fields that are valid UTF-8, are automatically set to be UTF-8, so that

```
$csv->parse ("\xC4\xA8\n");
```

results in

```
PV("\304\250"\0) [UTF8 "\x{128}"]
```

Sometimes it might not be a desired action. To prevent those upgrades, set this attribute to false, and the result will be

```
PV("\304\250"\0)
```

auto_diag

```
my $csv = Text::CSV->new ({ auto_diag => 1 });
        $csv->auto_diag (2);
my $l = $csv->auto_diag;
```

Set this attribute to a number between 1 and 9 causes "error_diag" to be automatically called in void context upon errors.

In case of error `2012 - EOF`, this call will be void.

If `auto_diag` is set to a numeric value greater than 1, it will `die` on errors instead of `warn`. If set to anything unrecognized, it will be silently ignored.

Future extensions to this feature will include more reliable auto-detection of `autodie` being active in the scope of which the error occurred which will increment the value of `auto_diag` with 1 the moment the error is detected.

diag_verbose

```
my $csv = Text::CSV->new ({ diag_verbose => 1 });
        $csv->diag_verbose (2);
my $l = $csv->diag_verbose;
```

Set the verbosity of the output triggered by `auto_diag`. Currently only adds the current input-record-number (if known) to the diagnostic output with an indication of the position of the error.

blank_is_undef

```
my $csv = Text::CSV->new ({ blank_is_undef => 1 });
        $csv->blank_is_undef (0);
my $f = $csv->blank_is_undef;
```

Under normal circumstances, `CSV` data makes no distinction between quoted- and unquoted empty fields. These both end up in an empty string field once read, thus

```
1,"",," ",2
```

is read as

```
("1", "", "", " ", "2")
```

When *writing* `CSV` files with either `always_quote` or `quote_empty` set, the unquoted *empty* field is the result of an undefined value. To enable this distinction when *reading* `CSV` data, the `blank_is_undef` attribute will cause unquoted empty fields to be set to `undef`, causing the above to be parsed as

```
("1", "", undef, " ", "2")
```

Note that this is specifically important when loading `CSV` fields into a database that allows `NULL` values, as the perl equivalent for `NULL` is `undef` in *DBI* land.

empty_is_undef

```
my $csv = Text::CSV->new ({ empty_is_undef => 1 });
        $csv->empty_is_undef (0);
my $f = $csv->empty_is_undef;
```

Going one step further than `blank_is_undef`, this attribute converts all empty fields to `undef`, so

```
1,"",," ",2
```

is read as

```
(1, undef, undef, " ", 2)
```

Note that this affects only fields that are originally empty, not fields that are empty after stripping allowed whitespace. YMMV.

allow_whitespace

```
my $csv = Text::CSV->new ({ allow_whitespace => 1 });
        $csv->allow_whitespace (0);
my $f = $csv->allow_whitespace;
```

When this option is set to true, the whitespace (TAB's and SPACE's) surrounding the separation character is removed when parsing. If either TAB or SPACE is one of the three characters `sep_char`, `quote_char`, or `escape_char` it will not be considered whitespace.

Now lines like:

```
1 , "foo" , bar , 3 , zapp
```

are parsed as valid CSV, even though it violates the CSV specs.

Note that **all** whitespace is stripped from both start and end of each field. That would make it *more* than a *feature* to enable parsing bad CSV lines, as

```
1,   2.0,  3,   ape  , monkey
```

will now be parsed as

```
("1", "2.0", "3", "ape", "monkey")
```

even if the original line was perfectly acceptable CSV.

allow_loose_quotes

```
my $csv = Text::CSV->new ({ allow_loose_quotes => 1 });
        $csv->allow_loose_quotes (0);
my $f = $csv->allow_loose_quotes;
```

By default, parsing unquoted fields containing `quote_char` characters like

```
1,foo "bar" baz,42
```

would result in parse error 2034. Though it is still bad practice to allow this format, we cannot help the fact that some vendors make their applications spit out lines styled this way.

If there is **really** bad CSV data, like

```
1,"foo "bar" baz",42
```

or

```
1,""foo bar baz"",42
```

there is a way to get this data-line parsed and leave the quotes inside the quoted field as-is. This can be achieved by setting `allow_loose_quotes` **AND** making sure that the `escape_char` is *not* equal to `quote_char`.

allow_loose_escapes

```
my $csv = Text::CSV->new ({ allow_loose_escapes => 1 });
        $csv->allow_loose_escapes (0);
my $f = $csv->allow_loose_escapes;
```

Parsing fields that have `escape_char` characters that escape characters that do not need to be escaped, like:

```
my $csv = Text::CSV->new ({ escape_char => "\\" });
$csv->parse (qq{1,"my bar\'s",baz,42});
```

would result in parse error 2025. Though it is bad practice to allow this format, this attribute enables you to treat all escape character sequences equal.

allow_unquoted_escape

```
my $csv = Text::CSV->new ({ allow_unquoted_escape => 1 });
        $csv->allow_unquoted_escape (0);
my $f = $csv->allow_unquoted_escape;
```

A backward compatibility issue where `escape_char` differs from `quote_char` prevents `escape_char` to be in the first position of a field. If `quote_char` is equal to the default `"` and `escape_char` is set to `\`, this would be illegal:

```
1,\0,2
```

Setting this attribute to 1 might help to overcome issues with backward compatibility and allow this style.

always_quote

```
my $csv = Text::CSV->new ({ always_quote => 1 });
        $csv->always_quote (0);
my $f = $csv->always_quote;
```

By default the generated fields are quoted only if they *need* to be. For example, if they contain the separator character. If you set this attribute to 1 then *all* defined fields will

be quoted. (undef fields are not quoted, see "blank_is_undef"). This makes it quite often easier to handle exported data in external applications.

quote_space

```
my $csv = Text::CSV->new ({ quote_space => 1 });
          $csv->quote_space (0);
my $f = $csv->quote_space;
```

By default, a space in a field would trigger quotation. As no rule exists this to be forced in CSV, nor any for the opposite, the default is true for safety. You can exclude the space from this trigger by setting this attribute to 0.

quote_empty

```
my $csv = Text::CSV->new ({ quote_empty => 1 });
          $csv->quote_empty (0);
my $f = $csv->quote_empty;
```

By default the generated fields are quoted only if they *need* to be. An empty (defined) field does not need quotation. If you set this attribute to 1 then *empty* defined fields will be quoted. (undef fields are not quoted, see "blank_is_undef"). See also `always_quote`.

quote_binary

```
my $csv = Text::CSV->new ({ quote_binary => 1 });
          $csv->quote_binary (0);
my $f = $csv->quote_binary;
```

By default, all "unsafe" bytes inside a string cause the combined field to be quoted. By setting this attribute to `0`, you can disable that trigger for bytes `>= 0x7F`.

escape_null

```
my $csv = Text::CSV->new ({ escape_null => 1 });
          $csv->escape_null (0);
my $f = $csv->escape_null;
```

By default, a `NULL` byte in a field would be escaped. This option enables you to treat the `NULL` byte as a simple binary character in binary mode (the `{ binary => 1 }` is set). The default is true. You can prevent `NULL` escapes by setting this attribute to `0`.

When the `escape_char` attribute is set to undefined, this attribute will be set to false.

The default setting will encode "=\x00=" as

```
"="0="
```

With `escape_null` set, this will result in

```
"=\x00="
```

The default when using the `csv` function is `false`.

For backward compatibility reasons, the deprecated old name `quote_null` is still recognized.

keep_meta_info

```
my $csv = Text::CSV->new ({ keep_meta_info => 1 });
        $csv->keep_meta_info (0);
my $f = $csv->keep_meta_info;
```

By default, the parsing of input records is as simple and fast as possible. However, some parsing information - like quotation of the original field - is lost in that process. Setting this flag to true enables retrieving that information after parsing with the methods "meta_info", "is_quoted", and "is_binary" described below. Default is false for performance.

If you set this attribute to a value greater than 9, then you can control output quotation style like it was used in the input of the the last parsed record (unless quotation was added because of other reasons).

```
my $csv = Text::CSV->new ({
   binary         => 1,
   keep_meta_info => 1,
   quote_space    => 0,
   });

my $row = $csv->parse (q{1,,"", ," ",f,"g","h""h",help,"help"});

$csv->print (*STDOUT, \@row);
# 1,,, , ,f,g,"h""h",help,help
$csv->keep_meta_info (11);
$csv->print (*STDOUT, \@row);
# 1,,"", ," ",f,"g","h""h",help,"help"
```

undef_str

```
my $csv = Text::CSV->new ({ undef_str => "\\N" });
        $csv->undef_str (undef);
my $s = $csv->undef_str;
```

This attribute optionally defines the output of undefined fields. The value passed is not changed at all, so if it needs quotation, the quotation needs to be included in the value of the attribute. Use with caution, as passing a value like `" , " , , , , """` will for sure mess up your output. The default for this attribute is `undef`, meaning no special treatment.

This attribute is useful when exporting CSV data to be imported in custom loaders, like for MySQL, that recognize special sequences for `NULL` data.

This attribute has no meaning when parsing CSV data.

comment_str

```
my $csv = Text::CSV->new ({ comment_str => "#" });
        $csv->comment_str (undef);
my $s = $csv->comment_str;
```

This attribute optionally defines a string to be recognized as comment. If this attribute is defined, all lines starting with this sequence will not be parsed as CSV but skipped as comment.

This attribute has no meaning when generating CSV.

Comment strings that start with any of the special characters/sequences are not supported (so it cannot start with any of "sep_char", "quote_char", "escape_char", "sep", "quote", or "eol").

For convenience, `comment` is an alias for `comment_str`.

verbatim

```
my $csv = Text::CSV->new ({ verbatim => 1 });
        $csv->verbatim (0);
my $f = $csv->verbatim;
```

This is a quite controversial attribute to set, but makes some hard things possible.

The rationale behind this attribute is to tell the parser that the normally special characters newline (`NL`) and Carriage Return (`CR`) will not be special when this flag is set, and be dealt with as being ordinary binary characters. This will ease working with data with embedded newlines.

When `verbatim` is used with "getline", "getline" auto-`chomp`'s every line.

Imagine a file format like

```
M^^Hans^Janssen^Klas 2\n2A^Ja^11-06-2007#\r\n
```

where, the line ending is a very specific `"#\r\n"`, and the sep_char is a `^` (caret). None of the fields is quoted, but embedded binary data is likely to be present. With the specific line ending, this should not be too hard to detect.

By default, Text::CSV' parse function is instructed to only know about `"\n"` and `"\r"` to be legal line endings, and so has to deal with the embedded newline as a real `end-of-line`, so it can scan the next line if binary is true, and the newline is inside a quoted field. With this option, we tell "parse" to parse the line as if `"\n"` is just nothing more than a binary character.

For "parse" this means that the parser has no more idea about line ending and "getline" `chomp`s line endings on reading.

types

A set of column types; the attribute is immediately passed to the "types" method.

callbacks

See the "Callbacks" section below.

accessors

To sum it up,

```
$csv = Text::CSV->new ();
```

is equivalent to

```
$csv = Text::CSV->new ({
    eol                    => undef, # \r, \n, or \r\n
    sep_char               => ',',
    sep                    => undef,
    quote_char             => '"',
    quote                  => undef,
    escape_char            => '"',
    binary                 => 0,
    decode_utf8            => 1,
    auto_diag              => 0,
    diag_verbose           => 0,
    blank_is_undef         => 0,
    empty_is_undef         => 0,
    allow_whitespace       => 0,
    allow_loose_quotes     => 0,
    allow_loose_escapes    => 0,
    allow_unquoted_escape  => 0,
    always_quote           => 0,
    quote_empty            => 0,
    quote_space            => 1,
    escape_null            => 1,
    quote_binary           => 1,
    keep_meta_info         => 0,
    strict                 => 0,
    skip_empty_rows        => 0,
    formula                => 0,
    verbatim               => 0,
    undef_str              => undef,
    comment_str            => undef,
    types                  => undef,
    callbacks              => undef,
    });
```

For all of the above mentioned flags, an accessor method is available where you can inquire the current value, or change the value

```
my $quote = $csv->quote_char;
$csv->binary (1);
```

It is not wise to change these settings halfway through writing CSV data to a stream. If however you want to create a new stream using the available CSV object, there is no harm in changing them.

If the "new" constructor call fails, it returns undef, and makes the fail reason available through the "error_diag" method.

```
$csv = Text::CSV->new ({ ecs_char => 1 }) or
    die "".Text::CSV->error_diag ();
```

"error_diag" will return a string like

```
"INI - Unknown attribute 'ecs_char'"
```

known_attributes

```
@attr = Text::CSV->known_attributes;
@attr = Text::CSV::known_attributes;
@attr = $csv->known_attributes;
```

This method will return an ordered list of all the supported attributes as described above. This can be useful for knowing what attributes are valid in classes that use or extend Text::CSV.

print

```
$status = $csv->print ($fh, $colref);
```

Similar to "combine" + "string" + "print", but much more efficient. It expects an array ref as input (not an array!) and the resulting string is not really created, but immediately written to the $fh object, typically an IO handle or any other object that offers a "print" method.

For performance reasons print does not create a result string, so all "string", "status", "fields", and "error_input" methods will return undefined information after executing this method.

If $colref is undef (explicit, not through a variable argument) and "bind_columns" was used to specify fields to be printed, it is possible to make performance improvements, as otherwise data would have to be copied as arguments to the method call:

```
$csv->bind_columns (\($foo, $bar));
$status = $csv->print ($fh, undef);
```

A short benchmark

```
my @data = ("aa" .. "zz");
$csv->bind_columns (\(@data));
```

(continues on next page)

(continued from previous page)

```
$csv->print ($fh, [ @data ]);   # 11800 recs/sec
$csv->print ($fh,  \@data  );   # 57600 recs/sec
$csv->print ($fh,   undef  );   # 48500 recs/sec
```

say

```
$status = $csv->say ($fh, $colref);
```

Like `print`, but `eol` defaults to `$\`.

print_hr

```
$csv->print_hr ($fh, $ref);
```

Provides an easy way to print a `$ref` (as fetched with "getline_hr") provided the column names are set with "column_names".

It is just a wrapper method with basic parameter checks over

```
$csv->print ($fh, [ map { $ref->{$_} } $csv->column_names ]);
```

combine

```
$status = $csv->combine (@fields);
```

This method constructs a CSV record from `@fields`, returning success or failure. Failure can result from lack of arguments or an argument that contains an invalid character. Upon success, "string" can be called to retrieve the resultant CSV string. Upon failure, the value returned by "string" is undefined and "error_input" could be called to retrieve the invalid argument.

string

```
$line = $csv->string ();
```

This method returns the input to "parse" or the resultant CSV string of "combine", whichever was called more recently.

getline

```
$colref = $csv->getline ($fh);
```

This is the counterpart to "print", as "parse" is the counterpart to "combine": it parses a row from the `$fh` handle using the "getline" method associated with `$fh` and parses this row into an array ref. This array ref is returned by the function or `undef` for failure. When `$fh` does not support `getline`, you are likely to hit errors.

When fields are bound with “bind_columns” the return value is a reference to an empty list.

The “string”, “fields”, and “status” methods are meaningless again.

getline_all

```
$arrayref = $csv->getline_all ($fh);
$arrayref = $csv->getline_all ($fh, $offset);
$arrayref = $csv->getline_all ($fh, $offset, $length);
```

This will return a reference to a list of getline ($fh) results. In this call, `keep_meta_info` is disabled. If `$offset` is negative, as with `splice`, only the last `abs ($offset)` records of `$fh` are taken into consideration. Parameters `$offset` and `$length` are expected to be integers. Non-integer values are interpreted as integer without check.

Given a CSV file with 10 lines:

```
lines call
----- --------------------------------------------------------
0..9  $csv->getline_all ($fh)          # all
0..9  $csv->getline_all ($fh,  0)      # all
8..9  $csv->getline_all ($fh,  8)      # start at 8
-     $csv->getline_all ($fh,  0,  0) # start at 0 first 0 rows
0..4  $csv->getline_all ($fh,  0,  5) # start at 0 first 5 rows
4..5  $csv->getline_all ($fh,  4,  2) # start at 4 first 2 rows
8..9  $csv->getline_all ($fh, -2)      # last 2 rows
6..7  $csv->getline_all ($fh, -4,  2) # first 2 of last  4 rows
```

getline_hr

The “getline_hr” and “column_names” methods work together to allow you to have rows returned as hashrefs. You must call “column_names” first to declare your column names.

```
$csv->column_names (qw( code name price description ));
$hr = $csv->getline_hr ($fh);
print "Price for $hr->{name} is $hr->{price} EUR\n";
```

“getline_hr” will croak if called before “column_names”.

Note that “getline_hr” creates a hashref for every row and will be much slower than the combined use of “bind_columns” and “getline” but still offering the same easy to use hashref inside the loop:

```
my @cols = @{$csv->getline ($fh)};
$csv->column_names (@cols);
while (my $row = $csv->getline_hr ($fh)) {
    print $row->{price};
    }
```

Could easily be rewritten to the much faster:

```
my @cols = @{$csv->getline ($fh)};
my $row = {};
$csv->bind_columns (\@{$row}{@cols});
while ($csv->getline ($fh)) {
    print $row->{price};
    }
```

Your mileage may vary for the size of the data and the number of rows. With perl-5.14.2 the comparison for a 100_000 line file with 14 columns:

```
           Rate hashrefs getlines
hashrefs 1.00/s       --     -76%
getlines 4.15/s     313%       --
```

getline_hr_all

```
$arrayref = $csv->getline_hr_all ($fh);
$arrayref = $csv->getline_hr_all ($fh, $offset);
$arrayref = $csv->getline_hr_all ($fh, $offset, $length);
```

This will return a reference to a list of getline_hr ($fh) results. In this call, `keep_meta_info` is disabled.

parse

```
$status = $csv->parse ($line);
```

This method decomposes a CSV string into fields, returning success or failure. Failure can result from a lack of argument or the given CSV string is improperly formatted. Upon success, “fields” can be called to retrieve the decomposed fields. Upon failure calling “fields” will return undefined data and “error_input” can be called to retrieve the invalid argument.

You may use the “types” method for setting column types. See “types”’ description below.

The `$line` argument is supposed to be a simple scalar. Everything else is supposed to croak and set error 1500.

fragment

This function tries to implement RFC7111 (URI Fragment Identifiers for the text/csv Media Type) - https://datatracker.ietf.org/doc/html/rfc7111

```
my $AoA = $csv->fragment ($fh, $spec);
```

In specifications, `*` is used to specify the *last* item, a dash (`-`) to indicate a range. All indices are 1-based: the first row or column has index 1. Selections can be combined with the semi-colon (`;`).

When using this method in combination with “column_names”, the returned reference will point to a list of hashes instead of a list of lists. A disjointed cell-based combined

selection might return rows with different number of columns making the use of hashes unpredictable.

```
$csv->column_names ("Name", "Age");
my $AoH = $csv->fragment ($fh, "col=3;8");
```

If the “after_parse” callback is active, it is also called on every line parsed and skipped before the fragment.

- row

  ```
  row=4
  row=5-7
  row=6-*
  row=1-2;4;6-*
  ```

- col

  ```
  col=2
  col=1-3
  col=4-*
  col=1-2;4;7-*
  ```

- cell

  In cell-based selection, the comma (`,`) is used to pair row and column

  ```
  cell=4,1
  ```

  The range operator (`-`) using `cell`s can be used to define top-left and bottom-right `cell` location

  ```
  cell=3,1-4,6
  ```

  The `*` is only allowed in the second part of a pair

  ```
  cell=3,2-*,2    # row 3 till end, only column 2
  cell=3,2-3,*    # column 2 till end, only row 3
  cell=3,2-*,*    # strip row 1 and 2, and column 1
  ```

  Cells and cell ranges may be combined with `;`, possibly resulting in rows with different numbers of columns

  ```
  cell=1,1-2,2;3,3-4,4;1,4;4,1
  ```

  Disjointed selections will only return selected cells. The cells that are not specified will not be included in the returned set, not even as `undef`. As an example given a `CSV` like

  ```
  11,12,13,...19
  21,22,...28,29
  :             :
  91,...97,98,99
  ```

with `cell=1,1-2,2;3,3-4,4;1,4;4,1` will return:

```
11,12,14
21,22
33,34
41,43,44
```

Overlapping cell-specs will return those cells only once, So `cell=1,1-3,3;2,2-4,4;2,3;4,2` will return:

```
11,12,13
21,22,23,24
31,32,33,34
42,43,44
```

RFC7111 does **not** allow different types of specs to be combined (either `row` *or* `col` *or* `cell`). Passing an invalid fragment specification will croak and set error 2013.

column_names

Set the "keys" that will be used in the "getline_hr" calls. If no keys (column names) are passed, it will return the current setting as a list.

"column_names" accepts a list of scalars (the column names) or a single array_ref, so you can pass the return value from "getline" too:

```
$csv->column_names ($csv->getline ($fh));
```

"column_names" does **no** checking on duplicates at all, which might lead to unexpected results. Undefined entries will be replaced with the string `"\cAUNDEF\cA"`, so

```
$csv->column_names (undef, "", "name", "name");
$hr = $csv->getline_hr ($fh);
```

will set `$hr->{"\cAUNDEF\cA"}` to the 1st field, `$hr->{""}` to the 2nd field, and `$hr->{name}` to the 4th field, discarding the 3rd field.

"column_names" croaks on invalid arguments.

header

This method does NOT work in perl-5.6.x

Parse the CSV header and set `sep`, column_names and encoding.

```
my @hdr = $csv->header ($fh);
$csv->header ($fh, { sep_set => [ ";", ",", "|", "\t" ] });
$csv->header ($fh, { detect_bom => 1, munge_column_names => "lc" });
```

The first argument should be a file handle.

This method resets some object properties, as it is supposed to be invoked only once per file or stream. It will leave attributes `column_names` and `bound_columns` alone if

setting column names is disabled. Reading headers on previously process objects might fail on perl-5.8.0 and older.

Assuming that the file opened for parsing has a header, and the header does not contain problematic characters like embedded newlines, read the first line from the open handle then auto-detect whether the header separates the column names with a character from the allowed separator list.

If any of the allowed separators matches, and none of the *other* allowed separators match, set `sep` to that separator for the current CSV instance and use it to parse the first line, map those to lowercase, and use that to set the instance "column_names":

```
my $csv = Text::CSV->new ({ binary => 1, auto_diag => 1 });
open my $fh, "<", "file.csv";
binmode $fh; # for Windows
$csv->header ($fh);
while (my $row = $csv->getline_hr ($fh)) {
    ...
    }
```

If the header is empty, contains more than one unique separator out of the allowed set, contains empty fields, or contains identical fields (after folding), it will croak with error 1010, 1011, 1012, or 1013 respectively.

If the header contains embedded newlines or is not valid CSV in any other way, this method will croak and leave the parse error untouched.

A successful call to `header` will always set the `sep` of the `$csv` object. This behavior can not be disabled.

return value

On error this method will croak.

In list context, the headers will be returned whether they are used to set "column_names" or not.

In scalar context, the instance itself is returned. **Note**: the values as found in the header will effectively be **lost** if `set_column_names` is false.

Options

- sep_set

  ```
  $csv->header ($fh, { sep_set => [ ";", ",", "|", "\t" ] });
  ```

  The list of legal separators defaults to `[ ";", "," ]` and can be changed by this option. As this is probably the most often used option, it can be passed on its own as an unnamed argument:

  ```
  $csv->header ($fh, [ ";", ",", "|", "\t", "::", "\x{2063}" ]);
  ```

  Multi-byte sequences are allowed, both multi-character and Unicode. See `sep`.

- detect_bom

```
$csv->header ($fh, { detect_bom => 1 });
```

The default behavior is to detect if the header line starts with a BOM. If the header has a BOM, use that to set the encoding of `$fh`. This default behavior can be disabled by passing a false value to `detect_bom`.

Supported encodings from BOM are: UTF-8, UTF-16BE, UTF-16LE, UTF-32BE, and UTF-32LE. BOM also supports UTF-1, UTF-EBCDIC, SCSU, BOCU-1, and GB-18030 but *Encode* does not (yet). UTF-7 is not supported.

If a supported BOM was detected as start of the stream, it is stored in the object attribute `ENCODING`.

```
my $enc = $csv->{ENCODING};
```

The encoding is used with `binmode` on `$fh`.

If the handle was opened in a (correct) encoding, this method will **not** alter the encoding, as it checks the leading **bytes** of the first line. In case the stream starts with a decoded BOM (`U+FEFF`), `{ENCODING}` will be `""` (empty) instead of the default `undef`.

- munge_column_names

  This option offers the means to modify the column names into something that is most useful to the application. The default is to map all column names to lower case.

  ```
  $csv->header ($fh, { munge_column_names => "lc" });
  ```

  The following values are available:

  ```
  lc     - lower case
  uc     - upper case
  db     - valid DB field names
  none   - do not change
  \%hash - supply a mapping
  \&cb   - supply a callback
  ```

  – Lower case

    ```
    $csv->header ($fh, { munge_column_names => "lc" });
    ```

    The header is changed to all lower-case

    ```
    $_ = lc;
    ```

  – Upper case

    ```
    $csv->header ($fh, { munge_column_names => "uc" });
    ```

    The header is changed to all upper-case

    ```
    $_ = uc;
    ```

- Literal

```
$csv->header ($fh, { munge_column_names => "none" });
```

- Hash

```
$csv->header ($fh, { munge_column_names => { foo => "sombrero
↪" });
```

if a value does not exist, the original value is used unchanged

- Database

```
$csv->header ($fh, { munge_column_names => "db" });
```

* .

lower-case

* .

all sequences of non-word characters are replaced with an underscore

* .

all leading underscores are removed

```
$_ = lc (s/\W+/_/gr =~ s/^_+//r);
```

- Callback

```
$csv->header ($fh, { munge_column_names => sub { fc } });
$csv->header ($fh, { munge_column_names => sub { "column_".
↪$col++ } });
$csv->header ($fh, { munge_column_names => sub { lc (s/\W+/_/
↪gr) } });
```

As this callback is called in a `map`, you can use `$_` directly.

- set_column_names

```
$csv->header ($fh, { set_column_names => 1 });
```

The default is to set the instances column names using "column_names" if the method is successful, so subsequent calls to "getline_hr" can return a hash. Disable setting the header can be forced by using a false value for this option.

As described in "return value" above, content is lost in scalar context.

Validation

When receiving CSV files from external sources, this method can be used to protect against changes in the layout by restricting to known headers (and typos in the header fields).

```
my %known = (
    "record key" => "c_rec",
    "rec id"     => "c_rec",
    "id_rec"     => "c_rec",
    "kode"       => "code",
    "code"       => "code",
    "vaule"      => "value",
    "value"      => "value",
    );
my $csv = Text::CSV->new ({ binary => 1, auto_diag => 1 });
open my $fh, "<", $source or die "$source: $!";
$csv->header ($fh, { munge_column_names => sub {
    s/\s+$//;
    s/^\s+//;
    $known{lc $_} or die "Unknown column '$_' in $source";
    }});
while (my $row = $csv->getline_hr ($fh)) {
    say join "\t", $row->{c_rec}, $row->{code}, $row->{value};
    }
```

bind_columns

Takes a list of scalar references to be used for output with "print" or to store in the fields fetched by "getline". When you do not pass enough references to store the fetched fields in, "getline" will fail with error `3006`. If you pass more than there are fields to return, the content of the remaining references is left untouched. Under `strict` the two should match, otherwise "getline" will fail with error `2014`.

```
$csv->bind_columns (\$code, \$name, \$price, \$description);
while ($csv->getline ($fh)) {
    print "The price of a $name is \x{20ac} $price\n";
    }
```

To reset or clear all column binding, call "bind_columns" with the single argument `undef`. This will also clear column names.

```
$csv->bind_columns (undef);
```

If no arguments are passed at all, "bind_columns" will return the list of current bindings or `undef` if no binds are active.

Note that in parsing with `bind_columns`, the fields are set on the fly. That implies that if the third field of a row causes an error (or this row has just two fields where the previous row had more), the first two fields already have been assigned the values of the current row, while the rest of the fields will still hold the values of the previous row. If you want the parser to fail in these cases, use the `strict` attribute.

eof

```
$eof = $csv->eof ();
```

If "parse" or "getline" was used with an IO stream, this method will return true (1) if the last call hit end of file, otherwise it will return false ('). This is useful to see the difference between a failure and end of file.

Note that if the parsing of the last line caused an error, `eof` is still true. That means that if you are *not* using "auto_diag", an idiom like

```
while (my $row = $csv->getline ($fh)) {
    # ...
    }
$csv->eof or $csv->error_diag;
```

will *not* report the error. You would have to change that to

```
while (my $row = $csv->getline ($fh)) {
    # ...
    }
+$csv->error_diag and $csv->error_diag;
```

types

```
$csv->types (\@tref);
```

This method is used to force that (all) columns are of a given type. For example, if you have an integer column, two columns with doubles and a string column, then you might do a

```
$csv->types ([Text::CSV::IV (),
              Text::CSV::NV (),
              Text::CSV::NV (),
              Text::CSV::PV ()]);
```

Column types are used only for *decoding* columns while parsing, in other words by the "parse" and "getline" methods.

You can unset column types by doing a

```
$csv->types (undef);
```

or fetch the current type settings with

```
$types = $csv->types ();
```

- IV
- CSV_TYPE_IV

  Set field type to integer.
- NV

- CSV_TYPE_NV

  Set field type to numeric/float.

- PV
- CSV_TYPE_PV

  Set field type to string.

fields

```
@columns = $csv->fields ();
```

This method returns the input to "combine" or the resultant decomposed fields of a successful "parse", whichever was called more recently.

Note that the return value is undefined after using "getline", which does not fill the data structures returned by "parse".

meta_info

```
@flags = $csv->meta_info ();
```

This method returns the "flags" of the input to "combine" or the flags of the resultant decomposed fields of "parse", whichever was called more recently.

For each field, a meta_info field will hold flags that inform something about the field returned by the "fields" method or passed to the "combine" method. The flags are bit-wise-or'd like:

- 0x0001
- CSV_FLAGS_IS_QUOTED

  The field was quoted.

- 0x0002
- CSV_FLAGS_IS_BINARY

  The field was binary.

- 0x0004
- CSV_FLAGS_ERROR_IN_FIELD

  The field was invalid.

  Currently only used when allow_loose_quotes is active.

- 0x0010
- CSV_FLAGS_IS_MISSING

  The field was missing.

See the is_*** methods below.

is_quoted

```
my $quoted = $csv->is_quoted ($column_idx);
```

where `$column_idx` is the (zero-based) index of the column in the last result of "parse".

This returns a true value if the data in the indicated column was enclosed in `quote_char` quotes. This might be important for fields where content `,20070108,` is to be treated as a numeric value, and where `,"20070108",` is explicitly marked as character string data.

This method is only valid when "keep_meta_info" is set to a true value.

is_binary

```
my $binary = $csv->is_binary ($column_idx);
```

where `$column_idx` is the (zero-based) index of the column in the last result of "parse".

This returns a true value if the data in the indicated column contained any byte in the range `[\x00-\x08,\x10-\x1F,\x7F-\xFF]`.

This method is only valid when "keep_meta_info" is set to a true value.

is_missing

```
my $missing = $csv->is_missing ($column_idx);
```

where `$column_idx` is the (zero-based) index of the column in the last result of "getline_hr".

```
$csv->keep_meta_info (1);
while (my $hr = $csv->getline_hr ($fh)) {
    $csv->is_missing (0) and next; # This was an empty line
    }
```

When using "getline_hr", it is impossible to tell if the parsed fields are `undef` because they where not filled in the `CSV` stream or because they were not read at all, as **all** the fields defined by "column_names" are set in the hash-ref. If you still need to know if all fields in each row are provided, you should enable `keep_meta_info` so you can check the flags.

If `keep_meta_info` is `false`, `is_missing` will always return `undef`, regardless of `$column_idx` being valid or not. If this attribute is `true` it will return either `0` (the field is present) or 1 (the field is missing).

A special case is the empty line. If the line is completely empty - after dealing with the flags - this is still a valid CSV line: it is a record of just one single empty field. However, if `keep_meta_info` is set, invoking `is_missing` with index `0` will now return true.

status

```
$status = $csv->status ();
```

This method returns the status of the last invoked "combine" or "parse" call. Status is success (true: 1) or failure (false: undef or 0).

Note that as this only keeps track of the status of above mentioned methods, you are probably looking for error_diag instead.

error_input

```
$bad_argument = $csv->error_input ();
```

This method returns the erroneous argument (if it exists) of "combine" or "parse", whichever was called more recently. If the last invocation was successful, error_input will return undef.

Depending on the type of error, it *might* also hold the data for the last error-input of "getline".

error_diag

```
Text::CSV->error_diag ();
$csv->error_diag ();
$error_code  = 0  + $csv->error_diag ();
$error_str   = "" . $csv->error_diag ();
($cde, $str, $pos, $rec, $fld, $xs) = $csv->error_diag ();
```

If (and only if) an error occurred, this function returns the diagnostics of that error.

If called in void context, this will print the internal error code and the associated error message to STDERR.

If called in list context, this will return the error code and the error message in that order. If the last error was from parsing, the rest of the values returned are a best guess at the location within the line that was being parsed. Their values are 1-based. The position currently is index of the byte at which the parsing failed in the current record. It might change to be the index of the current character in a later release. The records is the index of the record parsed by the csv instance. The field number is the index of the field the parser thinks it is currently trying to parse. See examples/csv-check for how this can be used.

If called in scalar context, it will return the diagnostics in a single scalar, a-la $!. It will contain the error code in numeric context, and the diagnostics message in string context.

When called as a class method or a direct function call, the diagnostics are that of the last "new" call.

_cache_diag

Note: This is an internal function only, and output cannot be relied upon. Use at own risk.

If debugging beyond what "error_diag" is able to show, the internal cache can be shown with this function.

```
# Something failed ..
$csv->error_diag;
$csv->_cache_diag ();
```

record_number

```
$recno = $csv->record_number ();
```

Returns the records parsed by this csv instance. This value should be more accurate than `$.` when embedded newlines come in play. Records written by this instance are not counted.

SetDiag

```
$csv->SetDiag (0);
```

Use to reset the diagnostics if you are dealing with errors.

ADDITIONAL METHODS

- backend

  Returns the backend module name called by Text::CSV. `module` is an alias.

- is_xs

- is_pp

  Returns true value if Text::CSV uses a pure-Perl backend.

FUNCTIONS

This section is also taken from Text::CSV_XS.

csv

This function is not exported by default and should be explicitly requested:

```
use Text::CSV qw( csv );
```

This is a high-level function that aims at simple (user) interfaces. This can be used to read/parse a `CSV` file or stream (the default behavior) or to produce a file or write to a stream (define the `out` attribute). It returns an array- or hash-reference on parsing (or `undef` on fail) or the numeric value of "error_diag" on writing. When this function fails you can get to the error using the class call to "error_diag"

```
my $aoa = csv (in => "test.csv") or
    die Text::CSV->error_diag;
```

Note that failure here is the inability to start the parser, like when the input does not exist or the arguments are unknown or conflicting. Run-time parsing errors will return a valid reference, which can be empty, but still contains all results up till the error. See "on_error".

This function takes the arguments as key-value pairs. This can be passed as a list or as an anonymous hash:

```
my $aoa = csv (  in => "test.csv", sep_char => ";");
my $aoh = csv ({ in => $fh, headers => "auto" });
```

The arguments passed consist of two parts: the arguments to "csv" itself and the optional attributes to the CSV object used inside the function as enumerated and explained in "new".

If not overridden, the default option used for CSV is

```
auto_diag   => 1
escape_null => 0
strict_eol  => 1
```

The option that is always set and cannot be altered is

```
binary      => 1
```

As this function will likely be used in one-liners, it allows `quote` to be abbreviated as quo, and `escape_char` to be abbreviated as `esc` or `escape`.

Alternative invocations:

```
my $aoa = Text::CSV::csv (in => "file.csv");

my $csv = Text::CSV->new ();
my $aoa = $csv->csv (in => "file.csv");
```

In the latter case, the object attributes are used from the existing object and the attribute arguments in the function call are ignored:

```
my $csv = Text::CSV->new ({ sep_char => ";" });
my $aoh = $csv->csv (in => "file.csv", bom => 1);
```

will parse using `;` as `sep_char`, not `,`.

in

Used to specify the source. `in` can be a file name (e.g. `"file.csv"`), which will be opened for reading and closed when finished, a file handle (e.g. `$fh` or `FH`), a reference to a glob (e.g. `\*ARGV`), the glob itself (e.g. `*STDIN`), or a reference to a scalar (e.g. `\q{1,2,"csv"}`).

When used with "out", in should be a reference to a CSV structure (AoA or AoH) or a CODE-ref that returns an array-reference or a hash-reference. The code-ref will be invoked with no arguments.

```
my $aoa = csv (in => "file.csv");

open my $fh, "<", "file.csv";
my $aoa = csv (in => $fh);

my $csv = [ [qw( Foo Bar )], [ 1, 2 ], [ 2, 3 ]];
my $err = csv (in => $csv, out => "file.csv");
```

If called in void context without the "out" attribute, the resulting ref will be used as input to a subsequent call to csv:

```
csv (in => "file.csv", filter => { 2 => sub { length > 2 }})
```

will be a shortcut to

```
csv (in => csv (in => "file.csv", filter => { 2 => sub { length > 2␣
 ↪}}))
```

where, in the absence of the out attribute, this is a shortcut to

```
csv (in  => csv (in => "file.csv", filter => { 2 => sub { length > 2␣
 ↪}}),
     out => *STDOUT)
```

out

```
csv (in => $aoa,  out => "file.csv");
csv (in => $aoa,  out => $fh);
csv (in => $aoa,  out =>   STDOUT);
csv (in => $aoa,  out =>  *STDOUT);
csv (in => $aoa,  out => \*STDOUT);
csv (in => $aoa,  out => \my $data);
csv (in => $aoa,  out =>  undef);
csv (in => $aoa,  out => \"skip");

csv (in => $fh,   out => \@aoa);
csv (in => $fh,   out => \@aoh, bom => 1);
csv (in => $fh,   out => \%hsh, key => "key");

csv (in => $file, out => $file);
csv (in => $file, out => $fh);
csv (in => $fh,   out => $file);
csv (in => $fh,   out => $fh);
```

In output mode, the default CSV options when producing CSV are

```
eol       => "\r\n"
```

The “fragment” attribute is ignored in output mode.

`out` can be a file name (e.g. `"file.csv"`), which will be opened for writing and closed when finished, a file handle (e.g. `$fh` or `FH`), a reference to a glob (e.g. `\*STDOUT`), the glob itself (e.g. `*STDOUT`), or a reference to a scalar (e.g. `\my $data`).

```
csv (in => sub { $sth->fetch },            out => "dump.csv");
csv (in => sub { $sth->fetchrow_hashref }, out => "dump.csv",
     headers => $sth->{NAME_lc});
```

When a code-ref is used for `in`, the output is generated per invocation, so no buffering is involved. This implies that there is no size restriction on the number of records. The `csv` function ends when the coderef returns a false value.

If `out` is set to a reference of the literal string `"skip"`, the output will be suppressed completely, which might be useful in combination with a filter for side effects only.

```
my %cache;
csv (in    => "dump.csv",
     out   => \"skip",
     on_in => sub { $cache{$_[1][1]}++ });
```

Currently, setting `out` to any false value (`undef`, `""`, 0) will be equivalent to `\"skip"`.

If the `in` argument point to something to parse, and the `out` is set to a reference to an `ARRAY` or a `HASH`, the output is appended to the data in the existing reference. The result of the parse should match what exists in the reference passed. This might come handy when you have to parse a set of files with similar content (like data stored per period) and you want to collect that into a single data structure:

```
my %hash;
csv (in => $_, out => \%hash, key => "id") for sort glob "foo-[0-
↪9]*.csv";

my @list; # List of arrays
csv (in => $_, out => \@list)              for sort glob "foo-[0-
↪9]*.csv";

my @list; # List of hashes
csv (in => $_, out => \@list, bom => 1)    for sort glob "foo-[0-
↪9]*.csv";
```

Streaming

If **both** `in` and `out` are files, file handles or globs, streaming is enforced by injecting an `after_parse` callback that immediately uses the `say ()` method of the same instance to output the result and then rejects the record.

If a `after_parse` was already passed as attribute, that will be included in the injected call. If `on_in` was passed and `after_parse` was not, it will be used instead. If both were passed, `on_in` is ignored.

The EOL of the first record of the `in` source is consistently used as EOL for all records in the `out` destination.

The `filter` attribute is not available.

All other attributes are shared for `in` and `out`, so you cannot define different encodings for `in` and `out`. You need to pass a `$fh`, where `binmode` was used to apply the encoding layers.

Note that this is work in progress and things might change.

encoding

If passed, it should be an encoding accepted by the `:encoding()` option to open. There is no default value. This attribute does not work in perl 5.6.x. `encoding` can be abbreviated to `enc` for ease of use in command line invocations.

If `encoding` is set to the literal value `"auto"`, the method "header" will be invoked on the opened stream to check if there is a BOM and set the encoding accordingly. This is equal to passing a true value in the option `detect_bom`.

Encodings can be stacked, as supported by `binmode`:

```
# Using PerlIO::via::gzip
csv (in       => \@csv,
     out      => "test.csv:via.gz",
     encoding => ":via(gzip):encoding(utf-8)",
     );
$aoa = csv (in => "test.csv:via.gz",  encoding => ":via(gzip)");

# Using PerlIO::gzip
csv (in       => \@csv,
     out      => "test.csv:via.gz",
     encoding => ":gzip:encoding(utf-8)",
     );
$aoa = csv (in => "test.csv:gzip.gz", encoding => ":gzip");
```

detect_bom

If `detect_bom` is given, the method "header" will be invoked on the opened stream to check if there is a BOM and set the encoding accordingly.

`detect_bom` can be abbreviated to `bom`.

This is the same as setting `encoding` to `"auto"`.

Note that as the method "header" is invoked, its default is to also set the headers.

headers

If this attribute is not given, the default behavior is to produce an array of arrays.

If `headers` is supplied, it should be an anonymous list of column names, an anonymous hashref, a coderef, or a literal flag: `auto`, `lc`, `uc`, or `skip`.

- skip

  When `skip` is used, the header will not be included in the output.

```
my $aoa = csv (in => $fh, headers => "skip");
```

  `skip` is invalid/ignored in combinations with `detect_bom`.

- auto

  If `auto` is used, the first line of the CSV source will be read as the list of field headers and used to produce an array of hashes.

```
my $aoh = csv (in => $fh, headers => "auto");
```

- lc

  If `lc` is used, the first line of the CSV source will be read as the list of field headers mapped to lower case and used to produce an array of hashes. This is a variation of `auto`.

```
my $aoh = csv (in => $fh, headers => "lc");
```

- uc

  If `uc` is used, the first line of the CSV source will be read as the list of field headers mapped to upper case and used to produce an array of hashes. This is a variation of `auto`.

```
my $aoh = csv (in => $fh, headers => "uc");
```

- CODE

  If a coderef is used, the first line of the CSV source will be read as the list of mangled field headers in which each field is passed as the only argument to the coderef. This list is used to produce an array of hashes.

```
my $aoh = csv (in      => $fh,
               headers => sub { lc ($_[0]) =~ s/kode/code/gr });
```

  this example is a variation of using `lc` where all occurrences of kode are replaced with code.

- ARRAY

  If `headers` is an anonymous list, the entries in the list will be used as field names. The first line is considered data instead of headers.

```
my $aoh = csv (in => $fh, headers => [qw( Foo Bar )]);
csv (in => $aoa, out => $fh, headers => [qw( code description␣
 ↪price )]);
```

- HASH

  If `headers` is a hash reference, this implies `auto`, but header fields that exist as key in the hashref will be replaced by the value for that key. Given a CSV file like

```
post-kode,city,name,id number,fubble
1234AA,Duckstad,Donald,13,"X313DF"
```

using

```
csv (headers => { "post-kode" => "pc", "id number" => "ID" }, ..
 ↪.
```

will return an entry like

```
{ pc     => "1234AA",
  city   => "Duckstad",
  name   => "Donald",
  ID     => "13",
  fubble => "X313DF",
  }
```

See also `munge_column_names` and `set_column_names`.

munge_column_names

If `munge_column_names` is set, the method "header" is invoked on the opened stream with all matching arguments to detect and set the headers.

`munge_column_names` can be abbreviated to `munge`.

key

If passed, will default `headers` to `"auto"` and return a hashref instead of an array of hashes. Allowed values are simple scalars or array-references where the first element is the joiner and the rest are the fields to join to combine the key.

```
my $ref = csv (in => "test.csv", key => "code");
my $ref = csv (in => "test.csv", key => [ ":" => "code", "color" ]);
```

with test.csv like

```
code,product,price,color
1,pc,850,gray
2,keyboard,12,white
3,mouse,5,black
```

the first example will return

```
{ 1   => {
      code    => 1,
      color   => 'gray',
      price   => 850,
      product => 'pc'
      },
  2   => {
```

(continues on next page)

(continued from previous page)

```
      code    => 2,
      color   => 'white',
      price   => 12,
      product => 'keyboard'
      },
  3   => {
      code    => 3,
      color   => 'black',
      price   => 5,
      product => 'mouse'
      }
  }
```

the second example will return

```
{ "1:gray"    => {
      code    => 1,
      color   => 'gray',
      price   => 850,
      product => 'pc'
      },
  "2:white"   => {
      code    => 2,
      color   => 'white',
      price   => 12,
      product => 'keyboard'
      },
  "3:black"   => {
      code    => 3,
      color   => 'black',
      price   => 5,
      product => 'mouse'
      }
  }
```

The key attribute can be combined with `headers` for `CSV` date that has no header line, like

```
my $ref = csv (
    in      => "foo.csv",
    headers => [qw( c_foo foo bar description stock )],
    key     =>     "c_foo",
    );
```

value

Used to create key-value hashes.

Only allowed when `key` is valid. A `value` can be either a single column label or an anonymous list of column labels. In the first case, the value will be a simple scalar value,

in the latter case, it will be a hashref.

```
my $ref = csv (in => "test.csv", key   => "code",
                                 value => "price");
my $ref = csv (in => "test.csv", key   => "code",
                                 value => [ "product", "price" ]);
my $ref = csv (in => "test.csv", key   => [ ":" => "code", "color"
 ↪],
                                 value => "price");
my $ref = csv (in => "test.csv", key   => [ ":" => "code", "color"
 ↪],
                                 value => [ "product", "price" ]);
```

with test.csv like

```
code,product,price,color
1,pc,850,gray
2,keyboard,12,white
3,mouse,5,black
```

the first example will return

```
{ 1 => 850,
  2 =>  12,
  3 =>   5,
  }
```

the second example will return

```
{ 1   => {
      price   => 850,
      product => 'pc'
      },
  2   => {
      price   => 12,
      product => 'keyboard'
      },
  3   => {
      price   => 5,
      product => 'mouse'
      }
  }
```

the third example will return

```
{ "1:gray"    => 850,
  "2:white"   =>  12,
  "3:black"   =>   5,
  }
```

the fourth example will return

```
{ "1:gray"    => {
      price   => 850,
      product => 'pc'
      },
  "2:white"   => {
      price   => 12,
      product => 'keyboard'
      },
  "3:black"   => {
      price   => 5,
      product => 'mouse'
      }
  }
```

keep_headers

When using hashes, keep the column names into the arrayref passed, so all headers are available after the call in the original order.

```
my $aoh = csv (in => "file.csv", keep_headers => \my @hdr);
```

This attribute can be abbreviated to `kh` or passed as `keep_column_names`.

This attribute implies a default of `auto` for the `headers` attribute.

The headers can also be kept internally to keep stable header order:

```
csv (in      => csv (in => "file.csv", kh => "internal"),
     out     => "new.csv",
     kh      => "internal");
```

where `internal` can also be 1, `yes`, or `true`. This is similar to

```
my @h;
csv (in      => csv (in => "file.csv", kh => \@h),
     out     => "new.csv",
     headers => \@h);
```

fragment

Only output the fragment as defined in the "fragment" method. This option is ignored when *generating* CSV. See "out".

Combining all of them could give something like

```
use Text::CSV qw( csv );
my $aoh = csv (
    in       => "test.txt",
    encoding => "utf-8",
    headers  => "auto",
    sep_char => "|",
```

(continues on next page)

(continued from previous page)

```
    fragment => "row=3;6-9;15-*",
    );
say $aoh->[15]{Foo};
```

sep_set

If `sep_set` is set, the method "header" is invoked on the opened stream to detect and set `sep_char` with the given set.

`sep_set` can be abbreviated to `seps`. If neither `sep_set` not `seps` is given, but `sep` is defined, `sep_set` defaults to `[ sep ]`. This is only supported for perl version 5.10 and up.

Note that as the "header" method is invoked, its default is to also set the headers.

set_column_names

If `set_column_names` is passed, the method "header" is invoked on the opened stream with all arguments meant for "header".

If `set_column_names` is passed as a false value, the content of the first row is only preserved if the output is AoA:

With an input-file like

```
bAr,foo
1,2
3,4,5
```

This call

```
my $aoa = csv (in => $file, set_column_names => 0);
```

will result in

```
[[ "bar", "foo"     ],
 [ "1",   "2"       ],
 [ "3",   "4",  "5" ]]
```

and

```
my $aoa = csv (in => $file, set_column_names => 0, munge => "none");
```

will result in

```
[[ "bAr", "foo"     ],
 [ "1",   "2"       ],
 [ "3",   "4",  "5" ]]
```

csv

The *function* "csv" can also be called as a method or with an existing Text::CSV object. This could help if the function is to be invoked a lot of times and the overhead of creating the object internally over and over again would be prevented by passing an existing instance.

```
my $csv = Text::CSV->new ({ binary => 1, auto_diag => 1 });

my $aoa = $csv->csv (in => $fh);
my $aoa = csv (in => $fh, csv => $csv);
```

both act the same. Running this 20000 times on a 20 lines CSV file, showed a 53% speedup.

Callbacks

Callbacks enable actions triggered from the *inside* of Text::CSV.

While most of what this enables can easily be done in an unrolled loop as described in the "SYNOPSIS" callbacks can be used to meet special demands or enhance the "csv" function.

- error

  ```
  $csv->callbacks (error => sub { $csv->SetDiag (0) });
  ```

  the `error` callback is invoked when an error occurs, but *only* when "auto_diag" is set to a true value. A callback is invoked with the values returned by "error_diag":

  ```
  my ($c, $s);

  sub ignore3006 {
      my ($err, $msg, $pos, $recno, $fldno) = @_;
      if ($err == 3006) {
          # ignore this error
          ($c, $s) = (undef, undef);
          Text::CSV->SetDiag (0);
          }
      # Any other error
      return;
      } # ignore3006

  $csv->callbacks (error => \&ignore3006);
  $csv->bind_columns (\$c, \$s);
  while ($csv->getline ($fh)) {
      # Error 3006 will not stop the loop
      }
  ```

- after_parse

```
$csv->callbacks (after_parse => sub { push @{$_[1]}, "NEW" });
while (my $row = $csv->getline ($fh)) {
    $row->[-1] eq "NEW";
    }
```

This callback is invoked after parsing with "getline" only if no error occurred. The callback is invoked with two arguments: the current CSV parser object and an array reference to the fields parsed.

The return code of the callback is ignored unless it is a reference to the string "skip", in which case the record will be skipped in "getline_all".

```
sub add_from_db {
    my ($csv, $row) = @_;
    $sth->execute ($row->[4]);
    push @$row, $sth->fetchrow_array;
    } # add_from_db

my $aoa = csv (in => "file.csv", callbacks => {
    after_parse => \&add_from_db });
```

This hook can be used for validation:

- FAIL

  Die if any of the records does not validate a rule:

  ```
  after_parse => sub {
      $_[1][4] =~ m/^[0-9]{4}\s?[A-Z]{2}$/ or
          die "5th field does not have a valid Dutch zipcode";
      }
  ```

- DEFAULT

  Replace invalid fields with a default value:

  ```
  after_parse => sub { $_[1][2] =~ m/^\d+$/ or $_[1][2] = 0 }
  ```

- SKIP

  Skip records that have invalid fields (only applies to "getline_all"):

  ```
  after_parse => sub { $_[1][0] =~ m/^\d+$/ or return \"skip";
  ↪}
  ```

- before_print

```
my $idx = 1;
$csv->callbacks (before_print => sub { $_[1][0] = $idx++ });
$csv->print (*STDOUT, [ 0, $_ ]) for @members;
```

This callback is invoked before printing with "print" only if no error occurred. The callback is invoked with two arguments: the current CSV parser object and an array reference to the fields passed.

The return code of the callback is ignored.

```
sub max_4_fields {
    my ($csv, $row) = @_;
    @$row > 4 and splice @$row, 4;
    } # max_4_fields

csv (in => csv (in => "file.csv"), out => *STDOUT,
    callbacks => { before_print => \&max_4_fields });
```

This callback is not active for “combine”.

Callbacks for csv ()

```
csv (in        => "file.csv",
     callbacks => {
         filter       => { 6 => sub { $_ > 15 } },    # first
         after_parse  => sub { say "AFTER PARSE";  }, # first
         after_in     => sub { say "AFTER IN";     }, # second
         on_in        => sub { say "ON IN";        }, # third
         },
     );

csv (in        => $aoh,
     out       => "file.csv",
     callbacks => {
         on_in        => sub { say "ON IN";        }, # first
         before_out   => sub { say "BEFORE OUT";   }, # second
         before_print => sub { say "BEFORE PRINT"; }, # third
         },
     );
```

- filter

  This callback can be used to filter records. It is called just after a new record has been scanned. The callback accepts a:

  - hashref

    The keys are the index to the row (the field name or field number, 1-based) and the values are subs to return a true or false value.

    ```
    csv (in => "file.csv", filter => {
              3 => sub { m/a/ },       # third field should␣
    ↪contain an "a"
              5 => sub { length > 4 }, # length of the 5th␣
    ↪field minimal 5
              });

    csv (in => "file.csv", filter => { foo => sub { $_ > 4 }});
    ```

    If the keys to the filter hash contain any character that is not a digit it will also implicitly set “headers” to `"auto"` unless “headers” was already passed

as argument. When headers are active, returning an array of hashes, the filter is not applicable to the header itself.

All sub results should match, as in AND.

The context of the callback sets $_ localized to the field indicated by the filter. The two arguments are as with all other callbacks, so the other fields in the current row can be seen:

```
filter => { 3 => sub { $_ > 100 ? $_[1][1] =~ m/A/ : $_
↪[1][6] =~ m/B/ }}
```

If the context is set to return a list of hashes ("headers" is defined), the current record will also be available in the localized %_:

```
filter => { 3 => sub { $_ > 100 && $_{foo} =~ m/A/ && $_{bar}
↪ < 1000  }}
```

If the filter is used to *alter* the content by changing $_, make sure that the sub returns true in order not to have that record skipped:

```
filter => { 2 => sub { $_ = uc }}
```

will upper-case the second field, and then skip it if the resulting content evaluates to false. To always accept, end with truth:

```
filter => { 2 => sub { $_ = uc; 1 }}
```

– coderef

```
csv (in => "file.csv", filter => sub { $n++; 0; });
```

If the argument to `filter` is a coderef, it is an alias or shortcut to a filter on column 0:

```
csv (filter => sub { $n++; 0 });
```

is equal to

```
csv (filter => { 0 => sub { $n++; 0 });
```

– filter-name

```
csv (in => "file.csv", filter => "not_blank");
csv (in => "file.csv", filter => "not_empty");
csv (in => "file.csv", filter => "filled");
```

These are predefined filters

Given a file like (line numbers prefixed for doc purpose only):

```
1:1,2,3
2:
3:,
```

(continues on next page)

(continued from previous page)

```
4:""
5:,,
6:, ,
7:"",
8:" "
9:4,5,6
```

* not_blank

  Filter out the blank lines

  This filter is a shortcut for

  ```
  filter => { 0 => sub { @{$_[1]} > 1 or
              defined $_[1][0] && $_[1][0] ne "" } }
  ```

  Due to the implementation, it is currently impossible to also filter lines that consists only of a quoted empty field. These lines are also considered blank lines.

  With the given example, lines 2 and 4 will be skipped.

* not_empty

  Filter out lines where all the fields are empty.

  This filter is a shortcut for

  ```
  filter => { 0 => sub { grep { defined && $_ ne "" } @{$_
  ↪[1]} } }
  ```

  A space is not regarded being empty, so given the example data, lines 2, 3, 4, 5, and 7 are skipped.

* filled

  Filter out lines that have no visible data

  This filter is a shortcut for

  ```
  filter => { 0 => sub { grep { defined && m/\S/ } @{$_[1]}␣
  ↪} }
  ```

  This filter rejects all lines that *not* have at least one field that does not evaluate to the empty string.

  With the given example data, this filter would skip lines 2 through 8.

One could also use modules like Types::Standard:

```
use Types::Standard -types;

my $type   = Tuple[Str, Str, Int, Bool, Optional[Num]];
my $check  = $type->compiled_check;

# filter with compiled check and warnings
```

(continues on next page)

(continued from previous page)

```
my $aoa = csv (
   in     => \$data,
   filter => {
       0 => sub {
           my $ok = $check->($_[1]) or
               warn $type->get_message ($_[1]), "\n";
           return $ok;
           },
       },
   );
```

- after_in

  This callback is invoked for each record after all records have been parsed but before returning the reference to the caller. The hook is invoked with two arguments: the current CSV parser object and a reference to the record. The reference can be a reference to a HASH or a reference to an ARRAY as determined by the arguments.

  This callback can also be passed as an attribute without the `callbacks` wrapper.

- before_out

  This callback is invoked for each record before the record is printed. The hook is invoked with two arguments: the current CSV parser object and a reference to the record. The reference can be a reference to a HASH or a reference to an ARRAY as determined by the arguments.

  This callback can also be passed as an attribute without the `callbacks` wrapper.

  This callback makes the row available in %_ if the row is a hashref. In this case %_ is writable and will change the original row.

- on_in

  This callback acts exactly as the "after_in" or the "before_out" hooks.

  This callback can also be passed as an attribute without the `callbacks` wrapper.

  This callback makes the row available in %_ if the row is a hashref. In this case %_ is writable and will change the original row. So e.g. with

```
my $aoh = csv (
    in      => \"foo\n1\n2\n",
    headers => "auto",
    on_in   => sub { $_{bar} = 2; },
    );
```

  $aoh will be:

```
[ { foo => 1,
    bar => 2,
    }
  { foo => 2,
    bar => 2,
```

(continues on next page)

(continued from previous page)

```
    }
  ]
```

- on_error

  This callback acts exactly as the “error” hook.

```
my @err;
my $aoa = csv (in => $fh, on_error => sub { @err = @_ });
```

  is identical to

```
my $aoa = csv (in => $fh, callbacks => {
    error => sub { @err = @_ },
    });
```

  It can be used for ignoring errors as well as for just keeping the error in case of analysis after the `csv ()` function has returned.

```
my @err;
my $aoa = csv (in => "bad.csv, on_error => sub { @err = @_ });
die Text::CSV->error_diag if @err or !$aoa;
```

DIAGNOSTICS

This section is also taken from Text::CSV_XS.

Still under construction ...

If an error occurs, `$csv->error_diag` can be used to get information on the cause of the failure. Note that for speed reasons the internal value is never cleared on success, so using the value returned by “error_diag” in normal cases - when no error occurred - may cause unexpected results.

If the constructor failed, the cause can be found using “error_diag” as a class method, like `Text::CSV->error_diag`.

The `$csv->error_diag` method is automatically invoked upon error when the contractor was called with `auto_diag` set to 1 or 2, or when *autodie* is in effect. When set to 1, this will cause a `warn` with the error message, when set to 2, it will `die`. `2012 - EOF` is excluded from `auto_diag` reports.

Errors can be (individually) caught using the “error” callback.

The errors as described below are available. I have tried to make the error itself explanatory enough, but more descriptions will be added. For most of these errors, the first three capitals describe the error category:

- INI

  Initialization error or option conflict.

- ECR

  Carriage-Return related parse error.

- EOF

  End-Of-File related parse error.

- EIQ

  Parse error inside quotation.

- EIF

  Parse error inside field.

- ECB

  Combine error.

- EHR

  HashRef parse related error.

And below should be the complete list of error codes that can be returned:

- 1001 “INI - sep_char is equal to quote_char or escape_char”

  The separation character cannot be equal to the quotation character or to the escape character, as this would invalidate all parsing rules.

- 1002 “INI - allow_whitespace with escape_char or quote_char SP or TAB”

  Using the `allow_whitespace` attribute when either `quote_char` or `escape_char` is equal to `SPACE` or `TAB` is too ambiguous to allow.

- 1003 “INI - \r or \n in main attr not allowed”

  Using default `eol` characters in either `sep_char`, `quote_char`, or `escape_char` is not allowed.

- 1004 “INI - callbacks should be undef or a hashref”

  The `callbacks` attribute only allows one to be `undef` or a hash reference.

- 1005 “INI - EOL too long”

  The value passed for EOL is exceeding its maximum length (16).

- 1006 “INI - SEP too long”

  The value passed for SEP is exceeding its maximum length (16).

- 1007 “INI - QUOTE too long”

  The value passed for QUOTE is exceeding its maximum length (16).

- 1008 “INI - SEP undefined”

  The value passed for SEP should be defined and not empty.

- 1010 “INI - the header is empty”

  The header line parsed in the “header” is empty.

- 1011 “INI - the header contains more than one valid separator”

  The header line parsed in the “header” contains more than one (unique) separator character out of the allowed set of separators.

- 1012 “INI - the header contains an empty field”

  The header line parsed in the “header” contains an empty field.

- 1013 “INI - the header contains nun-unique fields”

  The header line parsed in the “header” contains at least two identical fields.

- 1014 “INI - header called on undefined stream”

  The header line cannot be parsed from an undefined source.

- 1500 “PRM - Invalid/unsupported argument(s)”

  Function or method called with invalid argument(s) or parameter(s).

- 1501 “PRM - The key attribute is passed as an unsupported type”

  The `key` attribute is of an unsupported type.

- 1502 “PRM - The value attribute is passed without the key attribute”

  The `value` attribute is only allowed when a valid key is given.

- 1503 “PRM - The value attribute is passed as an unsupported type”

  The `value` attribute is of an unsupported type.

- 2010 “ECR - QUO char inside quotes followed by CR not part of EOL”

  When `eol` has been set to anything but the default, like `"\r\t\n"`, and the `"\r"` is following the **second** (closing) `quote_char`, where the characters following the `"\r"` do not make up the `eol` sequence, this is an error.

- 2011 “ECR - Characters after end of quoted field”

  Sequences like `1,foo,"bar"baz,22,1` are not allowed. `"bar"` is a quoted field and after the closing double-quote, there should be either a new-line sequence or a separation character.

- 2012 “EOF - End of data in parsing input stream”

  Self-explaining. End-of-file while inside parsing a stream. Can happen only when reading from streams with “getline”, as using “parse” is done on strings that are not required to have a trailing `eol`.

- 2013 “INI - Specification error for fragments RFC7111”

  Invalid specification for URI “fragment” specification.

- 2014 “ENF - Inconsistent number of fields”

  Inconsistent number of fields under strict parsing.

- 2015 “ERW - Empty row”

  An empty row was not allowed.

- 2016 “EOL - Inconsistent EOL”

  Inconsistent End-Of-Line detected under strict_eol parsing.

- 2021 “EIQ - NL char inside quotes, binary off”

  Sequences like `1,"foo\nbar",22,1` are allowed only when the binary option has been selected with the constructor.

- 2022 “EIQ - CR char inside quotes, binary off”

  Sequences like `1,"foo\rbar",22,1` are allowed only when the binary option has been selected with the constructor.

- 2023 “EIQ - QUO character not allowed”

  Sequences like `"foo "bar" baz",qu` and `2023,",2008-04-05,"Foo, Bar",\n` will cause this error.

- 2024 “EIQ - EOF cannot be escaped, not even inside quotes”

  The escape character is not allowed as last character in an input stream.

- 2025 “EIQ - Loose unescaped escape”

  An escape character should escape only characters that need escaping.

  Allowing the escape for other characters is possible with the attribute “allow_loose_escapes”.

- 2026 “EIQ - Binary character inside quoted field, binary off”

  Binary characters are not allowed by default. Exceptions are fields that contain valid UTF-8, that will automatically be upgraded if the content is valid UTF-8. Set `binary` to 1 to accept binary data.

- 2027 “EIQ - Quoted field not terminated”

  When parsing a field that started with a quotation character, the field is expected to be closed with a quotation character. When the parsed line is exhausted before the quote is found, that field is not terminated.

- 2030 “EIF - NL char inside unquoted verbatim, binary off”
- 2031 “EIF - CR char is first char of field, not part of EOL”
- 2032 “EIF - CR char inside unquoted, not part of EOL”
- 2034 “EIF - Loose unescaped quote”
- 2035 “EIF - Escaped EOF in unquoted field”
- 2036 “EIF - ESC error”
- 2037 “EIF - Binary character in unquoted field, binary off”
- 2110 “ECB - Binary character in Combine, binary off”
- 2200 “EIO - print to IO failed. See errno”
- 3001 “EHR - Unsupported syntax for column_names ()”
- 3002 “EHR - getline_hr () called before column_names ()”
- 3003 “EHR - bind_columns () and column_names () fields count mismatch”
- 3004 “EHR - bind_columns () only accepts refs to scalars”
- 3006 “EHR - bind_columns () did not pass enough refs for parsed fields”
- 3007 “EHR - bind_columns needs refs to writable scalars”
- 3008 “EHR - unexpected error in bound fields”
- 3009 “EHR - print_hr () called before column_names ()”

- 3010 “EHR - print_hr () called with invalid arguments”

SEE ALSO

Text::CSV_PP, Text::CSV_XS and Text::CSV::Encoded.

AUTHORS and MAINTAINERS

Alan Citterman <alan[at]mfgrtl.com> wrote the original Perl module. Please don’t send mail concerning Text::CSV to Alan, as he’s not a present maintainer.

Jochen Wiedmann <joe[at]ispsoft.de> rewrote the encoding and decoding in C by implementing a simple finite-state machine and added the variable quote, escape and separator characters, the binary mode and the print and getline methods. See ChangeLog releases 0.10 through 0.23.

H.Merijn Brand <h.m.brand[at]xs4all.nl> cleaned up the code, added the field flags methods, wrote the major part of the test suite, completed the documentation, fixed some RT bugs. See ChangeLog releases 0.25 and on.

Makamaka Hannyaharamitu, <makamaka[at]cpan.org> wrote Text::CSV_PP which is the pure-Perl version of Text::CSV_XS.

New Text::CSV (since 0.99) is maintained by Makamaka, and Kenichi Ishigaki since 1.91.

COPYRIGHT AND LICENSE

Text::CSV

Copyright (C) 1997 Alan Citterman. All rights reserved. Copyright (C) 2007-2015 Makamaka Hannyaharamitu. Copyright (C) 2017- Kenichi Ishigaki A large portion of the doc is taken from Text::CSV_XS. See below.

Text::CSV_PP:

Copyright (C) 2005-2015 Makamaka Hannyaharamitu. Copyright (C) 2017- Kenichi Ishigaki A large portion of the code/doc are also taken from Text::CSV_XS. See below.

Text:CSV_XS:

Copyright (C) 2007-2016 H.Merijn Brand for PROCURA B.V. Copyright (C) 1998-2001 Jochen Wiedmann. All rights reserved. Portions Copyright (C) 1997 Alan Citterman. All rights reserved.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

Text::CSV_PP

Text::CSV_XS compatible pure-Perl module

SYNOPSIS

This section is taken from Text::CSV_XS.

```
# Functional interface
use Text::CSV_PP qw( csv );

# Read whole file in memory
my $aoa = csv (in => "data.csv");    # as array of array
my $aoh = csv (in => "data.csv",
               headers => "auto");   # as array of hash

# Write array of arrays as csv file
csv (in => $aoa, out => "file.csv", sep_char => ";");

# Only show lines where "code" is odd
csv (in => "data.csv", filter => { code => sub { $_ % 2 }});

# Object interface
use Text::CSV_PP;

my @rows;
# Read/parse CSV
my $csv = Text::CSV_PP->new ({ binary => 1, auto_diag => 1 });
open my $fh, "<:encoding(utf8)", "test.csv" or die "test.csv: $!";
while (my $row = $csv->getline ($fh)) {
    $row->[2] =~ m/pattern/ or next; # 3rd field should match
    push @rows, $row;
    }
close $fh;

# and write as CSV
open $fh, ">:encoding(utf8)", "new.csv" or die "new.csv: $!";
$csv->say ($fh, $_) for @rows;
close $fh or die "new.csv: $!";
```

DESCRIPTION

Text::CSV_PP is a pure-perl module that provides facilities for the composition and decomposition of comma-separated values. This is (almost) compatible with much faster Text::CSV_XS, and mainly used as its fallback module when you use *Text::CSV* module without having installed Text::CSV_XS. If you don't have any reason to use this module directly, use Text::CSV for speed boost and portability (or maybe Text::CSV_XS when you write an one-off script and don't need to care about portability).

The following caveats are taken from the doc of Text::CSV_XS.

Embedded newlines

Important Note: The default behavior is to accept only ASCII characters in the range from `0x20` (space) to `0x7E` (tilde). This means that the fields can not contain newlines. If your data contains newlines embedded in fields, or characters above `0x7E` (tilde), or binary data, you ***must*** set `binary => 1` in the call to "new". To cover the widest range of parsing options, you will always want to set binary.

But you still have the problem that you have to pass a correct line to the "parse" method, which is more complicated from the usual point of usage:

```
my $csv = Text::CSV_PP->new ({ binary => 1, eol => $/ });
while (<>) {            #  WRONG!
    $csv->parse ($_);
    my @fields = $csv->fields ();
    }
```

this will break, as the `while` might read broken lines: it does not care about the quoting. If you need to support embedded newlines, the way to go is to **not** pass `eol` in the parser (it accepts `\n`, `\r`, **and** `\r\n` by default) and then

```
my $csv = Text::CSV_PP->new ({ binary => 1 });
open my $fh, "<", $file or die "$file: $!";
while (my $row = $csv->getline ($fh)) {
    my @fields = @$row;
    }
```

The old(er) way of using global file handles is still supported

```
while (my $row = $csv->getline (*ARGV)) { ... }
```

Unicode

Unicode is only tested to work with perl-5.8.2 and up.

See also "BOM".

The simplest way to ensure the correct encoding is used for in- and output is by either setting layers on the filehandles, or setting the "encoding" argument for "csv".

```
 open my $fh, "<:encoding(UTF-8)", "in.csv"  or die "in.csv: $!";
or
 my $aoa = csv (in => "in.csv",     encoding => "UTF-8");

 open my $fh, ">:encoding(UTF-8)", "out.csv" or die "out.csv: $!";
or
 csv (in => $aoa, out => "out.csv", encoding => "UTF-8");
```

On parsing (both for "getline" and "parse"), if the source is marked being UTF8, then all fields that are marked binary will also be marked UTF8.

On combining ("print" and "combine"): if any of the combining fields was marked UTF8, the resulting string will be marked as UTF8. Note however that all fields *before* the first field marked UTF8 and contained 8-bit characters that were not upgraded to UTF8, these

will be `bytes` in the resulting string too, possibly causing unexpected errors. If you pass data of different encoding, or you don't know if there is different encoding, force it to be upgraded before you pass them on:

```
$csv->print ($fh, [ map { utf8::upgrade (my $x = $_); $x } @data ]);
```

For complete control over encoding, please use Text::CSV::Encoded:

```
use Text::CSV::Encoded;
my $csv = Text::CSV::Encoded->new ({
    encoding_in  => "iso-8859-1", # the encoding comes into   Perl
    encoding_out => "cp1252",     # the encoding comes out of Perl
    });

$csv = Text::CSV::Encoded->new ({ encoding  => "utf8" });
# combine () and print () accept *literally* utf8 encoded data
# parse () and getline () return *literally* utf8 encoded data

$csv = Text::CSV::Encoded->new ({ encoding  => undef }); # default
# combine () and print () accept UTF8 marked data
# parse () and getline () return UTF8 marked data
```

BOM

BOM (or Byte Order Mark) handling is available only inside the "header" method. This method supports the following encodings: `utf-8`, `utf-1`, `utf-32be`, `utf-32le`, `utf-16be`, `utf-16le`, `utf-ebcdic`, `scsu`, `bocu-1`, and `gb-18030`. See Wikipedia.

If a file has a BOM, the easiest way to deal with that is

```
my $aoh = csv (in => $file, detect_bom => 1);
```

All records will be encoded based on the detected BOM.

This implies a call to the "header" method, which defaults to also set the "column_names". So this is **not** the same as

```
my $aoh = csv (in => $file, headers => "auto");
```

which only reads the first record to set "column_names" but ignores any meaning of possible present BOM.

METHODS

This section is also taken from Text::CSV_XS.

version

(Class method) Returns the current module version.

new

(Class method) Returns a new instance of class Text::CSV_PP. The attributes are described by the (optional) hash ref `\%attr`.

```
my $csv = Text::CSV_PP->new ({ attributes ... });
```

The following attributes are available:

eol

```
my $csv = Text::CSV_PP->new ({ eol => $/ });
          $csv->eol (undef);
my $eol = $csv->eol;
```

The end-of-line string to add to rows for "print" or the record separator for "getline".

When not passed in a **parser** instance, the default behavior is to accept `\n`, `\r`, and `\r\n`, so it is probably safer to not specify `eol` at all. Passing `undef` or the empty string behave the same.

When not passed in a **generating** instance, records are not terminated at all, so it is probably wise to pass something you expect. A safe choice for `eol` on output is either `$/` or `\r\n`.

Common values for `eol` are `"\012"` (`\n` or Line Feed), `"\015\012"` (`\r\n` or Carriage Return, Line Feed), and `"\015"` (`\r` or Carriage Return). The `eol` attribute cannot exceed 7 (ASCII) characters.

If both `$/` and `eol` equal `"\015"`, parsing lines that end on only a Carriage Return without Line Feed, will be "parse"d correct.

eol_type

```
my $eol = $csv->eol_type;
```

This read-only method returns the internal state of what is considered the valid EOL for parsing.

sep_char

```
my $csv = Text::CSV_PP->new ({ sep_char => ";" });
          $csv->sep_char (";");
my $c = $csv->sep_char;
```

The char used to separate fields, by default a comma. (`,`). Limited to a single-byte character, usually in the range from `0x20` (space) to `0x7E` (tilde). When longer sequences are required, use `sep`.

The separation character can not be equal to the quote character or to the escape character.

sep

```
my $csv = Text::CSV_PP->new ({ sep => "\N{FULLWIDTH COMMA}" });
          $csv->sep (";");
my $sep = $csv->sep;
```

The chars used to separate fields, by default undefined. Limited to 8 bytes.

When set, overrules `sep_char`. If its length is one byte it acts as an alias to `sep_char`.

quote_char

```
my $csv = Text::CSV_PP->new ({ quote_char => "'" });
        $csv->quote_char (undef);
my $c = $csv->quote_char;
```

The character to quote fields containing blanks or binary data, by default the double quote character (`"`). A value of undef suppresses quote chars (for simple cases only). Limited to a single-byte character, usually in the range from `0x20` (space) to `0x7E` (tilde). When longer sequences are required, use `quote`.

`quote_char` can not be equal to `sep_char`.

quote

```
my $csv = Text::CSV_PP->new ({ quote => "\N{FULLWIDTH QUOTATION␣
↪MARK}" });
            $csv->quote ("'");
my $quote = $csv->quote;
```

The chars used to quote fields, by default undefined. Limited to 8 bytes.

When set, overrules `quote_char`. If its length is one byte it acts as an alias to `quote_char`.

This method does not support `undef`. Use `quote_char` to disable quotation.

escape_char

```
my $csv = Text::CSV_PP->new ({ escape_char => "\\" });
        $csv->escape_char (":");
my $c = $csv->escape_char;
```

The character to escape certain characters inside quoted fields. This is limited to a single-byte character, usually in the range from `0x20` (space) to `0x7E` (tilde).

The `escape_char` defaults to being the double-quote mark (`"`). In other words the same as the default `quote_char`. This means that doubling the quote mark in a field escapes it:

```
"foo","bar","Escape ""quote mark"" with two ""quote marks""","baz"
```

If you change the quote_char without changing the escape_char, the escape_char will still be the double-quote ("). If instead you want to escape the quote_char by doubling it you will need to also change the escape_char to be the same as what you have changed the quote_char to.

Setting escape_char to undef or "" will completely disable escapes and is greatly discouraged. This will also disable escape_null.

The escape character can not be equal to the separation character.

binary

```
my $csv = Text::CSV_PP->new ({ binary => 1 });
        $csv->binary (0);
my $f = $csv->binary;
```

If this attribute is 1, you may use binary characters in quoted fields, including line feeds, carriage returns and NULL bytes. (The latter could be escaped as "0".) By default this feature is off.

If a string is marked UTF8, binary will be turned on automatically when binary characters other than CR and NL are encountered. Note that a simple string like "\x{00a0}" might still be binary, but not marked UTF8, so setting { binary => 1 } is still a wise option.

strict

```
my $csv = Text::CSV_PP->new ({ strict => 1 });
        $csv->strict (0);
my $f = $csv->strict;
```

If this attribute is set to 1, any row that parses to a different number of fields than the previous row will cause the parser to throw error 2014.

Empty rows or rows that result in no fields (like comment lines) are exempt from these checks.

strict_eol

```
my $csv = Text::CSV_PP->new ({ strict_eol => 1 });
        $csv->strict_eol (0);
my $f = $csv->strict_eol;
```

If this attribute is set to 0, no EOL consistency checks are done.

If this attribute is set to 1, any row that parses with a EOL other than the EOL from the first row will cause a warning. The error will be ignored and parsing continues. This warning is only thrown once. Note that in data with various different line endings, \r\r will still throw an error that cannot be ignored.

If this attribute is set to 2 or higher, any row that parses with a EOL other than the EOL from the first row will cause error 2016 to be thrown. The line being parsed to this error might not be stored in the result.

skip_empty_rows

```
my $csv = Text::CSV_PP->new ({ skip_empty_rows => 1 });
        $csv->skip_empty_rows ("eof");
my $f = $csv->skip_empty_rows;
```

This attribute defines the behavior for empty rows: an “eol” immediately following the start of line. Default behavior is to return one single empty field.

This attribute is only used in parsing. This attribute is ineffective when using “parse” and “fields”.

Possible values for this attribute are

- 0 | undef

  ```
  my $csv = Text::CSV_PP->new ({ skip_empty_rows => 0 });
  $csv->skip_empty_rows (undef);
  ```

  No special action is taken. The result will be one single empty field.

- 1 | “skip”

  ```
  my $csv = Text::CSV_PP->new ({ skip_empty_rows => 1 });
  $csv->skip_empty_rows ("skip");
  ```

  The row will be skipped.

- 2 | “eof” | “stop”

  ```
  my $csv = Text::CSV_PP->new ({ skip_empty_rows => 2 });
  $csv->skip_empty_rows ("eof");
  ```

  The parsing will stop as if an “eof” was detected.

- 3 | “die”

  ```
  my $csv = Text::CSV_PP->new ({ skip_empty_rows => 3 });
  $csv->skip_empty_rows ("die");
  ```

  The parsing will stop. The internal error code will be set to 2015 and the parser will `die`.

- 4 | “croak”

  ```
  my $csv = Text::CSV_PP->new ({ skip_empty_rows => 4 });
  $csv->skip_empty_rows ("croak");
  ```

  The parsing will stop. The internal error code will be set to 2015 and the parser will `croak`.

- 5 | “error”

  ```
  my $csv = Text::CSV_PP->new ({ skip_empty_rows => 5 });
  $csv->skip_empty_rows ("error");
  ```

  The parsing will fail. The internal error code will be set to 2015.

- callback

```
my $csv = Text::CSV_PP->new ({ skip_empty_rows => sub { [] } });
$csv->skip_empty_rows (sub { [ 42, $., undef, "empty" ] });
```

The callback is invoked and its result used instead. If you want the parse to stop after the callback, make sure to return a false value.

The returned value from the callback should be an array-ref. Any other type will cause the parse to stop, so these are equivalent in behavior:

```
csv (in => $fh, skip_empty_rows => "stop");
csv (in => $fh. skip_empty_rows => sub { 0; });
```

Without arguments, the current value is returned: `0`, `1`, `eof`, `die`, `croak` or the callback.

formula_handling

Alias for “formula”

formula

```
my $csv = Text::CSV_PP->new ({ formula => "none" });
        $csv->formula ("none");
my $f = $csv->formula;
```

This defines the behavior of fields containing *formulas*. As formulas are considered dangerous in spreadsheets, this attribute can define an optional action to be taken if a field starts with an equal sign (=).

For purpose of code-readability, this can also be written as

```
my $csv = Text::CSV_PP->new ({ formula_handling => "none" });
        $csv->formula_handling ("none");
my $f = $csv->formula_handling;
```

Possible values for this attribute are

- none

  Take no specific action. This is the default.

  ```
  $csv->formula ("none");
  ```

- die

  Cause the process to `die` whenever a leading = is encountered.

  ```
  $csv->formula ("die");
  ```

- croak

  Cause the process to `croak` whenever a leading = is encountered. (See *Carp*)

```
$csv->formula ("croak");
```

- diag

  Report position and content of the field whenever a leading = is found. The value of the field is unchanged.

```
$csv->formula ("diag");
```

- empty

  Replace the content of fields that start with a = with the empty string.

```
$csv->formula ("empty");
$csv->formula ("");
```

- undef

  Replace the content of fields that start with a = with `undef`.

```
$csv->formula ("undef");
$csv->formula (undef);
```

- a callback

  Modify the content of fields that start with a = with the return-value of the callback. The original content of the field is available inside the callback as `$_`;

```
# Replace all formula's with 42
$csv->formula (sub { 42; });

# same as $csv->formula ("empty") but slower
$csv->formula (sub { "" });

# Allow =4+12
$csv->formula (sub { s/^=(\d+\+\d+)$/$1/eer });

# Allow more complex calculations
$csv->formula (sub { eval { s{^=([-+*/0-9()]+)$}{$1}ee }; $_ });
```

All other values will give a warning and then fallback to `diag`.

decode_utf8

```
my $csv = Text::CSV_PP->new ({ decode_utf8 => 1 });
        $csv->decode_utf8 (0);
my $f = $csv->decode_utf8;
```

This attributes defaults to TRUE.

While *parsing*, fields that are valid UTF-8, are automatically set to be UTF-8, so that

```
$csv->parse ("\xC4\xA8\n");
```

results in

```
PV("\304\250"\0) [UTF8 "\x{128}"]
```

Sometimes it might not be a desired action. To prevent those upgrades, set this attribute to false, and the result will be

```
PV("\304\250"\0)
```

auto_diag

```
my $csv = Text::CSV_PP->new ({ auto_diag => 1 });
        $csv->auto_diag (2);
my $l = $csv->auto_diag;
```

Set this attribute to a number between 1 and 9 causes "error_diag" to be automatically called in void context upon errors.

In case of error `2012 - EOF`, this call will be void.

If `auto_diag` is set to a numeric value greater than 1, it will `die` on errors instead of `warn`. If set to anything unrecognized, it will be silently ignored.

Future extensions to this feature will include more reliable auto-detection of `autodie` being active in the scope of which the error occurred which will increment the value of `auto_diag` with 1 the moment the error is detected.

diag_verbose

```
my $csv = Text::CSV_PP->new ({ diag_verbose => 1 });
        $csv->diag_verbose (2);
my $l = $csv->diag_verbose;
```

Set the verbosity of the output triggered by `auto_diag`. Currently only adds the current input-record-number (if known) to the diagnostic output with an indication of the position of the error.

blank_is_undef

```
my $csv = Text::CSV_PP->new ({ blank_is_undef => 1 });
        $csv->blank_is_undef (0);
my $f = $csv->blank_is_undef;
```

Under normal circumstances, `CSV` data makes no distinction between quoted- and unquoted empty fields. These both end up in an empty string field once read, thus

```
1,"",," ",2
```

is read as

```
("1", "", "", " ", "2")
```

When *writing* CSV files with either always_quote or quote_empty set, the unquoted *empty* field is the result of an undefined value. To enable this distinction when *reading* CSV data, the blank_is_undef attribute will cause unquoted empty fields to be set to undef, causing the above to be parsed as

```
("1", "", undef, " ", "2")
```

Note that this is specifically important when loading CSV fields into a database that allows NULL values, as the perl equivalent for NULL is undef in *DBI* land.

empty_is_undef

```
my $csv = Text::CSV_PP->new ({ empty_is_undef => 1 });
        $csv->empty_is_undef (0);
my $f = $csv->empty_is_undef;
```

Going one step further than blank_is_undef, this attribute converts all empty fields to undef, so

```
1,"",," ",2
```

is read as

```
(1, undef, undef, " ", 2)
```

Note that this affects only fields that are originally empty, not fields that are empty after stripping allowed whitespace. YMMV.

allow_whitespace

```
my $csv = Text::CSV_PP->new ({ allow_whitespace => 1 });
        $csv->allow_whitespace (0);
my $f = $csv->allow_whitespace;
```

When this option is set to true, the whitespace (TAB's and SPACE's) surrounding the separation character is removed when parsing. If either TAB or SPACE is one of the three characters sep_char, quote_char, or escape_char it will not be considered whitespace.

Now lines like:

```
1 , "foo" , bar , 3 , zapp
```

are parsed as valid CSV, even though it violates the CSV specs.

Note that **all** whitespace is stripped from both start and end of each field. That would make it *more* than a *feature* to enable parsing bad CSV lines, as

```
1,   2.0,  3,   ape  , monkey
```

will now be parsed as

```
("1", "2.0", "3", "ape", "monkey")
```

even if the original line was perfectly acceptable CSV.

allow_loose_quotes

```
my $csv = Text::CSV_PP->new ({ allow_loose_quotes => 1 });
        $csv->allow_loose_quotes (0);
my $f = $csv->allow_loose_quotes;
```

By default, parsing unquoted fields containing quote_char characters like

```
1,foo "bar" baz,42
```

would result in parse error 2034. Though it is still bad practice to allow this format, we cannot help the fact that some vendors make their applications spit out lines styled this way.

If there is **really** bad CSV data, like

```
1,"foo "bar" baz",42
```

or

```
1,""foo bar baz"",42
```

there is a way to get this data-line parsed and leave the quotes inside the quoted field as-is. This can be achieved by setting allow_loose_quotes **AND** making sure that the escape_char is *not* equal to quote_char.

allow_loose_escapes

```
my $csv = Text::CSV_PP->new ({ allow_loose_escapes => 1 });
        $csv->allow_loose_escapes (0);
my $f = $csv->allow_loose_escapes;
```

Parsing fields that have escape_char characters that escape characters that do not need to be escaped, like:

```
my $csv = Text::CSV_PP->new ({ escape_char => "\\" });
$csv->parse (qq{1,"my bar\'s",baz,42});
```

would result in parse error 2025. Though it is bad practice to allow this format, this attribute enables you to treat all escape character sequences equal.

allow_unquoted_escape

```
my $csv = Text::CSV_PP->new ({ allow_unquoted_escape => 1 });
        $csv->allow_unquoted_escape (0);
my $f = $csv->allow_unquoted_escape;
```

A backward compatibility issue where `escape_char` differs from `quote_char` prevents `escape_char` to be in the first position of a field. If `quote_char` is equal to the default `"` and `escape_char` is set to `\`, this would be illegal:

```
1,\0,2
```

Setting this attribute to 1 might help to overcome issues with backward compatibility and allow this style.

always_quote

```
my $csv = Text::CSV_PP->new ({ always_quote => 1 });
          $csv->always_quote (0);
my $f = $csv->always_quote;
```

By default the generated fields are quoted only if they *need* to be. For example, if they contain the separator character. If you set this attribute to 1 then *all* defined fields will be quoted. (`undef` fields are not quoted, see "blank_is_undef"). This makes it quite often easier to handle exported data in external applications.

quote_space

```
my $csv = Text::CSV_PP->new ({ quote_space => 1 });
          $csv->quote_space (0);
my $f = $csv->quote_space;
```

By default, a space in a field would trigger quotation. As no rule exists this to be forced in CSV, nor any for the opposite, the default is true for safety. You can exclude the space from this trigger by setting this attribute to 0.

quote_empty

```
my $csv = Text::CSV_PP->new ({ quote_empty => 1 });
          $csv->quote_empty (0);
my $f = $csv->quote_empty;
```

By default the generated fields are quoted only if they *need* to be. An empty (defined) field does not need quotation. If you set this attribute to 1 then *empty* defined fields will be quoted. (`undef` fields are not quoted, see "blank_is_undef"). See also `always_quote`.

quote_binary

```
my $csv = Text::CSV_PP->new ({ quote_binary => 1 });
          $csv->quote_binary (0);
my $f = $csv->quote_binary;
```

By default, all "unsafe" bytes inside a string cause the combined field to be quoted. By setting this attribute to `0`, you can disable that trigger for bytes `>= 0x7F`.

escape_null

```
my $csv = Text::CSV_PP->new ({ escape_null => 1 });
        $csv->escape_null (0);
my $f = $csv->escape_null;
```

By default, a NULL byte in a field would be escaped. This option enables you to treat the NULL byte as a simple binary character in binary mode (the `{ binary => 1 }` is set). The default is true. You can prevent NULL escapes by setting this attribute to 0.

When the escape_char attribute is set to undefined, this attribute will be set to false.

The default setting will encode "=\x00=" as

```
"="0="
```

With escape_null set, this will result in

```
"=\x00="
```

The default when using the csv function is false.

For backward compatibility reasons, the deprecated old name quote_null is still recognized.

keep_meta_info

```
my $csv = Text::CSV_PP->new ({ keep_meta_info => 1 });
        $csv->keep_meta_info (0);
my $f = $csv->keep_meta_info;
```

By default, the parsing of input records is as simple and fast as possible. However, some parsing information - like quotation of the original field - is lost in that process. Setting this flag to true enables retrieving that information after parsing with the methods "meta_info", "is_quoted", and "is_binary" described below. Default is false for performance.

If you set this attribute to a value greater than 9, then you can control output quotation style like it was used in the input of the the last parsed record (unless quotation was added because of other reasons).

```
my $csv = Text::CSV_PP->new ({
   binary         => 1,
   keep_meta_info => 1,
   quote_space    => 0,
   });

my $row = $csv->parse (q{1,,"", ," ",f,"g","h""h",help,"help"});

$csv->print (*STDOUT, \@row);
# 1,,, , ,f,g,"h""h",help,help
$csv->keep_meta_info (11);
```

(continues on next page)

(continued from previous page)

```
$csv->print (*STDOUT, \@row);
# 1,,"", ," ",f,"g","h""h",help,"help"
```

undef_str

```
my $csv = Text::CSV_PP->new ({ undef_str => "\\N" });
        $csv->undef_str (undef);
my $s = $csv->undef_str;
```

This attribute optionally defines the output of undefined fields. The value passed is not changed at all, so if it needs quotation, the quotation needs to be included in the value of the attribute. Use with caution, as passing a value like " , " , , , , """ will for sure mess up your output. The default for this attribute is `undef`, meaning no special treatment.

This attribute is useful when exporting CSV data to be imported in custom loaders, like for MySQL, that recognize special sequences for `NULL` data.

This attribute has no meaning when parsing CSV data.

comment_str

```
my $csv = Text::CSV_PP->new ({ comment_str => "#" });
        $csv->comment_str (undef);
my $s = $csv->comment_str;
```

This attribute optionally defines a string to be recognized as comment. If this attribute is defined, all lines starting with this sequence will not be parsed as CSV but skipped as comment.

This attribute has no meaning when generating CSV.

Comment strings that start with any of the special characters/sequences are not supported (so it cannot start with any of “sep_char”, “quote_char”, “escape_char”, “sep”, “quote”, or “eol”).

For convenience, `comment` is an alias for `comment_str`.

verbatim

```
my $csv = Text::CSV_PP->new ({ verbatim => 1 });
        $csv->verbatim (0);
my $f = $csv->verbatim;
```

This is a quite controversial attribute to set, but makes some hard things possible.

The rationale behind this attribute is to tell the parser that the normally special characters newline (`NL`) and Carriage Return (`CR`) will not be special when this flag is set, and be dealt with as being ordinary binary characters. This will ease working with data with embedded newlines.

When `verbatim` is used with “getline”, “getline” auto-chomp’s every line.

Imagine a file format like

```
M^^Hans^Janssen^Klas 2\n2A^Ja^11-06-2007#\r\n
```

where, the line ending is a very specific "#\r\n", and the sep_char is a ^ (caret). None of the fields is quoted, but embedded binary data is likely to be present. With the specific line ending, this should not be too hard to detect.

By default, Text::CSV_PP' parse function is instructed to only know about "\n" and "\r" to be legal line endings, and so has to deal with the embedded newline as a real end-of-line, so it can scan the next line if binary is true, and the newline is inside a quoted field. With this option, we tell "parse" to parse the line as if "\n" is just nothing more than a binary character.

For "parse" this means that the parser has no more idea about line ending and "getline" chomps line endings on reading.

types

A set of column types; the attribute is immediately passed to the "types" method.

callbacks

See the "Callbacks" section below.

accessors

To sum it up,

```
$csv = Text::CSV_PP->new ();
```

is equivalent to

```
$csv = Text::CSV_PP->new ({
    eol                 => undef, # \r, \n, or \r\n
    sep_char            => ',',
    sep                 => undef,
    quote_char          => '"',
    quote               => undef,
    escape_char         => '"',
    binary              => 0,
    decode_utf8         => 1,
    auto_diag           => 0,
    diag_verbose        => 0,
    blank_is_undef      => 0,
    empty_is_undef      => 0,
    allow_whitespace    => 0,
    allow_loose_quotes  => 0,
    allow_loose_escapes => 0,
    allow_unquoted_escape => 0,
    always_quote        => 0,
    quote_empty         => 0,
```

(continues on next page)

(continued from previous page)

```
    quote_space             => 1,
    escape_null             => 1,
    quote_binary            => 1,
    keep_meta_info          => 0,
    strict                  => 0,
    skip_empty_rows         => 0,
    formula                 => 0,
    verbatim                => 0,
    undef_str               => undef,
    comment_str             => undef,
    types                   => undef,
    callbacks               => undef,
    });
```

For all of the above mentioned flags, an accessor method is available where you can inquire the current value, or change the value

```
my $quote = $csv->quote_char;
$csv->binary (1);
```

It is not wise to change these settings halfway through writing CSV data to a stream. If however you want to create a new stream using the available CSV object, there is no harm in changing them.

If the "new" constructor call fails, it returns undef, and makes the fail reason available through the "error_diag" method.

```
$csv = Text::CSV_PP->new ({ ecs_char => 1 }) or
    die "".Text::CSV_PP->error_diag ();
```

"error_diag" will return a string like

```
"INI - Unknown attribute 'ecs_char'"
```

known_attributes

```
@attr = Text::CSV_PP->known_attributes;
@attr = Text::CSV_PP::known_attributes;
@attr = $csv->known_attributes;
```

This method will return an ordered list of all the supported attributes as described above. This can be useful for knowing what attributes are valid in classes that use or extend Text::CSV_PP.

print

```
$status = $csv->print ($fh, $colref);
```

Similar to "combine" + "string" + "print", but much more efficient. It expects an array ref as input (not an array!) and the resulting string is not really created, but immediately

written to the `$fh` object, typically an IO handle or any other object that offers a "print" method.

For performance reasons `print` does not create a result string, so all "string", "status", "fields", and "error_input" methods will return undefined information after executing this method.

If `$colref` is `undef` (explicit, not through a variable argument) and "bind_columns" was used to specify fields to be printed, it is possible to make performance improvements, as otherwise data would have to be copied as arguments to the method call:

```
$csv->bind_columns (\($foo, $bar));
$status = $csv->print ($fh, undef);
```

A short benchmark

```
my @data = ("aa" .. "zz");
$csv->bind_columns (\(@data));

$csv->print ($fh, [ @data ]);   # 11800 recs/sec
$csv->print ($fh,  \@data  );   # 57600 recs/sec
$csv->print ($fh,   undef  );   # 48500 recs/sec
```

say

```
$status = $csv->say ($fh, $colref);
```

Like `print`, but `eol` defaults to `$\`.

print_hr

```
$csv->print_hr ($fh, $ref);
```

Provides an easy way to print a `$ref` (as fetched with "getline_hr") provided the column names are set with "column_names".

It is just a wrapper method with basic parameter checks over

```
$csv->print ($fh, [ map { $ref->{$_} } $csv->column_names ]);
```

combine

```
$status = $csv->combine (@fields);
```

This method constructs a CSV record from `@fields`, returning success or failure. Failure can result from lack of arguments or an argument that contains an invalid character. Upon success, "string" can be called to retrieve the resultant CSV string. Upon failure, the value returned by "string" is undefined and "error_input" could be called to retrieve the invalid argument.

string

```
$line = $csv->string ();
```

This method returns the input to “parse” or the resultant CSV string of “combine”, whichever was called more recently.

getline

```
$colref = $csv->getline ($fh);
```

This is the counterpart to “print”, as “parse” is the counterpart to “combine”: it parses a row from the `$fh` handle using the “getline” method associated with `$fh` and parses this row into an array ref. This array ref is returned by the function or `undef` for failure. When `$fh` does not support `getline`, you are likely to hit errors.

When fields are bound with “bind_columns” the return value is a reference to an empty list.

The “string”, “fields”, and “status” methods are meaningless again.

getline_all

```
$arrayref = $csv->getline_all ($fh);
$arrayref = $csv->getline_all ($fh, $offset);
$arrayref = $csv->getline_all ($fh, $offset, $length);
```

This will return a reference to a list of getline ($fh) results. In this call, `keep_meta_info` is disabled. If `$offset` is negative, as with `splice`, only the last `abs ($offset)` records of `$fh` are taken into consideration. Parameters `$offset` and `$length` are expected to be integers. Non-integer values are interpreted as integer without check.

Given a CSV file with 10 lines:

```
lines call
----- --------------------------------------------------------
0..9  $csv->getline_all ($fh)         # all
0..9  $csv->getline_all ($fh,  0)     # all
8..9  $csv->getline_all ($fh,  8)     # start at 8
-     $csv->getline_all ($fh,  0,  0) # start at 0 first 0 rows
0..4  $csv->getline_all ($fh,  0,  5) # start at 0 first 5 rows
4..5  $csv->getline_all ($fh,  4,  2) # start at 4 first 2 rows
8..9  $csv->getline_all ($fh, -2)     # last 2 rows
6..7  $csv->getline_all ($fh, -4,  2) # first 2 of last  4 rows
```

getline_hr

The “getline_hr” and “column_names” methods work together to allow you to have rows returned as hashrefs. You must call “column_names” first to declare your column names.

```
$csv->column_names (qw( code name price description ));
$hr = $csv->getline_hr ($fh);
print "Price for $hr->{name} is $hr->{price} EUR\n";
```

"getline_hr" will croak if called before "column_names".

Note that "getline_hr" creates a hashref for every row and will be much slower than the combined use of "bind_columns" and "getline" but still offering the same easy to use hashref inside the loop:

```
my @cols = @{$csv->getline ($fh)};
$csv->column_names (@cols);
while (my $row = $csv->getline_hr ($fh)) {
    print $row->{price};
    }
```

Could easily be rewritten to the much faster:

```
my @cols = @{$csv->getline ($fh)};
my $row = {};
$csv->bind_columns (\@{$row}{@cols});
while ($csv->getline ($fh)) {
    print $row->{price};
    }
```

Your mileage may vary for the size of the data and the number of rows. With perl-5.14.2 the comparison for a 100_000 line file with 14 columns:

```
           Rate hashrefs getlines
hashrefs 1.00/s       --     -76%
getlines 4.15/s     313%       --
```

getline_hr_all

```
$arrayref = $csv->getline_hr_all ($fh);
$arrayref = $csv->getline_hr_all ($fh, $offset);
$arrayref = $csv->getline_hr_all ($fh, $offset, $length);
```

This will return a reference to a list of getline_hr ($fh) results. In this call, `keep_meta_info` is disabled.

parse

```
$status = $csv->parse ($line);
```

This method decomposes a CSV string into fields, returning success or failure. Failure can result from a lack of argument or the given CSV string is improperly formatted. Upon success, "fields" can be called to retrieve the decomposed fields. Upon failure calling "fields" will return undefined data and "error_input" can be called to retrieve the invalid argument.

You may use the "types" method for setting column types. See "types"' description below.

The `$line` argument is supposed to be a simple scalar. Everything else is supposed to croak and set error 1500.

fragment

This function tries to implement RFC7111 (URI Fragment Identifiers for the text/csv Media Type) - https://datatracker.ietf.org/doc/html/rfc7111

```
my $AoA = $csv->fragment ($fh, $spec);
```

In specifications, `*` is used to specify the *last* item, a dash (`-`) to indicate a range. All indices are 1-based: the first row or column has index 1. Selections can be combined with the semi-colon (`;`).

When using this method in combination with "column_names", the returned reference will point to a list of hashes instead of a list of lists. A disjointed cell-based combined selection might return rows with different number of columns making the use of hashes unpredictable.

```
$csv->column_names ("Name", "Age");
my $AoH = $csv->fragment ($fh, "col=3;8");
```

If the "after_parse" callback is active, it is also called on every line parsed and skipped before the fragment.

- row

  ```
  row=4
  row=5-7
  row=6-*
  row=1-2;4;6-*
  ```

- col

  ```
  col=2
  col=1-3
  col=4-*
  col=1-2;4;7-*
  ```

- cell

  In cell-based selection, the comma (`,`) is used to pair row and column

  ```
  cell=4,1
  ```

  The range operator (`-`) using `cell`s can be used to define top-left and bottom-right `cell` location

  ```
  cell=3,1-4,6
  ```

  The `*` is only allowed in the second part of a pair

```
cell=3,2-*,2    # row 3 till end, only column 2
cell=3,2-3,*    # column 2 till end, only row 3
cell=3,2-*,*    # strip row 1 and 2, and column 1
```

Cells and cell ranges may be combined with `;`, possibly resulting in rows with different numbers of columns

```
cell=1,1-2,2;3,3-4,4;1,4;4,1
```

Disjointed selections will only return selected cells. The cells that are not specified will not be included in the returned set, not even as `undef`. As an example given a CSV like

```
11,12,13,...19
21,22,...28,29
:            :
91,...97,98,99
```

with `cell=1,1-2,2;3,3-4,4;1,4;4,1` will return:

```
11,12,14
21,22
33,34
41,43,44
```

Overlapping cell-specs will return those cells only once, So `cell=1,1-3,3;2,2-4,4;2,3;4,2` will return:

```
11,12,13
21,22,23,24
31,32,33,34
42,43,44
```

RFC7111 does **not** allow different types of specs to be combined (either `row` *or* `col` *or* `cell`). Passing an invalid fragment specification will croak and set error 2013.

column_names

Set the "keys" that will be used in the "getline_hr" calls. If no keys (column names) are passed, it will return the current setting as a list.

"column_names" accepts a list of scalars (the column names) or a single array_ref, so you can pass the return value from "getline" too:

```
$csv->column_names ($csv->getline ($fh));
```

"column_names" does **no** checking on duplicates at all, which might lead to unexpected results. Undefined entries will be replaced with the string `"\cAUNDEF\cA"`, so

```
$csv->column_names (undef, "", "name", "name");
$hr = $csv->getline_hr ($fh);
```

will set `$hr->{"\cAUNDEF\cA"}` to the 1st field, `$hr->{""}` to the 2nd field, and `$hr->{name}` to the 4th field, discarding the 3rd field.

"column_names" croaks on invalid arguments.

header

This method does NOT work in perl-5.6.x

Parse the CSV header and set `sep`, column_names and encoding.

```
my @hdr = $csv->header ($fh);
$csv->header ($fh, { sep_set => [ ";", ",", "|", "\t" ] });
$csv->header ($fh, { detect_bom => 1, munge_column_names => "lc" });
```

The first argument should be a file handle.

This method resets some object properties, as it is supposed to be invoked only once per file or stream. It will leave attributes `column_names` and `bound_columns` alone if setting column names is disabled. Reading headers on previously process objects might fail on perl-5.8.0 and older.

Assuming that the file opened for parsing has a header, and the header does not contain problematic characters like embedded newlines, read the first line from the open handle then auto-detect whether the header separates the column names with a character from the allowed separator list.

If any of the allowed separators matches, and none of the *other* allowed separators match, set `sep` to that separator for the current CSV_PP instance and use it to parse the first line, map those to lowercase, and use that to set the instance "column_names":

```
my $csv = Text::CSV_PP->new ({ binary => 1, auto_diag => 1 });
open my $fh, "<", "file.csv";
binmode $fh; # for Windows
$csv->header ($fh);
while (my $row = $csv->getline_hr ($fh)) {
    ...
    }
```

If the header is empty, contains more than one unique separator out of the allowed set, contains empty fields, or contains identical fields (after folding), it will croak with error 1010, 1011, 1012, or 1013 respectively.

If the header contains embedded newlines or is not valid CSV in any other way, this method will croak and leave the parse error untouched.

A successful call to `header` will always set the `sep` of the `$csv` object. This behavior can not be disabled.

return value

On error this method will croak.

In list context, the headers will be returned whether they are used to set "column_names" or not.

In scalar context, the instance itself is returned. **Note**: the values as found in the header will effectively be **lost** if `set_column_names` is false.

Options

- sep_set

```
$csv->header ($fh, { sep_set => [ ";", ",", "|", "\t" ] });
```

The list of legal separators defaults to `[ ";", "," ]` and can be changed by this option. As this is probably the most often used option, it can be passed on its own as an unnamed argument:

```
$csv->header ($fh, [ ";", ",", "|", "\t", "::", "\x{2063}" ]);
```

Multi-byte sequences are allowed, both multi-character and Unicode. See `sep`.

- detect_bom

```
$csv->header ($fh, { detect_bom => 1 });
```

The default behavior is to detect if the header line starts with a BOM. If the header has a BOM, use that to set the encoding of `$fh`. This default behavior can be disabled by passing a false value to `detect_bom`.

Supported encodings from BOM are: UTF-8, UTF-16BE, UTF-16LE, UTF-32BE, and UTF-32LE. BOM also supports UTF-1, UTF-EBCDIC, SCSU, BOCU-1, and GB-18030 but *Encode* does not (yet). UTF-7 is not supported.

If a supported BOM was detected as start of the stream, it is stored in the object attribute `ENCODING`.

```
my $enc = $csv->{ENCODING};
```

The encoding is used with `binmode` on `$fh`.

If the handle was opened in a (correct) encoding, this method will **not** alter the encoding, as it checks the leading **bytes** of the first line. In case the stream starts with a decoded BOM (`U+FEFF`), `{ENCODING}` will be `""` (empty) instead of the default `undef`.

- munge_column_names

This option offers the means to modify the column names into something that is most useful to the application. The default is to map all column names to lower case.

```
$csv->header ($fh, { munge_column_names => "lc" });
```

The following values are available:

```
lc      - lower case
uc      - upper case
db      - valid DB field names
none    - do not change
```

(continues on next page)

(continued from previous page)

```
\%hash - supply a mapping
\&cb   - supply a callback
```

- Lower case

```
$csv->header ($fh, { munge_column_names => "lc" });
```

  The header is changed to all lower-case

```
$_ = lc;
```

- Upper case

```
$csv->header ($fh, { munge_column_names => "uc" });
```

  The header is changed to all upper-case

```
$_ = uc;
```

- Literal

```
$csv->header ($fh, { munge_column_names => "none" });
```

- Hash

```
$csv->header ($fh, { munge_column_names => { foo => "sombrero
 ↪" });
```

  if a value does not exist, the original value is used unchanged

- Database

```
$csv->header ($fh, { munge_column_names => "db" });
```

  * .

    lower-case

  * .

    all sequences of non-word characters are replaced with an underscore

  * .

    all leading underscores are removed

```
$_ = lc (s/\W+/_/gr =~ s/^_+//r);
```

- Callback

```
$csv->header ($fh, { munge_column_names => sub { fc } });
$csv->header ($fh, { munge_column_names => sub { "column_".
 ↪$col++ } });
$csv->header ($fh, { munge_column_names => sub { lc (s/\W+/_/
 ↪gr) } });
```

As this callback is called in a `map`, you can use `$_` directly.

- set_column_names

```
$csv->header ($fh, { set_column_names => 1 });
```

The default is to set the instances column names using "column_names" if the method is successful, so subsequent calls to "getline_hr" can return a hash. Disable setting the header can be forced by using a false value for this option.

As described in "return value" above, content is lost in scalar context.

Validation

When receiving CSV files from external sources, this method can be used to protect against changes in the layout by restricting to known headers (and typos in the header fields).

```
my %known = (
    "record key" => "c_rec",
    "rec id"     => "c_rec",
    "id_rec"     => "c_rec",
    "kode"       => "code",
    "code"       => "code",
    "vaule"      => "value",
    "value"      => "value",
    );
my $csv = Text::CSV_PP->new ({ binary => 1, auto_diag => 1 });
open my $fh, "<", $source or die "$source: $!";
$csv->header ($fh, { munge_column_names => sub {
    s/\s+$//;
    s/^\s+//;
    $known{lc $_} or die "Unknown column '$_' in $source";
    }});
while (my $row = $csv->getline_hr ($fh)) {
    say join "\t", $row->{c_rec}, $row->{code}, $row->{value};
    }
```

bind_columns

Takes a list of scalar references to be used for output with "print" or to store in the fields fetched by "getline". When you do not pass enough references to store the fetched fields in, "getline" will fail with error `3006`. If you pass more than there are fields to return, the content of the remaining references is left untouched. Under `strict` the two should match, otherwise "getline" will fail with error `2014`.

```
$csv->bind_columns (\$code, \$name, \$price, \$description);
while ($csv->getline ($fh)) {
    print "The price of a $name is \x{20ac} $price\n";
    }
```

To reset or clear all column binding, call "bind_columns" with the single argument undef. This will also clear column names.

```
$csv->bind_columns (undef);
```

If no arguments are passed at all, "bind_columns" will return the list of current bindings or undef if no binds are active.

Note that in parsing with `bind_columns`, the fields are set on the fly. That implies that if the third field of a row causes an error (or this row has just two fields where the previous row had more), the first two fields already have been assigned the values of the current row, while the rest of the fields will still hold the values of the previous row. If you want the parser to fail in these cases, use the `strict` attribute.

eof

```
$eof = $csv->eof ();
```

If "parse" or "getline" was used with an IO stream, this method will return true (1) if the last call hit end of file, otherwise it will return false ('). This is useful to see the difference between a failure and end of file.

Note that if the parsing of the last line caused an error, `eof` is still true. That means that if you are *not* using "auto_diag", an idiom like

```
while (my $row = $csv->getline ($fh)) {
    # ...
    }
$csv->eof or $csv->error_diag;
```

will *not* report the error. You would have to change that to

```
while (my $row = $csv->getline ($fh)) {
    # ...
    }
+$csv->error_diag and $csv->error_diag;
```

types

```
$csv->types (\@tref);
```

This method is used to force that (all) columns are of a given type. For example, if you have an integer column, two columns with doubles and a string column, then you might do a

```
$csv->types ([Text::CSV_PP::IV (),
              Text::CSV_PP::NV (),
              Text::CSV_PP::NV (),
              Text::CSV_PP::PV ()]);
```

Column types are used only for *decoding* columns while parsing, in other words by the "parse" and "getline" methods.

You can unset column types by doing a

```
$csv->types (undef);
```

or fetch the current type settings with

```
$types = $csv->types ();
```

- IV
- CSV_TYPE_IV

  Set field type to integer.
- NV
- CSV_TYPE_NV

  Set field type to numeric/float.
- PV
- CSV_TYPE_PV

  Set field type to string.

fields

```
@columns = $csv->fields ();
```

This method returns the input to "combine" or the resultant decomposed fields of a successful "parse", whichever was called more recently.

Note that the return value is undefined after using "getline", which does not fill the data structures returned by "parse".

meta_info

```
@flags = $csv->meta_info ();
```

This method returns the "flags" of the input to "combine" or the flags of the resultant decomposed fields of "parse", whichever was called more recently.

For each field, a meta_info field will hold flags that inform something about the field returned by the "fields" method or passed to the "combine" method. The flags are bit-wise-or'd like:

- 0x0001
- CSV_FLAGS_IS_QUOTED

  The field was quoted.
- 0x0002
- CSV_FLAGS_IS_BINARY

  The field was binary.

- 0x0004
- CSV_FLAGS_ERROR_IN_FIELD

  The field was invalid.

  Currently only used when allow_loose_quotes is active.
- 0x0010
- CSV_FLAGS_IS_MISSING

  The field was missing.

See the is_*** methods below.

is_quoted

```
my $quoted = $csv->is_quoted ($column_idx);
```

where $column_idx is the (zero-based) index of the column in the last result of "parse".

This returns a true value if the data in the indicated column was enclosed in quote_char quotes. This might be important for fields where content ,20070108, is to be treated as a numeric value, and where ,"20070108", is explicitly marked as character string data.

This method is only valid when "keep_meta_info" is set to a true value.

is_binary

```
my $binary = $csv->is_binary ($column_idx);
```

where $column_idx is the (zero-based) index of the column in the last result of "parse".

This returns a true value if the data in the indicated column contained any byte in the range [\x00-\x08,\x10-\x1F,\x7F-\xFF].

This method is only valid when "keep_meta_info" is set to a true value.

is_missing

```
my $missing = $csv->is_missing ($column_idx);
```

where $column_idx is the (zero-based) index of the column in the last result of "getline_hr".

```
$csv->keep_meta_info (1);
while (my $hr = $csv->getline_hr ($fh)) {
    $csv->is_missing (0) and next; # This was an empty line
    }
```

When using "getline_hr", it is impossible to tell if the parsed fields are undef because they where not filled in the CSV stream or because they were not read at all, as **all** the fields defined by "column_names" are set in the hash-ref. If you still need to know if all

fields in each row are provided, you should enable `keep_meta_info` so you can check the flags.

If `keep_meta_info` is `false`, `is_missing` will always return `undef`, regardless of `$column_idx` being valid or not. If this attribute is `true` it will return either `0` (the field is present) or 1 (the field is missing).

A special case is the empty line. If the line is completely empty - after dealing with the flags - this is still a valid CSV line: it is a record of just one single empty field. However, if `keep_meta_info` is set, invoking `is_missing` with index `0` will now return true.

status

```
$status = $csv->status ();
```

This method returns the status of the last invoked "combine" or "parse" call. Status is success (true: 1) or failure (false: `undef` or `0`).

Note that as this only keeps track of the status of above mentioned methods, you are probably looking for `error_diag` instead.

error_input

```
$bad_argument = $csv->error_input ();
```

This method returns the erroneous argument (if it exists) of "combine" or "parse", whichever was called more recently. If the last invocation was successful, `error_input` will return `undef`.

Depending on the type of error, it *might* also hold the data for the last error-input of "getline".

error_diag

```
Text::CSV_PP->error_diag ();
$csv->error_diag ();
$error_code  =  0  + $csv->error_diag ();
$error_str   =  "" . $csv->error_diag ();
($cde, $str, $pos, $rec, $fld, $xs) = $csv->error_diag ();
```

If (and only if) an error occurred, this function returns the diagnostics of that error.

If called in void context, this will print the internal error code and the associated error message to STDERR.

If called in list context, this will return the error code and the error message in that order. If the last error was from parsing, the rest of the values returned are a best guess at the location within the line that was being parsed. Their values are 1-based. The position currently is index of the byte at which the parsing failed in the current record. It might change to be the index of the current character in a later release. The records is the index of the record parsed by the csv instance. The field number is the index of the field the parser thinks it is currently trying to parse. See `examples/csv-check` for how this can be used.

If called in scalar context, it will return the diagnostics in a single scalar, a-la `$!`. It will contain the error code in numeric context, and the diagnostics message in string context.

When called as a class method or a direct function call, the diagnostics are that of the last "new" call.

_cache_diag

Note: This is an internal function only, and output cannot be relied upon. Use at own risk.

If debugging beyond what "error_diag" is able to show, the internal cache can be shown with this function.

```
# Something failed ..
$csv->error_diag;
$csv->_cache_diag ();
```

record_number

```
$recno = $csv->record_number ();
```

Returns the records parsed by this csv instance. This value should be more accurate than `$.` when embedded newlines come in play. Records written by this instance are not counted.

SetDiag

```
$csv->SetDiag (0);
```

Use to reset the diagnostics if you are dealing with errors.

FUNCTIONS

This section is also taken from Text::CSV_XS.

csv

This function is not exported by default and should be explicitly requested:

```
use Text::CSV_PP qw( csv );
```

This is a high-level function that aims at simple (user) interfaces. This can be used to read/parse a `CSV` file or stream (the default behavior) or to produce a file or write to a stream (define the `out` attribute). It returns an array- or hash-reference on parsing (or `undef` on fail) or the numeric value of "error_diag" on writing. When this function fails you can get to the error using the class call to "error_diag"

```
my $aoa = csv (in => "test.csv") or
    die Text::CSV_PP->error_diag;
```

Note that failure here is the inability to start the parser, like when the input does not exist or the arguments are unknown or conflicting. Run-time parsing errors will return a valid reference, which can be empty, but still contains all results up till the error. See "on_error".

This function takes the arguments as key-value pairs. This can be passed as a list or as an anonymous hash:

```
my $aoa = csv (  in => "test.csv", sep_char => ";");
my $aoh = csv ({ in => $fh, headers => "auto" });
```

The arguments passed consist of two parts: the arguments to "csv" itself and the optional attributes to the CSV object used inside the function as enumerated and explained in "new".

If not overridden, the default option used for CSV is

```
auto_diag   => 1
escape_null => 0
strict_eol  => 1
```

The option that is always set and cannot be altered is

```
binary      => 1
```

As this function will likely be used in one-liners, it allows `quote` to be abbreviated as quo, and `escape_char` to be abbreviated as `esc` or `escape`.

Alternative invocations:

```
my $aoa = Text::CSV_PP::csv (in => "file.csv");

my $csv = Text::CSV_PP->new ();
my $aoa = $csv->csv (in => "file.csv");
```

In the latter case, the object attributes are used from the existing object and the attribute arguments in the function call are ignored:

```
my $csv = Text::CSV_PP->new ({ sep_char => ";" });
my $aoh = $csv->csv (in => "file.csv", bom => 1);
```

will parse using `;` as `sep_char`, not `,`.

in

Used to specify the source. `in` can be a file name (e.g. `"file.csv"`), which will be opened for reading and closed when finished, a file handle (e.g. `$fh` or `FH`), a reference to a glob (e.g. `\*ARGV`), the glob itself (e.g. `*STDIN`), or a reference to a scalar (e.g. `\q{1,2,"csv"}`).

When used with "out", `in` should be a reference to a CSV structure (AoA or AoH) or a CODE-ref that returns an array-reference or a hash-reference. The code-ref will be invoked with no arguments.

```
my $aoa = csv (in => "file.csv");

open my $fh, "<", "file.csv";
my $aoa = csv (in => $fh);

my $csv = [ [qw( Foo Bar )], [ 1, 2 ], [ 2, 3 ]];
my $err = csv (in => $csv, out => "file.csv");
```

If called in void context without the "out" attribute, the resulting ref will be used as input to a subsequent call to csv:

```
csv (in => "file.csv", filter => { 2 => sub { length > 2 }})
```

will be a shortcut to

```
csv (in => csv (in => "file.csv", filter => { 2 => sub { length > 2␣
↪}}))
```

where, in the absence of the out attribute, this is a shortcut to

```
csv (in  => csv (in => "file.csv", filter => { 2 => sub { length > 2␣
↪}}),
     out => *STDOUT)
```

out

```
csv (in => $aoa,  out => "file.csv");
csv (in => $aoa,  out => $fh);
csv (in => $aoa,  out =>   STDOUT);
csv (in => $aoa,  out =>  *STDOUT);
csv (in => $aoa,  out => \*STDOUT);
csv (in => $aoa,  out => \my $data);
csv (in => $aoa,  out =>  undef);
csv (in => $aoa,  out => \"skip");

csv (in => $fh,   out => \@aoa);
csv (in => $fh,   out => \@aoh, bom => 1);
csv (in => $fh,   out => \%hsh, key => "key");

csv (in => $file, out => $file);
csv (in => $file, out => $fh);
csv (in => $fh,   out => $file);
csv (in => $fh,   out => $fh);
```

In output mode, the default CSV options when producing CSV are

```
eol       => "\r\n"
```

The "fragment" attribute is ignored in output mode.

out can be a file name (e.g. "file.csv"), which will be opened for writing and closed

when finished, a file handle (e.g. `$fh` or `FH`), a reference to a glob (e.g. `\*STDOUT`), the glob itself (e.g. `*STDOUT`), or a reference to a scalar (e.g. `\my $data`).

```
csv (in => sub { $sth->fetch },            out => "dump.csv");
csv (in => sub { $sth->fetchrow_hashref }, out => "dump.csv",
     headers => $sth->{NAME_lc});
```

When a code-ref is used for `in`, the output is generated per invocation, so no buffering is involved. This implies that there is no size restriction on the number of records. The `csv` function ends when the coderef returns a false value.

If `out` is set to a reference of the literal string `"skip"`, the output will be suppressed completely, which might be useful in combination with a filter for side effects only.

```
my %cache;
csv (in    => "dump.csv",
     out   => \"skip",
     on_in => sub { $cache{$_[1][1]}++ });
```

Currently, setting `out` to any false value (`undef`, `""`, 0) will be equivalent to `\"skip"`.

If the `in` argument point to something to parse, and the `out` is set to a reference to an `ARRAY` or a `HASH`, the output is appended to the data in the existing reference. The result of the parse should match what exists in the reference passed. This might come handy when you have to parse a set of files with similar content (like data stored per period) and you want to collect that into a single data structure:

```
my %hash;
csv (in => $_, out => \%hash, key => "id") for sort glob "foo-[0-
↪9]*.csv";

my @list; # List of arrays
csv (in => $_, out => \@list)              for sort glob "foo-[0-
↪9]*.csv";

my @list; # List of hashes
csv (in => $_, out => \@list, bom => 1)    for sort glob "foo-[0-
↪9]*.csv";
```

Streaming

If **both** `in` and `out` are files, file handles or globs, streaming is enforced by injecting an `after_parse` callback that immediately uses the `say ()` method of the same instance to output the result and then rejects the record.

If a `after_parse` was already passed as attribute, that will be included in the injected call. If `on_in` was passed and `after_parse` was not, it will be used instead. If both were passed, `on_in` is ignored.

The EOL of the first record of the `in` source is consistently used as EOL for all records in the `out` destination.

The `filter` attribute is not available.

All other attributes are shared for `in` and `out`, so you cannot define different encodings for `in` and `out`. You need to pass a `$fh`, where `binmode` was used to apply the encoding layers.

Note that this is work in progress and things might change.

encoding

If passed, it should be an encoding accepted by the `:encoding()` option to open. There is no default value. This attribute does not work in perl 5.6.x. `encoding` can be abbreviated to `enc` for ease of use in command line invocations.

If `encoding` is set to the literal value `"auto"`, the method "header" will be invoked on the opened stream to check if there is a BOM and set the encoding accordingly. This is equal to passing a true value in the option `detect_bom`.

Encodings can be stacked, as supported by `binmode`:

```
# Using PerlIO::via::gzip
csv (in       => \@csv,
     out      => "test.csv:via.gz",
     encoding => ":via(gzip):encoding(utf-8)",
     );
$aoa = csv (in => "test.csv:via.gz",  encoding => ":via(gzip)");

# Using PerlIO::gzip
csv (in       => \@csv,
     out      => "test.csv:via.gz",
     encoding => ":gzip:encoding(utf-8)",
     );
$aoa = csv (in => "test.csv:gzip.gz", encoding => ":gzip");
```

detect_bom

If `detect_bom` is given, the method "header" will be invoked on the opened stream to check if there is a BOM and set the encoding accordingly.

`detect_bom` can be abbreviated to `bom`.

This is the same as setting `encoding` to `"auto"`.

Note that as the method "header" is invoked, its default is to also set the headers.

headers

If this attribute is not given, the default behavior is to produce an array of arrays.

If `headers` is supplied, it should be an anonymous list of column names, an anonymous hashref, a coderef, or a literal flag: `auto`, `lc`, `uc`, or `skip`.

- skip

  When `skip` is used, the header will not be included in the output.

```
my $aoa = csv (in => $fh, headers => "skip");
```

`skip` is invalid/ignored in combinations with `detect_bom`.

- auto

  If `auto` is used, the first line of the CSV source will be read as the list of field headers and used to produce an array of hashes.

```
my $aoh = csv (in => $fh, headers => "auto");
```

- lc

  If `lc` is used, the first line of the CSV source will be read as the list of field headers mapped to lower case and used to produce an array of hashes. This is a variation of `auto`.

```
my $aoh = csv (in => $fh, headers => "lc");
```

- uc

  If `uc` is used, the first line of the CSV source will be read as the list of field headers mapped to upper case and used to produce an array of hashes. This is a variation of `auto`.

```
my $aoh = csv (in => $fh, headers => "uc");
```

- CODE

  If a coderef is used, the first line of the CSV source will be read as the list of mangled field headers in which each field is passed as the only argument to the coderef. This list is used to produce an array of hashes.

```
my $aoh = csv (in      => $fh,
               headers => sub { lc ($_[0]) =~ s/kode/code/gr });
```

  this example is a variation of using `lc` where all occurrences of kode are replaced with code.

- ARRAY

  If `headers` is an anonymous list, the entries in the list will be used as field names. The first line is considered data instead of headers.

```
my $aoh = csv (in => $fh, headers => [qw( Foo Bar )]);
csv (in => $aoa, out => $fh, headers => [qw( code description␣
 ↪price )]);
```

- HASH

  If `headers` is a hash reference, this implies `auto`, but header fields that exist as key in the hashref will be replaced by the value for that key. Given a CSV file like

```
post-kode,city,name,id number,fubble
1234AA,Duckstad,Donald,13,"X313DF"
```

using

```
csv (headers => { "post-kode" => "pc", "id number" => "ID" }, ..
↪.
```

will return an entry like

```
{ pc     => "1234AA",
  city   => "Duckstad",
  name   => "Donald",
  ID     => "13",
  fubble => "X313DF",
  }
```

See also `munge_column_names` and `set_column_names`.

munge_column_names

If `munge_column_names` is set, the method "header" is invoked on the opened stream with all matching arguments to detect and set the headers.

`munge_column_names` can be abbreviated to `munge`.

key

If passed, will default `headers` to `"auto"` and return a hashref instead of an array of hashes. Allowed values are simple scalars or array-references where the first element is the joiner and the rest are the fields to join to combine the key.

```
my $ref = csv (in => "test.csv", key => "code");
my $ref = csv (in => "test.csv", key => [ ":" => "code", "color" ]);
```

with test.csv like

```
code,product,price,color
1,pc,850,gray
2,keyboard,12,white
3,mouse,5,black
```

the first example will return

```
{ 1   => {
      code    => 1,
      color   => 'gray',
      price   => 850,
      product => 'pc'
      },
  2   => {
      code    => 2,
      color   => 'white',
      price   => 12,
      product => 'keyboard'
```

(continues on next page)

(continued from previous page)

```
      },
  3   => {
      code    => 3,
      color   => 'black',
      price   => 5,
      product => 'mouse'
      }
  }
```

the second example will return

```
{ "1:gray"    => {
      code    => 1,
      color   => 'gray',
      price   => 850,
      product => 'pc'
      },
  "2:white"   => {
      code    => 2,
      color   => 'white',
      price   => 12,
      product => 'keyboard'
      },
  "3:black"   => {
      code    => 3,
      color   => 'black',
      price   => 5,
      product => 'mouse'
      }
  }
```

The key attribute can be combined with `headers` for `CSV` date that has no header line, like

```
my $ref = csv (
    in      => "foo.csv",
    headers => [qw( c_foo foo bar description stock )],
    key     =>     "c_foo",
    );
```

value

Used to create key-value hashes.

Only allowed when `key` is valid. A `value` can be either a single column label or an anonymous list of column labels. In the first case, the value will be a simple scalar value, in the latter case, it will be a hashref.

```
my $ref = csv (in => "test.csv", key   => "code",
                                 value => "price");
```

(continues on next page)

(continued from previous page)

```
my $ref = csv (in => "test.csv", key   => "code",
                                 value => [ "product", "price" ]);
my $ref = csv (in => "test.csv", key   => [ ":" => "code", "color"
 ↪],
                                 value => "price");
my $ref = csv (in => "test.csv", key   => [ ":" => "code", "color"
 ↪],
                                 value => [ "product", "price" ]);
```

with test.csv like

```
code,product,price,color
1,pc,850,gray
2,keyboard,12,white
3,mouse,5,black
```

the first example will return

```
{ 1 => 850,
  2 =>  12,
  3 =>   5,
  }
```

the second example will return

```
{ 1   => {
      price   => 850,
      product => 'pc'
      },
  2   => {
      price   => 12,
      product => 'keyboard'
      },
  3   => {
      price   => 5,
      product => 'mouse'
      }
  }
```

the third example will return

```
{ "1:gray"    => 850,
  "2:white"   =>  12,
  "3:black"   =>   5,
  }
```

the fourth example will return

```
{ "1:gray"    => {
      price   => 850,
```

(continues on next page)

(continued from previous page)

```
      product => 'pc'
      },
  "2:white"   => {
      price   => 12,
      product => 'keyboard'
      },
  "3:black"   => {
      price   => 5,
      product => 'mouse'
      }
  }
```

keep_headers

When using hashes, keep the column names into the arrayref passed, so all headers are available after the call in the original order.

```
my $aoh = csv (in => "file.csv", keep_headers => \my @hdr);
```

This attribute can be abbreviated to `kh` or passed as `keep_column_names`.

This attribute implies a default of `auto` for the `headers` attribute.

The headers can also be kept internally to keep stable header order:

```
csv (in      => csv (in => "file.csv", kh => "internal"),
     out     => "new.csv",
     kh      => "internal");
```

where `internal` can also be 1, `yes`, or `true`. This is similar to

```
my @h;
csv (in      => csv (in => "file.csv", kh => \@h),
     out     => "new.csv",
     headers => \@h);
```

fragment

Only output the fragment as defined in the "fragment" method. This option is ignored when *generating* CSV. See "out".

Combining all of them could give something like

```
use Text::CSV_PP qw( csv );
my $aoh = csv (
    in       => "test.txt",
    encoding => "utf-8",
    headers  => "auto",
    sep_char => "|",
    fragment => "row=3;6-9;15-*",
```

(continues on next page)

(continued from previous page)

```
    );
say $aoh->[15]{Foo};
```

sep_set

If `sep_set` is set, the method "header" is invoked on the opened stream to detect and set `sep_char` with the given set.

`sep_set` can be abbreviated to `seps`. If neither `sep_set` not `seps` is given, but `sep` is defined, `sep_set` defaults to `[ sep ]`. This is only supported for perl version 5.10 and up.

Note that as the "header" method is invoked, its default is to also set the headers.

set_column_names

If `set_column_names` is passed, the method "header" is invoked on the opened stream with all arguments meant for "header".

If `set_column_names` is passed as a false value, the content of the first row is only preserved if the output is AoA:

With an input-file like

```
bAr,foo
1,2
3,4,5
```

This call

```
my $aoa = csv (in => $file, set_column_names => 0);
```

will result in

```
[[ "bar", "foo"     ],
 [ "1",   "2"       ],
 [ "3",   "4",  "5" ]]
```

and

```
my $aoa = csv (in => $file, set_column_names => 0, munge => "none");
```

will result in

```
[[ "bAr", "foo"     ],
 [ "1",   "2"       ],
 [ "3",   "4",  "5" ]]
```

csv

The *function* "csv" can also be called as a method or with an existing Text::CSV_PP object. This could help if the function is to be invoked a lot of times and the overhead of creating the object internally over and over again would be prevented by passing an existing instance.

```
my $csv = Text::CSV_PP->new ({ binary => 1, auto_diag => 1 });

my $aoa = $csv->csv (in => $fh);
my $aoa = csv (in => $fh, csv => $csv);
```

both act the same. Running this 20000 times on a 20 lines CSV file, showed a 53% speedup.

Callbacks

Callbacks enable actions triggered from the *inside* of Text::CSV_PP.

While most of what this enables can easily be done in an unrolled loop as described in the "SYNOPSIS" callbacks can be used to meet special demands or enhance the "csv" function.

- error

  ```
  $csv->callbacks (error => sub { $csv->SetDiag (0) });
  ```

  the `error` callback is invoked when an error occurs, but *only* when "auto_diag" is set to a true value. A callback is invoked with the values returned by "error_diag":

  ```
  my ($c, $s);

  sub ignore3006 {
      my ($err, $msg, $pos, $recno, $fldno) = @_;
      if ($err == 3006) {
          # ignore this error
          ($c, $s) = (undef, undef);
          Text::CSV_PP->SetDiag (0);
          }
      # Any other error
      return;
      } # ignore3006

  $csv->callbacks (error => \&ignore3006);
  $csv->bind_columns (\$c, \$s);
  while ($csv->getline ($fh)) {
      # Error 3006 will not stop the loop
      }
  ```

- after_parse

```
$csv->callbacks (after_parse => sub { push @{$_[1]}, "NEW" });
while (my $row = $csv->getline ($fh)) {
    $row->[-1] eq "NEW";
    }
```

This callback is invoked after parsing with "getline" only if no error occurred. The callback is invoked with two arguments: the current CSV parser object and an array reference to the fields parsed.

The return code of the callback is ignored unless it is a reference to the string "skip", in which case the record will be skipped in "getline_all".

```
sub add_from_db {
    my ($csv, $row) = @_;
    $sth->execute ($row->[4]);
    push @$row, $sth->fetchrow_array;
    } # add_from_db

my $aoa = csv (in => "file.csv", callbacks => {
    after_parse => \&add_from_db });
```

This hook can be used for validation:

- FAIL

  Die if any of the records does not validate a rule:

  ```
  after_parse => sub {
      $_[1][4] =~ m/^[0-9]{4}\s?[A-Z]{2}$/ or
          die "5th field does not have a valid Dutch zipcode";
      }
  ```

- DEFAULT

  Replace invalid fields with a default value:

  ```
  after_parse => sub { $_[1][2] =~ m/^\d+$/ or $_[1][2] = 0 }
  ```

- SKIP

  Skip records that have invalid fields (only applies to "getline_all"):

  ```
  after_parse => sub { $_[1][0] =~ m/^\d+$/ or return \"skip";␣
  ↪}
  ```

- before_print

```
my $idx = 1;
$csv->callbacks (before_print => sub { $_[1][0] = $idx++ });
$csv->print (*STDOUT, [ 0, $_ ]) for @members;
```

This callback is invoked before printing with "print" only if no error occurred. The callback is invoked with two arguments: the current CSV parser object and an array reference to the fields passed.

The return code of the callback is ignored.

```
sub max_4_fields {
    my ($csv, $row) = @_;
    @$row > 4 and splice @$row, 4;
    } # max_4_fields

csv (in => csv (in => "file.csv"), out => *STDOUT,
    callbacks => { before_print => \&max_4_fields });
```

This callback is not active for “combine”.

Callbacks for csv ()

```
csv (in        => "file.csv",
     callbacks => {
         filter       => { 6 => sub { $_ > 15 } },    # first
         after_parse  => sub { say "AFTER PARSE";  }, # first
         after_in     => sub { say "AFTER IN";     }, # second
         on_in        => sub { say "ON IN";        }, # third
         },
     );

csv (in        => $aoh,
     out       => "file.csv",
     callbacks => {
         on_in        => sub { say "ON IN";        }, # first
         before_out   => sub { say "BEFORE OUT";   }, # second
         before_print => sub { say "BEFORE PRINT"; }, # third
         },
     );
```

- filter

  This callback can be used to filter records. It is called just after a new record has been scanned. The callback accepts a:

  – hashref

  The keys are the index to the row (the field name or field number, 1-based) and the values are subs to return a true or false value.

  ```
  csv (in => "file.csv", filter => {
             3 => sub { m/a/ },       # third field should␣
  ↪contain an "a"
             5 => sub { length > 4 }, # length of the 5th␣
  ↪field minimal 5
             });

  csv (in => "file.csv", filter => { foo => sub { $_ > 4 }});
  ```

  If the keys to the filter hash contain any character that is not a digit it will also implicitly set “headers” to "auto" unless “headers” was already passed

as argument. When headers are active, returning an array of hashes, the filter is not applicable to the header itself.

All sub results should match, as in AND.

The context of the callback sets $_ localized to the field indicated by the filter. The two arguments are as with all other callbacks, so the other fields in the current row can be seen:

```
filter => { 3 => sub { $_ > 100 ? $_[1][1] =~ m/A/ : $_
↪[1][6] =~ m/B/ }}
```

If the context is set to return a list of hashes ("headers" is defined), the current record will also be available in the localized %_:

```
filter => { 3 => sub { $_ > 100 && $_{foo} =~ m/A/ && $_{bar}
↪ < 1000  }}
```

If the filter is used to *alter* the content by changing $_, make sure that the sub returns true in order not to have that record skipped:

```
filter => { 2 => sub { $_ = uc }}
```

will upper-case the second field, and then skip it if the resulting content evaluates to false. To always accept, end with truth:

```
filter => { 2 => sub { $_ = uc; 1 }}
```

– coderef

```
csv (in => "file.csv", filter => sub { $n++; 0; });
```

If the argument to `filter` is a coderef, it is an alias or shortcut to a filter on column 0:

```
csv (filter => sub { $n++; 0 });
```

is equal to

```
csv (filter => { 0 => sub { $n++; 0 });
```

– filter-name

```
csv (in => "file.csv", filter => "not_blank");
csv (in => "file.csv", filter => "not_empty");
csv (in => "file.csv", filter => "filled");
```

These are predefined filters

Given a file like (line numbers prefixed for doc purpose only):

```
1:1,2,3
2:
3:,
```

(continues on next page)

(continued from previous page)

```
4:""
5:,,
6:, ,
7:"",
8:" "
9:4,5,6
```

* not_blank

  Filter out the blank lines

  This filter is a shortcut for

  ```
  filter => { 0 => sub { @{$_[1]} > 1 or
              defined $_[1][0] && $_[1][0] ne "" } }
  ```

  Due to the implementation, it is currently impossible to also filter lines that consists only of a quoted empty field. These lines are also considered blank lines.

  With the given example, lines 2 and 4 will be skipped.

* not_empty

  Filter out lines where all the fields are empty.

  This filter is a shortcut for

  ```
  filter => { 0 => sub { grep { defined && $_ ne "" } @{$_
  ↪[1]} } }
  ```

  A space is not regarded being empty, so given the example data, lines 2, 3, 4, 5, and 7 are skipped.

* filled

  Filter out lines that have no visible data

  This filter is a shortcut for

  ```
  filter => { 0 => sub { grep { defined && m/\S/ } @{$_[1]}␣
  ↪} }
  ```

  This filter rejects all lines that *not* have at least one field that does not evaluate to the empty string.

  With the given example data, this filter would skip lines 2 through 8.

One could also use modules like Types::Standard:

```
use Types::Standard -types;

my $type   = Tuple[Str, Str, Int, Bool, Optional[Num]];
my $check  = $type->compiled_check;

# filter with compiled check and warnings
```

(continues on next page)

(continued from previous page)

```
my $aoa = csv (
   in     => \$data,
   filter => {
       0 => sub {
           my $ok = $check->($_[1]) or
               warn $type->get_message ($_[1]), "\n";
           return $ok;
           },
       },
   );
```

- after_in

  This callback is invoked for each record after all records have been parsed but before returning the reference to the caller. The hook is invoked with two arguments: the current CSV parser object and a reference to the record. The reference can be a reference to a HASH or a reference to an ARRAY as determined by the arguments.

  This callback can also be passed as an attribute without the `callbacks` wrapper.

- before_out

  This callback is invoked for each record before the record is printed. The hook is invoked with two arguments: the current CSV parser object and a reference to the record. The reference can be a reference to a HASH or a reference to an ARRAY as determined by the arguments.

  This callback can also be passed as an attribute without the `callbacks` wrapper.

  This callback makes the row available in %_ if the row is a hashref. In this case %_ is writable and will change the original row.

- on_in

  This callback acts exactly as the "after_in" or the "before_out" hooks.

  This callback can also be passed as an attribute without the `callbacks` wrapper.

  This callback makes the row available in %_ if the row is a hashref. In this case %_ is writable and will change the original row. So e.g. with

```
my $aoh = csv (
    in      => \"foo\n1\n2\n",
    headers => "auto",
    on_in   => sub { $_{bar} = 2; },
    );
```

  $aoh will be:

```
[ { foo => 1,
    bar => 2,
    }
  { foo => 2,
    bar => 2,
```

(continues on next page)

(continued from previous page)

```
    }
  ]
```

- on_error

  This callback acts exactly as the “error” hook.

```
my @err;
my $aoa = csv (in => $fh, on_error => sub { @err = @_ });
```

  is identical to

```
my $aoa = csv (in => $fh, callbacks => {
    error => sub { @err = @_ },
    });
```

  It can be used for ignoring errors as well as for just keeping the error in case of analysis after the `csv ()` function has returned.

```
my @err;
my $aoa = csv (in => "bad.csv, on_error => sub { @err = @_ });
die Text::CSV_PP->error_diag if @err or !$aoa;
```

DIAGNOSTICS

This section is also taken from Text::CSV_XS.

Still under construction ...

If an error occurs, `$csv->error_diag` can be used to get information on the cause of the failure. Note that for speed reasons the internal value is never cleared on success, so using the value returned by “error_diag” in normal cases - when no error occurred - may cause unexpected results.

If the constructor failed, the cause can be found using “error_diag” as a class method, like `Text::CSV_PP->error_diag`.

The `$csv->error_diag` method is automatically invoked upon error when the contractor was called with `auto_diag` set to 1 or 2, or when *autodie* is in effect. When set to 1, this will cause a `warn` with the error message, when set to 2, it will `die`. `2012 - EOF` is excluded from `auto_diag` reports.

Errors can be (individually) caught using the “error” callback.

The errors as described below are available. I have tried to make the error itself explanatory enough, but more descriptions will be added. For most of these errors, the first three capitals describe the error category:

- INI

  Initialization error or option conflict.

- ECR

  Carriage-Return related parse error.

- EOF

  End-Of-File related parse error.

- EIQ

  Parse error inside quotation.

- EIF

  Parse error inside field.

- ECB

  Combine error.

- EHR

  HashRef parse related error.

And below should be the complete list of error codes that can be returned:

- 1001 “INI - sep_char is equal to quote_char or escape_char”

  The separation character cannot be equal to the quotation character or to the escape character, as this would invalidate all parsing rules.

- 1002 “INI - allow_whitespace with escape_char or quote_char SP or TAB”

  Using the `allow_whitespace` attribute when either `quote_char` or `escape_char` is equal to `SPACE` or `TAB` is too ambiguous to allow.

- 1003 “INI - \r or \n in main attr not allowed”

  Using default `eol` characters in either `sep_char`, `quote_char`, or `escape_char` is not allowed.

- 1004 “INI - callbacks should be undef or a hashref”

  The `callbacks` attribute only allows one to be `undef` or a hash reference.

- 1005 “INI - EOL too long”

  The value passed for EOL is exceeding its maximum length (16).

- 1006 “INI - SEP too long”

  The value passed for SEP is exceeding its maximum length (16).

- 1007 “INI - QUOTE too long”

  The value passed for QUOTE is exceeding its maximum length (16).

- 1008 “INI - SEP undefined”

  The value passed for SEP should be defined and not empty.

- 1010 “INI - the header is empty”

  The header line parsed in the “header” is empty.

- 1011 “INI - the header contains more than one valid separator”

  The header line parsed in the “header” contains more than one (unique) separator character out of the allowed set of separators.

- 1012 “INI - the header contains an empty field”

  The header line parsed in the “header” contains an empty field.

- 1013 “INI - the header contains nun-unique fields”

  The header line parsed in the “header” contains at least two identical fields.

- 1014 “INI - header called on undefined stream”

  The header line cannot be parsed from an undefined source.

- 1500 “PRM - Invalid/unsupported argument(s)”

  Function or method called with invalid argument(s) or parameter(s).

- 1501 “PRM - The key attribute is passed as an unsupported type”

  The `key` attribute is of an unsupported type.

- 1502 “PRM - The value attribute is passed without the key attribute”

  The `value` attribute is only allowed when a valid key is given.

- 1503 “PRM - The value attribute is passed as an unsupported type”

  The `value` attribute is of an unsupported type.

- 2010 “ECR - QUO char inside quotes followed by CR not part of EOL”

  When `eol` has been set to anything but the default, like `"\r\t\n"`, and the `"\r"` is following the **second** (closing) `quote_char`, where the characters following the `"\r"` do not make up the `eol` sequence, this is an error.

- 2011 “ECR - Characters after end of quoted field”

  Sequences like `1,foo,"bar"baz,22,1` are not allowed. `"bar"` is a quoted field and after the closing double-quote, there should be either a new-line sequence or a separation character.

- 2012 “EOF - End of data in parsing input stream”

  Self-explaining. End-of-file while inside parsing a stream. Can happen only when reading from streams with “getline”, as using “parse” is done on strings that are not required to have a trailing `eol`.

- 2013 “INI - Specification error for fragments RFC7111”

  Invalid specification for URI “fragment” specification.

- 2014 “ENF - Inconsistent number of fields”

  Inconsistent number of fields under strict parsing.

- 2015 “ERW - Empty row”

  An empty row was not allowed.

- 2016 “EOL - Inconsistent EOL”

  Inconsistent End-Of-Line detected under strict_eol parsing.

- 2021 “EIQ - NL char inside quotes, binary off”

  Sequences like `1,"foo\nbar",22,1` are allowed only when the binary option has been selected with the constructor.

- 2022 “EIQ - CR char inside quotes, binary off”

  Sequences like `1,"foo\rbar",22,1` are allowed only when the binary option has been selected with the constructor.

- 2023 “EIQ - QUO character not allowed”

  Sequences like `"foo "bar" baz",qu` and `2023,",2008-04-05,"Foo, Bar",\n` will cause this error.

- 2024 “EIQ - EOF cannot be escaped, not even inside quotes”

  The escape character is not allowed as last character in an input stream.

- 2025 “EIQ - Loose unescaped escape”

  An escape character should escape only characters that need escaping.

  Allowing the escape for other characters is possible with the attribute “allow_loose_escapes”.

- 2026 “EIQ - Binary character inside quoted field, binary off”

  Binary characters are not allowed by default. Exceptions are fields that contain valid UTF-8, that will automatically be upgraded if the content is valid UTF-8. Set `binary` to 1 to accept binary data.

- 2027 “EIQ - Quoted field not terminated”

  When parsing a field that started with a quotation character, the field is expected to be closed with a quotation character. When the parsed line is exhausted before the quote is found, that field is not terminated.

- 2030 “EIF - NL char inside unquoted verbatim, binary off”
- 2031 “EIF - CR char is first char of field, not part of EOL”
- 2032 “EIF - CR char inside unquoted, not part of EOL”
- 2034 “EIF - Loose unescaped quote”
- 2035 “EIF - Escaped EOF in unquoted field”
- 2036 “EIF - ESC error”
- 2037 “EIF - Binary character in unquoted field, binary off”
- 2110 “ECB - Binary character in Combine, binary off”
- 2200 “EIO - print to IO failed. See errno”
- 3001 “EHR - Unsupported syntax for column_names ()”
- 3002 “EHR - getline_hr () called before column_names ()”
- 3003 “EHR - bind_columns () and column_names () fields count mismatch”
- 3004 “EHR - bind_columns () only accepts refs to scalars”
- 3006 “EHR - bind_columns () did not pass enough refs for parsed fields”
- 3007 “EHR - bind_columns needs refs to writable scalars”
- 3008 “EHR - unexpected error in bound fields”
- 3009 “EHR - print_hr () called before column_names ()”

- 3010 “EHR - print_hr () called with invalid arguments”

SEE ALSO

Text::CSV_XS, *Text::CSV*

Older versions took many regexp from http://www.din.or.jp/~ohzaki/perl.htm

AUTHOR

Kenichi Ishigaki, <ishigaki[at]cpan.org> Makamaka Hannyaharamitu, <makamaka[at]cpan.org>

Text::CSV_XS was written by <joe[at]ispsoft.de> and maintained by <h.m.brand[at]xs4all.nl>.

Text::CSV was written by <alan[at]mfgrtl.com>.

COPYRIGHT AND LICENSE

Copyright 2017- by Kenichi Ishigaki, <ishigaki[at]cpan.org> Copyright 2005-2015 by Makamaka Hannyaharamitu, <makamaka[at]cpan.org>

Most of the code and doc is directly taken from the pure perl part of Text::CSV_XS.

Copyright (C) 2007-2016 H.Merijn Brand. All rights reserved. Copyright (C) 1998-2001 Jochen Wiedmann. All rights reserved. Copyright (C) 1997 Alan Citterman. All rights reserved.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

Text::LevenshteinXS

The Levenshtein edit distance of two strings: the fewest insertions, deletions and substitutions that turn one into the other.

`distance($s, $t)` is exported by default. It compares bytes: a character string counts the bytes of its UTF-8 form, and a string ends at its first NUL byte, as the C strings of upstream’s XS do.

This is a native implementation of Text::LevenshteinXS 0.03’s `LevenshteinXS.xs`; its `Text/LevenshteinXS.pm` is used unchanged.

Synopsis

```
use Text::LevenshteinXS;

print distance("foo", "four"), "\n";      # 2
print distance("kitten", "sitting"), "\n";  # 3
```

Differences from upstream

One, where perl 5.44 with LevenshteinXS.xs fails (t/81-xs-native/Text-LevenshteinXS/150-or-better.t): long strings. LevenshteinXS.xs allocates a table of (n+1)(m+1) ints for strings of n and m bytes, does not check the allocation, and indexes the table with an int; when the memory is not there (two strings of 20000 bytes need 1.6 GB), or once n*m passes 2**31, it dumps core. Here the distance is computed in two rows of the shorter string's length.

Functions

Edit distance

distance

`distance($s, $t)`: the Levenshtein distance of the two strings' bytes.

Text::LineFold

Line Folding for Plain Text

SYNOPSIS

```
use Text::LineFold;
$lf = Text::LineFold->new();

# Fold lines
$folded = $lf->fold($string, 'PLAIN');
$indented = $lf->fold(' ' x 8, ' ' x 4, $string);

# Unfold lines
$unfolded = $lf->unfold($string, 'FIXED');
```

DESCRIPTION

Text::LineFold folds or unfolds lines of plain text. As it mainly focuses on plain text e-mail messages, RFC 3676 flowed format is also supported.

Public Interface

- new ([KEY => VALUE, ...])

  Constructor. About KEY => VALUE pairs see config method.

- $self->config (KEY)

- $self->config ([KEY => VAL, ...])

  Instance method. Get or update configuration. Following KEY => VALUE pairs may be specified.

  – Charset => CHARSET

  Character set that is used to encode string. It may be string or *MIME::Charset* object. Default is `"UTF-8"`.

  - Language => LANGUAGE

    Along with Charset option, this may be used to define language/region context. Default is `"XX"`. See also *"Context" in Unicode::LineBreak* option.

  - Newline => STRING

    String to be used for newline sequence. Default is `"\n"`.

  - OutputCharset => CHARSET

    Character set that is used to encode result of fold()/unfold(). It may be string or *MIME::Charset* object. If a special value `"_UNICODE_"` is specified, result will be Unicode string. Default is the value of Charset option.

  - TabSize => NUMBER

    Column width of tab stops. When 0 is specified, tab stops are ignored. Default is 8.

  - BreakIndent
  - CharMax
  - ColMax
  - ColMin
  - ComplexBreaking
  - EAWidth
  - HangulAsAL
  - LBClass
  - LegacyCM
  - Prep
  - Urgent

    See *"Options" in Unicode::LineBreak*.

- $self->fold (STRING, [METHOD])
- $self->fold (INITIAL_TAB, SUBSEQUENT_TAB, STRING, ...)

  Instance method. fold() folds lines of string STRING and returns it. Surplus SPACEs and horizontal tabs at end of line are removed, newline sequences are replaced by that specified by Newline option and newline is appended at end of text if it does not exist. Horizontal tabs are treated as tab stops according to TabSize option.

  By the first style, following options may be specified for METHOD argument.

  - `"FIXED"`

    Lines preceded by `">"` won't be folded. Paragraphs are separated by empty line.

  - `"FLOWED"`

    `"Format=Flowed; DelSp=Yes"` formatting defined by RFC 3676.

- `"PLAIN"`

  Default method. All lines are folded.

Second style is similar to *"wrap()" in Text::Wrap*. All lines are folded. INITIAL_TAB is inserted at beginning of paragraphs and SUBSEQUENT_TAB at beginning of other broken lines.

- $self->unfold (STRING, METHOD)

  Conjunct folded paragraphs of string STRING and returns it.

  Following options may be specified for METHOD argument.

  - `"FIXED"`

    Default method. Lines preceded by `">"` won't be conjuncted. Treat empty line as paragraph separator.

  - `"FLOWED"`

    Unfold `"Format=Flowed; DelSp=Yes"` formatting defined by RFC 3676.

  - `"FLOWEDSP"`

    Unfold `"Format=Flowed; DelSp=No"` formatting defined by RFC 3676.

BUGS

Please report bugs or buggy behaviors to developer.

CPAN Request Tracker: http://rt.cpan.org/Public/Dist/Display.html?Name=Unicode-LineBreak.

VERSION

Consult $VERSION variable.

SEE ALSO

Unicode::LineBreak, *Text::Wrap*.

AUTHOR

Copyright (C) 2009-2012 Hatuka*nezumi - IKEDA Soji <hatuka(at)nezumi.nu>.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

Text::ParseWords

parse text into an array of tokens or array of arrays

SYNOPSIS

```
use Text::ParseWords;
@lists = nested_quotewords($delim, $keep, @lines);
@words = quotewords($delim, $keep, @lines);
@words = shellwords(@lines);
@words = parse_line($delim, $keep, $line);
@words = old_shellwords(@lines); # DEPRECATED!
```

DESCRIPTION

The `nested_quotewords()` and `quotewords()` functions accept a delimiter (which can be a regular expression) and a list of lines and then breaks those lines up into a list of words ignoring delimiters that appear inside quotes. `quotewords()` returns all of the tokens in a single long list, while `nested_quotewords()` returns a list of token lists corresponding to the elements of `@lines`. `parse_line()` does tokenizing on a single string. The `*quotewords()` functions simply call `parse_line()`, so if you're only splitting one line you can call `parse_line()` directly and save a function call.

The `$keep` controls what happens with delimters and special characters:

- true

  If true, then the tokens are split on the specified delimiter, but all other characters (including quotes and backslashes) are kept in the tokens.

- false

  If $keep is false then the `*quotewords()` functions remove all quotes and backslashes that are not themselves backslash-escaped or inside of single quotes (i.e., `quotewords()` tries to interpret these characters just like the Bourne shell). NB: these semantics are significantly different from the original version of this module shipped with Perl 5.000 through 5.004.

- `"delimiters"`

  As an additional feature, $keep may be the keyword "delimiters" which causes the functions to preserve the delimiters in each string as tokens in the token lists, in addition to preserving quote and backslash characters.

`shellwords()` is written as a special case of `quotewords()`, and it does token parsing with whitespace as a delimiter– similar to most Unix shells.

EXAMPLES

The sample program:

```
use Text::ParseWords;
@words = quotewords('\s+', 0, q{this   is "a test" of\ quotewords \
 ↪"for you});
$i = 0;
foreach (@words) {
    print "$i: <$_>\n";
```

(continues on next page)

(continued from previous page)

```
    $i++;
}
```

produces:

```
0: <this>
1: <is>
2: <a test>
3: <of quotewords>
4: <"for>
5: <you>
```

demonstrating:

- 0

  a simple word

- 1

  multiple spaces are skipped because of our $delim

- 2

  use of quotes to include a space in a word

- 3

  use of a backslash to include a space in a word

- 4

  use of a backslash to remove the special meaning of a double-quote

- 5

  another simple word (note the lack of effect of the backslashed double-quote)

Replacing `quotewords('\s+', 0, q{this is...})` with `shellwords(q{this is...})` is a simpler way to accomplish the same thing.

SEE ALSO

Text::CSV - for parsing CSV files

AUTHORS

The original author is unknown, but presumably this evolved from `shellwords.pl` in Perl 4.

Much of the code for `parse_line()` (including the primary regexp) came from Joerk Behrends jbehrends@multimediaproduzenten.de.

Examples section and other documentation provided by John Heidemann <johnh@ISI.EDU>.

Hal Pomeranz pomeranz@netcom.com maintained this from 1994 through 1999, and did the first CPAN release.

Alexandr Ciornii <alexchornyATgmail.com> maintained this from 2008 to 2015.

Many other people have contributed, with special thanks due to Michael Schwern schwern@envirolink.org and Jeff Friedl jfriedl@yahoo-inc.com.

COPYRIGHT AND LICENSE

This library is free software; you may redistribute and/or modify it under the same terms as Perl itself.

Text::Tabs

expand and unexpand tabs like unix expand(1) and unexpand(1)

SYNOPSIS

```
use Text::Tabs;

$tabstop = 4;  # default = 8
@lines_without_tabs = expand(@lines_with_tabs);
@lines_with_tabs = unexpand(@lines_without_tabs);
```

DESCRIPTION

Text::Tabs does most of what the unix utilities expand(1) and unexpand(1) do. Given a line with tabs in it, expand replaces those tabs with the appropriate number of spaces. Given a line with or without tabs in it, unexpand adds tabs when it can save bytes by doing so, like the unexpand -a command.

Unlike the old unix utilities, this module correctly accounts for any Unicode combining characters (such as diacriticals) that may occur in each line for both expansion and unexpansion. These are overstrike characters that do not increment the logical position. Make sure you have the appropriate Unicode settings enabled.

EXPORTS

The following are exported:

- expand
- unexpand
- $tabstop

  The `$tabstop` variable controls how many column positions apart each tabstop is. The default is 8.

  Please note that `local($tabstop)` doesn't do the right thing and if you want to use `local` to override `$tabstop`, you need to use `local($Text::Tabs::tabstop)`.

EXAMPLE

```
#!perl
# unexpand -a
use Text::Tabs;

while (<>) {
  print unexpand $_;
}
```

Instead of the shell's expand command, use:

```
perl -MText::Tabs -n -e 'print expand $_'
```

Instead of the shell's unexpand -a command, use:

```
perl -MText::Tabs -n -e 'print unexpand $_'
```

BUGS

Text::Tabs handles only tabs (`"\t"`) and combining characters (`/\pM/`). It doesn't count backwards for backspaces (`"\t"`), omit other non-printing control characters (`/\pC/`), or otherwise deal with any other zero-, half-, and full-width characters.

LICENSE

Copyright (C) 1996-2002,2005,2006 David Muir Sharnoff.
Copyright (C) 2005 Aristotle Pagaltzis Copyright (C) 2012-2013 Google, Inc. This module may be modified, used, copied, and redistributed at your own risk. Although allowed by the preceding license, please do not publicly redistribute modified versions of this code with the name "Text::Tabs" unless it passes the unmodified Text::Tabs test suite.

Text::Wrap

line wrapping to form simple paragraphs

SYNOPSIS

Example 1

```
use Text::Wrap;

$initial_tab = "\t";    # Tab before first line
$subsequent_tab = "";   # All other lines flush left

print wrap($initial_tab, $subsequent_tab, @text);
print fill($initial_tab, $subsequent_tab, @text);

$lines = wrap($initial_tab, $subsequent_tab, @text);

@paragraphs = fill($initial_tab, $subsequent_tab, @text);
```

Example 2

```
use Text::Wrap qw(wrap $columns $huge);

$columns = 132;         # Wrap at 132 characters
$huge = 'die';
$huge = 'wrap';
$huge = 'overflow';
```

Example 3

```
use Text::Wrap;

$Text::Wrap::columns = 72;
print wrap('', '', @text);
```

DESCRIPTION

`Text::Wrap::wrap()` is a very simple paragraph formatter. It formats a single paragraph at a time by breaking lines at word boundaries. Indentation is controlled for the first line (`$initial_tab`) and all subsequent lines (`$subsequent_tab`) independently. Please note: `$initial_tab` and `$subsequent_tab` are the literal strings that will be used: it is unlikely you would want to pass in a number.

`Text::Wrap::fill()` is a simple multi-paragraph formatter. It formats each paragraph separately and then joins them together when it's done. It will destroy any whitespace in the original text. It breaks text into paragraphs by looking for whitespace after a newline. In other respects, it acts like wrap().

`wrap()` compresses trailing whitespace into one newline, and `fill()` deletes all trailing whitespace.

Both `wrap()` and `fill()` return a single string.

Unlike the old Unix fmt(1) utility, this module correctly accounts for any Unicode combining characters (such as diacriticals) that may occur in each line for both expansion and unexpansion. These are overstrike characters that do not increment the logical position. Make sure you have the appropriate Unicode settings enabled.

OVERRIDES

`Text::Wrap::wrap()` has a number of variables that control its behavior. Because other modules might be using `Text::Wrap::wrap()` it is suggested that you leave these variables alone! If you can't do that, then use `local($Text::Wrap::VARIABLE) = YOURVALUE` when you change the values so that the original value is restored. This `local()` trick will not work if you import the variable into your own namespace.

Lines are wrapped at `$Text::Wrap::columns` columns (default value: 76). `$Text::Wrap::columns` should be set to the full width of your output device. In fact, every resulting line will have length of no more than `$columns - 1`.

It is possible to control which characters terminate words by modifying `$Text::Wrap::break`. Set this to a string such as `'[\s:]'` (to break before spaces or colons) or a pre-compiled regexp such as `qr/[\s']/` (to break before spaces or

apostrophes). The default is simply `'\s'`; that is, words are terminated by spaces. (This means, among other things, that trailing punctuation such as full stops or commas stay with the word they are "attached" to.) Setting `$Text::Wrap::break` to a regular expression that doesn't eat any characters (perhaps just a forward look-ahead assertion) will cause warnings.

Beginner note: In example 2, above `$columns` is imported into the local namespace, and set locally. In example 3, `$Text::Wrap::columns` is set in its own namespace without importing it.

`Text::Wrap::wrap()` starts its work by expanding all the tabs in its input into spaces. The last thing it does it to turn spaces back into tabs. If you do not want tabs in your results, set `$Text::Wrap::unexpand` to a false value. Likewise if you do not want to use 8-character tabstops, set `$Text::Wrap::tabstop` to the number of characters you do want for your tabstops.

If you want to separate your lines with something other than `\n` then set `$Text::Wrap::separator` to your preference. This replaces all newlines with `$Text::Wrap::separator`. If you just want to preserve existing newlines but add new breaks with something else, set `$Text::Wrap::separator2` instead.

When words that are longer than `$columns` are encountered, they are broken up. `wrap()` adds a `"\n"` at column `$columns`. This behavior can be overridden by setting `$huge` to 'die' or to 'overflow'. When set to 'die', large words will cause `die()` to be called. When set to 'overflow', large words will be left intact.

Historical notes: 'die' used to be the default value of `$huge`. Now, 'wrap' is the default value.

EXAMPLES

Code:

```
print wrap("\t","",<<END);
This is a bit of text that forms
a normal book-style indented paragraph
END
```

Result:

```
"     This is a bit of text that forms
a normal book-style indented paragraph
"
```

Code:

```
$Text::Wrap::columns=20;
$Text::Wrap::separator="|";
print wrap("","","This is a bit of text that forms a normal book-
 ↪style paragraph");
```

Result:

```
"This is a bit of|text that forms a|normal book-style|paragraph"
```

SEE ALSO

For correct handling of East Asian half- and full-width characters, see Text::WrapI18N. For more detailed controls: Text::Format.

AUTHOR

David Muir Sharnoff cpan@dave.sharnoff.org with help from Tim Pierce and many many others.

LICENSE

Copyright (C) 1996-2009 David Muir Sharnoff.
Copyright (C) 2012-2013 Google, Inc. This module may be modified, used, copied, and redistributed at your own risk. Although allowed by the preceding license, please do not publicly redistribute modified versions of this code with the name “Text::Wrap” unless it passes the unmodified Text::Wrap test suite.

5.2.104 Thread

Manipulate threads in Perl (for old code only)

DEPRECATED

The `Thread` module served as the frontend to the old-style thread model, called *5005threads*, that was introduced in release 5.005. That model was deprecated, and has been removed in version 5.10.

For old code and interim backwards compatibility, the `Thread` module has been reworked to function as a frontend for the new interpreter threads (*ithreads*) model. However, some previous functionality is not available. Further, the data sharing models between the two thread models are completely different, and anything to do with data sharing has to be thought differently. With *ithreads*, you must explicitly `share()` variables between the threads.

You are strongly encouraged to migrate any existing threaded code to the new model (i.e., use the `threads` and `threads::shared` modules) as soon as possible.

HISTORY

In Perl 5.005, the thread model was that all data is implicitly shared, and shared access to data has to be explicitly synchronized. This model is called *5005threads*.

In Perl 5.6, a new model was introduced in which all data is thread local and shared access to data has to be explicitly declared. This model is called *ithreads*, for “interpreter threads”.

In Perl 5.6, the *ithreads* model was not available as a public API; only as an internal API that was available for extension writers, and to implement fork() emulation on Win32 platforms.

In Perl 5.8, the *ithreads* model became available through the `threads` module, and the *5005threads* model was deprecated.

In Perl 5.10, the *5005threads* model was removed from the Perl interpreter.

SYNOPSIS

```
use Thread qw(:DEFAULT async yield);

my $t = Thread->new(\&start_sub, @start_args);

$result = $t->join;
$t->detach;

if ($t->done) {
    $t->join;
}

if($t->equal($another_thread)) {
    # ...
}

yield();

my $tid = Thread->self->tid;

lock($scalar);
lock(@array);
lock(%hash);

my @list = Thread->list;
```

DESCRIPTION

The `Thread` module provides multithreading support for Perl.

FUNCTIONS

- $thread = Thread->new(\&start_sub)
- $thread = Thread->new(\&start_sub, LIST)

  `new` starts a new thread of execution in the referenced subroutine. The optional list is passed as parameters to the subroutine. Execution continues in both the subroutine and the code after the `new` call.

  `Thread->new` returns a thread object representing the newly created thread.

- lock VARIABLE

  `lock` places a lock on a variable until the lock goes out of scope.

  If the variable is locked by another thread, the `lock` call will block until it's available. `lock` is recursive, so multiple calls to `lock` are safe–the variable will remain

locked until the outermost lock on the variable goes out of scope.

Locks on variables only affect `lock` calls–they do *not* affect normal access to a variable. (Locks on subs are different, and covered in a bit.) If you really, *really* want locks to block access, then go ahead and tie them to something and manage this yourself. This is done on purpose. While managing access to variables is a good thing, Perl doesn't force you out of its living room...

If a container object, such as a hash or array, is locked, all the elements of that container are not locked. For example, if a thread does a `lock @a`, any other thread doing a `lock($a[12])` won't block.

Finally, `lock` will traverse up references exactly *one* level. `lock(\$a)` is equivalent to `lock($a)`, while `lock(\\$a)` is not.

- async BLOCK;

  `async` creates a thread to execute the block immediately following it. This block is treated as an anonymous sub, and so must have a semi-colon after the closing brace. Like `Thread->new`, `async` returns a thread object.

- Thread->self

  The `Thread->self` function returns a thread object that represents the thread making the `Thread->self` call.

- Thread->list

  Returns a list of all non-joined, non-detached Thread objects.

- cond_wait VARIABLE

  The `cond_wait` function takes a **locked** variable as a parameter, unlocks the variable, and blocks until another thread does a `cond_signal` or `cond_broadcast` for that same locked variable. The variable that `cond_wait` blocked on is relocked after the `cond_wait` is satisfied. If there are multiple threads `cond_waiting` on the same variable, all but one will reblock waiting to re-acquire the lock on the variable. (So if you're only using `cond_wait` for synchronization, give up the lock as soon as possible.)

- cond_signal VARIABLE

  The `cond_signal` function takes a locked variable as a parameter and unblocks one thread that's `cond_waiting` on that variable. If more than one thread is blocked in a `cond_wait` on that variable, only one (and which one is indeterminate) will be unblocked.

  If there are no threads blocked in a `cond_wait` on the variable, the signal is discarded.

- cond_broadcast VARIABLE

  The `cond_broadcast` function works similarly to `cond_signal`. `cond_broadcast`, though, will unblock **all** the threads that are blocked in a `cond_wait` on the locked variable, rather than only one.

- yield

  The `yield` function allows another thread to take control of the CPU. The exact results are implementation-dependent.

METHODS

- join

  `join` waits for a thread to end and returns any values the thread exited with. `join` will block until the thread has ended, though it won't block if the thread has already terminated.

  If the thread being `joined` `died`, the error it died with will be returned at this time. If you don't want the thread performing the `join` to die as well, you should either wrap the `join` in an `eval` or use the `eval` thread method instead of `join`.

- detach

  `detach` tells a thread that it is never going to be joined i.e. that all traces of its existence can be removed once it stops running. Errors in detached threads will not be visible anywhere - if you want to catch them, you should use $SIG{__DIE__} or something like that.

- equal

  `equal` tests whether two thread objects represent the same thread and returns true if they do.

- tid

  The `tid` method returns the tid of a thread. The tid is a monotonically increasing integer assigned when a thread is created. The main thread of a program will have a tid of zero, while subsequent threads will have tids assigned starting with one.

- done

  The done method returns true if the thread you're checking has finished, and false otherwise.

DEFUNCT

The following were implemented with *5005threads*, but are no longer available with *ithreads*.

- lock(\&sub)

  With 5005threads, you could also `lock` a sub such that any calls to that sub from another thread would block until the lock was released.

  Also, subroutines could be declared with the `:locked` attribute which would serialize access to the subroutine, but allowed different threads non-simultaneous access.

- eval

  The `eval` method wrapped an `eval` around a `join`, and so waited for a thread to exit, passing along any values the thread might have returned and placing any errors into `$@`.

- flags

  The `flags` method returned the flags for the thread - an integer value corresponding to the internal flags for the thread.

SEE ALSO

threads, *threads::shared*, *Thread::Queue*, *Thread::Semaphore*

Modules

- `Thread::Queue` - Thread-safe queues
- `Thread::Semaphore` - Thread-safe semaphores

Thread::Queue

Thread-safe queues

VERSION

This document describes Thread::Queue version 3.14

SYNOPSIS

```
use strict;
use warnings;

use threads;
use Thread::Queue;

my $q = Thread::Queue->new();    # A new empty queue

# Worker thread
my $thr = threads->create(
    sub {
        # Thread will loop until no more work
        while (defined(my $item = $q->dequeue())) {
            # Do work on $item
            ...
        }
    }
);

# Send work to the thread
$q->enqueue($item1, ...);
# Signal that there is no more work to be sent
$q->end();
# Join up with the thread when it finishes
$thr->join();

...

# Count of items in the queue
my $left = $q->pending();
```

(continues on next page)

(continued from previous page)

```
# Non-blocking dequeue
if (defined(my $item = $q->dequeue_nb())) {
    # Work on $item
}

# Blocking dequeue with 5-second timeout
if (defined(my $item = $q->dequeue_timed(5))) {
    # Work on $item
}

# Set a size for a queue
$q->limit = 5;

# Get the second item in the queue without dequeuing anything
my $item = $q->peek(1);

# Insert two items into the queue just behind the head
$q->insert(1, $item1, $item2);

# Extract the last two items on the queue
my ($item1, $item2) = $q->extract(-2, 2);
```

DESCRIPTION

This module provides thread-safe FIFO queues that can be accessed safely by any number of threads.

Any data types supported by *threads::shared* can be passed via queues:

- Ordinary scalars
- Array refs
- Hash refs
- Scalar refs
- Objects based on the above

Ordinary scalars are added to queues as they are.

If not already thread-shared, the other complex data types will be cloned (recursively, if needed, and including any `bless`ings and read-only settings) into thread-shared structures before being placed onto a queue.

For example, the following would cause *Thread::Queue* to create a empty, shared array reference via `&shared([])`, copy the elements 'foo', 'bar' and 'baz' from `@ary` into it, and then place that shared reference onto the queue:

```
my @ary = qw/foo bar baz/;
$q->enqueue(\@ary);
```

However, for the following, the items are already shared, so their references are added directly to the queue, and no cloning takes place:

```
my @ary :shared = qw/foo bar baz/;
$q->enqueue(\@ary);

my $obj = &shared({});
$$obj{'foo'} = 'bar';
$$obj{'qux'} = 99;
bless($obj, 'My::Class');
$q->enqueue($obj);
```

See “LIMITATIONS” for caveats related to passing objects via queues.

QUEUE CREATION

- ->new()

  Creates a new empty queue.

- ->new(LIST)

  Creates a new queue pre-populated with the provided list of items.

BASIC METHODS

The following methods deal with queues on a FIFO basis.

- ->enqueue(LIST)

  Adds a list of items onto the end of the queue.

- ->dequeue()

- ->dequeue(COUNT)

  Removes the requested number of items (default is 1) from the head of the queue, and returns them. If the queue contains fewer than the requested number of items, then the thread will be blocked until the requisite number of items are available (i.e., until other threads enqueue more items).

- ->dequeue_nb()

- ->dequeue_nb(COUNT)

  Removes the requested number of items (default is 1) from the head of the queue, and returns them. If the queue contains fewer than the requested number of items, then it immediately (i.e., non-blocking) returns whatever items there are on the queue. If the queue is empty, then `undef` is returned.

- ->dequeue_timed(TIMEOUT)

- ->dequeue_timed(TIMEOUT, COUNT)

  Removes the requested number of items (default is 1) from the head of the queue, and returns them. If the queue contains fewer than the requested number of items, then the thread will be blocked until the requisite number of items are available,

or until the timeout is reached. If the timeout is reached, it returns whatever items there are on the queue, or undef if the queue is empty.

The timeout may be a number of seconds relative to the current time (e.g., 5 seconds from when the call is made), or may be an absolute timeout in *epoch* seconds the same as would be used with *cond_timedwait()*. Fractional seconds (e.g., 2.5 seconds) are also supported (to the extent of the underlying implementation).

If TIMEOUT is missing, undef, or less than or equal to 0, then this call behaves the same as dequeue_nb.

- ->pending()

  Returns the number of items still in the queue. Returns undef if the queue has been ended (see below), and there are no more items in the queue.

- ->limit

  Sets the size of the queue. If set, calls to enqueue() will block until the number of pending items in the queue drops below the limit. The limit does not prevent enqueuing items beyond that count:

```
my $q = Thread::Queue->new(1, 2);
$q->limit = 4;
$q->enqueue(3, 4, 5);   # Does not block
$q->enqueue(6);         # Blocks until at least 2 items are
                        # dequeued
my $size = $q->limit;   # Returns the current limit (may return
                        # 'undef')
$q->limit = 0;          # Queue size is now unlimited
```

  Calling any of the dequeue methods with COUNT greater than a queue's limit will generate an error.

- ->end()

  Declares that no more items will be added to the queue.

  All threads blocking on dequeue() calls will be unblocked with any remaining items in the queue and/or undef being returned. Any subsequent calls to dequeue() will behave like dequeue_nb().

  Once ended, no more items may be placed in the queue.

ADVANCED METHODS

The following methods can be used to manipulate items anywhere in a queue.

To prevent the contents of a queue from being modified by another thread while it is being examined and/or changed, *lock* the queue inside a local block:

```
{
    lock($q);   # Keep other threads from changing the queue's␣
↪contents
    my $item = $q->peek();
    if ($item ...) {
```

(continues on next page)

(continued from previous page)

```
        ...
    }
}
# Queue is now unlocked
```

- ->peek()
- ->peek(INDEX)

  Returns an item from the queue without dequeuing anything. Defaults to the head of queue (at index position 0) if no index is specified. Negative index values are supported as with *arrays* (i.e., -1 is the end of the queue, -2 is next to last, and so on).

  If no items exists at the specified index (i.e., the queue is empty, or the index is beyond the number of items on the queue), then `undef` is returned.

  Remember, the returned item is not removed from the queue, so manipulating a peeked at reference affects the item on the queue.

- ->insert(INDEX, LIST)

  Adds the list of items to the queue at the specified index position (0 is the head of the list). Any existing items at and beyond that position are pushed back past the newly added items:

  ```
  $q->enqueue(1, 2, 3, 4);
  $q->insert(1, qw/foo bar/);
  # Queue now contains:  1, foo, bar, 2, 3, 4
  ```

  Specifying an index position greater than the number of items in the queue just adds the list to the end.

  Negative index positions are supported:

  ```
  $q->enqueue(1, 2, 3, 4);
  $q->insert(-2, qw/foo bar/);
  # Queue now contains:  1, 2, foo, bar, 3, 4
  ```

  Specifying a negative index position greater than the number of items in the queue adds the list to the head of the queue.

- ->extract()
- ->extract(INDEX)
- ->extract(INDEX, COUNT)

  Removes and returns the specified number of items (defaults to 1) from the specified index position in the queue (0 is the head of the queue). When called with no arguments, `extract` operates the same as dequeue_nb.

  This method is non-blocking, and will return only as many items as are available to fulfill the request:

```
$q->enqueue(1, 2, 3, 4);
my $item  = $q->extract(2)     # Returns 3
                               # Queue now contains:  1, 2, 4
my @items = $q->extract(1, 3)  # Returns (2, 4)
                               # Queue now contains:  1
```

Specifying an index position greater than the number of items in the queue results in undef or an empty list being returned.

```
$q->enqueue('foo');
my $nada = $q->extract(3)      # Returns undef
my @nada = $q->extract(1, 3)   # Returns ()
```

Negative index positions are supported. Specifying a negative index position greater than the number of items in the queue may return items from the head of the queue (similar to dequeue_nb) if the count overlaps the head of the queue from the specified position (i.e. if queue size + index + count is greater than zero):

```
$q->enqueue(qw/foo bar baz/);
my @nada = $q->extract(-6, 2);  # Returns ()      - (3+(-6)+2)
 ↪<= 0
my @some = $q->extract(-6, 4);  # Returns (foo)   - (3+(-6)+4) >
 ↪ 0
                                # Queue now contains:  bar, baz
my @rest = $q->extract(-3, 4);  # Returns (bar, baz) -
                                #                  (2+(-3)+4) >
 ↪ 0
```

NOTES

Queues created by *Thread::Queue* can be used in both threaded and non-threaded applications.

LIMITATIONS

Passing objects on queues may not work if the objects' classes do not support sharing. See *"BUGS AND LIMITATIONS" in threads::shared* for more.

Passing array/hash refs that contain objects may not work for Perl prior to 5.10.0.

SEE ALSO

Thread::Queue on MetaCPAN: https://metacpan.org/release/Thread-Queue

Code repository for CPAN distribution: https://github.com/Dual-Life/Thread-Queue

threads, *threads::shared*

Sample code in the *examples* directory of this distribution on CPAN.

MAINTAINER

Jerry D. Hedden, <jdhedden AT cpan DOT org>

LICENSE

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

Thread::Semaphore

Thread-safe semaphores

VERSION

This document describes Thread::Semaphore version 2.13

SYNOPSIS

```
use Thread::Semaphore;
my $s = Thread::Semaphore->new();
$s->down();   # Also known as the semaphore P operation.
# The guarded section is here
$s->up();     # Also known as the semaphore V operation.

# Decrement the semaphore only if it would immediately succeed.
if ($s->down_nb()) {
    # The guarded section is here
    $s->up();
}

# Forcefully decrement the semaphore even if its count goes below 0.
$s->down_force();

# The default value for semaphore operations is 1
my $s = Thread::Semaphore->new($initial_value);
$s->down($down_value);
$s->up($up_value);
if ($s->down_nb($down_value)) {
    ...
    $s->up($up_value);
}
$s->down_force($down_value);
```

DESCRIPTION

Semaphores provide a mechanism to regulate access to resources. Unlike locks, semaphores aren't tied to particular scalars, and so may be used to control access to anything you care to use them for.

Semaphores don't limit their values to zero and one, so they can be used to control access to some resource that there may be more than one of (e.g., filehandles). Increment and decrement amounts aren't fixed at one either, so threads can reserve or return multiple resources at once.

METHODS

- ->new()
- ->new(NUMBER)

  new creates a new semaphore, and initializes its count to the specified number (which must be an integer). If no number is specified, the semaphore's count defaults to 1.

- ->down()
- ->down(NUMBER)

  The down method decreases the semaphore's count by the specified number (which must be an integer >= 1), or by one if no number is specified.

  If the semaphore's count would drop below zero, this method will block until such time as the semaphore's count is greater than or equal to the amount you're downing the semaphore's count by.

  This is the semaphore "P operation" (the name derives from the Dutch word "pak", which means "capture" – the semaphore operations were named by the late Dijkstra, who was Dutch).

- ->down_nb()
- ->down_nb(NUMBER)

  The down_nb method attempts to decrease the semaphore's count by the specified number (which must be an integer >= 1), or by one if no number is specified.

  If the semaphore's count would drop below zero, this method will return *false*, and the semaphore's count remains unchanged. Otherwise, the semaphore's count is decremented and this method returns *true*.

- ->down_force()
- ->down_force(NUMBER)

  The down_force method decreases the semaphore's count by the specified number (which must be an integer >= 1), or by one if no number is specified. This method does not block, and may cause the semaphore's count to drop below zero.

- ->down_timed(TIMEOUT)
- ->down_timed(TIMEOUT, NUMBER)

  The down_timed method attempts to decrease the semaphore's count by 1 or by the specified number within the specified timeout period given in seconds (which must be an integer >= 0).

  If the semaphore's count would drop below zero, this method will block until either the semaphore's count is greater than or equal to the amount you're downing the semaphore's count by, or until the timeout is reached.

If the timeout is reached, this method will return *false*, and the semaphore's count remains unchanged. Otherwise, the semaphore's count is decremented and this method returns *true*.

- ->up()
- ->up(NUMBER)

  The up method increases the semaphore's count by the number specified (which must be an integer >= 1), or by one if no number is specified.

  This will unblock any thread that is blocked trying to down the semaphore if the up raises the semaphore's count above the amount that the down is trying to decrement it by. For example, if three threads are blocked trying to down a semaphore by one, and another thread ups the semaphore by two, then two of the blocked threads (which two is indeterminate) will become unblocked.

  This is the semaphore "V operation" (the name derives from the Dutch word "vrij", which means "release").

NOTES

Semaphores created by *Thread::Semaphore* can be used in both threaded and non-threaded applications. This allows you to write modules and packages that potentially make use of semaphores, and that will function in either environment.

SEE ALSO

Thread::Semaphore on MetaCPAN: https://metacpan.org/release/Thread-Semaphore

Code repository for CPAN distribution: https://github.com/Dual-Life/Thread-Semaphore

threads, *threads::shared*

Sample code in the *examples* directory of this distribution on CPAN.

MAINTAINER

Jerry D. Hedden, <jdhedden AT cpan DOT org>

LICENSE

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

5.2.105 Tie

Namespace containing the following modules:

- `Tie::Array` - base class for tied arrays
- `Tie::File` - Access the lines of a disk file via a Perl array
- `Tie::Handle` - base class definitions for tied handles
- `Tie::Hash` - base class definitions for tied hashes

- `Tie::Memoize` - add data to hash when needed
- `Tie::RefHash` - Use references as hash keys
- `Tie::Scalar` - base class definitions for tied scalars
- `Tie::StdHandle` - base class definitions for tied handles
- `Tie::SubstrHash` - Fixed-table-size, fixed-key-length hashing

Tie::Array

base class for tied arrays

SYNOPSIS

```
package Tie::NewArray;
use Tie::Array;
@ISA = ('Tie::Array');

# mandatory methods
sub TIEARRAY { ... }
sub FETCH { ... }
sub FETCHSIZE { ... }

sub STORE { ... }       # mandatory if elements writeable
sub STORESIZE { ... }   # mandatory if elements can be added/deleted
sub EXISTS { ... }      # mandatory if exists() expected to work
sub DELETE { ... }      # mandatory if delete() expected to work

# optional methods - for efficiency
sub CLEAR { ... }
sub PUSH { ... }
sub POP { ... }
sub SHIFT { ... }
sub UNSHIFT { ... }
sub SPLICE { ... }
sub EXTEND { ... }
sub DESTROY { ... }

package Tie::NewStdArray;
use Tie::Array;

@ISA = ('Tie::StdArray');

# all methods provided by default

package main;

$object = tie @somearray,'Tie::NewArray';
$object = tie @somearray,'Tie::StdArray';
$object = tie @somearray,'Tie::NewStdArray';
```

DESCRIPTION

This module provides methods for array-tying classes. See perltie for a list of the functions required in order to tie an array to a package. The basic **Tie::Array** package provides stub `DESTROY`, and `EXTEND` methods that do nothing, stub `DELETE` and `EXISTS` methods that croak() if the delete() or exists() builtins are ever called on the tied array, and implementations of `PUSH`, `POP`, `SHIFT`, `UNSHIFT`, `SPLICE` and `CLEAR` in terms of basic `FETCH`, `STORE`, `FETCHSIZE`, `STORESIZE`.

The **Tie::StdArray** package provides efficient methods required for tied arrays which are implemented as blessed references to an "inner" perl array. It inherits from **Tie::Array**, and should cause tied arrays to behave exactly like standard arrays, allowing for selective overloading of methods.

For developers wishing to write their own tied arrays, the required methods are briefly defined below. See the perltie section for more detailed descriptive, as well as example code:

- TIEARRAY classname, LIST

  The class method is invoked by the command `tie @array, classname`. Associates an array instance with the specified class. `LIST` would represent additional arguments (along the lines of *AnyDBM_File* and compatriots) needed to complete the association. The method should return an object of a class which provides the methods below.

- STORE this, index, value

  Store datum *value* into *index* for the tied array associated with object *this*. If this makes the array larger then class's mapping of `undef` should be returned for new positions.

- FETCH this, index

  Retrieve the datum in *index* for the tied array associated with object *this*.

- FETCHSIZE this

  Returns the total number of items in the tied array associated with object *this*. (Equivalent to `scalar(@array)`).

- STORESIZE this, count

  Sets the total number of items in the tied array associated with object *this* to be *count*. If this makes the array larger then class's mapping of `undef` should be returned for new positions. If the array becomes smaller then entries beyond count should be deleted.

- EXTEND this, count

  Informative call that array is likely to grow to have *count* entries. Can be used to optimize allocation. This method need do nothing.

- EXISTS this, key

  Verify that the element at index *key* exists in the tied array *this*.

  The **Tie::Array** implementation is a stub that simply croaks.

- DELETE this, key

Delete the element at index *key* from the tied array *this*.

The **Tie::Array** implementation is a stub that simply croaks.

- CLEAR this

  Clear (remove, delete, ...) all values from the tied array associated with object *this*.

- DESTROY this

  Normal object destructor method.

- PUSH this, LIST

  Append elements of LIST to the array.

- POP this

  Remove last element of the array and return it.

- SHIFT this

  Remove the first element of the array (shifting other elements down) and return it.

- UNSHIFT this, LIST

  Insert LIST elements at the beginning of the array, moving existing elements up to make room.

- SPLICE this, offset, length, LIST

  Perform the equivalent of `splice` on the array.

  offset is optional and defaults to zero, negative values count back from the end of the array.

  length is optional and defaults to rest of the array.

  LIST may be empty.

  Returns a list of the original *length* elements at *offset*.

CAVEATS

There is no support at present for tied @ISA. There is a potential conflict between magic entries needed to notice setting of @ISA, and those needed to implement 'tie'.

AUTHOR

Nick Ing-Simmons nik@tiuk.ti.com

Tie::File

Access the lines of a disk file via a Perl array

SYNOPSIS

```
use Tie::File;

tie @array, 'Tie::File', filename or die ...;

$array[0] = 'blah';      # first line of the file is now 'blah'
                           # (line numbering starts at 0)
print $array[42];        # display line 43 of the file

$n_recs = @array;        # how many records are in the file?
$#array -= 2;            # chop two records off the end


for (@array) {
  s/PERL/Perl/g;        # Replace PERL with Perl everywhere in the
 ↪file
}

# These are just like regular push, pop, unshift, shift, and splice
# Except that they modify the file in the way you would expect

push @array, new recs...;
my $r1 = pop @array;
unshift @array, new recs...;
my $r2 = shift @array;
@old_recs = splice @array, 3, 7, new recs...;

untie @array;            # all finished
```

DESCRIPTION

`Tie::File` represents a regular text file as a Perl array. Each element in the array corresponds to a record in the file. The first line of the file is element 0 of the array; the second line is element 1, and so on.

The file is *not* loaded into memory, so this will work even for gigantic files.

Changes to the array are reflected in the file immediately.

Lazy people and beginners may now stop reading the manual.

unicode

You can read a unicode (UTF-8) file by providing a file handle opened with the desired encoding. It is not safe to write to one because the length in bytes and in characters is often different, Tie::File will miscalculate the length of writes, overwriting parts of other records.

recsep

What is a 'record'? By default, the meaning is the same as for the `<...>` operator: It's a string terminated by `$/`, which is probably `"\n"`. (Minor exception: on DOS and Win32 systems, a 'record' is a string terminated by `"\r\n"`.) You may change the definition of "record" by supplying the `recsep` option in the `tie` call:

```
tie @array, 'Tie::File', $file, recsep => 'es';
```

This says that records are delimited by the string `es`. If the file contained the following data:

```
Curse these pesky flies!\n
```

then the `@array` would appear to have four elements:

```
"Curse th"
"e p"
"ky fli"
"!\n"
```

An undefined value is not permitted as a record separator. Perl's special "paragraph mode" semantics (à la `$/ = ""`) are not emulated.

Records read from the tied array do not have the record separator string on the end; this is to allow

```
$array[17] .= "extra";
```

to work as expected.

(See "autochomp", below.) Records stored into the array will have the record separator string appended before they are written to the file, if they don't have one already. For example, if the record separator string is `"\n"`, then the following two lines do exactly the same thing:

```
$array[17] = "Cherry pie";
$array[17] = "Cherry pie\n";
```

The result is that the contents of line 17 of the file will be replaced with "Cherry pie"; a newline character will separate line 17 from line 18. This means that this code will do nothing:

```
chomp $array[17];
```

Because the `chomp`ed value will have the separator reattached when it is written back to the file. There is no way to create a file whose trailing record separator string is missing.

Inserting records that *contain* the record separator string is not supported by this module. It will probably produce a reasonable result, but what this result will be may change in a future version. Use 'splice' to insert records or to replace one record with several.

autochomp

Normally, array elements have the record separator removed, so that if the file contains the text

```
Gold
Frankincense
Myrrh
```

the tied array will appear to contain `("Gold", "Frankincense", "Myrrh")`. If you set `autochomp` to a false value, the record separator will not be removed. If the file above was tied with

```
tie @gifts, "Tie::File", $gifts, autochomp => 0;
```

then the array `@gifts` would appear to contain `("Gold\n", "Frankincense\n", "Myrrh\n")`, or (on Win32 systems) `("Gold\r\n", "Frankincense\r\n", "Myrrh\r\n")`.

mode

Normally, the specified file will be opened for read and write access, and will be created if it does not exist. (That is, the flags `O_RDWR | O_CREAT` are supplied in the open call.) If you want to change this, you may supply alternative flags in the `mode` option. See *Fcntl* for a listing of available flags. For example:

```
# open the file if it exists, but fail if it does not exist
use Fcntl 'O_RDWR';
tie @array, 'Tie::File', $file, mode => O_RDWR;

# create the file if it does not exist
use Fcntl 'O_RDWR', 'O_CREAT';
tie @array, 'Tie::File', $file, mode => O_RDWR | O_CREAT;

# open an existing file in read-only mode
use Fcntl 'O_RDONLY';
tie @array, 'Tie::File', $file, mode => O_RDONLY;
```

Opening the data file in write-only or append mode is not supported.

memory

This is an upper limit on the amount of memory that `Tie::File` will consume at any time while managing the file. This is used for two things: managing the *read cache* and managing the *deferred write buffer*.

Records read in from the file are cached, to avoid having to re-read them repeatedly. If you read the same record twice, the first time it will be stored in memory, and the second time it will be fetched from the *read cache*. The amount of data in the read cache will not exceed the value you specified for `memory`. If `Tie::File` wants to cache a new record, but the read cache is full, it will make room by expiring the least-recently visited records from the read cache.

The default memory limit is 2Mib. You can adjust the maximum read cache size by supplying the `memory` option. The argument is the desired cache size, in bytes.

```
# I have a lot of memory, so use a large cache to speed up access
tie @array, 'Tie::File', $file, memory => 20_000_000;
```

Setting the memory limit to 0 will inhibit caching; records will be fetched from disk every time you examine them.

The `memory` value is not an absolute or exact limit on the memory used. `Tie::File` objects contains some structures besides the read cache and the deferred write buffer, whose sizes are not charged against `memory`.

The cache itself consumes about 310 bytes per cached record, so if your file has many short records, you may want to decrease the cache memory limit, or else the cache overhead may exceed the size of the cached data.

dw_size

(This is an advanced feature. Skip this section on first reading.)

If you use deferred writing (See "Deferred Writing", below) then data you write into the array will not be written directly to the file; instead, it will be saved in the *deferred write buffer* to be written out later. Data in the deferred write buffer is also charged against the memory limit you set with the `memory` option.

You may set the `dw_size` option to limit the amount of data that can be saved in the deferred write buffer. This limit may not exceed the total memory limit. For example, if you set `dw_size` to 1000 and `memory` to 2500, that means that no more than 1000 bytes of deferred writes will be saved up. The space available for the read cache will vary, but it will always be at least 1500 bytes (if the deferred write buffer is full) and it could grow as large as 2500 bytes (if the deferred write buffer is empty.)

If you don't specify a `dw_size`, it defaults to the entire memory limit.

Option Format

`-mode` is a synonym for `mode`. `-recsep` is a synonym for `recsep`. `-memory` is a synonym for `memory`. You get the idea.

Public Methods

The `tie` call returns an object, say `$o`. You may call

```
$rec = $o->FETCH($n);
$o->STORE($n, $rec);
```

to fetch or store the record at line `$n`, respectively; similarly the other tied array methods. (See perltie for details.) You may also call the following methods on this object:

flock

```
$o->flock(MODE)
```

will lock the tied file. `MODE` has the same meaning as the second argument to the Perl built-in `flock` function; for example `LOCK_SH` or `LOCK_EX | LOCK_NB`. (These constants are provided by the `use Fcntl ':flock'` declaration.)

`MODE` is optional; the default is `LOCK_EX`.

`Tie::File` maintains an internal table of the byte offset of each record it has seen in the file.

When you use `flock` to lock the file, `Tie::File` assumes that the read cache is no longer trustworthy, because another process might have modified the file since the last time it was read. Therefore, a successful call to `flock` discards the contents of the read cache and the internal record offset table.

`Tie::File` promises that the following sequence of operations will be safe:

```
my $o = tie @array, "Tie::File", $filename;
$o->flock;
```

In particular, `Tie::File` will *not* read or write the file during the `tie` call. (Exception: Using `mode => O_TRUNC` will, of course, erase the file during the `tie` call. If you want to do this safely, then open the file without `O_TRUNC`, lock the file, and use `@array = ()`.)

The best way to unlock a file is to discard the object and untie the array. It is probably unsafe to unlock the file without also untying it, because if you do, changes may remain unwritten inside the object. That is why there is no shortcut for unlocking. If you really want to unlock the file prematurely, you know what to do; if you don't know what to do, then don't do it.

All the usual warnings about file locking apply here. In particular, note that file locking in Perl is **advisory**, which means that holding a lock will not prevent anyone else from reading, writing, or erasing the file; it only prevents them from getting another lock at the same time. Locks are analogous to green traffic lights: If you have a green light, that does not prevent the idiot coming the other way from plowing into you sideways; it merely guarantees to you that the idiot does not also have a green light at the same time.

autochomp

```
my $old_value = $o->autochomp(0);    # disable autochomp option
my $old_value = $o->autochomp(1);    #  enable autochomp option

my $ac = $o->autochomp();   # recover current value
```

See "autochomp", above.

defer, flush, discard, and autodefer

See “Deferred Writing”, below.

offset

```
$off = $o->offset($n);
```

This method returns the byte offset of the start of the $nth record in the file. If there is no such record, it returns an undefined value.

Tying to an already-opened filehandle

If $fh is a filehandle, such as is returned by IO::File or one of the other IO modules, you may use:

```
tie @array, 'Tie::File', $fh, ...;
```

Similarly if you opened that handle FH with regular open or sysopen, you may use:

```
tie @array, 'Tie::File', \*FH, ...;
```

Handles that were opened write-only won’t work. Handles that were opened read-only will work as long as you don’t try to modify the array. Handles must be attached to seekable sources of data—that means no pipes or sockets. If Tie::File can detect that you supplied a non-seekable handle, the tie call will throw an exception. (On Unix systems, it can detect this.)

Note that Tie::File will only close any filehandles that it opened internally. If you passed it a filehandle as above, you “own” the filehandle, and are responsible for closing it after you have untied the @array.

Tie::File calls binmode on filehandles that it opens internally, but not on filehandles passed in by the user. For consistency, especially if using the tied files cross-platform, you may wish to call binmode on the filehandle prior to tying the file.

Deferred Writing

(This is an advanced feature. Skip this section on first reading.)

Normally, modifying a Tie::File array writes to the underlying file immediately. Every assignment like $a[3] = ... rewrites as much of the file as is necessary; typically, everything from line 3 through the end will need to be rewritten. This is the simplest and most transparent behavior. Performance even for large files is reasonably good.

However, under some circumstances, this behavior may be excessively slow. For example, suppose you have a million-record file, and you want to do:

```
for (@FILE) {
  $_ = "> $_";
}
```

The first time through the loop, you will rewrite the entire file, from line 0 through the end. The second time through the loop, you will rewrite the entire file from line 1 through the end. The third time through the loop, you will rewrite the entire file from line 2 to the end. And so on.

If the performance in such cases is unacceptable, you may defer the actual writing, and then have it done all at once. The following loop will perform much better for large files:

```
(tied @a)->defer;
for (@a) {
  $_ = "> $_";
}
(tied @a)->flush;
```

If `Tie::File`'s memory limit is large enough, all the writing will done in memory. Then, when you call `->flush`, the entire file will be rewritten in a single pass.

(Actually, the preceding discussion is something of a fib. You don't need to enable deferred writing to get good performance for this common case, because `Tie::File` will do it for you automatically unless you specifically tell it not to. See "Autodeferring", below.)

Calling `->flush` returns the array to immediate-write mode. If you wish to discard the deferred writes, you may call `->discard` instead of `->flush`. Note that in some cases, some of the data will have been written already, and it will be too late for `->discard` to discard all the changes. Support for `->discard` may be withdrawn in a future version of `Tie::File`.

Deferred writes are cached in memory up to the limit specified by the `dw_size` option (see above). If the deferred-write buffer is full and you try to write still more deferred data, the buffer will be flushed. All buffered data will be written immediately, the buffer will be emptied, and the now-empty space will be used for future deferred writes.

If the deferred-write buffer isn't yet full, but the total size of the buffer and the read cache would exceed the `memory` limit, the oldest records will be expired from the read cache until the total size is under the limit.

`push`, `pop`, `shift`, `unshift`, and `splice` cannot be deferred. When you perform one of these operations, any deferred data is written to the file and the operation is performed immediately. This may change in a future version.

If you resize the array with deferred writing enabled, the file will be resized immediately, but deferred records will not be written. This has a surprising consequence: `@a = (...)` erases the file immediately, but the writing of the actual data is deferred. This might be a bug. If it is a bug, it will be fixed in a future version.

Autodeferring

`Tie::File` tries to guess when deferred writing might be helpful, and to turn it on and off automatically.

```
for (@a) {
  $_ = "> $_";
}
```

In this example, only the first two assignments will be done immediately; after this, all the changes to the file will be deferred up to the user-specified memory limit.

You should usually be able to ignore this and just use the module without thinking about deferring. However, special applications may require fine control over which writes are deferred, or may require that all writes be immediate. To disable the autodeferment feature, use

```
(tied @o)->autodefer(0);
```

or

```
tie @array, 'Tie::File', $file, autodefer => 0;
```

Similarly, `->autodefer(1)` re-enables autodeferment, and `->autodefer()` recovers the current value of the autodefer setting.

CONCURRENT ACCESS TO FILES

Caching and deferred writing are inappropriate if you want the same file to be accessed simultaneously from more than one process. Other optimizations performed internally by this module are also incompatible with concurrent access. A future version of this module will support a `concurrent => 1` option that enables safe concurrent access.

Previous versions of this documentation suggested using `memory => 0` for safe concurrent access. This was mistaken. Tie::File will not support safe concurrent access before version 0.96.

CAVEATS

(That's Latin for 'warnings'.)

- Reasonable effort was made to make this module efficient. Nevertheless, changing the size of a record in the middle of a large file will always be fairly slow, because everything after the new record must be moved.
- The behavior of tied arrays is not precisely the same as for regular arrays. For example:

  ```
  # This DOES print "How unusual!"
  undef $a[10];  print "How unusual!\n" if defined $a[10];
  ```

  `undef`-ing a `Tie::File` array element just blanks out the corresponding record in the file. When you read it back again, you'll get the empty string, so the supposedly-`undef`'ed value will be defined. Similarly, if you have `autochomp` disabled, then

  ```
  # This DOES print "How unusual!" if 'autochomp' is disabled
  undef $a[10];
  print "How unusual!\n" if $a[10];
  ```

  Because when `autochomp` is disabled, `$a[10]` will read back as `"\n"` (or whatever the record separator string is.)

There are other minor differences, particularly regarding `exists` and `delete`, but in general, the correspondence is extremely close.

- I have supposed that since this module is concerned with file I/O, almost all normal use of it will be heavily I/O bound. This means that the time to maintain complicated data structures inside the module will be dominated by the time to actually perform the I/O. When there was an opportunity to spend CPU time to avoid doing I/O, I usually tried to take it.
- You might be tempted to think that deferred writing is like transactions, with `flush` as `commit` and `discard` as `rollback`, but it isn't, so don't.
- There is a large memory overhead for each record offset and for each cache entry: about 310 bytes per cached data record, and about 21 bytes per offset table entry.

  The per-record overhead will limit the maximum number of records you can access per file. Note that *accessing* the length of the array via `$x = scalar @tied_file` accesses **all** records and stores their offsets. The same for `foreach (@tied_file)`, even if you exit the loop early.

SUBCLASSING

This version promises absolutely nothing about the internals, which may change without notice. A future version of the module will have a well-defined and stable subclassing API.

WHAT ABOUT DB_File?

People sometimes point out that *DB_File* will do something similar, and ask why `Tie::File` module is necessary.

There are a number of reasons that you might prefer `Tie::File`. A list is available at `http://perl.plover.com/TieFile/why-not-DB_File`.

AUTHOR

Mark Jason Dominus

To contact the author, send email to: `mjd-perl-tiefile+@plover.com`

To receive an announcement whenever a new version of this module is released, send a blank email message to `mjd-perl-tiefile-subscribe@plover.com`.

The most recent version of this module, including documentation and any news of importance, will be available at

```
http://perl.plover.com/TieFile/
```

LICENSE

`Tie::File` version 0.96 is copyright (C) 2003 Mark Jason Dominus.

This library is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

These terms are your choice of any of (1) the Perl Artistic Licence, or (2) version 2 of the GNU General Public License as published by the Free Software Foundation, or (3) any later version of the GNU General Public License.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this library program; it should be in the file `COPYING`. If not, see <https://www.gnu.org/licenses/>.

For licensing inquiries, contact the author at:

```
Mark Jason Dominus
255 S. Warnock St.
Philadelphia, PA 19107
```

WARRANTY

`Tie::File` version 0.98 comes with ABSOLUTELY NO WARRANTY. For details, see the license.

THANKS

Gigantic thanks to Jarkko Hietaniemi, for agreeing to put this in the core when I hadn't written it yet, and for generally being helpful, supportive, and competent. (Usually the rule is "choose any one.") Also big thanks to Abhijit Menon-Sen for all of the same things.

Special thanks to Craig Berry and Peter Prymmer (for VMS portability help), Randy Kobes (for Win32 portability help), Clinton Pierce and Autrijus Tang (for heroic eleventh-hour Win32 testing above and beyond the call of duty), Michael G Schwern (for testing advice), and the rest of the CPAN testers (for testing generally).

Special thanks to Tels for suggesting several speed and memory optimizations.

Additional thanks to: Edward Avis / Mattia Barbon / Tom Christiansen / Gerrit Haase / Gurusamy Sarathy / Jarkko Hietaniemi (again) / Nikola Knezevic / John Kominetz / Nick Ing-Simmons / Tassilo von Parseval / H. Dieter Pearcey / Slaven Rezic / Eric Roode / Peter Scott / Peter Somu / Autrijus Tang (again) / Tels (again) / Juerd Waalboer / Todd Rinaldo

TODO

More tests. (Stuff I didn't think of yet.)

Paragraph mode?

Fixed-length mode. Leave-blanks mode.

Maybe an autolocking mode?

For many common uses of the module, the read cache is a liability. For example, a program that inserts a single record, or that scans the file once, will have a cache hit rate of zero. This suggests a major optimization: The cache should be initially disabled. Here's a hybrid approach: Initially, the cache is disabled, but the cache code maintains statistics about how high the hit rate would be *if* it were enabled. When it sees the hit rate

get high enough, it enables itself. The STAT comments in this code are the beginning of an implementation of this.

Record locking with fcntl()? Then the module might support an undo log and get real transactions. What a tour de force that would be.

Keeping track of the highest cached record. This would allow reads-in-a-row to skip the cache lookup faster (if reading from 1..N with empty cache at start, the last cached value will be always N-1).

More tests.

Tie::Handle

base class definitions for tied handles

SYNOPSIS

```
package NewHandle;
require Tie::Handle;

@ISA = qw(Tie::Handle);

sub READ { ... }            # Provide a needed method
sub TIEHANDLE { ... }       # Overrides inherited method


package main;

tie *FH, 'NewHandle';
```

DESCRIPTION

This module provides some skeletal methods for handle-tying classes. See perltie for a list of the functions required in tying a handle to a package. The basic **Tie::Handle** package provides a new method, as well as methods `TIEHANDLE`, `PRINT`, `PRINTF` and `GETC`.

For developers wishing to write their own tied-handle classes, the methods are summarized below. The perltie section not only documents these, but has sample code as well:

- TIEHANDLE classname, LIST

  The method invoked by the command `tie *glob, classname`. Associates a new glob instance with the specified class. `LIST` would represent additional arguments (along the lines of *AnyDBM_File* and compatriots) needed to complete the association.

- WRITE this, scalar, length, offset

  Write *length* bytes of data from *scalar* starting at *offset*.

- PRINT this, LIST

Print the values in *LIST*

- PRINTF this, format, LIST

  Print the values in *LIST* using *format*

- READ this, scalar, length, offset

  Read *length* bytes of data into *scalar* starting at *offset*.

- READLINE this

  Read a single line

- GETC this

  Get a single character

- CLOSE this

  Close the handle

- OPEN this, filename

  (Re-)open the handle

- BINMODE this

  Specify content is binary

- EOF this

  Test for end of file.

- TELL this

  Return position in the file.

- SEEK this, offset, whence

  Position the file.

  Test for end of file.

- DESTROY this

  Free the storage associated with the tied handle referenced by *this*. This is rarely needed, as Perl manages its memory quite well. But the option exists, should a class wish to perform specific actions upon the destruction of an instance.

MORE INFORMATION

The perltie section contains an example of tying handles.

COMPATIBILITY

This version of Tie::Handle is neither related to nor compatible with the Tie::Handle (3.0) module available on CPAN. It was due to an accident that two modules with the same name appeared. The namespace clash has been cleared in favor of this module that comes with the perl core in September 2000 and accordingly the version number has been bumped up to 4.0.

Tie::Hash

base class definitions for tied hashes

SYNOPSIS

```
package NewHash;
require Tie::Hash;

@ISA = qw(Tie::Hash);

sub DELETE { ... }          # Provides needed method
sub CLEAR { ... }           # Overrides inherited method


package NewStdHash;
require Tie::Hash;

@ISA = qw(Tie::StdHash);

# All methods provided by default, define
# only those needing overrides
# Accessors access the storage in %{$_[0]};
# TIEHASH should return a reference to the actual storage
sub DELETE { ... }

package NewExtraHash;
require Tie::Hash;

@ISA = qw(Tie::ExtraHash);

# All methods provided by default, define
# only those needing overrides
# Accessors access the storage in %{$_[0][0]};
# TIEHASH should return an array reference with the first element
# being the reference to the actual storage
sub DELETE {
  $_[0][1]->('del', $_[0][0], $_[1]); # Call the report writer
  delete $_[0][0]->{$_[1]};           #  $_[0]->SUPER::DELETE($_[1])
}


package main;

tie %new_hash, 'NewHash';
tie %new_std_hash, 'NewStdHash';
tie %new_extra_hash, 'NewExtraHash',
    sub {warn "Doing \U$_[1]\E of $_[2].\n"};
```

DESCRIPTION

This module provides some skeletal methods for hash-tying classes. See perltie for a list of the functions required in order to tie a hash to a package. The basic **Tie::Hash** package provides a new method, as well as methods TIEHASH, EXISTS and CLEAR. The **Tie::StdHash** and **Tie::ExtraHash** packages provide most methods for hashes described in perltie (the exceptions are UNTIE and DESTROY). They cause tied hashes to behave exactly like standard hashes, and allow for selective overwriting of methods. **Tie::Hash** has legacy support for the new method: it is used if TIEHASH is not defined in the case a class forgets to include a TIEHASH method.

For developers wishing to write their own tied hashes, the required methods are briefly defined below. See the perltie section for more detailed descriptive, as well as example code:

- TIEHASH classname, LIST

  The method invoked by the command `tie %hash, classname`. Associates a new hash instance with the specified class. LIST would represent additional arguments (along the lines of *AnyDBM_File* and compatriots) needed to complete the association.

- STORE this, key, value

  Store datum *value* into *key* for the tied hash *this*.

- FETCH this, key

  Retrieve the datum in *key* for the tied hash *this*.

- FIRSTKEY this

  Return the first key in the hash.

- NEXTKEY this, lastkey

  Return the next key in the hash.

- EXISTS this, key

  Verify that *key* exists with the tied hash *this*.

  The **Tie::Hash** implementation is a stub that simply croaks.

- DELETE this, key

  Delete the key *key* from the tied hash *this*.

- CLEAR this

  Clear all values from the tied hash *this*.

- SCALAR this

  Returns what evaluating the hash in scalar context yields.

  Tie::Hash does not implement this method (but **Tie::StdHash** and **Tie::ExtraHash** do).

Inheriting from Tie::StdHash

The accessor methods assume that the actual storage for the data in the tied hash is in the hash referenced by `tied(%tiedhash)`. Thus overwritten `TIEHASH` method should return a hash reference, and the remaining methods should operate on the hash referenced by the first argument:

```
package ReportHash;
our @ISA = 'Tie::StdHash';

sub TIEHASH  {
  my $storage = bless {}, shift;
  warn "New ReportHash created, stored in $storage.\n";
  $storage
}
sub STORE    {
  warn "Storing data with key $_[1] at $_[0].\n";
  $_[0]{$_[1]} = $_[2]
}
```

Inheriting from Tie::ExtraHash

The accessor methods assume that the actual storage for the data in the tied hash is in the hash referenced by `(tied(%tiedhash))->[0]`. Thus overwritten `TIEHASH` method should return an array reference with the first element being a hash reference, and the remaining methods should operate on the hash `%{ $_[0]->[0] }`:

```
package ReportHash;
our @ISA = 'Tie::ExtraHash';

sub TIEHASH  {
  my $class = shift;
  my $storage = bless [{}, @_], $class;
  warn "New ReportHash created, stored in $storage.\n";
  $storage;
}
sub STORE    {
  warn "Storing data with key $_[1] at $_[0].\n";
  $_[0][0]{$_[1]} = $_[2]
}
```

The default `TIEHASH` method stores "extra" arguments to tie() starting from offset 1 in the array referenced by `tied(%tiedhash)`; this is the same storage algorithm as in TIEHASH subroutine above. Hence, a typical package inheriting from **Tie::ExtraHash** does not need to overwrite this method.

SCALAR, UNTIE and DESTROY

The methods `UNTIE` and `DESTROY` are not defined in **Tie::Hash**, **Tie::StdHash**, or **Tie::ExtraHash**. Tied hashes do not require presence of these methods, but if defined, the methods will be called in proper time, see perltie.

SCALAR is only defined in **Tie::StdHash** and **Tie::ExtraHash**.

If needed, these methods should be defined by the package inheriting from **Tie::Hash**, **Tie::StdHash**, or **Tie::ExtraHash**. See "SCALAR" in perltie to find out what happens when SCALAR does not exist.

MORE INFORMATION

The packages relating to various DBM-related implementations (DB_File, NDBM_File, etc.) show examples of general tied hashes, as does the *Config* module. While these do not utilize **Tie::Hash**, they serve as good working examples.

Modules

- Tie::Hash::Indexed - a hash that keeps its keys in insertion order
- Tie::Hash::NamedCapture - Named regexp capture buffers

Tie::Hash::Indexed

Tie::Hash::Indexed - a hash that keeps its keys in insertion order.

```
use Tie::Hash::Indexed;

tie my %h, 'Tie::Hash::Indexed';
%h = (one => 1, two => 2, three => 3);
print join(' ', keys %h), "\n";           # one two three

my $h = Tie::Hash::Indexed->new(a => 1, b => 2);
$h->push(c => 3);
$h->unshift(z => 0);
print join(' ', $h->keys), "\n";          # z a b c
my ($key, $value) = $h->pop;              # c 3
```

The module works as a tied hash and as an object. The object interface is the faster of the two: it has get, set, exists, delete and clear; keys, values and items (all of them, or those of the keys given); push, unshift, pop and shift for the two ends; merge and assign; and the in-place operators preinc, postinc, predec, postdec, add, subtract, multiply, divide, modulo, concat, or_assign and dor_assign.

Storing to a key that exists keeps its position; push and unshift move it to that end. Deleting a key takes the same time whatever the size of the hash.

iterator and reverse_iterator return an iterator object with next, prev, valid, key and value. An iterator dies with "invalid iterator access" once the hash was changed after it was made.

Objects can be frozen, thawed and cloned with Storable.

This is a native implementation of the CPAN module Tie::Hash::Indexed 0.08; its Indexed.pm is used unchanged.

Differences from upstream

Three, each where perl 5.44 with the XS module reads memory it has freed or a null pointer:

- pop or `shift` of the entry an `each` loop stands on: the loop continues with the next entry, as it does after `delete`.
- `clear`, `assign` or `%h = ()` in the middle of an `each` loop called through `NEXTKEY` directly: the loop ends.
- `NEXTKEY` on an object `FIRSTKEY` was never called on returns the first key.

Modules

- `Tie::Hash::Indexed::Iterator`

Functions

Other Functions

new

`TIEHASH`, `new`: a new hash holding the key/value pairs given.

get

`FETCH`, `get`: the value of a key, undef if there is none.

set

`STORE`, `set`: store a value. A new key goes to the end, an existing one keeps its place. `set` returns the value.

exists

`EXISTS`, `exists`, `has`: whether the hash has the key.

Also available as `Tie::Hash::Indexed::has`.

delete

`DELETE`, `delete`: remove a key and return its value, undef if there was none.

clear

`CLEAR`, `clear`: remove everything. `clear` returns the object.

items

`items`, `as_list`, `keys`, `values`: all entries in order, or those of the keys given (undef for a value that is not there). In scalar context the number of elements the list would have.

Also available as `Tie::Hash::Indexed::as_list`, `Tie::Hash::Indexed::keys`, `Tie::Hash::Indexed::values`.

merge

`merge`, `assign`, `push`, `unshift`: store key/value pairs. `merge` leaves an existing key in place; `assign` clears the hash first; `push` and `unshift` put every pair at that end, moving a key that exists. Returns the number of keys.

Also available as `Tie::Hash::Indexed::assign`, `Tie::Hash::Indexed::push`, `Tie::Hash::Indexed::unshift`.

pop

`pop`, `shift`: remove the last or the first entry. Returns its key and value in list context, its value otherwise, nothing when the hash is empty.

Also available as `Tie::Hash::Indexed::shift`.

iterator

`iterator`, `reverse_iterator`: an iterator standing on the first, or the last, entry.

Also available as `Tie::Hash::Indexed::reverse_iterator`.

preinc

`preinc`, `predec`, `postinc`, `postdec`: `++` and `--` on the value of a key, which starts at 0 if the key is new.

Also available as `Tie::Hash::Indexed::predec`, `Tie::Hash::Indexed::postinc`, `Tie::Hash::Indexed::postdec`.

add

`add`, `subtract`, `multiply`, `divide`, `modulo`, `concat`, `dor_assign`, `or_assign` (and `dor_equals`, `or_equals`): the assignment operator `+=`, `-=`, `*=`, `/=`, `%=`, `.=`, `//=`, `||=` on the value of a key, which starts undefined if the key is new. Returns the new value.

Also available as `Tie::Hash::Indexed::subtract`, `Tie::Hash::Indexed::multiply`, `Tie::Hash::Indexed::divide`, `Tie::Hash::Indexed::modulo`, `Tie::Hash::Indexed::concat`, `Tie::Hash::Indexed::dor_assign`, `Tie::Hash::Indexed::dor_equals`, `Tie::Hash::Indexed::or_assign`, `Tie::Hash::Indexed::or_equals`.

STORABLE_freeze

STORABLE_freeze: Storable's hook. The frozen form is a six-byte header and a reference to every key and value, in order.

STORABLE_thaw

STORABLE_thaw: Storable's hook, the inverse of STORABLE_freeze.

Tie::Hash::Indexed::Iterator

Functions

Other Functions

next

next, prev: in list context the key and value the iterator stands on (nothing at the end); then it moves one entry on, or back.

Also available as Tie::Hash::Indexed::Iterator::prev.

valid

valid: true while the hash is unchanged and the iterator is not at the end.

key

key, value: of the entry the iterator stands on; undef at the end.

Also available as Tie::Hash::Indexed::Iterator::value.

Tie::Hash::NamedCapture

Named regexp capture buffers

SYNOPSIS

```
tie my %hash, "Tie::Hash::NamedCapture";
# %hash now behaves like %+

tie my %hash, "Tie::Hash::NamedCapture", all => 1;
# %hash now access buffers from regexp in $qr like %-
```

DESCRIPTION

This module is used to implement the special hashes %+ and %-, but it can be used to tie other variables as you choose.

When the all parameter is provided, then the tied hash elements will be array refs listing the contents of each capture buffer whose name is the same as the associated

hash key. If none of these buffers were involved in the match, the contents of that array ref will be as many `undef` values as there are capture buffers with that name. In other words, the tied hash will behave as `%-`.

When the `all` parameter is omitted or false, then the tied hash elements will be the contents of the leftmost defined buffer with the name of the associated hash key. In other words, the tied hash will behave as `%+`.

The keys of `%-`-like hashes correspond to all buffer names found in the regular expression; the keys of `%+`-like hashes list only the names of buffers that have captured (and that are thus associated to defined values).

This implementation has been moved into the core executable, but you can still load this module for backward compatibility.

SEE ALSO

perlreapi, *re*, "Pragmatic Modules" in perlmodlib, *"%+" in perlvar*, *"%-" in perlvar*.

Tie::Memoize

add data to hash when needed

SYNOPSIS

```
require Tie::Memoize;
tie %hash, 'Tie::Memoize',
    \&fetch,                  # The rest is optional
    $DATA, \&exists,
    {%ini_value}, {%ini_existence};
```

DESCRIPTION

This package allows a tied hash to autoload its values on the first access, and to use the cached value on the following accesses.

Only read-accesses (via fetching the value or `exists`) result in calls to the functions; the modify-accesses are performed as on a normal hash.

The required arguments during `tie` are the hash, the package, and the reference to the `FETCH`ing function. The optional arguments are an arbitrary scalar `$data`, the reference to the `EXISTS` function, and initial values of the hash and of the existence cache.

Both the `FETCH`ing function and the `EXISTS` functions have the same signature: the arguments are `$key, $data`; `$data` is the same value as given as argument during tie()ing. Both functions should return an empty list if the value does not exist. If `EXISTS` function is different from the `FETCH`ing function, it should return a TRUE value on success. The `FETCH`ing function should return the intended value if the key is valid.

Inheriting from Tie::Memoize

The structure of the tied() data is an array reference with elements

```
0:  cache of known values
1:  cache of known existence of keys
2:  FETCH  function
3:  EXISTS function
4:  $data
```

The rest is for internal usage of this package. In particular, if TIEHASH is overwritten, it should call SUPER::TIEHASH.

EXAMPLE

```
sub slurp {
  my ($key, $dir) = @_;
  open my $h, '<', "$dir/$key" or return;
  local $/; <$h>                        # slurp it all
}
sub exists { my ($key, $dir) = @_; return -f "$dir/$key" ? 1 : () }

tie %hash, 'Tie::Memoize', \&slurp, $directory, \&exists,
    { fake_file1 => $content1, fake_file2 => $content2 },
    { pretend_does_not_exists => 0, known_to_exist => 1 };
```

This example treats the slightly modified contents of $directory as a hash. The modifications are that the keys `fake_file1` and `fake_file2` fetch values $content1 and $content2, and `pretend_does_not_exists` will never be accessed. Additionally, the existence of `known_to_exist` is never checked (so if it does not exists when its content is needed, the user of %hash may be confused).

BUGS

FIRSTKEY and NEXTKEY methods go through the keys which were already read, not all the possible keys of the hash.

AUTHOR

Ilya Zakharevich mailto:perl-module-hash-memoize@ilyaz.org.

Tie::RefHash

Use references as hash keys

VERSION

version 1.41

SYNOPSIS

```
require 5.004;
use Tie::RefHash;
tie HASHVARIABLE, 'Tie::RefHash', LIST;
tie HASHVARIABLE, 'Tie::RefHash::Nestable', LIST;

untie HASHVARIABLE;
```

DESCRIPTION

This module provides the ability to use references as hash keys if you first `tie` the hash variable to this module. Normally, only the keys of the tied hash itself are preserved as references; to use references as keys in hashes-of-hashes, use Tie::RefHash::Nestable, included as part of Tie::RefHash.

It is implemented using the standard perl TIEHASH interface. Please see the `tie` entry in perlfunc(1) and perltie(1) for more information.

The Nestable version works by looking for hash references being stored and converting them to tied hashes so that they too can have references as keys. This will happen without warning whenever you store a reference to one of your own hashes in the tied hash.

EXAMPLE

```
use Tie::RefHash;
tie %h, 'Tie::RefHash';
$a = [];
$b = {};
$c = \*main;
$d = \"gunk";
$e = sub { 'foo' };
%h = ($a => 1, $b => 2, $c => 3, $d => 4, $e => 5);
$a->[0] = 'foo';
$b->{foo} = 'bar';
for (keys %h) {
   print ref($_), "\n";
}

tie %h, 'Tie::RefHash::Nestable';
$h{$a}->{$b} = 1;
for (keys %h, keys %{$h{$a}}) {
   print ref($_), "\n";
}
```

THREAD SUPPORT

Tie::RefHash fully supports threading using the `CLONE` method.

STORABLE SUPPORT

Storable hooks are provided for semantically correct serialization and cloning of tied refhashes.

SEE ALSO

perl(1), perlfunc(1), perltie(1)

SUPPORT

Bugs may be submitted through the RT bug tracker (or bug-Tie-RefHash@rt.cpan.org).

AUTHORS

Gurusamy Sarathy gsar@activestate.com

Tie::RefHash::Nestable by Ed Avis ed@membled.com

CONTRIBUTORS

- Yuval Kogman nothingmuch@woobling.org
- Karen Etheridge ether@cpan.org
- Florian Ragwitz rafl@debian.org
- Lukas Mai lukasmai.403@gmail.com
- Jerry D. Hedden jdhedden@cpan.org
- tusooa tusooa@kazv.moe

COPYRIGHT AND LICENCE

This software is copyright (c) 2006 by יובל קוג'מן (Yuval Kogman) nothingmuch@woobling.org.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

Tie::Scalar

base class definitions for tied scalars

SYNOPSIS

```
package NewScalar;
require Tie::Scalar;

@ISA = qw(Tie::Scalar);

sub FETCH { ... }           # Provide a needed method
sub TIESCALAR { ... }       # Overrides inherited method


package NewStdScalar;
require Tie::Scalar;

@ISA = qw(Tie::StdScalar);

# All methods provided by default, so define
# only what needs be overridden
sub FETCH { ... }


package main;

tie $new_scalar, 'NewScalar';
tie $new_std_scalar, 'NewStdScalar';
```

DESCRIPTION

This module provides some skeletal methods for scalar-tying classes. See perltie for a list of the functions required in tying a scalar to a package. The basic **Tie::Scalar** package provides a new method, as well as methods TIESCALAR, FETCH and STORE. The **Tie::StdScalar** package provides all the methods specified in perltie. It inherits from **Tie::Scalar** and causes scalars tied to it to behave exactly like the built-in scalars, allowing for selective overloading of methods. The new method is provided as a means of legacy support for classes that forget to provide their own TIESCALAR method.

For developers wishing to write their own tied-scalar classes, the methods are summarized below. The perltie section not only documents these, but has sample code as well:

- TIESCALAR classname, LIST

  The method invoked by the command `tie $scalar, classname`. Associates a new scalar instance with the specified class. LIST would represent additional arguments (along the lines of *AnyDBM_File* and compatriots) needed to complete the association.

- FETCH this

  Retrieve the value of the tied scalar referenced by *this*.

- STORE this, value

  Store data *value* in the tied scalar referenced by *this*.

- DESTROY this

Free the storage associated with the tied scalar referenced by *this*. This is rarely needed, as Perl manages its memory quite well. But the option exists, should a class wish to perform specific actions upon the destruction of an instance.

Tie::Scalar vs Tie::StdScalar

`Tie::Scalar` provides all the necessary methods, but one should realize they do not do anything useful. Calling `Tie::Scalar::FETCH` or `Tie::Scalar::STORE` results in a (trappable) croak. And if you inherit from `Tie::Scalar`, you *must* provide either a `new` or a `TIESCALAR` method.

If you are looking for a class that does everything for you that you don't define yourself, use the `Tie::StdScalar` class, not the `Tie::Scalar` one.

MORE INFORMATION

The perltie section uses a good example of tying scalars by associating process IDs with priority.

Tie::StdHandle

base class definitions for tied handles

SYNOPSIS

```
package NewHandle;
require Tie::Handle;

@ISA = qw(Tie::Handle);

sub READ { ... }             # Provide a needed method
sub TIEHANDLE { ... }        # Overrides inherited method


package main;

tie *FH, 'NewHandle';
```

DESCRIPTION

The **Tie::StdHandle** package provide most methods for file handles described in perltie (the exceptions are `UNTIE` and `DESTROY`). It causes tied file handles to behave exactly like standard file handles and allow for selective overwriting of methods.

Tie::SubstrHash

Fixed-table-size, fixed-key-length hashing

SYNOPSIS

```
require Tie::SubstrHash;

tie %myhash, 'Tie::SubstrHash', $key_len, $value_len, $table_size;
```

DESCRIPTION

The **Tie::SubstrHash** package provides a hash-table-like interface to an array of determinate size, with constant key size and record size.

Upon tying a new hash to this package, the developer must specify the size of the keys that will be used, the size of the value fields that the keys will index, and the size of the overall table (in terms of key-value pairs, not size in hard memory). *These values will not change for the duration of the tied hash.* The newly-allocated hash table may now have data stored and retrieved. Efforts to store more than `$table_size` elements will result in a fatal error, as will efforts to store a value not exactly `$value_len` characters in length, or reference through a key not exactly `$key_len` characters in length. While these constraints may seem excessive, the result is a hash table using much less internal memory than an equivalent freely-allocated hash table.

CAVEATS

Because the current implementation uses the table and key sizes for the hashing algorithm, there is no means by which to dynamically change the value of any of the initialization parameters.

The hash does not support exists().

5.2.106 Time

Namespace containing the following modules:

- `Time::HiRes` - High-resolution timers and sleeps: microsecond `gettimeofday`, sub-second `sleep`, `alarm`, and POSIX clocks.
- `Time::Local` - Efficiently compute time from local and GMT time
- `Time::Piece` - Date and time objects: `use Time::Piece` makes `localtime` and `gmtime` return an object with accessors for every field, formatting and parsing, comparison, and arithmetic with the durations of `Time::Seconds`.
- `Time::Seconds` - a simple API to convert seconds to other date values
- `Time::gmtime` - by-name interface to Perl's built-in gmtime() function
- `Time::localtime` - by-name interface to Perl's built-in localtime() function
- `Time::tm` - internal object used by Time::gmtime and Time::localtime

Time::HiRes

High-resolution timers and sleeps: microsecond `gettimeofday`, sub-second `sleep`, `alarm`, and POSIX clocks.

Where the built-in `time`, `sleep` and `alarm` work in whole seconds, these take and return fractional ones. Use `clock_gettime` with `CLOCK_MONOTONIC` to measure an elapsed interval: it cannot jump backwards the way wall-clock time can.

This is Time::HiRes 1.978; `Time/HiRes.pm` is upstream's. Nothing is exported by default; `use Time::HiRes qw(time sleep alarm)` replaces the built-ins in the importing package.

Synopsis

```
use Time::HiRes qw(time sleep gettimeofday tv_interval clock_gettime␣
↪CLOCK_MONOTONIC);

my $t0 = time;                          # e.g. 1700000000.123456
sleep 0.25;                             # a quarter of a second
printf "%.2f\n", time - $t0;            # about 0.25

my @start = gettimeofday;               # (seconds, microseconds)
my $elapsed = tv_interval(\@start);     # seconds since then, as a␣
↪float

my $m0 = clock_gettime(CLOCK_MONOTONIC);    # for intervals: never␣
↪jumps back
my $dt = clock_gettime(CLOCK_MONOTONIC) - $m0;
```

`tv_interval($start, $end)` (written in Perl) is the time between two `gettimeofday` pairs, `$end` defaulting to now. The clock constants (`CLOCK_REALTIME`, `CLOCK_MONOTONIC`, `CLOCK_PROCESS_CPUTIME_ID`, ...) and `ITIMER_REAL`/`ITIMER_VIRTUAL`/`ITIMER_PROF` for `setitimer` are exported on request.

Differences from upstream

None: upstream's tests pass unchanged (t/81-xs-native/Time-HiRes, the `upstream` files), and the rest there checks against perl 5.44.

Functions

Other Functions

time

Return the current epoch time as a float, with sub-second precision.

gettimeofday

Return the current time as `($seconds, $microseconds)`.

usleep

Sleep for the given number of MICROseconds.

nanosleep

Sleep for the given number of NANOseconds.

sleep

Sleep for a possibly fractional number of seconds.

alarm

Schedule a `SIGALRM` after a possibly fractional delay.

ualarm

Schedule a `SIGALRM` after a delay given in MICROseconds.

setitimer

Arm one of the interval timers (`ITIMER_REAL` and friends).

getitimer

Return an interval timer's current `($value, $interval)` without changing it.

clock_gettime

Read the named POSIX clock, returning the time as a float.

clock_getres

Return the resolution of the named POSIX clock, in seconds.

clock

Return processor time consumed by this process, in seconds.

clock_nanosleep

Sleep against a named POSIX clock, in nanoseconds.

utime

Set access and modification times on files, with sub-second precision.

stat

The built-in `stat`, with the access, modification and change times (elements 8, 9 and 10) in fractional seconds, as precise as the file system keeps them. Takes a file name or a filehandle, `$_` when omitted; returns the empty list for a file that cannot be stat'ed, with `$!` set.

lstat

`stat` for a symbolic link itself rather than what it points to, as the built-in `lstat`, with fractional times.

tv_interval

`tv_interval(\@t0, \@t1?)` - pure Perl in `HiRes.pm`

Time::Local

Efficiently compute time from local and GMT time

VERSION

version 1.35

SYNOPSIS

```
use Time::Local qw( timelocal_posix timegm_posix );

my $time = timelocal_posix( $sec, $min, $hour, $mday, $mon, $year );
my $time = timegm_posix( $sec, $min, $hour, $mday, $mon, $year );
```

DESCRIPTION

This module provides functions that are the inverse of built-in perl functions `localtime()` and `gmtime()`. They accept a date as a six-element array, and return the corresponding `time(2)` value in seconds since the system epoch (Midnight, January 1, 1970 GMT on Unix, for example). This value can be positive or negative, though POSIX only requires support for positive values, so dates before the system's epoch may not work on all operating systems.

It is worth drawing particular attention to the expected ranges for the values provided. The value for the day of the month is the actual day (i.e. 1..31), while the month is the number of months since January (0..11). This is consistent with the values returned from `localtime()` and `gmtime()`.

FUNCTIONS

`timelocal_posix()` and `timegm_posix()`

Since version 1.30.

These functions are the exact inverse of Perl's built-in `localtime` and `gmtime` functions. That means that calling `timelocal_posix( localtime($value) )` will always give you the same `$value` you started with. The same applies to `timegm_posix( gmtime($value) )`.

The one exception is when the value returned from `localtime()` represents an ambiguous local time because of a DST change. See the documentation below for more details.

These functions expect the year value to be the number of years since 1900, which is what the `localtime()` and `gmtime()` built-ins returns.

They perform range checking by default on the input `$sec`, `$min`, `$hour`, `$mday`, and `$mon` values and will croak (using `Carp::croak()`) if given a value outside the allowed ranges.

While it would be nice to make this the default behavior, that would almost certainly break a lot of code, so you must explicitly import these functions and use them instead of the default `timelocal()` and `timegm()`.

You are **strongly** encouraged to use these functions in any new code which uses this module. It will almost certainly make your code's behavior less surprising.

`timelocal_modern()` and `timegm_modern()`

Since version 1.27.

When `Time::Local` was first written, it was a common practice to represent years as a two-digit value like `99` for `1999` or `1` for `2001`. This caused all sorts of problems (google "Y2K problem" if you're very young) and developers eventually realized that this was a terrible idea.

The default exports of `timelocal()` and `timegm()` do a complicated calculation when given a year value less than 1000. This leads to surprising results in many cases. See "Year Value Interpretation" for details.

The `time*_modern()` functions do not do this year munging and simply take the year value as provided.

They perform range checking by default on the input `$sec`, `$min`, `$hour`, `$mday`, and `$mon` values and will croak (using `Carp::croak()`) if given a value outside the allowed ranges.

`timelocal()` and `timegm()`

This module exports two functions by default, `timelocal()` and `timegm()`.

They perform range checking by default on the input `$sec`, `$min`, `$hour`, `$mday`, and `$mon` values and will croak (using `Carp::croak()`) if given a value outside the allowed ranges.

Warning: The year value interpretation that these functions and their nocheck variants use will almost certainly lead to bugs in your code, if not now, then in the future. You are strongly discouraged from using these in new code, and you should convert old code to using either the `*_posix` or `*_modern` functions if possible.

`timelocal_nocheck()` and `timegm_nocheck()`

If you are working with data you know to be valid, you can use the "nocheck" variants, `timelocal_nocheck()` and `timegm_nocheck()`. These variants must be explicitly imported.

If you supply data which is not valid (month 27, second 1,000) the results will be unpredictable (so don't do that).

Note that my benchmarks show that this is just a 3% speed increase over the checked versions, so unless calling `Time::Local` is the hottest spot in your application, using these nocheck variants is unlikely to have much impact on your application.

Year Value Interpretation

This does not apply to the `*_posix` or `*_modern` functions. Use those exports if you want to ensure consistent behavior as your code ages.

Strictly speaking, the year should be specified in a form consistent with `localtime()`, i.e. the offset from 1900. In order to make the interpretation of the year easier for humans, however, who are more accustomed to seeing years as two-digit or four-digit values, the following conventions are followed:

- Years greater than 999 are interpreted as being the actual year, rather than the offset from 1900. Thus, 1964 would indicate the year Martin Luther King won the Nobel prize, not the year 3864.
- Years in the range 100..999 are interpreted as offset from 1900, so that 112 indicates 2012. This rule also applies to years less than zero (but see note below regarding date range).
- Years in the range 0..99 are interpreted as shorthand for years in the rolling "current century," defined as 50 years on either side of the current year. Thus, today, in 1999, 0 would refer to 2000, and 45 to 2045, but 55 would refer to 1955. Twenty years from now, 55 would instead refer to 2055. This is messy, but matches the way people currently think about two digit dates. Whenever possible, use an absolute four digit year instead.

The scheme above allows interpretation of a wide range of dates, particularly if 4-digit years are used. But it also means that the behavior of your code changes as time passes, because the rolling "current century" changes each year.

Limits of time_t

On perl versions older than 5.12.0, the range of dates that can be actually be handled depends on the size of `time_t` (usually a signed integer) on the given platform. Currently, this is 32 bits for most systems, yielding an approximate range from Dec 1901 to Jan 2038.

Both `timelocal()` and `timegm()` croak if given dates outside the supported range.

As of version 5.12.0, perl has stopped using the time implementation of the operating system it's running on. Instead, it has its own implementation of those routines with a safe range of at least +/- 2**52 (about 142 million years)

Ambiguous Local Times (DST)

Because of DST changes, there are many time zones where the same local time occurs for two different GMT times on the same day. For example, in the "Europe/Paris" time zone, the local time of 2001-10-28 02:30:00 can represent either 2001-10-28 00:30:00 GMT, **or** 2001-10-28 01:30:00 GMT.

When given an ambiguous local time, the timelocal() function will always return the epoch for the *earlier* of the two possible GMT times.

Non-Existent Local Times (DST)

When a DST change causes a locale clock to skip one hour forward, there will be an hour's worth of local times that don't exist. Again, for the "Europe/Paris" time zone, the local clock jumped from 2001-03-25 01:59:59 to 2001-03-25 03:00:00.

If the `timelocal()` function is given a non-existent local time, it will simply return an epoch value for the time one hour later.

Negative Epoch Values

On perl version 5.12.0 and newer, negative epoch values are fully supported.

On older versions of perl, negative epoch (`time_t`) values, which are not officially supported by the POSIX standards, are known not to work on some systems. These include MacOS (pre-OSX) and Win32.

On systems which do support negative epoch values, this module should be able to cope with dates before the start of the epoch, down the minimum value of time_t for the system.

IMPLEMENTATION

These routines are quite efficient and yet are always guaranteed to agree with `localtime()` and `gmtime()`. We manage this by caching the start times of any months we've seen before. If we know the start time of the month, we can always calculate any time within the month. The start times are calculated using a mathematical formula. Unlike other algorithms that do multiple calls to `gmtime()`.

The `timelocal()` function is implemented using the same cache. We just assume that we're translating a GMT time, and then fudge it when we're done for the timezone and daylight savings arguments. Note that the timezone is evaluated for each date because countries occasionally change their official timezones. Assuming that `localtime()` corrects for these changes, this routine will also be correct.

AUTHORS EMERITUS

This module is based on a Perl 4 library, timelocal.pl, that was included with Perl 4.036, and was most likely written by Tom Christiansen.

The current version was written by Graham Barr.

BUGS

The whole scheme for interpreting two-digit years can be considered a bug.

Bugs may be submitted at https://github.com/houseabsolute/Time-Local/issues.

There is a mailing list available for users of this distribution, mailto:datetime@perl.org.

SOURCE

The source code repository for Time-Local can be found at https://github.com/houseabsolute/Time-Local.

AUTHOR

Dave Rolsky autarch@urth.org

CONTRIBUTORS

- Florian Ragwitz rafl@debian.org
- Gregory Oschwald oschwald@gmail.com
- J. Nick Koston nick@cpanel.net
- Tom Wyant wyant@cpan.org
- Unknown unknown@example.com

COPYRIGHT AND LICENSE

This software is copyright (c) 1997 - 2023 by Graham Barr & Dave Rolsky.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

The full text of the license can be found in the `LICENSE` file included with this distribution.

Time::Piece

Date and time objects: `use Time::Piece` makes `localtime` and `gmtime` return an object with accessors for every field, formatting and parsing, comparison, and arithmetic with the durations of `Time::Seconds`. In string context the object is still the familiar `scalar localtime` text (`Tue Nov 14 22:13:20 2023`).

This is Time::Piece 1.41; `Time/Piece.pm` and `Time/Seconds.pm` are upstream's. Formatting, parsing and the breaking down of an epoch run natively; parsing (`strptime`) does not depend on the system's library, so a format parses the same way everywhere.

Synopsis

```
use Time::Piece;
use Time::Seconds;

my $t = gmtime(1_700_000_000);
print $t->ymd, ' ', $t->hms, "\n";              # 2023-11-14␣
↪22:13:20
print $t->year, ' ', $t->mon, ' ', $t->monname, "\n";   # 2023 11␣
↪Nov
print $t->strftime('%A, %d %B %Y'), "\n";       # Tuesday, 14␣
↪November 2023
print $t->epoch, "\n";                          # 1700000000

my $later = $t + ONE_DAY * 3;
print $later->ymd, "\n";                        # 2023-11-17
print +($later - $t)->days, " days\n";          # 3 days
print $t->add_months(3)->ymd, "\n";             # 2024-02-14

my $p = Time::Piece->strptime('2024-02-29 12:00', '%Y-%m-%d %H:%M');
print $p->is_leap_year ? "leap\n" : "common\n"; # leap
print $p > $t ? "later\n" : "earlier\n";        # later
```

The interface

- **Making one** - `localtime($epoch)`, `gmtime($epoch)` (now without an argument), `Time::Piece->strptime($string, $format)`, and `Time::Piece->new($epoch)`.
- **Fields** - `sec`, `min`, `hour`, `mday` (`day_of_month`), `mon` (1-12), `_mon` (0-11), `monname`/`month`, `fullmonth`, `year`, `yy`, `wday` (1 = Sunday), `_wday` (0 = Sunday), `day_of_week`, `wdayname`, `fullday`, `yday` (`day_of_year`), `isdst`, `tzoffset`, `epoch`, `week` (ISO 8601), `julian_day`, `mjd`, `is_leap_year`, `month_last_day`.
- **Text** - `ymd`, `mdy`, `dmy`, `hms` (each with an optional separator), `datetime` (ISO 8601), `cdate`, `strftime($format)`.
- **Arithmetic** - `+` and `-` with seconds or a `Time::Seconds` (`ONE_DAY`, `ONE_WEEK`, `ONE_HOUR`, ...); two objects subtract to a `Time::Seconds` (`->days`, `->hours`, `->minutes`); `add_months`, `add_years`; `<`, `==`, `<=>` and `cmp` compare.
- **Locale** - `day_list`, `mon_list` and their `full` forms set or read the names `strftime` and `strptime` use; `Time::Piece::use_locale()` takes them from the current locale.

Differences from upstream

None: upstream's tests pass unchanged (t/81-xs-native/Time-Piece, the `upstream` files), and 010-050 there check formatting, parsing and the field breakdown against perl 5.44.

Time::Seconds

a simple API to convert seconds to other date values

SYNOPSIS

```
use Time::Piece;
use Time::Seconds;

my $t = localtime;
$t += ONE_DAY;

my $t2 = localtime;
my $s = $t - $t2;

print "Difference is: ", $s->days, "\n";
```

DESCRIPTION

This module is part of the Time::Piece distribution. It allows the user to find out the number of minutes, hours, days, weeks or years in a given number of seconds. It is returned by Time::Piece when you delta two Time::Piece objects.

Time::Seconds also exports the following constants:

```
ONE_DAY
ONE_WEEK
ONE_HOUR
ONE_MINUTE
ONE_MONTH
ONE_YEAR
ONE_FINANCIAL_MONTH
LEAP_YEAR
NON_LEAP_YEAR
```

Since perl does not (yet?) support constant objects, these constants are in seconds only, so you cannot, for example, do this: `print ONE_WEEK->minutes;`

METHODS

The following methods are available:

```
my $val = Time::Seconds->new(SECONDS);
$val->seconds;
$val->minutes;
$val->hours;
$val->days;
$val->weeks;
$val->months;
$val->financial_months; # 30 days
```

(continues on next page)

(continued from previous page)

```
$val->years;
$val->pretty; # gives English representation of the delta
```

The usual arithmetic (+,-,+=,-=) is also available on the objects.

The methods make the assumption that there are 24 hours in a day, 7 days in a week, 365.24225 days in a year and 12 months in a year. (from The Calendar FAQ at http://www.tondering.dk/claus/calendar.html)

AUTHOR

Matt Sergeant, matt@sergeant.org

Tobias Brox, tobiasb@tobiasb.funcom.com

Balázs Szabó (dLux), dlux@kapu.hu

COPYRIGHT AND LICENSE

Copyright 2001, Larry Wall.

This module is free software, you may distribute it under the same terms as Perl.

Bugs

Currently the methods aren't as efficient as they could be, for reasons of clarity. This is probably a bad idea.

Time::gmtime

by-name interface to Perl's built-in gmtime() function

SYNOPSIS

```
use Time::gmtime;
$gm = gmtime();
printf "The day in Greenwich is %s\n",
   (qw(Sun Mon Tue Wed Thu Fri Sat Sun))[ $gm->wday() ];

use Time::gmtime qw(:FIELDS);
gmtime();
printf "The day in Greenwich is %s\n",
   (qw(Sun Mon Tue Wed Thu Fri Sat Sun))[ $tm_wday ];

$now = gmctime();

use Time::gmtime;
use File::stat;
$date_string = gmctime(stat($file)->mtime);
```

DESCRIPTION

This module's default exports override the core gmtime() function, replacing it with a version that returns "Time::tm" objects. This object has methods that return the similarly named structure field name from the C's tm structure from `time.h`; namely sec, min, hour, mday, mon, year, wday, yday, and isdst.

You may also import all the structure fields directly into your namespace as regular variables using the :FIELDS import tag. (Note that this still overrides your core functions.) Access these fields as variables named with a preceding `tm_` in front their method names. Thus, `$tm_obj->mday()` corresponds to $tm_mday if you import the fields.

The gmctime() function provides a way of getting at the scalar sense of the original CORE::gmtime() function.

To access this functionality without the core overrides, pass the `use` an empty import list, and then access function functions with their full qualified names. On the other hand, the built-ins are still available via the `CORE::` pseudo-package.

NOTE

While this class is currently implemented using the Class::Struct module to build a struct-like class, you shouldn't rely upon this.

AUTHOR

Tom Christiansen

Time::localtime

by-name interface to Perl's built-in localtime() function

SYNOPSIS

```
use Time::localtime;
printf "Year is %d\n", localtime->year() + 1900;

$now = ctime();

use Time::localtime;
use File::stat;
$date_string = ctime(stat($file)->mtime);
```

DESCRIPTION

This module's default exports override the core localtime() function, replacing it with a version that returns "Time::tm" objects. This object has methods that return the similarly named structure field name from the C's tm structure from `time.h`; namely sec, min, hour, mday, mon, year, wday, yday, and isdst.

You may also import all the structure fields directly into your namespace as regular variables using the :FIELDS import tag. (Note that this still overrides your core func-

tions.) Access these fields as variables named with a preceding `tm_` in front their method names. Thus, `$tm_obj->mday()` corresponds to $tm_mday if you import the fields.

The ctime() function provides a way of getting at the scalar sense of the original CORE::localtime() function.

To access this functionality without the core overrides, pass the `use` an empty import list, and then access function functions with their full qualified names. On the other hand, the built-ins are still available via the `CORE::` pseudo-package.

NOTE

While this class is currently implemented using the Class::Struct module to build a struct-like class, you shouldn't rely upon this.

AUTHOR

Tom Christiansen

Time::tm

internal object used by Time::gmtime and Time::localtime

SYNOPSIS

Don't use this module directly.

DESCRIPTION

This module is used internally as a base class by Time::localtime And Time::gmtime functions. It creates a Time::tm struct object which is addressable just like's C's tm structure from `time.h`; namely with sec, min, hour, mday, mon, year, wday, yday, and isdst.

This class is an internal interface only.

AUTHOR

Tom Christiansen

5.2.107 Types

Namespace containing the following modules:

- `Types::Serialiser` - simple data types for common serialisation formats

Types::Serialiser

simple data types for common serialisation formats

SYNOPSIS

DESCRIPTION

This module provides some extra datatypes that are used by common serialisation formats such as JSON or CBOR. The idea is to have a repository of simple/small constants and containers that can be shared by different implementations so they become interoperable between each other.

SIMPLE SCALAR CONSTANTS

Simple scalar constants are values that are overloaded to act like simple Perl values, but have (class) type to differentiate them from normal Perl scalars. This is necessary because these have different representations in the serialisation formats.

In the following, functions with zero or one arguments have a prototype of `()` and `($)`, respectively, so act as constants and unary operators.

BOOLEANS (Types::Serialiser::Boolean class)

This type has only two instances, true and false. A natural representation for these in Perl is `1` and `0`, but serialisation formats need to be able to differentiate between them and mere numbers.

- $Types::Serialiser::true, Types::Serialiser::true

  This value represents the "true" value. In most contexts is acts like the number `1`. It is up to you whether you use the variable form (`$Types::Serialiser::true`) or the constant form (`Types::Serialiser::true`).

  The constant is represented as a reference to a scalar containing `1` - implementations are allowed to directly test for this.

- $Types::Serialiser::false, Types::Serialiser::false

  This value represents the "false" value. In most contexts is acts like the number `0`. It is up to you whether you use the variable form (`$Types::Serialiser::false`) or the constant form (`Types::Serialiser::false`).

  The constant is represented as a reference to a scalar containing `0` - implementations are allowed to directly test for this.

- Types::Serialiser::as_bool $value

  Converts a Perl scalar into a boolean, which is useful syntactic sugar. Strictly equivalent to:

  ```
  $value ? $Types::Serialiser::true : $Types::Serialiser::false
  ```

- $is_bool = Types::Serialiser::is_bool $value

  Returns true iff the `$value` is either `$Types::Serialiser::true` or `$Types::Serialiser::false`.

  For example, you could differentiate between a perl true value and a `Types::Serialiser::true` by using this:

```
$value && Types::Serialiser::is_bool $value
```

- $is_true = Types::Serialiser::is_true $value

  Returns true iff `$value` is `$Types::Serialiser::true`.

- $is_false = Types::Serialiser::is_false $value

  Returns false iff `$value` is `$Types::Serialiser::false`.

ERROR (Types::Serialiser::Error class)

This class has only a single instance, `error`. It is used to signal an encoding or decoding error. In CBOR for example, and object that couldn't be encoded will be represented by a CBOR undefined value, which is represented by the error value in Perl.

- $Types::Serialiser::error, Types::Serialiser::error

  This value represents the "error" value. Accessing values of this type will throw an exception.

  The constant is represented as a reference to a scalar containing `undef` - implementations are allowed to directly test for this.

- $is_error = Types::Serialiser::is_error $value

  Returns false iff `$value` is `$Types::Serialiser::error`.

A GENERIC OBJECT SERIALIATION PROTOCOL

This section explains the object serialisation protocol used by CBOR::XS. It is meant to be generic enough to support any kind of generic object serialiser.

This protocol is called "the Types::Serialiser object serialisation protocol".

ENCODING

When the encoder encounters an object that it cannot otherwise encode (for example, CBOR::XS can encode a few special types itself, and will first attempt to use the special `TO_CBOR` serialisation protocol), it will look up the `FREEZE` method on the object.

Note that the `FREEZE` method will normally be called *during* encoding, and *MUST NOT* change the data structure that is being encoded in any way, or it might cause memory corruption or worse.

If it exists, it will call it with two arguments: the object to serialise, and a constant string that indicates the name of the data model. For example CBOR::XS uses `CBOR`, and the *JSON* and *JSON::XS* modules (or any other JSON serialiser), would use `JSON` as second argument.

The `FREEZE` method can then return zero or more values to identify the object instance. The serialiser is then supposed to encode the class name and all of these return values (which must be encodable in the format) using the relevant form for Perl objects. In CBOR for example, there is a registered tag number for encoded perl objects.

The values that `FREEZE` returns must be serialisable with the serialiser that calls it. Therefore, it is recommended to use simple types such as strings and numbers, and

maybe array references and hashes (basically, the JSON data model). You can always use a more complex format for a specific data model by checking the second argument, the data model.

The "data model" is not the same as the "data format" - the data model indicates what types and kinds of return values can be returned from FREEZE. For example, in CBOR it is permissible to return tagged CBOR values, while JSON does not support these at all, so JSON would be a valid (but too limited) data model name for CBOR::XS. similarly, a serialising format that supports more or less the same data model as JSON could use JSON as data model without losing anything.

DECODING

When the decoder then encounters such an encoded perl object, it should look up the THAW method on the stored classname, and invoke it with the classname, the constant string to identify the data model/data format, and all the return values returned by FREEZE.

EXAMPLES

See the OBJECT SERIALISATION section in the CBOR::XS manpage for more details, an example implementation, and code examples.

Here is an example FREEZE/THAW method pair:

```
sub My::Object::FREEZE {
   my ($self, $model) = @_;

   ($self->{type}, $self->{id}, $self->{variant})
}

sub My::Object::THAW {
   my ($class, $model, $type, $id, $variant) = @_;

   $class->new (type => $type, id => $id, variant => $variant)
}
```

BUGS

The use of *overload* makes this module much heavier than it should be (on my system, this module: 4kB RSS, overload: 260kB RSS).

SEE ALSO

Currently, *JSON::XS* and CBOR::XS use these types.

AUTHOR

```
Marc Lehmann <schmorp@schmorp.de>
http://home.schmorp.de/
```

Modules

- `Types::Serialiser::Error` - dummy module for Types::Serialiser

Types::Serialiser::Error

dummy module for Types::Serialiser

SYNOPSIS

```
# do not "use" yourself
```

DESCRIPTION

This module exists only to provide overload resolution for Storable and similar modules that assume that class name equals module name. See *Types::Serialiser* for more info about this class.

AUTHOR

```
Marc Lehmann <schmorp@schmorp.de>
http://home.schmorp.de/
```

5.2.108 UNIVERSAL

base class for ALL classes (blessed references)

SYNOPSIS

```
my $obj_is_io    = $fd->isa("IO::Handle");
my $cls_is_io    = Class->isa("IO::Handle");

my $obj_does_log = $obj->DOES("Logger");
my $cls_does_log = Class->DOES("Logger");

my $obj_sub      = $obj->can("print");
my $cls_sub      = Class->can("print");

my $eval_sub     = eval { $ref->can("fandango") };
my $ver          = $obj->VERSION;

# but never do this!
```

(continues on next page)

(continued from previous page)

```
my $is_io          = UNIVERSAL::isa($fd, "IO::Handle");
my $sub            = UNIVERSAL::can($obj, "print");
```

DESCRIPTION

UNIVERSAL is the base class from which all blessed references inherit. See perlobj.

UNIVERSAL provides the following methods:

- `$obj->isa( TYPE )`
- `CLASS->isa( TYPE )`
- `eval { VAL->isa( TYPE ) }`

  Where

  - `TYPE`

    is a package name

  - `$obj`

    is a blessed reference or a package name

  - `CLASS`

    is a package name

  - `VAL`

    is any of the above or an unblessed reference

  When used as an instance or class method (`$obj->isa( TYPE )`), `isa` returns *true* if $obj is blessed into package `TYPE` or inherits from package `TYPE`.

  When used as a class method (`CLASS->isa( TYPE )`, sometimes referred to as a static method), `isa` returns *true* if `CLASS` inherits from (or is itself) the name of the package `TYPE` or inherits from package `TYPE`.

  If you're not sure what you have (the `VAL` case), wrap the method call in an `eval` block to catch the exception if `VAL` is undefined or an unblessed reference. The *`isa` operator* is an alternative that simply returns false in this case, so the `eval` is not needed.

  If you want to be sure that you're calling `isa` on an instance, not a class, check the invocant with `blessed` from *Scalar::Util* first:

  ```
  use Scalar::Util 'blessed';

  if ( blessed( $obj ) && $obj->isa("Some::Class") ) {
      ...
  }
  ```

- `$obj->DOES( ROLE )`
- `CLASS->DOES( ROLE )`

DOES checks if the object or class performs the role ROLE. A role is a named group of specific behavior (often methods of particular names and signatures), similar to a class, but not necessarily a complete class by itself. For example, logging or serialization may be roles.

DOES and isa are similar, in that if either is true, you know that the object or class on which you call the method can perform specific behavior. However, DOES is different from isa in that it does not care *how* the invocant performs the operations, merely that it does. (isa of course mandates an inheritance relationship. Other relationships include aggregation, delegation, and mocking.)

By default, classes in Perl only perform the UNIVERSAL role, as well as the role of all classes in their inheritance. In other words, by default DOES responds identically to isa.

There is a relationship between roles and classes, as each class implies the existence of a role of the same name. There is also a relationship between inheritance and roles, in that a subclass that inherits from an ancestor class implicitly performs any roles its parent performs. Thus you can use DOES in place of isa safely, as it will return true in all places where isa will return true (provided that any overridden DOES *and* isa methods behave appropriately).

- `$obj->can( METHOD )`
- `CLASS->can( METHOD )`
- `eval { VAL->can( METHOD ) }`

  can checks if the object or class has a method called METHOD. If it does, then it returns a reference to the sub. If it does not, then it returns *undef*. This includes methods inherited or imported by $obj, CLASS, or VAL.

  can cannot know whether an object will be able to provide a method through AUTOLOAD (unless the object's class has overridden can appropriately), so a return value of *undef* does not necessarily mean the object will not be able to handle the method call. To get around this some module authors use a forward declaration (see *perlsub*) for methods they will handle via AUTOLOAD. For such 'dummy' subs, can will still return a code reference, which, when called, will fall through to the AUTOLOAD. If no suitable AUTOLOAD is provided, calling the coderef will cause an error.

  You may call can as a class (static) method or an object method.

  Again, the same rule about having a valid invocant applies – use an eval block or blessed if you need to be extra paranoid.

- `VERSION ( [ REQUIRE ] )`

  VERSION will return the value of the variable $VERSION in the package the object is blessed into. If REQUIRE is given then it will do a comparison and die if the package version is not greater than or equal to REQUIRE, or if either $VERSION or REQUIRE is not a "lax" version number (as defined by the *version* module).

  The return from VERSION will actually be the stringified version object using the package $VERSION scalar, which is guaranteed to be equivalent but may not be precisely the contents of the $VERSION scalar. If you want the actual contents of $VERSION, use $CLASS::VERSION instead.

`VERSION` can be called as either a class (static) method or an object method.

WARNINGS

NOTE: `can` directly uses Perl's internal code for method lookup, and `isa` uses a very similar method and cache-ing strategy. This may cause strange effects if the Perl code dynamically changes @ISA in any package.

You may add other methods to the UNIVERSAL class via Perl code. You do not need to `use UNIVERSAL` to make these methods available to your program (and you should not do so).

EXPORTS

None.

Previous versions of this documentation suggested using `isa` as a function to determine the type of a reference:

```
$yes = UNIVERSAL::isa($h, "HASH");
$yes = UNIVERSAL::isa("Foo", "Bar");
```

The problem is that this code would *never* call an overridden `isa` method in any class. Instead, use `reftype` from *Scalar::Util* for the first case:

```
use Scalar::Util 'reftype';

$yes = reftype( $h ) eq "HASH";
```

and the method form of `isa` for the second:

```
$yes = Foo->isa("Bar");
```

5.2.109 URL

Namespace containing the following modules:

- `URL::Encode` - Encoding and decoding of `application/x-www-form-urlencoded` encoding.

URL::Encode

Encoding and decoding of `application/x-www-form-urlencoded` encoding.

SYNOPSIS

```
$octets = url_decode($octets);
$string = url_decode_utf8($octets);

$octets = url_encode($octets);
$octets = url_encode_utf8($string);
```

(continues on next page)

(continued from previous page)

```
$params = url_params_flat($octets [, $utf8 = false]);
$params = url_params_mixed($octets [, $utf8 = false]);
$params = url_params_multi($octets [, $utf8 = false]);
          url_params_each($octets, $callback [, $utf8 = false]);
```

DESCRIPTION

This module provides functions to encode and decode strings into and from the `application/x-www-form-urlencoded` encoding.

The `application/x-www-form-urlencoded` format encodes a ordered data sets of pairs consisting of a name and a value, with pairs seperated by ampersand or semicolon and names and values seperated by the equal sign. Space characters are replaced with plus sign and any characters not in the unreserved character set is encoded using the percent-encoding scheme also used for resource identifiers. A percent-encoded octet is encoded as a character triplet, consisting of the percent character "%" followed by the two hexadecimal digits representing that octet's numeric value.

The unreserved character set includes the uppercase and lowercase letters, decimal digits, hyphen, period, underscore, and tilde.

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
0123456789
- . _ ~
```

FUNCTIONS

url_decode

```
$octets = url_decode($octets);
```

Returns a decoded representation of the given URL-encoded `$octets` as an octet string.

url_decode_utf8

```
$string = url_decode_utf8($octets);
```

Returns a decoded representation of the given URL-encoded `$octets` in UTF-8 encoding as a character string.

url_encode

```
$octets = url_encode($octets);
```

Returns a URL-encoded representation of the given `$octets` as an octet string.

url_encode_utf8

```
$octets = url_encode_utf8($string);
```

Returns a URL-encoded representation of `$string` in UTF-8 encoding as an octet string.

url_params_flat

```
$params = url_params_flat($octets);
$params = url_params_flat($octets, $utf8);
```

Parses a URL-encoded data set of name/value pairs from the given octets. Returns an ARRAY reference containing the URL-decoded name/value pairs in order.

```
$params = url_params_flat('foo=A&foo=B&bar=C');
$params; # [ foo => 'A', foo => 'B', bar => 'C' ]
```

url_params_mixed

```
$params = url_params_mixed($octets);
$params = url_params_mixed($octets, $utf8);
```

Parses a URL-encoded data set of name/value pairs from the given octets. Returns a HASH reference containing the URL-decoded name/value pairs. Multiple occurrences of a parameter will result in an ARRAY reference holding all the values for that parameter in order.

```
$params = url_params_mixed('foo=A&foo=B&bar=C');
$params; # { foo => [ 'A', 'B' ], bar => 'C' }
```

url_params_multi

```
$params = url_params_multi($octets);
$params = url_params_multi($octets, $utf8);
```

Parses a URL-encoded data set of name/value pairs from the given octets. Returns a HASH reference containing the URL-decoded name/value pairs. Values are stored in an ARRAY reference.

```
$params = url_params_multi('foo=A&foo=B&bar=C');
$params; # { foo => [ 'A', 'B' ], bar => [ 'C' ] }
```

url_params_each

```
url_params_each($octets, $callback);
url_params_each($octets, $callback, $utf8);
```

Parses a URL-encoded data set of name/value pairs from the given octets. Invokes the given callback for each URL-decoded name/value pair.

```
$callback = sub {
    my ($name, $value) = @_;
};

url_params_each($octets, $callback);
```

EXPORTS

None by default. All functions can be exported using the `:all` tag or individually.

DIAGNOSTICS

- **(F)** Usage: %s

  Subroutine called with wrong number of arguments.
- **(F)** Wide character in octet string
- **(F)** Malformed UTF-8 in URL-decoded octets

PERFORMANCE

This module will automatically use *URL::Encode::XS* if it's installed.

SEE ALSO

- *URL::Encode::XS*
- CGI::Deurl::XS

SUPPORT

BUGS

Please report any bugs by email to `bug-url-encode at rt.cpan.org`, or through the web interface at http://rt.cpan.org/Public/Dist/Display.html?Name=URL-Encode. You will be automatically notified of any progress on the request by the system.

SOURCE CODE

This is open source software. The code repository is available for public review and contribution under the terms of the license.

http://github.com/chansen/p5-url-encode

```
git clone http://github.com/chansen/p5-url-encode
```

AUTHOR

Christian Hansen chansen@cpan.org

COPYRIGHT

Copyright 2011-2014 by Christian Hansen.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

Modules

- URL::Encode::XS - Encode and decode the application/x-www-form-urlencoded form of strings, and split a query string into its parameters.

URL::Encode::XS

Encode and decode the application/x-www-form-urlencoded form of strings, and split a query string into its parameters.

This is the fast half of URL::Encode: URL::Encode loads it when it is there, and falls back to its pure-Perl URL::Encode::PP, which gives the same results, otherwise (or when URL_ENCODE_PP is set in the environment). Use URL::Encode; nothing is exported by default.

url_encode escapes every byte except A-Z a-z 0-9 - . _ ~ as %XX, a space as +; url_decode reverses it, and leaves a % that is not followed by two hex digits as it is. The _utf8 variants take and return character strings, the others octets: an octet argument with a character above 255 croaks "Wide character in octet string".

The url_params_* functions split on & and ;, and = separates a name from its value; a pair without = has the value undef. With a true second argument, names and values are decoded as UTF-8. url_params_flat returns an array reference of names and values in order, url_params_mixed a hash reference whose repeated names hold an array reference of their values, url_params_multi a hash reference of array references, and url_params_each calls a code reference with each name and value.

This is a native implementation of URL::Encode::XS 0.03's XS.xs; its URL/Encode/XS.pm is used unchanged.

Synopsis

```
use URL::Encode qw(url_encode_utf8 url_decode_utf8 url_params_
 ↪mixed);

my $q = url_encode_utf8("K\x{f6}ln & more");   # "K%C3%B6ln+%26+more
 ↪"
print url_decode_utf8($q), "\n";               # "Köln & more"

my $p = url_params_mixed("a=1&b=2&a=3");
# { a => [1, 3], b => 2 }
```

Functions

Encoding

url_decode

`url_decode($octets)`: the octets with + and the %XX escapes decoded.

url_decode_utf8

`url_decode_utf8($octets)`: `url_decode`, then the result decoded as UTF-8; croaks "Malformed UTF-8 in URL decoded string" when it is not.

url_encode

`url_encode($octets)`: every byte except `A-Z a-z 0-9 - . _ ~` as %XX, a space as +.

url_encode_utf8

`url_encode_utf8($string)`: the string encoded as UTF-8, then `url_encoded`.

Query strings

url_params_flat

`url_params_flat($octets [, $utf8])`: an array reference of the names and values in order.

url_params_mixed

`url_params_mixed($octets [, $utf8])`: a hash reference; a name given more than once holds an array reference of its values.

url_params_multi

`url_params_multi($octets [, $utf8])`: a hash reference of array references of the values.

url_params_each

`url_params_each($octets, $callback [, $utf8])`: call `$callback` with each name and value; returns nothing.

5.2.110 Unicode

Namespace containing the following modules:

- `Unicode::Collate` - Unicode Collation Algorithm
- `Unicode::GCString`

- `Unicode::LineBreak` - Line breaking for Unicode text, and strings of grapheme clusters.
- `Unicode::Normalize` - the Unicode normalization forms NFD, NFC, NFKD
- `Unicode::UCD` - the Unicode Character Database: what perl knows about

Unicode::Collate

Unicode Collation Algorithm

SYNOPSIS

```
use Unicode::Collate;

#construct
$Collator = Unicode::Collate->new(%tailoring);

#sort
@sorted = $Collator->sort(@not_sorted);

#compare
$result = $Collator->cmp($a, $b); # returns 1, 0, or -1.
```

Note: Strings in `@not_sorted`, `$a` and `$b` are interpreted according to Perl's Unicode support. See perlunicode, perluniintro, perlunitut, perlunifaq, *utf8*. Otherwise you can use `preprocess` or should decode them before.

DESCRIPTION

This module is an implementation of Unicode Technical Standard #10 (a.k.a. UTS #10) - Unicode Collation Algorithm (a.k.a. UCA).

Constructor and Tailoring

The new method returns a collator object. If new() is called with no parameters, the collator should do the default collation.

```
$Collator = Unicode::Collate->new(
   UCA_Version => $UCA_Version,
   alternate => $alternate, # alias for 'variable'
   backwards => $levelNumber, # or \@levelNumbers
   entry => $element,
   hangul_terminator => $term_primary_weight,
   highestFFFF => $bool,
   identical => $bool,
   ignoreName => qr/$ignoreName/,
   ignoreChar => qr/$ignoreChar/,
   ignore_level2 => $bool,
   katakana_before_hiragana => $bool,
   level => $collationLevel,
```

(continues on next page)

(continued from previous page)

```
   long_contraction => $bool,
   minimalFFFE => $bool,
   normalization  => $normalization_form,
   overrideCJK => \&overrideCJK,
   overrideHangul => \&overrideHangul,
   preprocess => \&preprocess,
   rearrange => \@charList,
   rewrite => \&rewrite,
   suppress => \@charList,
   table => $filename,
   undefName => qr/$undefName/,
   undefChar => qr/$undefChar/,
   upper_before_lower => $bool,
   variable => $variable,
);
```

- UCA_Version

  If the revision (previously "tracking version") number of UCA is given, behavior of that revision is emulated on collating. If omitted, the return value of `UCA_Version()` is used.

  The following revisions are supported. The default is 43.

```
  UCA       Unicode Standard          DUCET (@version)
--------------------------------------------------------
   8              3.1                 3.0.1 (3.0.1d9)
   9     3.1 with Corrigendum 3       3.1.1
  11             4.0.0
  14             4.1.0
  16             5.0.0
  18             5.1.0
  20             5.2.0
  22             6.0.0
  24             6.1.0
  26             6.2.0
  28             6.3.0
  30             7.0.0
  32             8.0.0
  34             9.0.0
  36            10.0.0
  38            11.0.0
  40            12.0.0
  41            12.1.0
  43            13.0.0
```

  * See below for `long_contraction` with `UCA_Version` 22 and 24.

  * Noncharacters (e.g. U+FFFF) are not ignored, and can be overridden since `UCA_Version` 22.

  * Out-of-range codepoints (greater than U+10FFFF) are not ignored, and can be

overridden since `UCA_Version` 22.

* Fully ignorable characters were ignored, and would not interrupt contractions with `UCA_Version` 9 and 11.

* Treatment of ignorables after variables and some behaviors were changed at `UCA_Version` 9.

* Characters regarded as CJK unified ideographs (cf. `overrideCJK`) depend on `UCA_Version`.

* Many hangul jamo are assigned at `UCA_Version` 20, that will affect `hangul_terminator`.

- alternate

  – see 3.2.2 Alternate Weighting, version 8 of UTS #10

  For backward compatibility, `alternate` (old name) can be used as an alias for `variable`.

- backwards

  – see 3.4 Backward Accents, UTS #10.

```
backwards => $levelNumber or \@levelNumbers
```

  Weights in reverse order; ex. level 2 (diacritic ordering) in French. If omitted (or `$levelNumber` is `undef` or `\@levelNumbers` is `[]`), forwards at all the levels.

- entry

  – see 5 Tailoring; 9.1 Allkeys File Format, UTS #10.

  If the same character (or a sequence of characters) exists in the collation element table through `table`, mapping to collation elements is overridden. If it does not exist, the mapping is defined additionally.

```
    entry => <<'ENTRY', # for DUCET v4.0.0 (allkeys-4.0.0.txt)
0063 0068 ; [.0E6A.0020.0002.0063] # ch
0043 0068 ; [.0E6A.0020.0007.0043] # Ch
0043 0048 ; [.0E6A.0020.0008.0043] # CH
006C 006C ; [.0F4C.0020.0002.006C] # ll
004C 006C ; [.0F4C.0020.0007.004C] # Ll
004C 004C ; [.0F4C.0020.0008.004C] # LL
00F1      ; [.0F7B.0020.0002.00F1] # n-tilde
006E 0303 ; [.0F7B.0020.0002.00F1] # n-tilde
00D1      ; [.0F7B.0020.0008.00D1] # N-tilde
004E 0303 ; [.0F7B.0020.0008.00D1] # N-tilde
ENTRY

    entry => <<'ENTRY', # for DUCET v4.0.0 (allkeys-4.0.0.txt)
00E6 ; [.0E33.0020.0002.00E6][.0E8B.0020.0002.00E6] # ae␣
 ↪ligature as <a><e>
00C6 ; [.0E33.0020.0008.00C6][.0E8B.0020.0008.00C6] # AE␣
 ↪ligature as <A><E>
ENTRY
```

NOTE: The code point in the UCA file format (before `';'`) **must** be a Unicode code point (defined as hexadecimal), but not a native code point. So `0063` must always denote `U+0063`, but not a character of `"\x63"`.

Weighting may vary depending on collation element table. So ensure the weights defined in `entry` will be consistent with those in the collation element table loaded via `table`.

In DUCET v4.0.0, primary weight of `C` is `0E60` and that of `D` is `0E6D`. So setting primary weight of `CH` to `0E6A` (as a value between `0E60` and `0E6D`) makes ordering as `C < CH < D`. Exactly speaking DUCET already has some characters between `C` and `D`: `small capital C` (`U+1D04`) with primary weight `0E64`, `c-hook/C-hook` (`U+0188/U+0187`) with `0E65`, and `c-curl` (`U+0255`) with `0E69`. Then primary weight `0E6A` for `CH` makes `CH` ordered between `c-curl` and `D`.

- hangul_terminator

  – see 7.1.4 Trailing Weights, UTS #10.

  If a true value is given (non-zero but should be positive), it will be added as a terminator primary weight to the end of every standard Hangul syllable. Secondary and any higher weights for terminator are set to zero. If the value is false or `hangul_terminator` key does not exist, insertion of terminator weights will not be performed.

  Boundaries of Hangul syllables are determined according to conjoining Jamo behavior in `the Unicode Standard` and `HangulSyllableType.txt`.

  Implementation Note: (1) For expansion mapping (Unicode character mapped to a sequence of collation elements), a terminator will not be added between collation elements, even if Hangul syllable boundary exists there. Addition of terminator is restricted to the next position to the last collation element.

  (2) Non-conjoining Hangul letters (Compatibility Jamo, halfwidth Jamo, and enclosed letters) are not automatically terminated with a terminator primary weight. These characters may need terminator included in a collation element table beforehand.

- highestFFFF

  – see 2.4 Tailored noncharacter weights, UTS #35 (LDML) Part 5: Collation.

  If the parameter is made true, `U+FFFF` has a highest primary weight. When a boolean of `$coll->ge($str, "abc")` and `$coll->le($str, "abc\x{FFFF}")` is true, it is expected that `$str` begins with `"abc"`, or another primary equivalent. `$str` may be `"abcd"`, `"abc012"`, but should not include `U+FFFF` such as `"abc\x{FFFF}xyz"`.

  `$coll->le($str, "abc\x{FFFF}")` works like `$coll->lt($str, "abd")` almost, but the latter has a problem that you should know which letter is next to `c`. For a certain language where `ch` as the next letter, `"abch"` is greater than `"abc\x{FFFF}"`, but less than `"abd"`.

  Note: This is equivalent to `(entry => 'FFFF ; [.FFFE.0020.0005.FFFF]')`. Any other character than `U+FFFF` can be tailored by `entry`.

- identical

  – see A.3 Deterministic Comparison, UTS #10.

By default, strings whose weights are equal should be equal, even though their code points are not equal. Completely ignorable characters are ignored.

If the parameter is made true, a final, tie-breaking level is used. If no difference of weights is found after the comparison through all the level specified by `level`, the comparison with code points will be performed. For the tie-breaking comparison, the sort key has code points of the original string appended. Completely ignorable characters are not ignored.

If `preprocess` and/or `normalization` is applied, the code points of the string after them (in NFD by default) are used.

- ignoreChar
- ignoreName

  – see 3.6 Variable Weighting, UTS #10.

  Makes the entry in the table completely ignorable; i.e. as if the weights were zero at all level.

  Through `ignoreChar`, any character matching `qr/$ignoreChar/` will be ignored. Through `ignoreName`, any character whose name (given in the `table` file as a comment) matches `qr/$ignoreName/` will be ignored.

  E.g. when 'a' and 'e' are ignorable, 'element' is equal to 'lament' (or 'lmnt').

- ignore_level2

  – see 5.1 Parametric Tailoring, UTS #10.

  By default, case-sensitive comparison (that is level 3 difference) won't ignore accents (that is level 2 difference).

  If the parameter is made true, accents (and other primary ignorable characters) are ignored, even though cases are taken into account.

  NOTE: `level` should be 3 or greater.

- katakana_before_hiragana

  – see 7.2 Tertiary Weight Table, UTS #10.

  By default, hiragana is before katakana. If the parameter is made true, this is reversed.

  NOTE: This parameter simplemindedly assumes that any hiragana/katakana distinctions must occur in level 3, and their weights at level 3 must be same as those mentioned in 7.3.1, UTS #10. If you define your collation elements which violate this requirement, this parameter does not work validly.

- level

  – see 4.3 Form Sort Key, UTS #10.

  Set the maximum level. Any higher levels than the specified one are ignored.

```
Level 1: alphabetic ordering
Level 2: diacritic ordering
Level 3: case ordering
Level 4: tie-breaking (e.g. in the case when variable is
```

(continues on next page)

(continued from previous page)

```
↪'shifted')

ex.level => 2,
```

If omitted, the maximum is the 4th.

NOTE: The DUCET includes weights over 0xFFFF at the 4th level. But this module only uses weights within 0xFFFF. When `variable` is 'blanked' or 'non-ignorable' (other than 'shifted' and 'shift-trimmed'), the level 4 may be unreliable.

See also `identical`.

- long_contraction

  – see 3.8.2 Well-Formedness of the DUCET, 4.2 Produce Array, UTS #10.

  If the parameter is made true, for a contraction with three or more characters (here nicknamed "long contraction"), initial substrings will be handled. For example, a contraction ABC, where A is a starter, and B and C are non-starters (character with non-zero combining character class), will be detected even if there is not AB as a contraction.

  Default: Usually false. If `UCA_Version` is 22 or 24, and the value of `long_contraction` is not specified in `new()`, a true value is set implicitly. This is a workaround to pass Conformance Tests for Unicode 6.0.0 and 6.1.0.

  `change()` handles `long_contraction` explicitly only. If `long_contraction` is not specified in `change()`, even though `UCA_Version` is changed, `long_contraction` will not be changed.

  Limitation: Scanning non-starters is one-way (no back tracking). If AB is found but not ABC is not found, other long contraction where the first character is A and the second is not B may not be found.

  Under `(normalization => undef)`, detection step of discontiguous contractions will be skipped.

  Note: The following contractions in DUCET are not considered in steps S2.1.1 to S2.1.3, where they are discontiguous.

```
0FB2 0F71 0F80 (TIBETAN VOWEL SIGN VOCALIC RR)
0FB3 0F71 0F80 (TIBETAN VOWEL SIGN VOCALIC LL)
```

  For example `TIBETAN VOWEL SIGN VOCALIC RR` with `COMBINING TILDE OVERLAY` (`U+0344`) is `0FB2 0344 0F71 0F80` in NFD. In this case `0FB2 0F80` (`TIBETAN VOWEL SIGN VOCALIC R`) is detected, instead of `0FB2 0F71 0F80`. Inserted `0344` makes `0FB2 0F71 0F80` discontiguous and lack of contraction `0FB2 0F71` prohibits `0FB2 0F71 0F80` from being detected.

- minimalFFFE

  – see 1.1.1 U+FFFE, UTS #35 (LDML) Part 5: Collation.

  If the parameter is made true, `U+FFFE` has a minimal primary weight. The comparison between `"$a1\x{FFFE}$a2"` and `"$b1\x{FFFE}$b2"` first compares `$a1` and `$b1` at level 1, and then `$a2` and `$b2` at level 1, as followed.

```
"ab\x{FFFE}a"
"Ab\x{FFFE}a"
"ab\x{FFFE}c"
"Ab\x{FFFE}c"
"ab\x{FFFE}xyz"
"abc\x{FFFE}def"
"abc\x{FFFE}xYz"
"aBc\x{FFFE}xyz"
"abcX\x{FFFE}def"
"abcx\x{FFFE}xyz"
"b\x{FFFE}aaa"
"bbb\x{FFFE}a"
```

Note: This is equivalent to `(entry => 'FFFE ; [.0001.0020.0005.FFFE]')`. Any other character than `U+FFFE` can be tailored by `entry`.

- normalization

  – see 4.1 Normalize, UTS #10.

  If specified, strings are normalized before preparation of sort keys (the normalization is executed after preprocess).

  A form name `Unicode::Normalize::normalize()` accepts will be applied as `$normalization_form`. Acceptable names include `'NFD'`, `'NFC'`, `'NFKD'`, and `'NFKC'`. See `Unicode::Normalize::normalize()` for detail. If omitted, `'NFD'` is used.

  `normalization` is performed after `preprocess` (if defined).

  Furthermore, special values, `undef` and `"prenormalized"`, can be used, though they are not concerned with `Unicode::Normalize::normalize()`.

  If `undef` (not a string `"undef"`) is passed explicitly as the value for this key, any normalization is not carried out (this may make tailoring easier if any normalization is not desired). Under `(normalization => undef)`, only contiguous contractions are resolved; e.g. even if `A-ring` (and `A-ring-cedilla`) is ordered after `Z`, `A-cedilla-ring` would be primary equal to `A`. In this point, `(normalization => undef, preprocess => sub { NFD(shift) })` **is not** equivalent to `(normalization => 'NFD')`.

  In the case of `(normalization => "prenormalized")`, any normalization is not performed, but discontiguous contractions with combining characters are performed. Therefore `(normalization => 'prenormalized', preprocess => sub { NFD(shift) })` **is** equivalent to `(normalization => 'NFD')`. If source strings are finely prenormalized, `(normalization => 'prenormalized')` may save time for normalization.

  Except `(normalization => undef)`, **Unicode::Normalize** is required (see also **CAVEAT**).

- overrideCJK

  – see 7.1 Derived Collation Elements, UTS #10.

  By default, CJK unified ideographs are ordered in Unicode codepoint order, but those in the CJK Unified Ideographs block are less than those in the CJK Unified

Ideographs Extension A etc.

```
In the CJK Unified Ideographs block:
U+4E00..U+9FA5 if UCA_Version is 8, 9 or 11.
U+4E00..U+9FBB if UCA_Version is 14 or 16.
U+4E00..U+9FC3 if UCA_Version is 18.
U+4E00..U+9FCB if UCA_Version is 20 or 22.
U+4E00..U+9FCC if UCA_Version is 24 to 30.
U+4E00..U+9FD5 if UCA_Version is 32 or 34.
U+4E00..U+9FEA if UCA_Version is 36.
U+4E00..U+9FEF if UCA_Version is 38, 40 or 41.
U+4E00..U+9FFC if UCA_Version is 43.

In the CJK Unified Ideographs Extension blocks:
Ext.A (U+3400..U+4DB5)   if UCA_Version is  8 to 41.
Ext.A (U+3400..U+4DBF)   if UCA_Version is 43.
Ext.B (U+20000..U+2A6D6) if UCA_Version is  8 to 41.
Ext.B (U+20000..U+2A6DD) if UCA_Version is 43.
Ext.C (U+2A700..U+2B734) if UCA_Version is 20 or later.
Ext.D (U+2B740..U+2B81D) if UCA_Version is 22 or later.
Ext.E (U+2B820..U+2CEA1) if UCA_Version is 32 or later.
Ext.F (U+2CEB0..U+2EBE0) if UCA_Version is 36 or later.
Ext.G (U+30000..U+3134A) if UCA_Version is 43.
```

Through `overrideCJK`, ordering of CJK unified ideographs (including extensions) can be overridden.

ex. CJK unified ideographs in the JIS code point order.

```
overrideCJK => sub {
    my $u = shift;              # get a Unicode codepoint
    my $b = pack('n', $u);      # to UTF-16BE
    my $s = your_unicode_to_sjis_converter($b); # convert
    my $n = unpack('n', $s);    # convert sjis to short
    [ $n, 0x20, 0x2, $u ];      # return the collation element
},
```

The return value may be an arrayref of 1st to 4th weights as shown above. The return value may be an integer as the primary weight as shown below. If `undef` is returned, the default derived collation element will be used.

```
overrideCJK => sub {
    my $u = shift;              # get a Unicode codepoint
    my $b = pack('n', $u);      # to UTF-16BE
    my $s = your_unicode_to_sjis_converter($b); # convert
    my $n = unpack('n', $s);    # convert sjis to short
    return $n;                  # return the primary weight
},
```

The return value may be a list containing zero or more of an arrayref, an integer, or `undef`.

ex. ignores all CJK unified ideographs.

```
overrideCJK => sub {()}, # CODEREF returning empty list

 # where ->eq("Pe\x{4E00}rl", "Perl") is true
 # as U+4E00 is a CJK unified ideograph and to be ignorable.
```

If a false value (including undef) is passed, overrideCJK has no effect. `$Collator->change(overrideCJK => 0)` resets the old one.

But assignment of weight for CJK unified ideographs in table or entry is still valid. If undef is passed explicitly as the value for this key, weights for CJK unified ideographs are treated as undefined. However when UCA_Version > 8, `(overrideCJK => undef)` has no special meaning.

Note: In addition to them, 12 CJK compatibility ideographs (U+FA0E, U+FA0F, U+FA11, U+FA13, U+FA14, U+FA1F, U+FA21, U+FA23, U+FA24, U+FA27, U+FA28, U+FA29) are also treated as CJK unified ideographs. But they can't be overridden via overrideCJK when you use DUCET, as the table includes weights for them. table or entry has priority over overrideCJK.

- overrideHangul

  – see 7.1 Derived Collation Elements, UTS #10.

  By default, Hangul syllables are decomposed into Hangul Jamo, even if `(normalization => undef)`. But the mapping of Hangul syllables may be overridden.

  This parameter works like overrideCJK, so see there for examples.

  If you want to override the mapping of Hangul syllables, NFD and NFKD are not appropriate, since NFD and NFKD will decompose Hangul syllables before overriding. FCD may decompose Hangul syllables as the case may be.

  If a false value (but not undef) is passed, overrideHangul has no effect. `$Collator->change(overrideHangul => 0)` resets the old one.

  If undef is passed explicitly as the value for this key, weight for Hangul syllables is treated as undefined without decomposition into Hangul Jamo. But definition of weight for Hangul syllables in table or entry is still valid.

- overrideOut

  – see 7.1.1 Handling Ill-Formed Code Unit Sequences, UTS #10.

  Perl seems to allow out-of-range values (greater than 0x10FFFF). By default, out-of-range values are replaced with U+FFFD (REPLACEMENT CHARACTER) when UCA_Version >= 22, or ignored when UCA_Version <= 20.

  When UCA_Version >= 22, the weights of out-of-range values can be overridden. Though table or entry are available for them, out-of-range values are too many.

  overrideOut can perform it algorithmically. This parameter works like overrideCJK, so see there for examples.

  ex. ignores all out-of-range values.

```
overrideOut => sub {()}, # CODEREF returning empty list
```

If a false value (including undef) is passed, overrideOut has no effect. $Collator->change(overrideOut => 0) resets the old one.

NOTE ABOUT U+FFFD:

UCA recommends that out-of-range values should not be ignored for security reasons. Say, "pe\x{110000}rl" should not be equal to "perl". However, U+FFFD is wrongly mapped to a variable collation element in DUCET for Unicode 6.0.0 to 6.2.0, that means out-of-range values will be ignored when variable isn't Non-ignorable.

The mapping of U+FFFD is corrected in Unicode 6.3.0. see http://www.unicode.org/reports/tr10/tr10-28.html#Trailing_Weights (7.1.4 Trailing Weights). Such a correction is reproduced by this.

```
overrideOut => sub { 0xFFFD }, # CODEREF returning a very large␣
↪integer
```

This workaround is unnecessary since Unicode 6.3.0.

- preprocess

  – see 5.4 Preprocessing, UTS #10.

  If specified, the coderef is used to preprocess each string before the formation of sort keys.

  ex. dropping English articles, such as "a" or "the". Then, "the pen" is before "a pencil".

```
preprocess => sub {
      my $str = shift;
      $str =~ s/\b(?:an?|the)\s+//gi;
      return $str;
   },
```

  preprocess is performed before normalization (if defined).

  ex. decoding strings in a legacy encoding such as shift-jis:

```
$sjis_collator = Unicode::Collate->new(
    preprocess => \&your_shiftjis_to_unicode_decoder,
);
@result = $sjis_collator->sort(@shiftjis_strings);
```

  Note: Strings returned from the coderef will be interpreted according to Perl's Unicode support. See perlunicode, perluniintro, perlunitut, perlunifaq, *utf8*.

- rearrange

  – see 3.5 Rearrangement, UTS #10.

  Characters that are not coded in logical order and to be rearranged. If UCA_Version is equal to or less than 11, default is:

```
rearrange => [ 0x0E40..0x0E44, 0x0EC0..0x0EC4 ],
```

If you want to disallow any rearrangement, pass `undef` or `[]` (a reference to empty list) as the value for this key.

If `UCA_Version` is equal to or greater than 14, default is `[]` (i.e. no rearrangement).

According to the version 9 of UCA, this parameter shall not be used; but it is not warned at present.

- rewrite

  If specified, the coderef is used to rewrite lines in `table` or `entry`. The coderef will get each line, and then should return a rewritten line according to the UCA file format. If the coderef returns an empty line, the line will be skipped.

  e.g. any primary ignorable characters into tertiary ignorable:

```
rewrite => sub {
    my $line = shift;
    $line =~ s/\[\.0000\..{4}\..{4}\./[.0000.0000.0000./g;
    return $line;
},
```

  This example shows rewriting weights. `rewrite` is allowed to affect code points, weights, and the name.

  NOTE: `table` is available to use another table file; preparing a modified table once would be more efficient than rewriting lines on reading an unmodified table every time.

- suppress

  – see 3.12 Special-Purpose Commands, UTS #35 (LDML) Part 5: Collation.

  Contractions beginning with the specified characters are suppressed, even if those contractions are defined in `table`.

  An example for Russian and some languages using the Cyrillic script:

```
suppress => [0x0400..0x0417, 0x041A..0x0437, 0x043A..0x045F],
```

  where 0x0400 stands for `U+0400`, CYRILLIC CAPITAL LETTER IE WITH GRAVE.

  NOTE: Contractions via `entry` will not be suppressed.

- table

  – see 3.8 Default Unicode Collation Element Table, UTS #10.

  You can use another collation element table if desired.

  The table file should locate in the `Unicode/Collate` directory on `@INC`. Say, if the filename is `Foo.txt`, the table file is searched as `Unicode/Collate/Foo.txt` in `@INC`.

  By default, `allkeys.txt` (as the filename of DUCET) is used. If you will prepare your own table file, any name other than `allkeys.txt` may be better to avoid namespace conflict.

Explicit saying `(table => 'allkeys.txt')`, or using another table, or using `ignoreChar`, `ignoreName`, `undefChar`, `undefName` or `rewrite` will prevent this module from using the compiled DUCET.

If `undef` is passed explicitly as the value for this key, no file is read (but you can define collation elements via `entry`).

A typical way to define a collation element table without any file of table:

```
   $onlyABC = Unicode::Collate->new(
       table => undef,
       entry => << 'ENTRIES',
0061 ; [.0101.0020.0002.0061] # LATIN SMALL LETTER A
0041 ; [.0101.0020.0008.0041] # LATIN CAPITAL LETTER A
0062 ; [.0102.0020.0002.0062] # LATIN SMALL LETTER B
0042 ; [.0102.0020.0008.0042] # LATIN CAPITAL LETTER B
0063 ; [.0103.0020.0002.0063] # LATIN SMALL LETTER C
0043 ; [.0103.0020.0008.0043] # LATIN CAPITAL LETTER C
ENTRIES
    );
```

If `ignoreName` or `undefName` is used, character names should be specified as a comment (following #) on each line.

- undefChar
- undefName

  – see 6.3.3 Reducing the Repertoire, UTS #10.

  Undefines the collation element as if it were unassigned in the `table`. This reduces the size of the table. If an unassigned character appears in the string to be collated, the sort key is made from its codepoint as a single-character collation element, as it is greater than any other assigned collation elements (in the codepoint order among the unassigned characters). But, it'd be better to ignore characters unfamiliar to you and maybe never used.

  Through `undefChar`, any character matching `qr/$undefChar/` will be undefined. Through `undefName`, any character whose name (given in the `table` file as a comment) matches `qr/$undefName/` will be undefined.

  ex. Collation weights for beyond-BMP characters are not stored in object:

```
undefChar => qr/[^\0-\x{fffd}]/,
```

- upper_before_lower

  – see 6.6 Case Comparisons, UTS #10.

  By default, lowercase is before uppercase. If the parameter is made true, this is reversed.

  NOTE: This parameter simplemindedly assumes that any lowercase/uppercase distinctions must occur in level 3, and their weights at level 3 must be same as those mentioned in 7.3.1, UTS #10. If you define your collation elements which differs from this requirement, this parameter doesn't work validly.

- variable

  – see 3.6 Variable Weighting, UTS #10.

  This key allows for variable weighting of variable collation elements, which are marked with an ASTERISK in the table (NOTE: Many punctuation marks and symbols are variable in `allkeys.txt`).

```
variable => 'blanked', 'non-ignorable', 'shifted', or 'shift-
↪trimmed'.
```

  These names are case-insensitive. By default (if specification is omitted), 'shifted' is adopted.

```
'Blanked'        Variable elements are made ignorable at levels␣
↪1 through 3;
                 considered at the 4th level.

'Non-Ignorable'  Variable elements are not reset to ignorable.

'Shifted'        Variable elements are made ignorable at levels␣
↪1 through 3
                 their level 4 weight is replaced by the old␣
↪level 1 weight.
                 Level 4 weight for Non-Variable elements is␣
↪0xFFFF.

'Shift-Trimmed'  Same as 'shifted', but all FFFF's at the 4th␣
↪level
                 are trimmed.
```

Methods for Collation

- `@sorted = $Collator->sort(@not_sorted)`

  Sorts a list of strings.

- `$result = $Collator->cmp($a, $b)`

  Returns 1 (when `$a` is greater than `$b`) or 0 (when `$a` is equal to `$b`) or -1 (when `$a` is less than `$b`).

- `$result = $Collator->eq($a, $b)`
- `$result = $Collator->ne($a, $b)`
- `$result = $Collator->lt($a, $b)`
- `$result = $Collator->le($a, $b)`
- `$result = $Collator->gt($a, $b)`
- `$result = $Collator->ge($a, $b)`

  They works like the same name operators as theirs.

```
eq : whether $a is equal to $b.
ne : whether $a is not equal to $b.
lt : whether $a is less than $b.
le : whether $a is less than $b or equal to $b.
gt : whether $a is greater than $b.
ge : whether $a is greater than $b or equal to $b.
```

- `$sortKey = $Collator->getSortKey($string)`

  – see 4.3 Form Sort Key, UTS #10.

  Returns a sort key.

  You compare the sort keys using a binary comparison and get the result of the comparison of the strings using UCA.

```
$Collator->getSortKey($a) cmp $Collator->getSortKey($b)

   is equivalent to

$Collator->cmp($a, $b)
```

- `$sortKeyForm = $Collator->viewSortKey($string)`

  Converts a sorting key into its representation form. If `UCA_Version` is 8, the output is slightly different.

```
use Unicode::Collate;
my $c = Unicode::Collate->new();
print $c->viewSortKey("Perl"),"\n";

# output:
# [0B67 0A65 0B7F 0B03 | 0020 0020 0020 0020 | 0008 0002 0002
 ↪0002 | FFFF FFFF FFFF FFFF]
#  Level 1              Level 2              Level 3
 ↪     Level 4
```

Methods for Searching

The `match`, `gmatch`, `subst`, `gsubst` methods work like `m//`, `m//g`, `s///`, `s///g`, respectively, but they are not aware of any pattern, but only a literal substring.

DISCLAIMER: If `preprocess` or `normalization` parameter is true for `$Collator`, calling these methods (`index`, `match`, `gmatch`, `subst`, `gsubst`) is croaked, as the position and the length might differ from those on the specified string.

`rearrange` and `hangul_terminator` parameters are neglected. `katakana_before_hiragana` and `upper_before_lower` don't affect matching and searching, as it doesn't matter whether greater or less.

- `$position = $Collator->index($string, $substring[, $position])`
- `($position, $length) = $Collator->index($string, $substring[, $position])`

If `$substring` matches a part of `$string`, returns the position of the first occurrence of the matching part in scalar context; in list context, returns a two-element list of the position and the length of the matching part.

If `$substring` does not match any part of `$string`, returns `-1` in scalar context and an empty list in list context.

e.g. when the content of `$str` is `"Ich muß studieren Perl."`, you say the following where `$sub` is `"MüSS"`,

```
my $Collator = Unicode::Collate->new( normalization => undef,␣
↪level => 1 );
                                    # (normalization => undef)␣
↪is REQUIRED.
my $match;
if (my($pos,$len) = $Collator->index($str, $sub)) {
    $match = substr($str, $pos, $len);
}
```

and get `"muß"` in `$match`, since `"muß"` is primary equal to `"MüSS"`.

- `$match_ref = $Collator->match($string, $substring)`
- `($match) = $Collator->match($string, $substring)`

If `$substring` matches a part of `$string`, in scalar context, returns **a reference to** the first occurrence of the matching part (`$match_ref` is always true if matches, since every reference is **true**); in list context, returns the first occurrence of the matching part.

If `$substring` does not match any part of `$string`, returns `undef` in scalar context and an empty list in list context.

e.g.

```
if ($match_ref = $Collator->match($str, $sub)) { # scalar␣
↪context
    print "matches [$$match_ref].\n";
} else {
    print "doesn't match.\n";
}

 or

if (($match) = $Collator->match($str, $sub)) { # list context
    print "matches [$match].\n";
} else {
    print "doesn't match.\n";
}
```

- `@match = $Collator->gmatch($string, $substring)`

If `$substring` matches a part of `$string`, returns all the matching parts (or matching count in scalar context).

If `$substring` does not match any part of `$string`, returns an empty list.

- `$count = $Collator->subst($string, $substring, $replacement)`

  If `$substring` matches a part of `$string`, the first occurrence of the matching part is replaced by `$replacement` (`$string` is modified) and `$count` (always equals to 1) is returned.

  `$replacement` can be a CODEREF, taking the matching part as an argument, and returning a string to replace the matching part (a bit similar to `s/(..)/$coderef->($1)/e`).

- `$count = $Collator->gsubst($string, $substring, $replacement)`

  If `$substring` matches a part of `$string`, all the occurrences of the matching part are replaced by `$replacement` (`$string` is modified) and `$count` is returned.

  `$replacement` can be a CODEREF, taking the matching part as an argument, and returning a string to replace the matching part (a bit similar to `s/(..)/$coderef->($1)/eg`).

  e.g.

```
my $Collator = Unicode::Collate->new( normalization => undef,␣
↪level => 1 );
                                    # (normalization => undef)␣
↪is REQUIRED.
my $str = "Camel donkey zebra came\x{301}l CAMEL horse cam\0e\
↪0l...";
$Collator->gsubst($str, "camel", sub { "<b>$_[0]</b>" });

# now $str is "<b>Camel</b> donkey zebra <b>came\x{301}l</b> <b>
↪CAMEL</b> horse <b>cam\0e\0l</b>...";
# i.e., all the camels are made bold-faced.

 Examples: levels and ignore_level2 - what does camel match?
-------------------------------------------------------------
↪-----------
 level  ignore_level2  |  camel  Camel  came\x{301}l  c-a-m-e-l␣
↪ cam\0e\0l
----------------------|--------------------------------------
↪-----------
   1        false      |   yes    yes      yes          yes     ␣
↪   yes
   2        false      |   yes    yes      no           yes     ␣
↪   yes
   3        false      |   yes    no       no           yes     ␣
↪   yes
   4        false      |   yes    no       no           no      ␣
↪   yes
----------------------|--------------------------------------
↪-----------
   1        true       |   yes    yes      yes          yes     ␣
↪   yes
   2        true       |   yes    yes      yes          yes     ␣
↪   yes
```

(continues on next page)

(continued from previous page)

```
   3        true       |   yes    no       yes          yes
↪   yes
   4        true       |   yes    no       yes          no
↪   yes
-------------------------------------------------------------------
↪-----------
 note: if variable => non-ignorable, camel doesn't match c-a-m-
↪e-l
       at any level.
```

Other Methods

- `%old_tailoring = $Collator->change(%new_tailoring)`
- `$modified_collator = $Collator->change(%new_tailoring)`

  Changes the value of specified keys and returns the changed part.

```
$Collator = Unicode::Collate->new(level => 4);

$Collator->eq("perl", "PERL"); # false

%old = $Collator->change(level => 2); # returns (level => 4).

$Collator->eq("perl", "PERL"); # true

$Collator->change(%old); # returns (level => 2).

$Collator->eq("perl", "PERL"); # false
```

  Not all `(key,value)`s are allowed to be changed. See also `@Unicode::Collate::ChangeOK` and `@Unicode::Collate::ChangeNG`.

  In the scalar context, returns the modified collator (but it is **not** a clone from the original).

```
$Collator->change(level => 2)->eq("perl", "PERL"); # true

$Collator->eq("perl", "PERL"); # true; now max level is 2nd.

$Collator->change(level => 4)->eq("perl", "PERL"); # false
```

- `$version = $Collator->version()`

  Returns the version number (a string) of the Unicode Standard which the `table` file used by the collator object is based on. If the table does not include a version line (starting with `@version`), returns `"unknown"`.

- `UCA_Version()`

  Returns the revision number of UTS #10 this module consults, that should correspond with the DUCET incorporated.

- `Base_Unicode_Version()`

  Returns the version number of UTS #10 this module consults, that should correspond with the DUCET incorporated.

EXPORT

No method will be exported.

INSTALL

Though this module can be used without any `table` file, to use this module easily, it is recommended to install a table file in the UCA format, by copying it under the directory <a place in @INC>/Unicode/Collate.

The most preferable one is "The Default Unicode Collation Element Table" (aka DUCET), available from the Unicode Consortium's website:

```
http://www.unicode.org/Public/UCA/

http://www.unicode.org/Public/UCA/latest/allkeys.txt
(latest version)
```

If DUCET is not installed, it is recommended to copy the file from http://www.unicode.org/Public/UCA/latest/allkeys.txt to <a place in @INC>/Unicode/Collate/allkeys.txt manually.

CAVEATS

- Normalization

  Use of the `normalization` parameter requires the **Unicode::Normalize** module (see *Unicode::Normalize*).

  If you need not it (say, in the case when you need not handle any combining characters), assign `(normalization => undef)` explicitly.

  – see 6.5 Avoiding Normalization, UTS #10.

- Conformance Test

  The Conformance Test for the UCA is available under http://www.unicode.org/Public/UCA/.

  For `CollationTest_SHIFTED.txt`, a collator via `Unicode::Collate->new( )` should be used; for `CollationTest_NON_IGNORABLE.txt`, a collator via `Unicode::Collate->new(variable => "non-ignorable", level => 3)`.

  If `UCA_Version` is 26 or later, the `identical` level is preferred; `Unicode::Collate->new(identical => 1)` and `Unicode::Collate->new(identical => 1, variable => "non-ignorable", level => 3)` should be used.

  Unicode::Normalize is required to try The Conformance Test.

  EBCDIC-SUPPORT IS EXPERIMENTAL.

AUTHOR, COPYRIGHT AND LICENSE

The Unicode::Collate module for perl was written by SADAHIRO Tomoyuki, SADAHIRO@cpan.org. This module is Copyright(C) 2001-2021, SADAHIRO Tomoyuki. Japan. All rights reserved.

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

The file Unicode/Collate/allkeys.txt was copied verbatim from http://www.unicode.org/Public/UCA/13.0.0/allkeys.txt. For this file, Copyright (c) 2020 Unicode, Inc.; distributed under the Terms of Use in http://www.unicode.org/terms_of_use.html

SEE ALSO

- Unicode Collation Algorithm - UTS #10

  http://www.unicode.org/reports/tr10/

- The Default Unicode Collation Element Table (DUCET)

  http://www.unicode.org/Public/UCA/latest/allkeys.txt

- The conformance test for the UCA

  http://www.unicode.org/Public/UCA/latest/CollationTest.html

  http://www.unicode.org/Public/UCA/latest/CollationTest.zip

- Hangul Syllable Type

  http://www.unicode.org/Public/UNIDATA/HangulSyllableType.txt

- Unicode Normalization Forms - UAX #15

  http://www.unicode.org/reports/tr15/

- Unicode Locale Data Markup Language (LDML) - UTS #35

  http://www.unicode.org/reports/tr35/

Modules

- `Unicode::Collate::CJK`
- `Unicode::Collate::Locale` - Linguistic tailoring for DUCET via Unicode::Collate

Unicode::Collate::CJK

Namespace containing the following modules:

- `Unicode::Collate::CJK::Big5` - weighting CJK Unified Ideographs for Unicode::Collate
- `Unicode::Collate::CJK::GB2312` - weighting CJK Unified Ideographs for Unicode::Collate
- `Unicode::Collate::CJK::JISX0208` - weighting JIS KANJI for Unicode::Collate

- `Unicode::Collate::CJK::Korean` - weighting CJK Unified Ideographs for Unicode::Collate
- `Unicode::Collate::CJK::Pinyin` - weighting CJK Unified Ideographs for Unicode::Collate
- `Unicode::Collate::CJK::Stroke` - weighting CJK Unified Ideographs for Unicode::Collate
- `Unicode::Collate::CJK::Zhuyin` - weighting CJK Unified Ideographs for Unicode::Collate

Unicode::Collate::CJK::Big5

weighting CJK Unified Ideographs for Unicode::Collate

SYNOPSIS

```
use Unicode::Collate;
use Unicode::Collate::CJK::Big5;

my $collator = Unicode::Collate->new(
    overrideCJK => \&Unicode::Collate::CJK::Big5::weightBig5
);
```

DESCRIPTION

`Unicode::Collate::CJK::Big5` provides `weightBig5()`, that is adequate for `overrideCJK` of `Unicode::Collate` and makes tailoring of CJK Unified Ideographs in the order of CLDR's big5han ordering.

SEE ALSO

- CLDR - Unicode Common Locale Data Repository

  http://cldr.unicode.org/
- Unicode Locale Data Markup Language (LDML) - UTS #35

  http://www.unicode.org/reports/tr35/
- *Unicode::Collate*
- *Unicode::Collate::Locale*

Unicode::Collate::CJK::GB2312

weighting CJK Unified Ideographs for Unicode::Collate

SYNOPSIS

```
use Unicode::Collate;
use Unicode::Collate::CJK::GB2312;

my $collator = Unicode::Collate->new(
    overrideCJK => \&Unicode::Collate::CJK::GB2312::weightGB2312
);
```

DESCRIPTION

`Unicode::Collate::CJK::GB2312` provides `weightGB2312()`, that is adequate for `overrideCJK` of `Unicode::Collate` and makes tailoring of CJK Unified Ideographs in the order of CLDR's gb2312han ordering.

CAVEAT

The gb2312han ordering includes 5 code points in private use area (E2D8..E2DC), that can't utilize `weightGB2312()` for collation. For them, use `entry` instead.

SEE ALSO

- CLDR - Unicode Common Locale Data Repository

  http://cldr.unicode.org/

- Unicode Locale Data Markup Language (LDML) - UTS #35

  http://www.unicode.org/reports/tr35/

- *Unicode::Collate*
- *Unicode::Collate::Locale*

Unicode::Collate::CJK::JISX0208

weighting JIS KANJI for Unicode::Collate

SYNOPSIS

```
use Unicode::Collate;
use Unicode::Collate::CJK::JISX0208;

my $collator = Unicode::Collate->new(
    overrideCJK => \&Unicode::Collate::CJK::JISX0208::weightJISX0208
);
```

DESCRIPTION

`Unicode::Collate::CJK::JISX0208` provides `weightJISX0208()`, that is adequate for `overrideCJK` of `Unicode::Collate` and makes tailoring of 6355 kanji (CJK Unified Ideographs) in the JIS X 0208 order.

SEE ALSO

- *Unicode::Collate*
- *Unicode::Collate::Locale*

Unicode::Collate::CJK::Korean

weighting CJK Unified Ideographs for Unicode::Collate

SYNOPSIS

```
use Unicode::Collate;
use Unicode::Collate::CJK::Korean;

my $collator = Unicode::Collate->new(
    overrideCJK => \&Unicode::Collate::CJK::Korean::weightKorean
);
```

DESCRIPTION

`Unicode::Collate::CJK::Korean` provides `weightKorean()`, that is adequate for `overrideCJK` of `Unicode::Collate` and makes tailoring of CJK Unified Ideographs in the order of CLDR's Korean ordering.

SEE ALSO

- CLDR - Unicode Common Locale Data Repository

  http://cldr.unicode.org/
- Unicode Locale Data Markup Language (LDML) - UTS #35

  http://www.unicode.org/reports/tr35/
- *Unicode::Collate*
- *Unicode::Collate::Locale*

Unicode::Collate::CJK::Pinyin

weighting CJK Unified Ideographs for Unicode::Collate

SYNOPSIS

```
use Unicode::Collate;
use Unicode::Collate::CJK::Pinyin;

my $collator = Unicode::Collate->new(
    overrideCJK => \&Unicode::Collate::CJK::Pinyin::weightPinyin
);
```

DESCRIPTION

`Unicode::Collate::CJK::Pinyin` provides `weightPinyin()`, that is adequate for `overrideCJK` of `Unicode::Collate` and makes tailoring of CJK Unified Ideographs in the order of CLDR's pinyin ordering.

CAVEAT

The pinyin ordering includes some characters that are not CJK Unified Ideographs and can't utilize `weightPinyin()` for collation. For them, use `entry` instead.

SEE ALSO

- CLDR - Unicode Common Locale Data Repository

  http://cldr.unicode.org/

- Unicode Locale Data Markup Language (LDML) - UTS #35

  http://www.unicode.org/reports/tr35/

- *Unicode::Collate*
- *Unicode::Collate::Locale*

Unicode::Collate::CJK::Stroke

weighting CJK Unified Ideographs for Unicode::Collate

SYNOPSIS

```
use Unicode::Collate;
use Unicode::Collate::CJK::Stroke;

my $collator = Unicode::Collate->new(
    overrideCJK => \&Unicode::Collate::CJK::Stroke::weightStroke
);
```

DESCRIPTION

`Unicode::Collate::CJK::Stroke` provides `weightStroke()`, that is adequate for `overrideCJK` of `Unicode::Collate` and makes tailoring of CJK Unified Ideographs in the order of CLDR's stroke ordering.

CAVEAT

The stroke ordering includes some characters that are not CJK Unified Ideographs and can't utilize `weightStroke()` for collation. For them, use `entry` instead.

SEE ALSO

- CLDR - Unicode Common Locale Data Repository

  http://cldr.unicode.org/

- Unicode Locale Data Markup Language (LDML) - UTS #35

  http://www.unicode.org/reports/tr35/

- *Unicode::Collate*
- *Unicode::Collate::Locale*

Unicode::Collate::CJK::Zhuyin

weighting CJK Unified Ideographs for Unicode::Collate

SYNOPSIS

```
use Unicode::Collate;
use Unicode::Collate::CJK::Zhuyin;

my $collator = Unicode::Collate->new(
    overrideCJK => \&Unicode::Collate::CJK::Zhuyin::weightZhuyin
);
```

DESCRIPTION

`Unicode::Collate::CJK::Zhuyin` provides `weightZhuyin()`, that is adequate for `overrideCJK` of `Unicode::Collate` and makes tailoring of CJK Unified Ideographs in the order of CLDR's zhuyin (bopomofo) ordering.

CAVEAT

The zhuyin ordering includes some characters that are not CJK Unified Ideographs and can't utilize `weightZhuyin()` for collation. For them, use `entry` instead.

SEE ALSO

- CLDR - Unicode Common Locale Data Repository

  http://cldr.unicode.org/
- Unicode Locale Data Markup Language (LDML) - UTS #35

  http://www.unicode.org/reports/tr35/
- *Unicode::Collate*
- *Unicode::Collate::Locale*

Unicode::Collate::Locale

Linguistic tailoring for DUCET via Unicode::Collate

SYNOPSIS

```
use Unicode::Collate::Locale;

#construct
$Collator = Unicode::Collate::Locale->
    new(locale => $locale_name, %tailoring);

#sort
@sorted = $Collator->sort(@not_sorted);

#compare
$result = $Collator->cmp($a, $b); # returns 1, 0, or -1.
```

Note: Strings in `@not_sorted`, `$a` and `$b` are interpreted according to Perl's Unicode support. See perlunicode, perluniintro, perlunitut, perlunifaq, *utf8*. Otherwise you can use `preprocess` (cf. `Unicode::Collate`) or should decode them before.

DESCRIPTION

This module provides linguistic tailoring for it taking advantage of `Unicode::Collate`.

Constructor

The new method returns a collator object.

A parameter list for the constructor is a hash, which can include a special key `locale` and its value (case-insensitive) standing for a Unicode base language code (two or three-letter). For example, `Unicode::Collate::Locale->new(locale => 'ES')` returns a collator tailored for Spanish.

`$locale_name` may be suffixed with a Unicode script code (four-letter), a Unicode region (territory) code, a Unicode language variant code. These codes are case-insensitive, and separated with `'_'` or `'-'`. E.g. `en_US` for English in USA, `az_Cyrl` for Azerbaijani in the Cyrillic script, `es_ES_traditional` for Spanish in Spain (Traditional).

If `$locale_name` is not available, fallback is selected in the following order:

```
1. language with a variant code
2. language with a script code
3. language with a region code
4. language
5. default
```

Tailoring tags provided by `Unicode::Collate` are allowed as long as they are not used for `locale` support. Esp. the `table` tag is always untailorable, since it is reserved for DUCET.

However `entry` is allowed, even if it is used for `locale` support, to add or override mappings.

E.g. a collator for Spanish, which ignores diacritics and case difference (i.e. level 1), with reversed case ordering and no normalization.

```
Unicode::Collate::Locale->new(
    level => 1,
    locale => 'es',
    upper_before_lower => 1,
    normalization => undef
)
```

Overriding a behavior already tailored by `locale` is disallowed if such a tailoring is passed to `new()`.

```
Unicode::Collate::Locale->new(
    locale => 'da',
    upper_before_lower => 0, # causes error as reserved by 'da'
)
```

However `change()` inherited from `Unicode::Collate` allows such a tailoring that is reserved by `locale`. Examples:

```
new(locale => 'fr_ca')->change(backwards => undef)
new(locale => 'da')->change(upper_before_lower => 0)
new(locale => 'ja')->change(overrideCJK => undef)
```

Methods

`Unicode::Collate::Locale` is a subclass of `Unicode::Collate` and methods other than new are inherited from `Unicode::Collate`.

Here is a list of additional methods:

- `$Collator->getlocale`

  Returns a language code accepted and used actually on collation. If linguistic tailoring is not provided for a language code you passed (intensionally for some languages, or due to the incomplete implementation), this method returns a string `'default'` meaning no special tailoring.

- `$Collator->locale_version`

(Since Unicode::Collate::Locale 0.87) Returns the version number (perhaps `/\d\.\d\d/`) of the locale, as that of `Locale/*.pl`.

Note: `Locale/*.pl` that a collator uses should be identified by a combination of return values from `getlocale` and `locale_version`.

A list of tailorable locales

```
  locale name       description
------------------------------------------------------------
  af                Afrikaans
  ar                Arabic
  as                Assamese
  az                Azerbaijani (Azeri)
  be                Belarusian
  bn                Bengali
  bs                Bosnian (tailored as Croatian)
  bs_Cyrl           Bosnian in Cyrillic (tailored as Serbian)
  ca                Catalan
  cs                Czech
  cu                Church Slavic
  cy                Welsh
  da                Danish
  de__phonebook     German (umlaut as 'ae', 'oe', 'ue')
  de_AT_phonebook   Austrian German (umlaut primary greater)
  dsb               Lower Sorbian
  ee                Ewe
  eo                Esperanto
  es                Spanish
  es__traditional   Spanish ('ch' and 'll' as a grapheme)
  et                Estonian
  fa                Persian
  fi                Finnish (v and w are primary equal)
  fi__phonebook     Finnish (v and w as separate characters)
  fil               Filipino
  fo                Faroese
  fr_CA             Canadian French
  gu                Gujarati
  ha                Hausa
  haw               Hawaiian
  he                Hebrew
  hi                Hindi
  hr                Croatian
  hu                Hungarian
  hy                Armenian
  ig                Igbo
  is                Icelandic
  ja                Japanese [1]
  kk                Kazakh
  kl                Kalaallisut
  kn                Kannada
```

(continues on next page)

(continued from previous page)

```
  ko                Korean [2]
  kok               Konkani
  lkt               Lakota
  ln                Lingala
  lt                Lithuanian
  lv                Latvian
  mk                Macedonian
  ml                Malayalam
  mr                Marathi
  mt                Maltese
  nb                Norwegian Bokmal
  nn                Norwegian Nynorsk
  nso               Northern Sotho
  om                Oromo
  or                Oriya
  pa                Punjabi
  pl                Polish
  ro                Romanian
  sa                Sanskrit
  se                Northern Sami
  si                Sinhala
  si__dictionary    Sinhala (U+0DA5 = U+0DA2,0DCA,0DA4)
  sk                Slovak
  sl                Slovenian
  sq                Albanian
  sr                Serbian
  sr_Latn           Serbian in Latin (tailored as Croatian)
  sv                Swedish (v and w are primary equal)
  sv__reformed      Swedish (v and w as separate characters)
  ta                Tamil
  te                Telugu
  th                Thai
  tn                Tswana
  to                Tonga
  tr                Turkish
  ug_Cyrl           Uyghur in Cyrillic
  uk                Ukrainian
  ur                Urdu
  vi                Vietnamese
  vo                Volapu"k
  wae               Walser
  wo                Wolof
  yo                Yoruba
  zh                Chinese
  zh__big5han       Chinese (ideographs: big5 order)
  zh__gb2312han     Chinese (ideographs: GB-2312 order)
  zh__pinyin        Chinese (ideographs: pinyin order) [3]
  zh__stroke        Chinese (ideographs: stroke order) [3]
  zh__zhuyin        Chinese (ideographs: zhuyin order) [3]
```

(continues on next page)

(continued from previous page)

```
------------------------------------------------------------
```

Locales according to the default UCA rules include am (Amharic) without `[reorder Ethi]`, bg (Bulgarian) without `[reorder Cyrl]`, chr (Cherokee) without `[reorder Cher]`, de (German), en (English), fr (French), ga (Irish), id (Indonesian), it (Italian), ka (Georgian) without `[reorder Geor]`, mn (Mongolian) without `[reorder Cyrl Mong]`, ms (Malay), nl (Dutch), pt (Portuguese), ru (Russian) without `[reorder Cyrl]`, sw (Swahili), zu (Zulu).

Note

[1] ja: Ideographs are sorted in JIS X 0208 order. Fullwidth and halfwidth forms are identical to their regular form. The difference between hiragana and katakana is at the 4th level, the comparison also requires `(variable => 'Non-ignorable')`, and then `katakana_before_hiragana` has no effect.

[2] ko: Plenty of ideographs are sorted by their reading. Such an ideograph is primary (level 1) equal to, and secondary (level 2) greater than, the corresponding hangul syllable.

[3] zh__pinyin, zh__stroke and zh__zhuyin: implemented alt='short', where a smaller number of ideographs are tailored.

A list of variant codes and their aliases

```
  variant code       alias
-----------------------------------------
  dictionary         dict
  phonebook          phone     phonebk
  reformed           reform
  traditional        trad
-----------------------------------------
  big5han            big5
  gb2312han          gb2312
  pinyin
  stroke
  zhuyin
-----------------------------------------
```

Note: 'pinyin' is Han in Latin, 'zhuyin' is Han in Bopomofo.

INSTALL

Installation of `Unicode::Collate::Locale` requires `Collate/Locale.pm`, `Collate/Locale/*.pm`, `Collate/CJK/*.pm` and `Collate/allkeys.txt`. On building, `Unicode::Collate::Locale` doesn't require any of `data/*.txt`, `gendata/*`, and `mklocale`. Tests for `Unicode::Collate::Locale` are named `t/loc_*.t`.

CAVEAT

- Tailoring is not maximum

  Even if a certain letter is tailored, its equivalent would not always tailored as well as it. For example, even though W is tailored, fullwidth W (U+FF37), W with acute (U+1E82), etc. are not tailored. The result may depend on whether source strings are normalized or not, and whether decomposed or composed. Thus `(normalization => undef)` is less preferred.

- Collation reordering is not supported

  The order of any groups including scripts is not changed.

Reference

```
  locale             based CLDR or other reference
-----------------------------------------------------------------------
  af                 30 = 1.8.1
  ar                 30 = 28 ("compat" wo [reorder Arab]) = 1.9.0
  as                 30 = 28 (without [reorder Beng..]) = 23
  az                 30 = 24 ("standard" wo [reorder Latn Cyrl])
  be                 30 = 28 (without [reorder Cyrl])
  bn                 30 = 28 ("standard" wo [reorder Beng..]) = 2.0.1
  bs                 30 = 28 (type="standard": [import hr])
  bs_Cyrl            30 = 28 (type="standard": [import sr])
  ca                 30 = 23 (alt="proposed" type="standard")
  cs                 30 = 1.8.1 (type="standard")
  cu                 34 = 30 (without [reorder Cyrl])
  cy                 30 = 1.8.1
  da                 22.1 = 1.8.1 (type="standard")
  de__phonebook      30 = 2.0 (type="phonebook")
  de_AT_phonebook    30 = 27 (type="phonebook")
  dsb                30 = 26
  ee                 30 = 21
  eo                 30 = 1.8.1
  es                 30 = 1.9.0 (type="standard")
  es__traditional    30 = 1.8.1 (type="traditional")
  et                 30 = 26
  fa                 22.1 = 1.8.1
  fi                 22.1 = 1.8.1 (type="standard" alt="proposed")
  fi__phonebook      22.1 = 1.8.1 (type="phonebook")
  fil                30 = 1.9.0 (type="standard") = 1.8.1
  fo                 22.1 = 1.8.1 (alt="proposed" type="standard")
  fr_CA              30 = 1.9.0
  gu                 30 = 28 ("standard" wo [reorder Gujr..]) = 1.9.0
  ha                 30 = 1.9.0
  haw                30 = 24
  he                 30 = 28 (without [reorder Hebr]) = 23
  hi                 30 = 28 (without [reorder Deva..]) = 1.9.0
  hr                 30 = 28 ("standard" wo [reorder Latn Cyrl]) = 1.
↪9.0
```

(continues on next page)

(continued from previous page)

```
  hu                  22.1 = 1.8.1 (alt="proposed" type="standard")
  hy                  30 = 28 (without [reorder Armn]) = 1.8.1
  ig                  30 = 1.8.1
  is                  22.1 = 1.8.1 (type="standard")
  ja                  22.1 = 1.8.1 (type="standard")
  kk                  30 = 28 (without [reorder Cyrl])
  kl                  22.1 = 1.8.1 (type="standard")
  kn                  30 = 28 ("standard" wo [reorder Knda..]) = 1.9.0
  ko                  22.1 = 1.8.1 (type="standard")
  kok                 30 = 28 (without [reorder Deva..]) = 1.8.1
  lkt                 30 = 25
  ln                  30 = 2.0 (type="standard") = 1.8.1
  lt                  22.1 = 1.9.0
  lv                  22.1 = 1.9.0 (type="standard") = 1.8.1
  mk                  30 = 28 (without [reorder Cyrl])
  ml                  22.1 = 1.9.0
  mr                  30 = 28 (without [reorder Deva..]) = 1.8.1
  mt                  22.1 = 1.9.0
  nb                  22.1 = 2.0   (type="standard")
  nn                  22.1 = 2.0   (type="standard")
  nso              [*] 26 = 1.8.1
  om                  22.1 = 1.8.1
  or                  30 = 28 (without [reorder Orya..]) = 1.9.0
  pa                  22.1 = 1.8.1
  pl                  30 = 1.8.1
  ro                  30 = 1.9.0 (type="standard")
  sa               [*] 1.9.1 = 1.8.1 (type="standard" alt="proposed")
  se                  22.1 = 1.8.1 (type="standard")
  si                  30 = 28 ("standard" wo [reorder Sinh..]) = 1.9.0
  si__dictionary      30 = 28 ("dictionary" wo [reorder Sinh..]) = 1.
↪9.0
  sk                  22.1 = 1.9.0 (type="standard")
  sl                  22.1 = 1.8.1 (type="standard" alt="proposed")
  sq                  22.1 = 1.8.1 (alt="proposed" type="standard")
  sr                  30 = 28 (without [reorder Cyrl])
  sr_Latn             30 = 28 (type="standard": [import hr])
  sv                  22.1 = 1.9.0 (type="standard")
  sv__reformed        22.1 = 1.8.1 (type="reformed")
  ta                  22.1 = 1.9.0
  te                  30 = 28 (without [reorder Telu..]) = 1.9.0
  th                  22.1 = 22
  tn               [*] 26 = 1.8.1
  to                  22.1 = 22
  tr                  22.1 = 1.8.1 (type="standard")
  uk                  30 = 28 (without [reorder Cyrl])
  ug_Cyrl             https://en.wikipedia.org/wiki/Uyghur_Cyrillic_
↪alphabet
  ur                  22.1 = 1.9.0
  vi                  22.1 = 1.8.1
```

(continues on next page)

(continued from previous page)

```
  vo                30 = 25
  wae               30 = 2.0
  wo            [*] 1.9.1 = 1.8.1
  yo                30 = 1.8.1
  zh                22.1 = 1.8.1 (type="standard")
  zh__big5han       22.1 = 1.8.1 (type="big5han")
  zh__gb2312han     22.1 = 1.8.1 (type="gb2312han")
  zh__pinyin        22.1 = 2.0   (type='pinyin' alt='short')
  zh__stroke        22.1 = 1.9.1 (type='stroke' alt='short')
  zh__zhuyin        22.1 = 22    (type='zhuyin' alt='short')
----------------------------------------------------------------------
```

[*] http://www.unicode.org/repos/cldr/tags/latest/seed/collation/

AUTHOR

The Unicode::Collate::Locale module for perl was written by SADAHIRO Tomoyuki, SADAHIRO@cpan.org. This module is Copyright(C) 2004-2020, SADAHIRO Tomoyuki. Japan. All rights reserved.

This module is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

SEE ALSO

- Unicode Collation Algorithm - UTS #10

  http://www.unicode.org/reports/tr10/

- The Default Unicode Collation Element Table (DUCET)

  http://www.unicode.org/Public/UCA/latest/allkeys.txt

- Unicode Locale Data Markup Language (LDML) - UTS #35

  http://www.unicode.org/reports/tr35/

- CLDR - Unicode Common Locale Data Repository

  http://cldr.unicode.org/

- *Unicode::Collate*
- *Unicode::Normalize*

Unicode::GCString

Functions

Grapheme cluster strings

as_array

The grapheme clusters as one-cluster `Unicode::GCStrings` - what `@{$gcstring}` gives.

as_string

The string, as a Perl (UTF-8) string - what `"$gcstring"` gives.

chars

The number of characters (code points).

cmp

Compare with another string by code points, as `cmp` does - the overloaded `cmp`.

columns

The number of columns the string occupies (UAX #11).

concat

Join with another string, re-segmenting where they meet - the overloaded `.` and `.=`.

copy

A copy of the string.

eos

True when the iterator position is at the end.

flag

Get, or set, the break flag of cluster `i` (default: the current position).

item

Cluster `i` (default: the current position) as a string.

join

The strings given, joined with this one between them.

length

The number of grapheme clusters.

next

The cluster at the iterator position, advancing it; undef at the end - the overloaded `<>`.

pos

The iterator position, after setting it when an argument is given (negative counts from the end).

substr

`length` clusters from `offset`, as `substr` counts characters; with a replacement, those clusters are replaced in the string.

Overload glue

as_scalarref

A reference to the object's name, "Unicode::GCString(0x...)".

Properties

lbc

The line breaking class of the first grapheme base.

lbcext

The line breaking class of the last grapheme extender, or of its base when the extender has none of its own.

lbclass

Obsolete: the class of cluster `i`'s base; use `lbc`.

lbclass_ext

Obsolete: the class of cluster `i`'s extender; use `lbcext`.

Unicode::LineBreak

Line breaking for Unicode text, and strings of grapheme clusters.

`Unicode::LineBreak` folds text into lines following UAX #14, the Unicode Line Breaking Algorithm: it knows where a line may end in any script, keeps a column budget the way terminals and fixed-width fonts count it (UAX #11 East Asian widths), and lets you tailor classes, widths, line formatting and what happens to a word too long for a line. `Unicode::GCString` is the string type it works in: a sequence of grapheme clusters - what a reader sees as one character - with their columns and line breaking classes. `Text::LineFold` builds on both to fold and unfold plain text.

```
use Unicode::LineBreak;
use Unicode::GCString;

my $lb = Unicode::LineBreak->new(ColMax => 20);
```

(continues on next page)

(continued from previous page)

```
print $lb->break("The quick brown fox jumps over the lazy dog.\n");

my $s = Unicode::GCString->new("e\x{301}\x{65E5}\x{672C}");
print $s->length, " clusters, ", $s->columns, " columns\n";
# The quick brown fox
# jumps over the lazy
# dog.
# 3 clusters, 5 columns
```

Scripts written without spaces between words - Thai, Lao, Khmer, Myanmar - are broken between words with the dictionaries of the system's ICU library when it is installed (`ComplexBreaking`), and kept whole when it is not.

This is a native implementation of the XS half of Unicode-LineBreak 2019.001 (sombok 2.4.0); its `Unicode/LineBreak.pm`, `Unicode/GCString.pm` and `Text/LineFold.pm` are used unchanged.

Differences from upstream

- **Unicode version.** Classes, widths and clusters come from the interpreter's own Unicode tables (`UNICODE_VERSION` is 17.0.0), and the rules are Unicode 17's UAX #14 and UAX #29, the same perl's `\b{lb}` and `\X` follow. Upstream is frozen at Unicode 8.0.0. So emoji are wide, emoji modifier and ZWJ sequences are one cluster, Indic conjuncts join (UAX #29 GB9c), and the Unicode 9-17 classes (EB, EM, ZWJ, HH, AK, AP, AS, VF, VI) exist; their `LB_*` constants follow upstream's 40.
- `ViramaAsJoiner` is accepted and reported but changes nothing: what it did beyond UAX #29 is UAX #29 itself now (GB9c).
- SA text (Thai and its neighbours) is segmented by ICU's dictionaries instead of libthai, so `SouthEastAsian::supported()` names ICU and all four scripts, and Thai words may break differently.
- `LB_CJ` and `LB_XX` are the classes their names say. Upstream's constants swap them: its library numbers CJ 39 and XX 38 while the constants it exports give CJ 38 and XX 39.
- A string that is not valid UTF-8 where a string or GCString is expected (`$gc . "\xE9"`) is read with U+FFFD for each bad sequence; upstream loops forever.
- A `Prep` pattern that matches the empty string keeps the character after the match; upstream drops it.
- `break` breaks a whole text in one pass instead of 1000-character slices; the lines are the same.
- A setting call (`$lb->config(Key => $value)`) returns undef; upstream returns a null pointer to Perl.

Modules

- `Unicode::LineBreak::SouthEastAsian`

Functions

Line breaking

`lbrule`

Obsolete: the breaking action between two classes; use `breakingRule`.

`breakingRule`

The breaking action between the end of one string and the start of another: `MANDATORY`, `DIRECT` (a break may come there), `INDIRECT` (only if spaces come between them) or `PROHIBITED`.

`reset`

Forget a partially broken text, starting afresh.

`strsize`

Obsolete: `len` plus the columns of `spc` and `str`; use `Unicode::GCString::columns`.

`break`

Break a whole text into lines: in list context the lines as `Unicode::GCStrings`, in scalar context the text with line breaks.

`break_partial`

Break text arriving in pieces: each call returns the lines that are complete so far; `undef` ends the text and returns the rest.

Configuration

`copy`

A copy of the object: its settings, tailoring and any partially broken text.

Properties

`EAWidths`

The East_Asian_Width values, in the order of their `EA_*` constants.

LBClasses

The line breaking classes, in the order of their `LB_*` constants.

Overload glue

as_hashref

The object's user hash: what `$lb->{key}` reaches.

as_scalarref

A reference to the object's name, "Unicode::LineBreak(0x...)".

as_string

The object's name, "Unicode::LineBreak(0x...)".

Library info

UNICODE_VERSION

The Unicode version of the character data, "17.0.0".

SOMBOK_VERSION

The version of the sombok library this module reimplements.

Unicode::LineBreak::SouthEastAsian

Functions

Library info

supported

Which scripts written without spaces are broken into words, and by what: "Thai:ICU/78.3 Lao:ICU/78.3 ...", or undef without ICU.

Unicode::Normalize

`Unicode::Normalize` - the Unicode normalization forms NFD, NFC, NFKD and NFKC (UAX #15), and the FCD and FCC forms of UTN #5.

```
use Unicode::Normalize qw(NFD NFC NFKC normalize checkNFC␣
 ↪getCombinClass getCanon);

binmode STDOUT, ':encoding(UTF-8)';
my $s = "Cafe\x{301} \x{FB01}";              # e + combining acute,
 ↪"fi" ligature
printf "NFC  %s (%d chars)\n", NFC($s), length NFC($s);   # Café fi␣
```

(continues on next page)

(continued from previous page)

```
 ↪(6 chars)
printf "NFD  %d chars\n", length NFD("Caf\x{e9}");        # 5 chars
printf "NFKC %s\n", NFKC($s);                             # Café fi
print  "same: ", (normalize('C', $s) eq NFC($s) ? "yes" : "no"), "\n
 ↪";
print  "checkNFC: ", (defined checkNFC($s) ? checkNFC($s) || 'NO' :
 ↪'MAYBE'), "\n";
printf "class of U+0301: %d, U+00E9 = %vX\n", getCombinClass(0x301),␣
 ↪getCanon(0xE9);
# class of U+0301: 230, U+00E9 = 65.301
```

Two strings that look the same can be different sequences of characters: é is one character, U+00E9, or two, e and the combining acute U+0301. Normalizing both to the same form makes them compare equal. NFD and NFKD decompose every character as far as it goes; NFC and NFKC decompose and then compose again, giving the short form. The K forms also replace compatibility characters - ligatures, full-width forms, superscripts - by their plain equivalents, which loses formatting; NFC is the form to store and compare text in.

The data are those of the Unicode version perl itself was built with (17.0.0 for perl 5.44), so normalization agrees with perl's own character properties.

This is a native implementation of `Unicode::Normalize` 1.32; its `Unicode/Normalize.pm` is used unchanged. `use Unicode::Normalize` exports NFC, NFD, NFKC and NFKD; every other function named on this page is exported on request. The `.pm` adds, over these functions:

- `normalize($form, $string)`, where `$form` is C, D, KC, KD (or NFC ...), FCD or FCC, and `check($form, $string)` for the quick checks.
- `FCD($string)`: the string unchanged when it is already in FCD, else a string in FCD.
- `normalize_partial($form, $unprocessed)` and `NFD_partial`, `NFC_partial`, `NFKD_partial`, `NFKC_partial`: normalize what can be normalized now and leave the rest in `$unprocessed`, for text that arrives in pieces (see `splitOnLastStarter`).

A code point argument is converted as an unsigned integer: pass a plain non-negative integer.

Differences from upstream

None.

Functions

Normalization forms

NFD

`NFD($string)`: Normalization Form D, the canonical decomposition with the combining marks in canonical order. `NFKD($string)` is Form KD, the same with compatibility

decomposition. Both exported by default.

Also available as `Unicode::Normalize::NFKD`.

FCC

`NFC($string)`: Normalization Form C, canonical decomposition followed by canonical composition - the composed form, and the usual one for storing and comparing text. `NFKC($string)` is Form KC, from the compatibility decomposition; `FCC($string)` is UTN #5's "Fast C Contiguous" form. `NFC` and `NFKC` are exported by default.

Also available as `Unicode::Normalize::NFC`, `Unicode::Normalize::NFKC`.

Quick checks

checkNFD

`checkNFD($string)`, `checkNFKD($string)`: the quick check of UAX #15, 1 when the string is in NFD (NFKD) and `""` when it is not. For these forms the answer is always definite.

Also available as `Unicode::Normalize::checkNFKD`.

checkNFC

`checkNFC($string)`, `checkNFKC($string)`: the quick check - 1 when the string is in NFC (NFKC), `""` when it is not, and `undef` for "maybe": a combining mark the check cannot decide on alone. `checkNFC("A\x{301}")` and `checkNFC("B\x{301}")` both say maybe, though only the first is not in NFC; for a definite answer compare `$string eq NFC($string)`.

Also available as `Unicode::Normalize::checkNFKC`.

checkFCC

`checkFCD($string)`: 1 when the string is in FCD, `""` when not. `checkFCC($string)`: the same for FCC, with `undef` for "maybe"; a string not in FCD is never in FCC.

Also available as `Unicode::Normalize::checkFCD`.

Building blocks

decompose

`decompose($string [, $compat])`: each character replaced by its canonical decomposition, or by its compatibility decomposition when `$compat` is true. The combining marks are not put in canonical order: `reorder(decompose($s))` is `NFD($s)`, and `reorder(decompose($s, 1))` is `NFKD($s)`.

reorder

`reorder($string)`: every run of combining marks put in canonical order. Joined NFD (or NFKD) strings are NFD again after `reorder`: `reorder(join '', @nfd_strings)`.

compose

`compose($string)`: canonical composition, without decomposing first; `compose($nfd)` is the NFC of an NFD string, `compose($nfkd)` the NFKC of an NFKD one. `composeContiguous($string)` composes only characters that are next to each other, which is how FCC is made.

Also available as `Unicode::Normalize::composeContiguous`.

splitOnLastStarter

`splitOnLastStarter($normalized)`: two strings, the part before the last starter (a character of combining class 0) and the rest. The rest may still change when more text follows, so text normalized in pieces is put together as:

Character data

getCombinClass

`getCombinClass($code_point)`: the canonical combining class, 0 for a starter (230 for U+0301 COMBINING ACUTE ACCENT).

isExclusion

`isExclusion($code_point)`: whether it is a composition exclusion, a character composition never produces.

isSingleton

`isSingleton($code_point)`: whether it decomposes canonically to a single other character.

isNonStDecomp

`isNonStDecomp($code_point)`: whether its decomposition starts with a non-starter (a character of combining class other than 0).

isComp2nd

`isComp2nd($code_point)`: whether it can compose with the character before it (Hangul included, exclusions not). `isNFC_MAYBE` and `isNFKC_MAYBE` are the same property under its quick-check names, `NFC_Quick_Check=Maybe` and `NFKC_Quick_Check=Maybe`.

Also available as `Unicode::Normalize::isNFC_MAYBE`, `Unicode::Normalize::isNFKC_MAYBE`.

isNFD_NO

`isNFD_NO($code_point)`: the derived property `NFD_Quick_Check=No` - whether it decomposes canonically. `isNFKD_NO` is `NFKD_Quick_Check=No`: whether it decomposes at all.

Also available as `Unicode::Normalize::isNFKD_NO`.

isComp_Ex

`isComp_Ex($code_point)`: the derived property Full_Composition_Exclusion - an exclusion, a singleton or a non-starter decomposition. `isNFC_NO` is the same property, as `NFC_Quick_Check=No`; `isNFKC_NO` is `NFKC_Quick_Check=No`, which adds every character whose compatibility decomposition differs from its canonical one (U+FB01, the "fi" ligature).

Also available as `Unicode::Normalize::isNFC_NO`, `Unicode::Normalize::isNFKC_NO`.

getComposite

`getComposite($code_point, $next_code_point)`: the code point the two compose to (Hangul included, exclusions excluded), or `undef`.

getCanon

`getCanon($code_point)`: the full canonical decomposition as a string, or `undef` for a character that does not decompose canonically. `getCompat($code_point)` is the same for the compatibility decomposition. Hangul syllables decompose to their jamo.

Also available as `Unicode::Normalize::getCompat`.

splitOnLastStarter

`splitOnLastStarter($normalized)`: two strings, the part before the last starter (a character of combining class 0) and the rest. The rest may still change when more text follows, so text normalized in pieces is put together as:

```
my ($done, $pending) = splitOnLastStarter($normalized);
my $all = $done . NFC($pending . $more);
```

`$normalized` must already be in the form being built. `normalize_partial` wraps this.

Unicode::UCD

`Unicode::UCD` - the Unicode Character Database: what perl knows about each character, block and script.

```
use Unicode::UCD qw(charblock charinfo);

my $ci = charinfo(0xE9);
printf "Unicode %s | %s | %s, class %s\n",␣
```

(continues on next page)

(continued from previous page)

```
 ↪Unicode::UCD::UnicodeVersion(),
    charblock(0x263A), $ci->{name}, $ci->{category};
# Unicode 17.0.0 | Miscellaneous Symbols | LATIN SMALL LETTER E WITH␣
 ↪ACUTE, class Ll
```

`charinfo`, `charblock`, `charscript`, `charblocks`, `charscripts`, `charprop`, `prop_invlist`, `casefold`, `casespec`, `num` and the rest are `Unicode/UCD.pm` 0.83's own Perl, unchanged, over the data of the Unicode version perl 5.44 was built with (17.0.0) whatever the host has installed.

Upstream reads three of its tables - the Unicode version, the block list and the special casing rules - from files along `@INC` (`unicore/version`, `unicore/Blocks.txt`, `unicore/SpecialCasing.txt`). pperl stores them in the binary when it is built and hands them to `UCD.pm` when it loads, so nothing is read from disk.

Differences from upstream

None in behaviour.

5.2.111 User

Namespace containing the following modules:

- `User::grent` - by-name interface to Perl's built-in getgr*() functions
- `User::pwent` - by-name interface to Perl's built-in getpw*() functions

User::grent

by-name interface to Perl's built-in getgr*() functions

SYNOPSIS

```
use User::grent;
my $gr = getgrgid(0) or die "No group zero";
if ( $gr->name eq 'wheel' && @{$gr->members} > 1 ) {
    print "gid zero name wheel, with other members";
}

use User::grent qw(:FIELDS);
getgrgid(0) or die "No group zero";
if ( $gr_name eq 'wheel' && @gr_members > 1 ) {
    print "gid zero name wheel, with other members";
}

my $gr = getgr($whoever);
```

DESCRIPTION

This module's default exports override the core getgrent(), getgrgid(), and getgrnam() functions, replacing them with versions that return "User::grent" objects. This object has methods that return the similarly named structure field name from the C's passwd structure from `grp.h`; namely name, passwd, gid, and members (not mem). The first three return scalars, the last an array reference.

You may also import all the structure fields directly into your namespace as regular variables using the :FIELDS import tag. (Note that this still overrides your core functions.) Access these fields as variables named with a preceding `gr_`. Thus, `$group_obj->gid()` corresponds to $gr_gid if you import the fields. Array references are available as regular array variables, so `@{ $group_obj->members() }` would be simply @gr_members.

The getgr() function is a simple front-end that forwards a numeric argument to getgrgid() and the rest to getgrnam().

To access this functionality without the core overrides, pass the `use` an empty import list, and then access function functions with their full qualified names. On the other hand, the built-ins are still available via the `CORE::` pseudo-package.

NOTE

While this class is currently implemented using the Class::Struct module to build a struct-like class, you shouldn't rely upon this.

AUTHOR

Tom Christiansen

User::pwent

by-name interface to Perl's built-in getpw*() functions

SYNOPSIS

```
use User::pwent;
my $pw = getpwnam('daemon')       || die "No daemon user";
if ( $pw->uid == 1 && $pw->dir =~ m#^/(bin|tmp)?\z#s ) {
    print "gid 1 on root dir";
}

my $real_shell = $pw->shell || '/bin/sh';

for (my ($fullname, $office, $workphone, $homephone) =
       split /\s*,\s*/, $pw->gecos)
{
   s/&/ucfirst(lc($pw->name))/ge;
}

use User::pwent qw(:FIELDS);
```

(continues on next page)

(continued from previous page)

```
getpwnam('daemon')              || die "No daemon user";
if ( $pw_uid == 1 && $pw_dir =~ m#^/(bin|tmp)?\z#s ) {
    print "gid 1 on root dir";
}

my $pw = getpw($whoever);

use User::pwent qw/:DEFAULT pw_has/;
if (pw_has(qw[gecos expire quota])) { .... }
if (pw_has("name uid gid passwd"))  { .... }
print "Your struct pwd has: ", scalar pw_has(), "\n";
```

DESCRIPTION

This module's default exports override the core getpwent(), getpwuid(), and getpwnam() functions, replacing them with versions that return `User::pwent` objects. This object has methods that return the similarly named structure field name from the C's passwd structure from pwd.h, stripped of their leading "pw_" parts, namely `name`, `passwd`, `uid`, `gid`, `change`, `age`, `quota`, `comment`, `class`, `gecos`, `dir`, `shell`, and `expire`. The `passwd`, `gecos`, and `shell` fields are tainted when running in taint mode.

You may also import all the structure fields directly into your namespace as regular variables using the :FIELDS import tag. (Note that this still overrides your core functions.) Access these fields as variables named with a preceding `pw_` in front their method names. Thus, `$passwd_obj->shell` corresponds to $pw_shell if you import the fields.

The getpw() function is a simple front-end that forwards a numeric argument to getpwuid() and the rest to getpwnam().

To access this functionality without the core overrides, pass the `use` an empty import list, and then access function functions with their full qualified names. The built-ins are always still available via the `CORE::` pseudo-package.

System Specifics

Perl believes that no machine ever has more than one of `change`, `age`, or `quota` implemented, nor more than one of either `comment` or `class`. Some machines do not support `expire`, `gecos`, or allegedly, `passwd`. You may call these methods no matter what machine you're on, but they return `undef` if unimplemented.

You may ask whether one of these was implemented on the system Perl was built on by asking the importable `pw_has` function about them. This function returns true if all parameters are supported fields on the build platform, false if one or more were not, and raises an exception if you asked about a field that Perl never knows how to provide. Parameters may be in a space-separated string, or as separate arguments. If you pass no parameters, the function returns the list of `struct pwd` fields supported by your build platform's C library, as a list in list context, or a space-separated string in scalar context. Note that just because your C library had a field doesn't necessarily mean that it's fully implemented on that system.

Interpretation of the `gecos` field varies between systems, but traditionally holds 4 comma-separated fields containing the user's full name, office location, work phone

number, and home phone number. An & in the gecos field should be replaced by the user's properly capitalized login `name`. The `shell` field, if blank, must be assumed to be `/bin/sh`. Perl does not do this for you. The `passwd` is one-way hashed garble, not clear text, and may not be unhashed save by brute-force guessing. Secure systems use more a more secure hashing than DES. On systems supporting shadow password systems, Perl automatically returns the shadow password entry when called by a suitably empowered user, even if your underlying vendor-provided C library was too short-sighted to realize it should do this.

See passwd(5) and getpwent(3) for details.

NOTE

While this class is currently implemented using the Class::Struct module to build a struct-like class, you shouldn't rely upon this.

AUTHOR

Tom Christiansen

HISTORY

- March 18th, 2000

  Reworked internals to support better interface to dodgey fields than normal Perl function provides. Added pw_has() field. Improved documentation.

5.2.112 XML

Namespace containing the following modules:

- `XML::Parser` - A perl module for parsing XML documents

XML::Parser

A perl module for parsing XML documents

SYNOPSIS

```
use XML::Parser;

$p1 = XML::Parser->new(Style => 'Debug');
$p1->parsefile('REC-xml-19980210.xml');
$p1->parse('<foo id="me">Hello World</foo>');

# Alternative
$p2 = XML::Parser->new(Handlers => {Start => \&handle_start,
                                    End   => \&handle_end,
                                    Char  => \&handle_char});
$p2->parse($socket);
```

(continues on next page)

(continued from previous page)

```
# Another alternative
$p3 = XML::Parser->new(ErrorContext => 2);

$p3->setHandlers(Char    => \&text,
                 Default => \&other);

open(my $fh, 'xmlgenerator |');
$p3->parse($fh, ProtocolEncoding => 'ISO-8859-1');
close($fh);

$p3->parsefile('junk.xml', ErrorContext => 3);
```

DESCRIPTION

This module provides ways to parse XML documents. It is built on top of *XML::Parser::Expat*, which is a lower level interface to James Clark's expat library. Each call to one of the parsing methods creates a new instance of XML::Parser::Expat which is then used to parse the document. Expat options may be provided when the XML::Parser object is created. These options are then passed on to the Expat object on each parse call. They can also be given as extra arguments to the parse methods, in which case they override options given at XML::Parser creation time.

The behavior of the parser is controlled either by "`STYLES`" and/or "`HANDLERS`" options, or by "setHandlers" method. These all provide mechanisms for XML::Parser to set the handlers needed by XML::Parser::Expat. If neither `Style` nor `Handlers` are specified, then parsing just checks the document for being well-formed.

When underlying handlers get called, they receive as their first parameter the *Expat* object, not the Parser object.

METHODS

- new

  This is a class method, the constructor for XML::Parser. Options are passed as keyword value pairs. Recognized options are:

  – Style

  This option provides an easy way to create a given style of parser. The built in styles are: "Debug", "Subs", "Tree", "Objects", and "Stream". These are all defined in separate packages under `XML::Parser::Style::*`, and you can find further documentation for each style both below, and in those packages.

  Custom styles can be provided by giving a full package name containing at least one '::'. This package should then have subs defined for each handler it wishes to have installed. See "STYLES" below for a discussion of each built in style.

  – Handlers

  When provided, this option should be an anonymous hash containing as keys the type of handler and as values a sub reference to handle that type of event.

All the handlers get passed as their 1st parameter the instance of expat that is parsing the document. Further details on handlers can be found in "HANDLERS". Any handler set here overrides the corresponding handler set with the Style option.

– Pkg

Some styles will refer to subs defined in this package. If not provided, it defaults to the package which called the constructor.

– ErrorContext

This is an Expat option. When this option is defined, errors are reported in context. The value should be the number of lines to show on either side of the line in which the error occurred.

– ProtocolEncoding

This is an Expat option. This sets the protocol encoding name. It defaults to none. The built-in encodings are: `UTF-8`, `ISO-8859-1`, `UTF-16`, and `US-ASCII`. Other encodings may be used if they have encoding maps in one of the directories in the @Encoding_Path list. Check "ENCODINGS" for more information on encoding maps. Setting the protocol encoding overrides any encoding in the XML declaration.

– Namespaces

This is an Expat option. If this is set to a true value, then namespace processing is done during the parse. See *"Namespaces" in XML::Parser::Expat* for further discussion of namespace processing.

– NoExpand

This is an Expat option. Normally, the parser will try to expand references to entities defined in the internal subset. If this option is set to a true value, and a default handler is also set, then the default handler will be called when an entity reference is seen in text. This has no effect if a default handler has not been registered, and it has no effect on the expansion of entity references inside attribute values.

– Stream_Delimiter

This is an Expat option. It takes a string value. When this string is found alone on a line while parsing from a stream, then the parse is ended as if it saw an end of file. The intended use is with a stream of xml documents in a MIME multipart format. The string should not contain a trailing newline.

– ParseParamEnt

This is an Expat option. Unless standalone is set to "yes" in the XML declaration, setting this to a true value allows the external DTD to be read, and parameter entities to be parsed and expanded.

Implicit vs explicit parameter entity parsing: When `ParseParamEnt` is not set, parameter entity references (e.g. `%foo;`) in the internal DTD subset are passed through to the **Default** handler as literal text. This is the mode that XML::Twig and other DTD round-tripping tools rely on.

When `ParseParamEnt` is set to a true value, or when a declaration handler (**Entity**, **Element**, or **Attlist**) is registered, parameter entity parsing is activated. In this mode, PE references are resolved by expat (via the **ExternEnt** handler) and subsequent declarations are routed to their dedicated declaration handlers instead of the Default handler.

– NoLWP

This option has no effect if the ExternEnt or ExternEntFin handlers are directly set. Otherwise, if true, it forces the use of a file based external entity handler.

– BillionLaughsAttackProtectionMaximumAmplification

Sets the maximum amplification factor for the Billion Laughs attack protection. See “SECURITY” below for details.

This is an Expat option. Requires libexpat >= 2.4.0 built with `XML_DTD` or `XML_GE`.

– BillionLaughsAttackProtectionActivationThreshold

Sets the activation threshold (in bytes) for the Billion Laughs attack protection. See “SECURITY” below for details.

This is an Expat option. Requires libexpat >= 2.4.0 built with `XML_DTD` or `XML_GE`.

– AllocTrackerMaximumAmplification

Sets the maximum amplification factor for the allocation tracker. See “SECURITY” below for details.

This is an Expat option. Requires libexpat >= 2.7.2 built with `XML_DTD` or `XML_GE`.

– AllocTrackerActivationThreshold

Sets the activation threshold (in bytes) for the allocation tracker. See “SECURITY” below for details.

This is an Expat option. Requires libexpat >= 2.7.2 built with `XML_DTD` or `XML_GE`.

– ReparseDeferralEnabled

Enables or disables reparse deferral, a security mechanism that prevents certain token-boundary attacks. See “SECURITY” below for details.

This is an Expat option. Requires libexpat >= 2.6.0.

– Non_Expat_Options

If provided, this should be an anonymous hash whose keys are options that shouldn’t be passed to Expat. This should only be of concern to those subclassing XML::Parser.

- setHandlers(TYPE, HANDLER [, TYPE, HANDLER [...]])

This method registers handlers for various parser events. It overrides any previous handlers registered through the Style or Handler options or through earlier calls to setHandlers. By providing a false or undefined value as the handler, the existing handler can be unset.

This method returns a list of type, handler pairs corresponding to the input. The handlers returned are the ones that were in effect prior to the call.

See a description of the handler types in "HANDLERS".

- parse(SOURCE [, OPT => OPT_VALUE [...]])

  The SOURCE parameter should either be a string containing the whole XML document, or it should be an open IO::Handle. Constructor options to XML::Parser::Expat given as keyword-value pairs may follow the SOURCE parameter. These override, for this call, any options or attributes passed through from the XML::Parser instance.

  A die call is thrown if a parse error occurs. Otherwise it will return 1 or whatever is returned from the **Final** handler, if one is installed. In other words, what parse may return depends on the style.

  See "ERROR HANDLING" below for how to catch and handle parse errors.

- parsestring

  This is just an alias for parse for backwards compatibility.

- parsefile(FILE [, OPT => OPT_VALUE [...]])

  Open FILE for reading, then call parse with the open handle. The file is closed no matter how parse returns. A die call is thrown if the file cannot be opened or if a parse error occurs. Returns what parse returns.

- parse_start([OPT => OPT_VALUE [...]])

  Create and return a new instance of XML::Parser::ExpatNB. Constructor options may be provided. If an init handler has been provided, it is called before returning the ExpatNB object. Documents are parsed by making incremental calls to the parse_more method of this object, which takes a string. A single call to the parse_done method of this object, which takes no arguments, indicates that the document is finished.

  If there is a final handler installed, it is executed by the parse_done method before returning and the parse_done method returns whatever is returned by the final handler.

HANDLERS

Expat is an event based parser. As the parser recognizes parts of the document (say the start or end tag for an XML element), then any handlers registered for that type of an event are called with suitable parameters. All handlers receive an instance of XML::Parser::Expat as their first argument. See *"METHODS" in XML::Parser::Expat* for a discussion of the methods that can be called on this object.

Init (Expat)

This is called just before the parsing of the document starts.

Final (Expat)

This is called just after parsing has finished, but only if no errors occurred during the parse. Parse returns what this returns.

Start (Expat, Element [, Attr, Val [,...]])

This event is generated when an XML start tag is recognized. Element is the name of the XML element type that is opened with the start tag. The Attr & Val pairs are generated for each attribute in the start tag.

End (Expat, Element)

This event is generated when an XML end tag is recognized. Note that an XML empty tag (<foo/>) generates both a start and an end event.

Char (Expat, String)

This event is generated when non-markup is recognized. The non-markup sequence of characters is in String. A single non-markup sequence of characters may generate multiple calls to this handler. Whatever the encoding of the string in the original document, this is given to the handler in UTF-8.

Important: Because the underlying expat library parses in fixed-size chunks, character data that spans a buffer boundary will arrive as two or more consecutive Char events. This typically occurs with files larger than about 32 KiB and is not a bug. To obtain the complete text of an element, accumulate the strings delivered between Start and End events:

```
my $current_text;
sub start_handler { $current_text = ''; }
sub char_handler  { $current_text .= $_[1]; }
sub end_handler   { print "complete text: $current_text\n"; }
```

The Stream style (`XML::Parser::Style::Stream`) already performs this accumulation automatically.

Proc (Expat, Target, Data)

This event is generated when a processing instruction is recognized.

Comment (Expat, Data)

This event is generated when a comment is recognized.

CdataStart (Expat)

This is called at the start of a CDATA section.

CdataEnd (Expat)

This is called at the end of a CDATA section.

Default (Expat, String)

This is called for any characters that don't have a registered handler. This includes both characters that are part of markup for which no events are generated (markup declarations) and characters that could generate events, but for which no handler has been registered.

Whatever the encoding in the original document, the string is returned to the handler in UTF-8.

Unparsed (Expat, Entity, Base, Sysid, Pubid, Notation)

This is called for a declaration of an unparsed entity. Entity is the name of the entity. Base is the base to be used for resolving a relative URI. Sysid is the system id. Pubid is the public id. Notation is the notation name. Base and Pubid may be undefined.

Notation (Expat, Notation, Base, Sysid, Pubid)

This is called for a declaration of notation. Notation is the notation name. Base is the base to be used for resolving a relative URI. Sysid is the system id. Pubid is the public id. Base, Sysid, and Pubid may all be undefined.

ExternEnt (Expat, Base, Sysid, Pubid)

This is called when an external entity is referenced. Base is the base to be used for resolving a relative URI. Sysid is the system id. Pubid is the public id. Base, and Pubid may be undefined.

This handler should either return a string, which represents the contents of the external entity, or return an open filehandle that can be read to obtain the contents of the external entity, or return undef, which indicates the external entity couldn't be found and will generate a parse error.

If an open filehandle is returned, it must be returned as either a glob (*FOO) or as a reference to a glob (e.g. an instance of IO::Handle).

A default handler is installed for this event. The default handler is XML::Parser::lwp_ext_ent_handler unless the NoLWP option was provided with a true value, otherwise XML::Parser::file_ext_ent_handler is the default handler for external entities. Even without the NoLWP option, if the URI or LWP modules are missing, the file based handler ends up being used after giving a warning on the first external entity reference.

The LWP external entity handler will use proxies defined in the environment (http_proxy, ftp_proxy, etc.).

Please note that the LWP external entity handler reads the entire entity into a string and returns it, where as the file handler opens a filehandle.

Also note that the file external entity handler will likely choke on absolute URIs or file names that don't fit the conventions of the local operating system.

The expat base method can be used to set a basename for relative pathnames. If no basename is given, or if the basename is itself a relative name, then it is relative to the current working directory.

ExternEntFin (Expat)

This is called after parsing an external entity. It's not called unless an ExternEnt handler is also set. There is a default handler installed that pairs with the default ExternEnt handler.

If you're going to install your own ExternEnt handler, then you should set (or unset) this handler too.

Entity (Expat, Name, Val, Sysid, Pubid, Ndata, IsParam)

This is called when an entity is declared. For internal entities, the Val parameter will contain the value and the remaining three parameters will be undefined. For external entities, the Val parameter will be undefined, the Sysid parameter will have the system id, the Pubid parameter will have the public id if it was provided (it will be undefined otherwise), the Ndata parameter will contain the notation for unparsed entities. If this is a parameter entity declaration, then the IsParam parameter is true.

Note that this handler and the Unparsed handler above overlap. If both are set, then this handler will not be called for unparsed entities.

Element (Expat, Name, Model)

The element handler is called when an element declaration is found. Name is the element name, and Model is the content model as an XML::Parser::Content object. See *"XML::Parser::ContentModel Methods" in XML::Parser::Expat* for methods available for this class.

Attlist (Expat, Elname, Attname, Type, Default, Fixed)

This handler is called for each attribute in an ATTLIST declaration. So an ATTLIST declaration that has multiple attributes will generate multiple calls to this handler. The Elname parameter is the name of the element with which the attribute is being associated. The Attname parameter is the name of the attribute. Type is the attribute type, given as a string. Default is the default value, which will either be "#REQUIRED", "#IMPLIED" or a quoted string (i.e. the returned string will begin and end with a quote character). If Fixed is true, then this is a fixed attribute.

Doctype (Expat, Name, Sysid, Pubid, Internal)

This handler is called for DOCTYPE declarations. Name is the document type name. Sysid is the system id of the document type, if it was provided, otherwise it's undefined. Pubid is the public id of the document type, which will be undefined if no public id was given. Internal will be true or false, indicating whether or not the doctype declaration contains an internal subset.

* DoctypeFin (Expat)

This handler is called after parsing of the DOCTYPE declaration has finished, including any internal or external DTD declarations.

XMLDecl (Expat, Version, Encoding, Standalone)

This handler is called for xml declarations. Version is a string containing the version. Encoding is either undefined or contains an encoding string. Standalone will be either true, false, or undefined if the standalone attribute is yes, no, or not made respectively.

STYLES

Debug

This just prints out the document in outline form. Nothing special is returned by parse.

Subs

Each time an element starts, a sub by that name in the package specified by the Pkg option is called with the same parameters that the Start handler gets called with.

Each time an element ends, a sub with that name appended with an underscore ("_"), is called with the same parameters that the End handler gets called with.

Nothing special is returned by parse.

Tree

Parse will return a parse tree for the document. Each node in the tree takes the form of a tag, content pair. Text nodes are represented with a pseudo-tag of "0" and the string that is their content. For elements, the content is an array reference. The first item in the array is a (possibly empty) hash reference containing attributes. The remainder of the array is a sequence of tag-content pairs representing the content of the element.

So for example the result of parsing:

```
<foo><head id="a">Hello <em>there</em></head><bar>Howdy<ref/></bar>
 ↪do</foo>
```

would be:

```
           Tag   Content
=============================================================
[foo, [{}, head, [{id => "a"}, 0, "Hello ",  em, [{}, 0, "there"]],
            bar, [         {}, 0, "Howdy",  ref, [{}]],
              0, "do"
      ]
]
```

The root document "foo", has 3 children: a "head" element, a "bar" element and the text "do". After the empty attribute hash, these are represented in it's contents by 3 tag-content pairs.

Objects

This is similar to the Tree style, except that a hash object is created for each element. The corresponding object will be in the class whose name is created by appending "::" and the element name to the package set with the Pkg option. Non-markup text will be in the ::Characters class. The contents of the corresponding object will be in an anonymous array that is the value of the Kids property for that object.

Stream

This style also uses the Pkg package. If none of the subs that this style looks for is there, then the effect of parsing with this style is to print a canonical copy of the document without comments or declarations. All the subs receive as their 1st parameter the Expat instance for the document they're parsing.

It looks for the following routines:

- StartDocument

  Called at the start of the parse .

- StartTag

  Called for every start tag with a second parameter of the element type. The $_ variable will contain a copy of the tag and the %_ variable will contain attribute values supplied for that element.

- EndTag

  Called for every end tag with a second parameter of the element type. The $_ variable will contain a copy of the end tag.

- Text

  Called just before start or end tags with accumulated non-markup text in the $_ variable.

- PI

  Called for processing instructions. The $_ variable will contain a copy of the PI and the target and data are sent as 2nd and 3rd parameters respectively.

- EndDocument

  Called at conclusion of the parse.

ENCODINGS

XML documents may be encoded in character sets other than Unicode as long as they may be mapped into the Unicode character set. Expat has further restrictions on encodings. Read the xmlparse.h header file in the expat distribution to see details on these restrictions.

Expat has built-in encodings for: `UTF-8`, `ISO-8859-1`, `UTF-16`, and `US-ASCII`. Encodings are set either through the XML declaration encoding attribute or through the ProtocolEncoding option to XML::Parser or XML::Parser::Expat.

For encodings other than the built-ins, expat calls the function load_encoding in the Expat package with the encoding name. This function looks for a file in the path list

@XML::Parser::Expat::Encoding_Path, that matches the lower-cased name with a '.enc' extension. The first one it finds, it loads.

If you wish to build your own encoding maps, check out the XML::Encoding module from CPAN.

ERROR HANDLING

XML::Parser throws an exception (dies) when it encounters a parse error. This includes malformed XML, encoding errors, and other problems detected by the underlying expat library.

The `parse`, `parsefile`, and `parse_done` methods may all throw exceptions. To handle parse errors gracefully in your application, wrap the parse call in an `eval` block:

```
my $parser = XML::Parser->new(Style => 'Tree');

my $tree = eval { $parser->parsefile('data.xml') };
if ($@) {
  # Handle the parse error
  warn "Parse failed: $@";
}
```

The error message (in `$@`) will include the line number, column number, and byte position where the error was detected. For additional context around the error location, set the **ErrorContext** option when constructing the parser:

```
my $parser = XML::Parser->new(
  Style        => 'Tree',
  ErrorContext  => 2,
);
```

This will include 2 lines of context on either side of the error in the error message.

SECURITY

XML::Parser relies on the expat C library for parsing. Modern versions of expat include several security mechanisms that can be tuned through constructor options passed to `new()`. These options are forwarded directly to *XML::Parser::Expat* and take effect for every subsequent `parse`, `parsefile`, or `parse_start` call on the parser instance.

All of these options will `croak` at runtime if the underlying libexpat does not support them.

Billion Laughs Attack Protection

The Billion Laughs attack (also known as an XML bomb) uses deeply nested entity definitions to cause exponential expansion, consuming memory and CPU. Expat >= 2.4.0 (built with `XML_DTD` or `XML_GE`) includes built-in protection controlled by two parameters:

- **BillionLaughsAttackProtectionMaximumAmplification**

The maximum ratio between the size of the expanded output and the size of the input. For example, a value of `100.0` means the parser will abort if entity expansion would produce output more than 100 times the size of the input.

- **BillionLaughsAttackProtectionActivationThreshold**

  The number of bytes of expanded output before the amplification limit takes effect. This prevents false positives on small documents that happen to have a high amplification ratio.

Allocation Tracker

Expat >= 2.7.2 (built with `XML_DTD` or `XML_GE`) adds a second layer of amplification tracking through the allocation tracker, which measures memory allocation rather than output size:

- **AllocTrackerMaximumAmplification**

  The maximum ratio of allocated memory to input size.

- **AllocTrackerActivationThreshold**

  The number of bytes of allocation before the limit takes effect.

Reparse Deferral

Expat >= 2.6.0 includes reparse deferral, which prevents attacks that exploit token boundaries. Rather than reparsing incomplete tokens immediately, the parser defers until more input arrives.

- **ReparseDeferralEnabled**

  A boolean. Set to a true value to enable reparse deferral, or `0` to disable it.

For full details on each option, see *"new" in XML::Parser::Expat*.

```
# Example: tighten Billion Laughs limits
my $parser = XML::Parser->new(
  Style => 'Tree',
  BillionLaughsAttackProtectionMaximumAmplification => 50,
  BillionLaughsAttackProtectionActivationThreshold  => 1024,
);
```

LICENSE

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See https://dev.perl.org/licenses/ for more information.

AUTHORS

Larry Wall <larry@wall.org> wrote version 1.0.

Clark Cooper <coopercc@netheaven.com> picked up support, changed the API for this version (2.x), provided documentation, and added some standard package features.

Matt Sergeant <matt@sergeant.org> was maintaining XML::Parser from 2003 to 2007.

Alexandr Ciornii <alexchorny@gmail.com> was maintaining XML::Parser from 2007 to 2013.

Todd Rinaldo <toddr@cpan.org> has been maintaining XML::Parser since 2013.

The project started making use of Claude Code <https://claude.ai/code> in January 2026.

Modules

- XML::Parser::Expat - the low-level interface of XML::Parser
- XML::Parser::Style

XML::Parser::Expat

XML::Parser::Expat - the low-level interface of XML::Parser.

```
use XML::Parser;

my $p = XML::Parser->new(Handlers => {
    Start => sub { my ($expat, $elem, %attr) = @_; print "<$elem>\n"
 ↪},
    Char  => sub { my ($expat, $text) = @_; print $text },
});
$p->parse('<doc><item id="1">hello</item></doc>');
```

Most programs use XML::Parser (or a module built on it such as XML::Twig, XML::Simple or XML::Dumper) and never call this module directly. It parses with pperl's XML engine over libxml2, the one XML::LibXML uses too; libxml2 2.15 or newer is required, and the module fails to load with a message saying so when it is older or missing.

This is a native implementation of the CPAN distribution XML-Parser 2.59, which parses with the expat library; its XML/Parser.pm, XML/Parser/Expat.pm and the XML::Parser::Style::* modules are used unchanged. Every handler sees what it sees under expat: the same calls with the same arguments, character data split where expat splits it, and the same text, positions and lengths from recognized_string, original_string, current_byte, current_line, current_column and current_length. Errors carry expat's messages and positions.

Differences from upstream

expat_version reports the expat release whose interface this module provides, expat_2.6.0, since no expat is loaded.

A document with standalone="yes" has the parameter entities of its internal subset expanded, as XML 1.0 requires; expat leaves them unread, and then refuses an entity reference they declare as undefined.

Functions

Other Functions

ParserCreate

`ParserCreate($self, $encoding, $namespaces)`: a new parser for the XML::Parser::Expat object $self, as an integer handle.

ParserRelease

`ParserRelease($parser)`: drop the parser's reference to its object, breaking the cycle so that the object can be freed.

ParserFree

`ParserFree($parser)`: free the parser and every handler it holds.

ParseString

`ParseString($parser, $string)`: parse a whole document.

ParseStream

`ParseStream($parser, $handle, $delim)`: parse from a handle, up to a line holding only `$delim` when it is defined.

ParsePartial

`ParsePartial($parser, $chunk)`: parse one more chunk of a document.

ParseDone

`ParseDone($parser)`: end a document parsed with ParsePartial.

UnsetAllHandlers

`UnsetAllHandlers($parser)`: expat's handlers go, the element ones and the namespace ones included; what is left is the Default handler.

SetBase

`SetBase($parser, $base)`.

GetBase

`GetBase($parser)`: the base, or undef.

PositionContext

`PositionContext($parser, $lines)`: the input around the current position, `$lines` lines either side, and the position's offset in it.

GenerateNSName

`GenerateNSName($name, $namespace, \%table, \@list)`: the name object for `$name` in `$namespace`, as the parser makes them.

DefaultCurrent

`DefaultCurrent($parser)`: pass the current event's text to the Default handler.

RecognizedString

`RecognizedString($parser)`: the text of the current event.

GetErrorCode

`GetErrorCode($parser)`.

GetCurrentLineNumber

`GetCurrentLineNumber($parser)`.

GetCurrentColumnNumber

`GetCurrentColumnNumber($parser)`.

GetCurrentByteIndex

`GetCurrentByteIndex($parser)`.

GetCurrentByteCount

`GetCurrentByteCount($parser)`.

GetSpecifiedAttributeCount

`GetSpecifiedAttributeCount($parser)`.

ErrorString

`ErrorString($code)`: the message for an error code.

ExpatVersion

`ExpatVersion()`: “expat_X.Y.Z”, the interface this module provides.

ExpatVersionInfo

`ExpatVersionInfo()`: `(major => X, minor => Y, micro => Z)`.

LoadEncoding

`LoadEncoding($data, $size)`: register a binary encoding map (an .enc file’s contents) in %Encoding_Table; returns its name, or undef when the data is not a map.

FreeEncoding

`FreeEncoding($encinfo)`: free a map LoadEncoding made.

OriginalString

`OriginalString($parser)`: the input text of the current event.

ElementIndex

`ElementIndex($parser)`: the serial number of the current element.

SkipUntil

`SkipUntil($parser, $index)`: suspend most handlers until the element with serial number `$index` starts.

Do_External_Parse

`Do_External_Parse($parser, $result)`: the ExternEnt handler’s result is read by the loader that called the handler, so outside one there is nothing to parse into.

SetReparseDeferralEnabled

`SetReparseDeferralEnabled($parser, $enabled)`: expat’s deferral of reparsing a token split across buffers (`parse_buffer`); XML_TRUE.

XML::Parser::Style

Namespace containing the following modules:

- `XML::Parser::Style::Debug` - Debug style for XML::Parser
- `XML::Parser::Style::Objects` - Objects styler parser
- `XML::Parser::Style::Stream` - Stream style for XML::Parser
- `XML::Parser::Style::Subs` - glue for handling element callbacks
- `XML::Parser::Style::Tree` - Tree style parser

XML::Parser::Style::Debug

Debug style for XML::Parser

SYNOPSIS

```
use XML::Parser;
my $p = XML::Parser->new(Style => 'Debug');
$p->parsefile('foo.xml');
```

DESCRIPTION

This just prints out the document in outline form to STDERR. Nothing special is returned by parse.

XML::Parser::Style::Objects

Objects styler parser

SYNOPSIS

```
use XML::Parser;
my $p = XML::Parser->new(Style => 'Objects', Pkg => 'MyNode');
my $tree = $p->parsefile('foo.xml');
```

DESCRIPTION

This module implements XML::Parser's Objects style parser.

This is similar to the Tree style, except that a hash object is created for each element. The corresponding object will be in the class whose name is created by appending "::" and the element name to the package set with the Pkg option. Non-markup text will be in the ::Characters class. The contents of the corresponding object will be in an anonymous array that is the value of the Kids property for that object.

SEE ALSO

XML::Parser::Style::Tree

XML::Parser::Style::Stream

Stream style for XML::Parser

SYNOPSIS

```
use XML::Parser;
my $p = XML::Parser->new(Style => 'Stream', Pkg => 'MySubs');
$p->parsefile('foo.xml');
```

(continues on next page)

(continued from previous page)

```
{
  package MySubs;

  sub StartTag {
    my ($e, $name) = @_;
    # do something with start tags
  }

  sub EndTag {
    my ($e, $name) = @_;
    # do something with end tags
  }

  sub Text {
    my ($e) = @_;
    # $_ contains accumulated text
    # do something with text nodes
  }
}
```

DESCRIPTION

This style uses the Pkg option to find subs in a given package to call for each event. If none of the subs that this style looks for is there, then the effect of parsing with this style is to print a canonical copy of the document without comments or declarations. All the subs receive as their 1st parameter the Expat instance for the document they're parsing.

It looks for the following routines:

- StartDocument

  Called at the start of the parse .

- StartTag

  Called for every start tag with a second parameter of the element type. The $_ variable will contain a copy of the tag and the %_ variable will contain attribute values supplied for that element.

- EndTag

  Called for every end tag with a second parameter of the element type. The $_ variable will contain a copy of the end tag.

- Text

  Called just before start or end tags with accumulated non-markup text in the $_ variable.

- PI

  Called for processing instructions. The $_ variable will contain a copy of the PI and the target and data are sent as 2nd and 3rd parameters respectively.

- EndDocument

  Called at conclusion of the parse.

XML::Parser::Style::Subs

glue for handling element callbacks

SYNOPSIS

```
use XML::Parser;
my $p = XML::Parser->new(Style => 'Subs', Pkg => 'MySubs');
$p->parsefile('foo.xml');

{
  package MySubs;

  sub foo {
    # start of foo tag
  }

  sub foo_ {
    # end of foo tag
  }
}
```

DESCRIPTION

Each time an element starts, a sub by that name in the package specified by the Pkg option is called with the same parameters that the Start handler gets called with.

Each time an element ends, a sub with that name appended with an underscore (“_”), is called with the same parameters that the End handler gets called with.

Nothing special is returned by parse.

XML::Parser::Style::Tree

Tree style parser

SYNOPSIS

```
use XML::Parser;
my $p = XML::Parser->new(Style => 'Tree');
my $tree = $p->parsefile('foo.xml');
```

DESCRIPTION

This module implements XML::Parser's Tree style parser.

When parsing a document, `parse()` will return a parse tree for the document. Each node in the tree takes the form of a tag, content pair. Text nodes are represented with a pseudo-tag of "0" and the string that is their content. For elements, the content is an array reference. The first item in the array is a (possibly empty) hash reference containing attributes. The remainder of the array is a sequence of tag-content pairs representing the content of the element.

So for example the result of parsing:

```
<foo><head id="a">Hello <em>there</em></head><bar>Howdy<ref/></bar>
↪do</foo>
```

would be:

```
           Tag   Content
============================================================
[foo, [{}, head, [{id => "a"}, 0, "Hello ",  em, [{}, 0, "there"]],
            bar, [         {}, 0, "Howdy",  ref, [{}]],
              0, "do"
      ]
]
```

The root document "foo", has 3 children: a "head" element, a "bar" element and the text "do". After the empty attribute hash, these are represented in it's contents by 3 tag-content pairs.

Entity Expansion

The underlying Expat parser always expands predefined XML entity references (`<` , `>`, `&`, `"`, `'`) in both text content and attribute values before they reach the Tree style handlers. This is required by the XML specification and cannot be prevented. For example, `<` in the source XML will appear as < in the resulting tree structure.

If you need access to the original unexpanded text, consider using the handler-based API with the `original_string` method on the Expat object instead of the Tree style.

5.2.113 XSLoader

Start the native part of a module

VERSION

Version 0.32, the XSLoader of perl 5.44, unchanged.

SYNOPSIS

```
package YourPackage;
require XSLoader;

XSLoader::load(__PACKAGE__, $VERSION);
```

DESCRIPTION

`XSLoader::load` starts the compiled part of a module. The module's `.pm` calls it, usually right after setting `$VERSION`; without arguments it uses the calling package.

Under pperl, `XSLoader::load` loads no shared object (DynaLoader's `dl_*` functions do, for a library a program asks for). Where pperl implements a module natively, the compiled part is in the pperl binary, and `XSLoader::load` starts it the way perl5 starts an extension linked statically into its binary. That is how the native modules of the module reference load, such as Socket, POSIX, List::Util and Storable: their upstream `.pm` runs unchanged and calls `XSLoader::load`.

For a module pperl has no native part for, `XSLoader::load` dies with perl's own message for a compiled part that is not there:

```
Can't locate loadable object for module Some::Module in @INC (@INC␣
↪contains: ...)
```

That is the message modules with a pure-Perl fallback test for, so a module that calls `XSLoader::load` inside `eval` and falls back when the call fails runs its pure-Perl part under pperl.

5.2.114 XString

The string quoting helpers of `B`, without loading `B`.

`XString::cstring` and `XString::perlstring` return a string quoted and escaped so that it reads back as the same bytes: as a C string literal, or as a Perl double-quoted string. They are `B::cstring` and `B::perlstring`, packaged so that a module can quote strings without pulling in the compiler backend.

Synopsis

```
use XString;
print XString::cstring(qq[a"b\n]), "\n";      # "a\"b\n"
print XString::perlstring('cost: $5 @ 2'), "\n"; # "cost: \$5 \@ 2"
print XString::cstring(undef), "\n";          # 0
```

Nothing is exported; call them by their full names.

This is a native implementation of XString 0.005's `XString.xs`; its `XString.pm` is used unchanged.

Differences from upstream

None.

Functions

Quoting

cstring

Quote a string as a C string literal.

perlstring

Quote a string as a Perl double-quoted string.

cstring

Quote a string as a C string literal.

Synopsis

```
my $c = XString::cstring($str);
```

What you get back

`$str` between double quotes, with `"` and `\` escaped, `\n`, `\r`, `\t`, `\a`, `\b`, `\f` and `\v` in their C spelling, `??` written so that it cannot form a trigraph, and every other unprintable byte as a three-digit octal escape. `undef` gives `0`, unquoted. The same as `B::cstring`.

perlstring

Quote a string as a Perl double-quoted string.

Synopsis

```
my $code = XString::perlstring($str);
my $back = eval $code;        # eq $str
```

What you get back

`$str` between double quotes, escaped so that perl reads it back as the same string: `"`, `\`, `$` and `@` are escaped, control characters take their backslash spelling or an octal escape, and a string of characters is written with `\x{...}` escapes. `undef` gives `0`, unquoted. The same as `B::perlstring`.

5.2.115 YAML

Namespace containing the following modules:

- `YAML::LibYAML` - `YAML-LibYAML` is the CPAN *distribution* name for the `YAML::XS` module.
- `YAML::PP` - YAML 1.2 processor
- `YAML::XS` - use YAML::XS;

YAML::LibYAML

NOTE

`YAML-LibYAML` is the CPAN *distribution* name for the `YAML::XS` module.

See the YAML::XS documentation instead.

AUTHOR

Ingy döt Net ingy@cpan.org

COPYRIGHT AND LICENSE

Copyright 2007-2022 - Ingy döt Net

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See http://www.perl.com/perl/misc/Artistic.html

YAML::PP

YAML 1.2 processor

SYNOPSIS

WARNING: Most of the inner API is not stable yet.

Here are a few examples of the basic load and dump methods:

```
use YAML::PP;
my $ypp = YAML::PP->new;

my $yaml = <<'EOM';
--- # Document one is a mapping
name: Tina
age: 29
favourite language: Perl

--- # Document two is a sequence
- plain string
- 'in single quotes'
```

(continues on next page)

(continued from previous page)

```
- "in double quotes we have escapes! like \t and \n"
- | # a literal block scalar
  line1
  line2
- > # a folded block scalar
  this is all one
  single line because the
  linebreaks will be folded
EOM

my @documents = $ypp->load_string($yaml);
my @documents = $ypp->load_file($filename);

my $yaml = $ypp->dump_string($data1, $data2);
$ypp->dump_file($filename, $data1, $data2);

# Enable perl data types and objects
my $ypp = YAML::PP->new(schema => [qw/ + Perl /]);
my $yaml = $yp->dump_string($data_with_perl_objects);

# Legacy interface
use YAML::PP qw/ Load Dump LoadFile DumpFile /;
my @documents = Load($yaml);
my @documents = LoadFile($filename);
my @documents = LoadFile($filehandle);
my $yaml = = Dump(@documents);
DumpFile($filename, @documents);
DumpFile($filenhandle @documents);
```

Some utility scripts, mostly useful for debugging:

```
# Load YAML into a data structure and dump with Data::Dumper
yamlpp-load < file.yaml

# Load and Dump
yamlpp-load-dump < file.yaml

# Print the events from the parser in yaml-test-suite format
yamlpp-events < file.yaml

# Parse and emit events directly without loading
yamlpp-parse-emit < file.yaml

# Create ANSI colored YAML. Can also be useful for invalid YAML,
 ↪showing
# you the exact location of the error
yamlpp-highlight < file.yaml
```

DESCRIPTION

YAML::PP is a modular YAML processor.

It aims to support `YAML 1.2` and `YAML 1.1`. See https://yaml.org/. Some (rare) syntax elements are not yet supported and documented below.

YAML is a serialization language. The YAML input is called "YAML Stream". A stream consists of one or more "Documents", separated by a line with a document start marker `---`. A document optionally ends with the document end marker `...`

This allows one to process continuous streams additionally to a fixed input file or string.

The YAML::PP frontend will currently load all documents, and return only the first if called with scalar context.

The YAML backend is implemented in a modular way that allows one to add custom handling of YAML tags, perl objects and data types. The inner API is not yet stable. Suggestions welcome.

You can check out all current parse and load results from the yaml-test-suite here: https://perlpunk.github.io/YAML-PP-p5/test-suite.html

METHODS

new

```
my $ypp = YAML::PP->new;

# use YAML 1.2 Failsafe Schema
my $ypp = YAML::PP->new( schema => ['Failsafe'] );
# use YAML 1.2 JSON Schema
my $ypp = YAML::PP->new( schema => ['JSON'] );
# use YAML 1.2 Core Schema
my $ypp = YAML::PP->new( schema => ['Core'] );

# Die when detecting cyclic references
my $ypp = YAML::PP->new( cyclic_refs => 'fatal' );

my $ypp = YAML::PP->new(
    boolean => 'perl',
    schema => ['Core'],
    cyclic_refs => 'fatal',
    indent => 4,
    header => 1,
    footer => 0,
    version_directive => 0,
    require_footer => 0,
    max_depth => 512,
);
```

Options:

- boolean

Values: `perl` (currently default), `JSON::PP`, `boolean`, `perl_experimental`

This option is for loading and dumping.

In case of perl 5.36 and later, builtin booleans should work out of the box (since YAML::PP >= 0.38.0).

```
print YAML::PP->new->dump_string([ builtin::true, !1 ]);
# ---
# - true
# - false
```

For earlier perl versions, you can use "pseudo" booleans like documented in the following examples.

```
# load/dump booleans via boolean.pm
my $ypp = YAML::PP->new( boolean => 'boolean' );
# load/dump booleans via JSON::PP::true/false
my $ypp = YAML::PP->new( boolean => 'JSON::PP' );
```

You can also specify more than one class, comma separated. This is important for dumping.

```
boolean => 'JSON::PP,boolean'
Booleans will be loaded as JSON::PP::Booleans, but when dumping,
 ↪ also
'boolean' objects will be recognized

boolean => 'JSON::PP,*'
Booleans will be loaded as JSON::PP::Booleans, but when dumping,
 ↪ all
currently supported boolean classes will be recognized

boolean => '*'
Booleans will be loaded as perl booleans, but when dumping, all
currently supported boolean classes will be recognized

boolean => ''
Booleans will be loaded as perl booleans, but when dumping,␣
 ↪nothing
will be recognized as booleans.
This option is for backwards compatibility for perl versions <␣
 ↪5.36,
if you rely on [!!1, !1] being dumped as [1, ''].
```

The option `perl_experimental` was introduced when experimental boolean support was added to perl 5.36. Since it will not be experimental anymore in perl 5.40 \o/ the option is deprecated and the same as `perl`.

- schema

  Default: `['Core']`

  This option is for loading and dumping.

Array reference. Here you can define what schema to use. Supported standard Schemas are: `Failsafe`, `JSON`, `Core`, `YAML1_1`.

To get an overview how the different Schemas behave, see https://perlpunk.github.io/YAML-PP-p5/schemas.html

Additionally you can add further schemas, for example `Merge`.

- cyclic_refs

  Default: `fatal` (since 0.037)

  Before the default was `allow`, but this can lead to memory leaks when loading on untrusted data, so it was changed to `fatal` by default.

  This option is for loading only.

  Defines what to do when a cyclic reference is detected when loading.

```
# fatal  - die
# warn   - Just warn about them and replace with undef
# ignore - replace with undef
# allow  - Default
```

- duplicate_keys

  Default: 0

  Since version 0.027

  This option is for loading.

  The YAML Spec says duplicate mapping keys should be forbidden.

  When set to true, duplicate keys in mappings are allowed (and will overwrite the previous key).

  When set to false, duplicate keys will result in an error when loading.

  This is especially useful when you have a longer mapping and don't see the duplicate key in your editor:

```
---
a: 1
b: 2
# ............
a: 23 # error
```

- indent

  Default: 2

  This option is for dumping.

  Use that many spaces for indenting

- width

  Since version 0.025

  Default: 80

This option is for dumping.

Maximum columns when dumping.

This is only respected when dumping flow collections right now.

in the future it will be used also for wrapping long strings.

- header

  Default: 1

  This option is for dumping.

  Print document header `---`

- footer

  Default: 0

  Since version v0.39

  This option is for dumping.

  Print document footer `...`

- require_footer

  Default: 0

  Will require a `...` at the end of each document. This can be useful in a context where you want to make sure you received the complete content, for example over network.

```
# Good
---
a: 1
...
---
a: 2
...

# Bad
---
a: 1
---
a: 2
```

- max_depth

  Since version v0.40

  default: 512

  Loading deeply nested, but still small YAML documents, like

```
[[[[[[[[[[...]]]]]]]]]]
```

  can be significantly slower than a “normal” document with the same size. By default the depth is limited to 512, which should be more than enough for 99% of the use cases.

- yaml_version

  Since version 0.020

  This option is for loading and dumping.

  Default: `1.2`

  Note that in this case, a directive `%YAML 1.1` will basically be ignored and everything loaded with the `1.2 Core` Schema.

  If you want to support both YAML 1.1 and 1.2, you have to specify that, and the schema (`Core` or `YAML1_1`) will be chosen automatically.

```
my $yp = YAML::PP->new(
    yaml_version => ['1.2', '1.1'],
);
```

  This is the same as

```
my $yp = YAML::PP->new(
    schema => ['+'],
    yaml_version => ['1.2', '1.1'],
);
```

  because the `+` stands for the default schema per version.

  When loading, and there is no `%YAML` directive, `1.2` will be considered as default, and the `Core` schema will be used.

  If there is a `%YAML 1.1` directive, the `YAML1_1` schema will be used.

  Of course, you can also make `1.1` the default:

```
my $yp = YAML::PP->new(
    yaml_version => ['1.1', '1.2'],
);
```

  You can also specify `1.1` only:

```
my $yp = YAML::PP->new(
    yaml_version => ['1.1'],
);
```

  In this case also documents with `%YAML 1.2` will be loaded with the `YAML1_1` schema.

- version_directive

  Since version 0.020

  This option is for dumping.

  Default: 0

  Print Version Directive `%YAML 1.2` (or `%YAML 1.1`) on top of each YAML document. It will use the first version specified in the `yaml_version` option.

- preserve

  Since version 0.021

  Default: false

  This option is for loading and dumping.

  Preserving scalar styles is still experimental.

```
use YAML::PP::Common qw/ :PRESERVE /;

# Preserve the order of hash keys
my $yp = YAML::PP->new( preserve => PRESERVE_ORDER );

# Preserve the quoting style of scalars
my $yp = YAML::PP->new( preserve => PRESERVE_SCALAR_STYLE );

# Preserve block/flow style (since 0.024)
my $yp = YAML::PP->new( preserve => PRESERVE_FLOW_STYLE );

# Preserve alias names (since 0.027)
my $yp = YAML::PP->new( preserve => PRESERVE_ALIAS );

# Combine, e.g. preserve order and scalar style
my $yp = YAML::PP->new( preserve => PRESERVE_ORDER | PRESERVE_
 ↪SCALAR_STYLE );
```

  Do NOT rely on the internal implementation of it.

  If you load the following input:

```
---
z: 1
a: 2
---
- plain
- 'single'
- "double"
- |
  literal
- >
  folded
---
block mapping: &alias
  flow sequence: [a, b]
same mapping: *alias
flow mapping: {a: b}
```

  with this code:

```
my $yp = YAML::PP->new(
    preserve => PRESERVE_ORDER | PRESERVE_SCALAR_STYLE
                | PRESERVE_FLOW_STYLE | PRESERVE_ALIAS
```

(continues on next page)

(continued from previous page)

```
);
my ($hash, $styles, $flow) = $yp->load_file($file);
$yp->dump_file($hash, $styles, $flow);
```

Then dumping it will return the same output.

Note that YAML allows repeated definition of anchors. They cannot be preserved with YAML::PP right now. Example:

```
---
- &seq [a]
- *seq
- &seq [b]
- *seq
```

Because the data could be shuffled before dumping again, the anchor definition could be broken. In this case repeated anchor names will be discarded when loading and dumped with numeric anchors like usual.

Implementation:

When loading, hashes will be tied to an internal class (`YAML::PP::Preserve::Hash`) that keeps the key order.

Scalars will be returned as objects of an internal class (`YAML::PP::Preserve::Scalar`) with overloading. If you assign to such a scalar, the object will be replaced by a simple scalar.

```
# assignment, style gets lost
$styles->[1] .= ' append';
```

You can also pass 1 as a value. In this case all preserving options will be enabled, also if there are new options added in the future.

There are also methods to create preserved nodes from scratch. See the `preserved_(scalar|mapping|sequence)` "METHODS" below.

load_string

```
my $doc = $ypp->load_string("foo: bar");
my @docs = $ypp->load_string("foo: bar\n---\n- a");
```

Input should be Unicode characters.

So if you read from a file, you should decode it, for example with `Encode::decode()`.

Note that in scalar context, `load_string` and `load_file` return the first document (like YAML::Syck), while *YAML* and *YAML::XS* return the last.

load_file

```
my $doc = $ypp->load_file("file.yaml");
my @docs = $ypp->load_file("file.yaml");
```

Strings will be loaded as unicode characters.

dump_string

```
my $yaml = $ypp->dump_string($doc);
my $yaml = $ypp->dump_string($doc1, $doc2);
my $yaml = $ypp->dump_string(@docs);
```

Input strings should be Unicode characters.

Output will return Unicode characters.

So if you want to write that to a file (or pass to YAML::XS, for example), you typically encode it via `Encode::encode()`.

dump_file

```
$ypp->dump_file("file.yaml", $doc);
$ypp->dump_file("file.yaml", $doc1, $doc2);
$ypp->dump_file("file.yaml", @docs);
```

Input data should be Unicode characters.

dump

This will dump to a predefined writer. By default it will just use the *YAML::PP::Writer* and output a string.

```
my $writer = MyWriter->new(\my $output);
my $yp = YAML::PP->new(
    writer => $writer,
);
$yp->dump($data);
```

preserved_scalar

Since version 0.024

Experimental. Please report bugs or let me know this is useful and works.

You can define a certain scalar style when dumping data. Figuring out the best style is a hard task and practically impossible to get it right for all cases. It's also a matter of taste.

```
use YAML::PP::Common qw/ PRESERVE_SCALAR_STYLE YAML_LITERAL_SCALAR_
 ↪STYLE /;
my $yp = YAML::PP->new(
```

(continues on next page)

(continued from previous page)

```
    preserve => PRESERVE_SCALAR_STYLE,
);
# a single linebreak would normally be dumped with double quotes: "\
 ↪n"
my $scalar = $yp->preserved_scalar("\n", style => YAML_LITERAL_
 ↪SCALAR_STYLE );

my $data = { literal => $scalar };
my $dump = $yp->dump_string($data);
# output
---
literal: |+


...
```

preserved_mapping, preserved_sequence

Since version 0.024

Experimental. Please report bugs or let me know this is useful and works.

With this you can define which nodes are dumped with the more compact flow style instead of block style.

If you add PRESERVE_ORDER to the preserve option, it will also keep the order of the keys in a hash.

```
use YAML::PP::Common qw/
    PRESERVE_ORDER PRESERVE_FLOW_STYLE
    YAML_FLOW_MAPPING_STYLE YAML_FLOW_SEQUENCE_STYLE
/;
my $yp = YAML::PP->new(
    preserve => PRESERVE_FLOW_STYLE | PRESERVE_ORDER
);

my $hash = $yp->preserved_mapping({}, style => YAML_FLOW_MAPPING_
 ↪STYLE);
# Add values after initialization to preserve order
%$hash = (z => 1, a => 2, y => 3, b => 4);

my $array = $yp->preserved_sequence([23, 24], style => YAML_FLOW_
 ↪SEQUENCE_STYLE);

my $data = $yp->preserved_mapping({});
%$data = ( map => $hash, seq => $array );

my $dump = $yp->dump_string($data);
# output
---
map: {z: 1, a: 2, y: 3, b: 4}
seq: [23, 24]
```

loader

Returns or sets the loader object, by default YAML::PP::Loader

dumper

Returns or sets the dumper object, by default YAML::PP::Dumper

schema

Returns or sets the schema object

default_schema

Creates and returns the default schema

FUNCTIONS

The functions `Load`, `LoadFile`, `Dump` and `DumpFile` are provided as a drop-in replacement for other existing YAML processors. No function is exported by default.

Note that in scalar context, `Load` and `LoadFile` return the first document (like YAML::Syck), while *YAML* and *YAML::XS* return the last.

- Load

```
use YAML::PP qw/ Load /;
my $doc = Load($yaml);
my @docs = Load($yaml);
```

  Works like `load_string`.

- LoadFile

```
use YAML::PP qw/ LoadFile /;
my $doc = LoadFile($file);
my @docs = LoadFile($file);
my @docs = LoadFile($filehandle);
```

  Works like `load_file`.

- Dump

```
use YAML::PP qw/ Dump /;
my $yaml = Dump($doc);
my $yaml = Dump(@docs);
```

  Works like `dump_string`.

- DumpFile

```
use YAML::PP qw/ DumpFile /;
DumpFile($file, $doc);
```

(continues on next page)

(continued from previous page)

```
DumpFile($file, @docs);
DumpFile($filehandle, @docs);
```

Works like `dump_file`.

PLUGINS

You can alter the behaviour of YAML::PP by using the following schema classes:

- *YAML::PP::Schema::Failsafe*

  One of the three YAML 1.2 official schemas

- *YAML::PP::Schema::JSON*

  One of the three YAML 1.2 official schemas.

- *YAML::PP::Schema::Core*

  One of the three YAML 1.2 official schemas. Default

- *YAML::PP::Schema::YAML1_1*

  Schema implementing the most common YAML 1.1 types

- *YAML::PP::Schema::Perl*

  Serializing Perl objects and types

- *YAML::PP::Schema::Binary*

  Serializing binary data

- *YAML::PP::Schema::Merge*

  YAML 1.1 merge keys for mappings

- *YAML::PP::Schema::Catchall*

  Experimental.

  It was accidentally added in 0.38.1, and unknown tags forbidden. This was reverted in 0.39.0

  By default they will result in an error.

- *YAML::PP::Schema::Include*

  Include other YAML files via `!include` tags

- *YAML::PP::Schema::Tie::IxHash*

  Deprecated. See option `preserve`

To make the parsing process faster, you can plugin the libyaml parser with YAML::PP::LibYAML.

IMPLEMENTATION

The process of loading and dumping is split into the following steps:

```
Load:

YAML Stream        Tokens        Event List        Data Structure
          --------->    --------->        --------->
            lex           parse           construct


Dump:

Data Structure       Event List        YAML Stream
            --------->        --------->
            represent           emit
```

You can dump basic perl types like hashes, arrays, scalars (strings, numbers). For dumping blessed objects and things like coderefs have a look at *YAML::PP::Perl*/*YAML::PP::Schema::Perl*.

- YAML::PP::Lexer

  The Lexer is reading the YAML stream into tokens. This makes it possible to generate syntax highlighted YAML output.

  Note that the API to retrieve the tokens will change.

- YAML::PP::Parser

  The Parser retrieves the tokens from the Lexer. The main YAML content is then parsed with the Grammar.

- *YAML::PP::Grammar*

- *YAML::PP::Constructor*

  The Constructor creates a data structure from the Parser events.

- YAML::PP::Loader

  The Loader combines the constructor and parser.

- YAML::PP::Dumper

  The Dumper will delegate to the Representer

- YAML::PP::Representer

  The Representer will create Emitter events from the given data structure.

- *YAML::PP::Emitter*

  The Emitter creates a YAML stream.

YAML::PP::Parser

Still TODO:

- Implicit collection keys

```
---
[ a, b, c ]: value
```

- Implicit mapping in flow style sequences

  This is supported since 0.029 (except some not relevant cases):

```
---
[ a, b, c: d ]
# equals
[ a, b, { c: d } ]
```

- Plain mapping keys ending with colons

```
---
key ends with two colons::: value
```

  This was implemented in 0.037.

- Supported Characters

  If you have valid YAML that's not parsed, or the other way round, please create an issue.

- Line and Column Numbers

  You will see line and column numbers in the error message. The column numbers might still be wrong in some cases.

- Error Messages

  The error messages need to be improved.

- Unicode Surrogate Pairs

  Currently loaded as single characters without validating

- Possibly more

YAML::PP::Constructor

The Constructor now supports all three YAML 1.2 Schemas, Failsafe, JSON and Core. Additionally you can choose the schema for YAML 1.1 as `YAML1_1`.

Too see what strings are resolved as booleans, numbers, null etc. look at https://perlpunk.github.io/YAML-PP-p5/schema-examples.html.

You can choose the Schema like this:

```
my $ypp = YAML::PP->new(schema => ['JSON']); # default is 'Core'
```

The Tags `!!seq` and `!!map` are still ignored for now.

It supports:

- Handling of Anchors/Aliases

  Like in modules like *YAML*, the Constructor will use references for mappings and sequences, but obviously not for scalars.

  YAML::XS uses real aliases, which allows also aliasing scalars. I might add an option for that since aliasing is now available in pure perl.

- Boolean Handling

  You can choose between `'perl'` (1/”, currently default), `'JSON::PP'` and `'boolean'`.pm for handling boolean types. That allows you to dump the data structure with one of the JSON modules without losing information about booleans.

- Numbers

  Numbers are created as real numbers instead of strings, so that they are dumped correctly by modules like *JSON::PP* or *JSON::XS*, for example.

- Complex Keys

  Mapping Keys in YAML can be more than just scalars. Of course, you can't load that into a native perl structure. The Constructor will stringify those keys with *Data::Dumper* instead of just returning something like `HASH(0x55dc1b5d0178)`.

  Example:

```
use YAML::PP;
use JSON::PP;
my $ypp = YAML::PP->new;
my $coder = JSON::PP->new->ascii->pretty->allow_nonref->
 ↪canonical;
my $yaml = <<'EOM';
complex:
    ?
        ?
            a: 1
            c: 2
        : 23
    : 42
EOM
my $data = $yppl->load_string($yaml);
say $coder->encode($data);
__END__
{
   "complex" : {
      "{'{a => 1,c => 2}' => 23}" : 42
   }
}
```

TODO:

- Parse Tree

  I would like to generate a complete parse tree, that allows you to manipulate the

data structure and also dump it, including all whitespaces and comments. The spec says that this is throwaway content, but I read that many people wish to be able to keep the comments.

YAML::PP::Dumper, YAML::PP::Emitter

The Dumper should be able to dump strings correctly, adding quotes whenever a plain scalar would look like a special string, like `true`, or when it contains or starts with characters that are not allowed.

Most strings will be dumped as plain scalars without quotes. If they contain special characters or have a special meaning, they will be dumped with single quotes. If they contain control characters, including <"\n">, they will be dumped with double quotes.

It will recognize JSON::PP::Boolean and boolean.pm objects and dump them correctly.

Numbers which also have a PV flag will be recognized as numbers and not as strings:

```
my $int = 23;
say "int: $int"; # $int will now also have a PV flag
```

That means that if you accidentally use a string in numeric context, it will also be recognized as a number:

```
my $string = "23";
my $something = $string + 0;
print $yp->dump_string($string);
# will be emitted as an integer without quotes!
```

The layout is like libyaml output:

```
key:
- a
- b
- c
---
- key1: 1
  key2: 2
  key3: 3
---
- - a1
  - a2
- - b1
  - b2
```

FAQ - Frequently Asked Questions

- Are << merge keys supported?

  Yes, this can be enabled optionally, see *YAML::PP::Schema::Merge*

- Is there a linter / formatter for YAML

There is the widely used "yamllint", based on python's PyYAML. It is very configurable and will report errors or warnings. It cannot format.

Now there is also YAML::Tidy, which will format the given file according to your configuration. So far only a few configuration options exist, but they can already be quite helpful.

Which YAML module should I use?

There are many YAML modules on CPAN. For historical reasons some of them aren't handling YAML correctly.

Most of them are not compatible with the YAML spec and with each other, meaning they can interpret the same YAML differently.

The behaviours we are discussing here can be divided into parsing issues (syntax) and loading/constructing issues (for example type resolving which decides what is a number, boolean or null).

See also https://matrix.yaml.info/ (parsing) and https://perlpunk.github.io/YAML-PP-p5/schema-examples.html (loading).

- *"YAML.pm"*

  It was written even before the YAML 1.0 spec was finished and by that enabled perl users to process YAML very early. It might work for you if you have simple data, but it's missing quite some features and can also produce YAML that doesn't roundtrip. Nowadays it might be a good idea to switch.

- *YAML::XS*

  A libyaml binding that is robust and widely used. However, there are two things to consider.

  1. (syntax) libyaml diverged from the spec in several aspects. They are rare though.
  2. The type resolving does not adhere to YAML 1.1 or YAML 1.2, meaning it is incompatible with other YAML libraries in perl or other languages.

- YAML::Tiny

  It implements both a tiny subset of YAML, but also a superset. Meaning it will happily accept some YAML documents that are not officially valid. Type resolving is also not implemented according to the spec.

- YAML::Syck

  A binding to libsyck. It is even less compatible to YAML than libyaml. Also type resolving is not implemented according to the spec.

- *YAML::PP*

  Regarding YAML syntax, it is the second most YAML 1.2 compatible perl module. The cases it cannot (yet) parse are not relevant in perl programming, e.g. hash keys that are not strings. Regarding type resolving, it is compatible with the YAML 1.2 Core schema, so it should be possible to exchange data as YAML with other libraries in other languages. One downside is that it is the slowest perl YAML module.

- YAML::Parser

This is a parser generated by the YAML grammar, and it's passing all official tests. A YAML::PP::Ref frontend exists that you can use just like YAML::PP. It is quite slow (although it might be ok for small files depending on the use case). The error messages it creates on invalid YAML are not helpful currently.

- YAML::PP::LibYAML

  This combines the YAML::LibYAML::API binding for parsing with the YAML::PP frontend for loading and type resolving. It is faster than YAML::PP but slower than YAML::XS. The divergence from the YAML spec regarding syntax is usually not a problem, and at the same time you have the advantage of being compatible to the YAML 1.2 Core Schema.

WHY

Why did I start to write a new YAML module?

All the available parsers and loaders for Perl are behaving differently, and more important, aren't conforming to the spec. *YAML::XS* is doing pretty well, but `libyaml` only handles YAML 1.1 and diverges a bit from the spec. The pure perl loaders lack support for a number of features.

I was going over *YAML*.pm issues end of 2016, integrating old patches from rt.cpan.org and creating some pull requests myself. I realized that it would be difficult to patch YAML.pm to parse YAML 1.1 or even 1.2, and it would also break existing usages relying on the current behaviour.

In 2016 Ingy döt Net initiated two really cool projects:

- "YAML TEST SUITE"
- "YAML EDITOR"

These projects are a big help for any developer. So I got the idea to write my own parser and started on New Year's Day 2017. Without the test suite and the editor I would have never started this.

I also started another YAML Test project which allows one to get a quick overview of which frameworks support which YAML features:

- "YAML TEST MATRIX"

YAML TEST SUITE

https://github.com/yaml/yaml-test-suite

It contains almost 400 test cases and expected parsing events and more. There will be more tests coming. This test suite allows you to write parsers without turning the examples from the Specification into tests yourself. Also the examples aren't completely covering all cases - the test suite aims to do that.

Thanks also to Felix Krause, who is writing a YAML parser in Nim. He turned all the spec examples into test cases.

YAML EDITOR

This is a tool to play around with several YAML parsers and loaders in vim.

https://github.com/yaml/yaml-editor

The project contains the code to build the frameworks (16 as of this writing) and put it into one big Docker image.

It also contains the yaml-editor itself, which will start a vim in the docker container. It uses a lot of funky vimscript that makes playing with it easy and useful. You can choose which frameworks you want to test and see the output in a grid of vim windows.

Especially when writing a parser it is extremely helpful to have all the test cases and be able to play around with your own examples to see how they are handled.

YAML TEST MATRIX

I was curious to see how the different frameworks handle the test cases, so, using the test suite and the docker image, I wrote some code that runs the tests, manipulates the output to compare it with the expected output, and created a matrix view.

https://github.com/perlpunk/yaml-test-matrix

You can find the latest build at https://matrix.yaml.info

CONTRIBUTORS

- Ingy döt Net

  Ingy is one of the creators of YAML. In 2016 he started the YAML Test Suite and the YAML Editor. He also made useful suggestions on the class hierarchy of YAML::PP.

- Felix “flyx” Krause

  Felix answered countless questions about the YAML Specification.

SEE ALSO

- *YAML*
- *YAML::XS*
- YAML::Syck
- YAML::Tiny
- YAML::PP::LibYAML
- YAML::LibYAML::API
- YAML::Tidy
- https://www.yaml.info/

SPONSORS

The Perl Foundation https://www.perlfoundation.org/ sponsored this project (and the YAML Test Suite) with a grant of 2500 USD in 2017-2018.

COPYRIGHT AND LICENSE

Copyright 2017-2022 by Tina Müller

This library is free software and may be distributed under the same terms as perl itself.

Modules

- `YAML::PP::Common` - Constants and common functions
- `YAML::PP::Constructor` - Constructing data structure from parsing events
- `YAML::PP::Emitter` - Emitting events
- `YAML::PP::Grammar` - YAML grammar
- `YAML::PP::Highlight` - Syntax highlighting utilities
- `YAML::PP::Perl` - Convenience module for loading and dumping Perl objects
- `YAML::PP::Schema` - Schema for YAML::PP
- `YAML::PP::Type`
- `YAML::PP::Writer` - Write YAML output

YAML::PP::Common

Constants and common functions

SYNOPSIS

```
use YAML::PP::Common ':STYLES';
# or
use YAML::PP::Common qw/
    YAML_ANY_SCALAR_STYLE YAML_PLAIN_SCALAR_STYLE
    YAML_SINGLE_QUOTED_SCALAR_STYLE YAML_DOUBLE_QUOTED_SCALAR_STYLE
    YAML_LITERAL_SCALAR_STYLE YAML_FOLDED_SCALAR_STYLE
    YAML_QUOTED_SCALAR_STYLE

    YAML_ANY_SEQUENCE_STYLE
    YAML_BLOCK_SEQUENCE_STYLE YAML_FLOW_SEQUENCE_STYLE

    YAML_ANY_MAPPING_STYLE
    YAML_BLOCK_MAPPING_STYLE YAML_FLOW_MAPPING_STYLE
/;

use YAML::PP::Common ':PRESERVE';
# or
```

(continues on next page)

(continued from previous page)

```
use YAML::PP::Common qw/
    PRESERVE_ALL PRESERVE_ORDER PRESERVE_SCALAR_STYLE PRESERVE_FLOW_
 ↪STYLE
    PRESERVE_ALIAS
/:
```

DESCRIPTION

This module provides common constants and functions for modules working with YAML::PP events.

FUNCTIONS

- event_to_test_suite

  ```
  my $string = YAML::PP::Common::event_to_test_suite($event_prom_
   ↪parser);
  ```

  For examples of the returned format look into this distributions's directory `yaml-test-suite` which is a copy of https://github.com/yaml/yaml-test-suite.

- test_suite_to_event

  ```
  my $event = YAML::PP::Common::test_suite_to_event($str);
  ```

  Turns an event string in test suite format into an event hashref. Not complete yet.

YAML::PP::Constructor

Constructing data structure from parsing events

METHODS

- new

  The Constructor constructor

  ```
  my $constructor = YAML::PP::Constructor->new(
      schema => $schema,
      cyclic_refs => $cyclic_refs,
  );
  ```

- init

  Resets any data being used during construction.

  ```
  $constructor->init;
  ```

- document_start_event, document_end_event, mapping_start_event, mapping_end_event, sequence_start_event, sequence_end_event, scalar_event, alias_event, stream_start_event, stream_end_event

These methods are called from YAML::PP::Parser:

```
$constructor->document_start_event($event);
```

- anchors, set_anchors

  Helper for storing anchors during construction

- docs, set_docs

  Helper for storing resulting documents during construction

- stack, set_stack

  Helper for storing data during construction

- cyclic_refs, set_cyclic_refs

  Option for controlling the behaviour when finding circular references

- schema, set_schema

  Holds a *YAML::PP::Schema* object

- stringify_complex

  When constructing a hash and getting a non-scalar key, this method is used to stringify the key.

  It uses a terse Data::Dumper output. Other modules, like *YAML::XS*, use the default stringification, `ARRAY(0x55617c0c7398)` for example.

YAML::PP::Emitter

Emitting events

SYNOPSIS

```
my $emitter = YAML::PP::Emitter->new(
    indent => 4,
);

$emitter->init;

$emitter->stream_start_event;
$emitter->document_start_event({ implicit => 1 });
$emitter->sequence_start_event;
$emitter->scalar_event({ value => $input, style => $style });
$emitter->sequence_end_event;
$emitter->document_end_event({ implicit => 1 });
$emitter->stream_end_event;

my $yaml = $emitter->writer->output;
$emitter->finish;
```

DESCRIPTION

The emitter emits events to YAML. It provides methods for each event type. The arguments are mostly the same as the events from YAML::PP::Parser.

METHODS

- new

```
my $emitter = YAML::PP::Emitter->new(
    indent => 4,
);
```

  Constructor. Currently takes these options:

  - indent
  - writer

- stream_start_event, stream_end_event, document_start_event, document_end_event, sequence_start_event, sequence_end_event, mapping_start_event, mapping_end_event, scalar_event, alias_event
- indent, set_indent

  Getter/setter for number of indentation spaces.

  TODO: Currently sequences are always zero-indented.

- writer, set_writer

  Getter/setter for the writer object. By default *YAML::PP::Writer*. You can pass your own writer if you want to output the resulting YAML yourself.

- init

  Initialize

- finish

YAML::PP::Grammar

YAML grammar

GRAMMAR

This is the Grammar in YAML

```
# START OF YAML INLINE

# DO NOT CHANGE THIS
# This grammar is automatically generated from etc/grammar.yaml

---
NODETYPE_NODE:
  DASH:
```

(continues on next page)

(continued from previous page)

```
    match: cb_seqstart
    EOL: { new: FULLNODE }
    WS: { new: FULLNODE }

  DEFAULT: { new: NODETYPE_SCALAR_OR_MAP }

NODETYPE_SCALAR_OR_MAP:

  # Flow nodes can follow tabs
  WS: { new: FULLMAPVALUE_INLINE }

  ALIAS:
    match: cb_alias
    EOL: { match: cb_send_alias_from_stack }
    WS:
      COLON:
        match: cb_insert_map_alias
        EOL: { new: FULLNODE }
        WS: { new: FULLMAPVALUE_INLINE }

  QUESTION:
    match: cb_questionstart
    EOL: { new: FULLNODE }
    WS: { new: FULLNODE }

  QUOTED:
    match: cb_take_quoted
    EOL: { match: cb_send_scalar }
    WS:
      COLON:
        match: cb_insert_map
        EOL: { new: FULLNODE }
        WS: { new: FULLMAPVALUE_INLINE }
    COLON:
      match: cb_insert_map
      EOL: { new: FULLNODE }
      WS: { new: FULLMAPVALUE_INLINE }

  QUOTED_MULTILINE:
    match: cb_quoted_multiline
    EOL: {  }


  PLAIN:
    match: cb_start_plain
    EOL:
      match: cb_send_scalar
    WS:
      COLON:
```

(continues on next page)

(continued from previous page)

```
        match: cb_insert_map
        EOL: { new: FULLNODE }
        WS: { new: FULLMAPVALUE_INLINE }
    COLON:
      match: cb_insert_map
      EOL: { new: FULLNODE }
      WS: { new: FULLMAPVALUE_INLINE }

  PLAIN_MULTI:
    match: cb_send_plain_multi
    EOL: { }

  COLON:
    match: cb_insert_empty_map
    EOL: { new: FULLNODE }
    WS: { new: FULLMAPVALUE_INLINE }

  BLOCK_SCALAR:
    match: cb_send_block_scalar
    EOL: { }

  FLOWSEQ_START:
    match: cb_start_flowseq
    new: NEWFLOWSEQ

  FLOWMAP_START:
    match: cb_start_flowmap
    new: NEWFLOWMAP

  DOC_END:
    match: cb_end_document
    EOL: { }

  DOC_START:
    match: cb_end_doc_start_document
    EOL: { new: FULLNODE }
    WS: { new: FULLNODE }

  EOL:
    new: NODETYPE_SCALAR_OR_MAP

NODETYPE_COMPLEX:
  COLON:
    match: cb_complexcolon
    EOL: { new: FULLNODE }
    WS: { new: FULLNODE }
  DEFAULT:
    match: cb_empty_complexvalue
    new: NODETYPE_MAP
```

(continues on next page)

(continued from previous page)

```
  EOL:
    new: NODETYPE_COMPLEX

RULE_FULLFLOWSCALAR:
  ANCHOR:
    match: cb_anchor
    EOL: { new: RULE_FULLFLOWSCALAR_ANCHOR }
    DEFAULT: { new: RULE_FULLFLOWSCALAR_ANCHOR }
  TAG:
    match: cb_tag
    EOL: { new: RULE_FULLFLOWSCALAR_TAG }
    DEFAULT: { new: RULE_FULLFLOWSCALAR_TAG }
  DEFAULT: { new: RULE_FLOWSCALAR }

RULE_FULLFLOWSCALAR_ANCHOR:
  WS: { new: RULE_FULLFLOWSCALAR_ANCHOR }
  TAG:
    match: cb_tag
    WS: { new: RULE_FLOWSCALAR }
    EOL: { new: RULE_FLOWSCALAR }
  DEFAULT: { new: RULE_FLOWSCALAR }

RULE_FULLFLOWSCALAR_TAG:
  WS: { new: RULE_FULLFLOWSCALAR_TAG }
  ANCHOR:
    match: cb_anchor
    WS: { new: RULE_FLOWSCALAR }
    EOL: { new: RULE_FLOWSCALAR }
  DEFAULT: { new: RULE_FLOWSCALAR }

RULE_FLOWSCALAR:
  FLOWSEQ_START: { match: cb_start_flowseq, new: NEWFLOWSEQ }
  FLOWMAP_START: { match: cb_start_flowmap, new: NEWFLOWMAP }

  ALIAS: { match: cb_send_alias, return: 1 }

  QUOTED:
    match: cb_take_quoted
    EOL: { match: cb_send_scalar }
    WS: { match: cb_send_scalar, return: 1 }
    DEFAULT: { match: cb_send_scalar, return: 1 }

  QUOTED_MULTILINE: { match: cb_quoted_multiline, return: 1 }

  PLAIN:
    match: cb_start_plain
    EOL: { match: cb_send_scalar }
    DEFAULT: { match: cb_send_scalar, return: 1 }
```

(continues on next page)

(continued from previous page)

```
  PLAIN_MULTI: { match: cb_send_plain_multi, return: 1 }

  FLOW_COMMA: { match: cb_empty_flow_mapkey, return: 1 }

  FLOWMAP_END:
    match: cb_end_flowmap_empty
    return: 1

FLOWSEQ:
  FLOWSEQ_START: { match: cb_start_flowseq, new: NEWFLOWSEQ }
  FLOWMAP_START: { match: cb_start_flowmap, new: NEWFLOWMAP }

  ALIAS: { match: cb_send_flow_alias, new: FLOWSEQ_NEXT }

  PLAIN:
    match: cb_start_plain
    EOL:
      match: cb_send_scalar
      new: FLOWSEQ_NEXT
    DEFAULT:
      new: FLOWSEQ_MAYBE_KEY
  PLAIN_MULTI: { match: cb_send_plain_multi, new: FLOWSEQ_NEXT }

  QUOTED:
    match: cb_take_quoted
    EOL:
      match: cb_send_scalar
      new: FLOWSEQ_NEXT
    DEFAULT:
      new: FLOWSEQ_MAYBE_KEY
  QUOTED_MULTILINE: { match: cb_quoted_multiline, new: FLOWSEQ_NEXT␣
↪}

  COLON:
    WS:
      match: cb_insert_empty_implicit_flowseq_map
      new: RULE_FULLFLOWSCALAR
    EOL:
      match: cb_insert_empty_implicit_flowseq_map
      new: RULE_FULLFLOWSCALAR

FLOWSEQ_PROPS:
  FLOWSEQ_START: { match: cb_start_flowseq, new: NEWFLOWSEQ }
  FLOWMAP_START: { match: cb_start_flowmap, new: NEWFLOWMAP }

  PLAIN:
    match: cb_start_plain
    EOL:
      match: cb_send_scalar
```

(continues on next page)

(continued from previous page)

```
      new: FLOWSEQ_NEXT
    DEFAULT:
      new: FLOWSEQ_MAYBE_KEY
  PLAIN_MULTI: { match: cb_send_plain_multi, new: FLOWSEQ_NEXT }

  QUOTED:
    match: cb_take_quoted
    EOL:
      match: cb_send_scalar
      new: FLOWSEQ_NEXT
    DEFAULT:
      new: FLOWSEQ_MAYBE_KEY
  QUOTED_MULTILINE: { match: cb_quoted_multiline, new: FLOWSEQ_NEXT␣
↪}

  FLOW_COMMA:
    match: cb_empty_flowseq_comma
    return: 1
  FLOWSEQ_END:
    match: cb_empty_flowseq_end
    return: 1

  COLON:
    WS:
      match: cb_insert_empty_implicit_flowseq_map
      new: RULE_FULLFLOWSCALAR
    EOL:
      match: cb_insert_empty_implicit_flowseq_map
      new: RULE_FULLFLOWSCALAR

FLOWSEQ_EMPTY:
  FLOW_COMMA:
    match: cb_empty_flowseq_comma
    return: 1
  FLOWSEQ_END:
    match: cb_empty_flowseq_end
    return: 1

FLOWSEQ_NEXT:
  WS: { new: FLOWSEQ_NEXT }
  EOL: { new: FLOWSEQ_NEXT }

  FLOW_COMMA:
    match: cb_flow_comma
    return: 1
  FLOWSEQ_END:
    match: cb_end_flowseq
    return: 1
```

(continues on next page)

(continued from previous page)

```
FLOWSEQ_MAYBE_KEY:
  WS: { new: FLOWSEQ_MAYBE_KEY }

  COLON:
    WS:
      match: cb_insert_implicit_flowseq_map
      new: RULE_FULLFLOWSCALAR
    EOL:
      match: cb_insert_implicit_flowseq_map
      new: RULE_FULLFLOWSCALAR
    DEFAULT:
      match: cb_insert_implicit_flowseq_map
      new: RULE_FULLFLOWSCALAR
  DEFAULT:
    new: FLOWSEQ_NEXT

FLOWMAP_CONTENT:
  FLOWSEQ_START: { match: cb_start_flowseq, new: NEWFLOWSEQ }
  FLOWMAP_START: { match: cb_start_flowmap, new: NEWFLOWMAP }

  ALIAS: { match: cb_send_alias, return: 1 }

  PLAIN: { match: cb_flowkey_plain, return: 1 }
  PLAIN_MULTI: { match: cb_send_plain_multi, return: 1 }

  QUOTED: { match: cb_flowkey_quoted, return: 1 }
  QUOTED_MULTILINE: { match: cb_quoted_multiline, return: 1 }

  COLON:
    WS:
      match: cb_empty_flow_mapkey
      new: RULE_FULLFLOWSCALAR
    EOL:
      match: cb_empty_flow_mapkey
      new: RULE_FULLFLOWSCALAR

FLOWMAP_PROPS:
  FLOWSEQ_START: { match: cb_start_flowseq, new: NEWFLOWSEQ }
  FLOWMAP_START: { match: cb_start_flowmap, new: NEWFLOWMAP }

  PLAIN: { match: cb_flowkey_plain, return: 1 }
  PLAIN_MULTI: { match: cb_send_plain_multi, return: 1 }

  QUOTED: { match: cb_flowkey_quoted, return: 1 }
  QUOTED_MULTILINE: { match: cb_quoted_multiline, return: 1 }

  COLON:
    WS:
      match: cb_empty_flow_mapkey
```

(continues on next page)

(continued from previous page)

```
      new: RULE_FULLFLOWSCALAR
    EOL:
      match: cb_empty_flow_mapkey
      new: RULE_FULLFLOWSCALAR
  FLOW_COMMA:
    match: cb_empty_flowmap_key_value
    return: 1
  FLOWMAP_END:
    match: cb_end_empty_flowmap_key_value
    return: 1

FLOWMAP_EMPTYKEY:
  FLOW_COMMA:
    match: cb_empty_flowmap_key_value
    return: 1
  FLOWMAP_END:
    match: cb_end_empty_flowmap_key_value
    return: 1

NEWFLOWSEQ:
  EOL: { new: NEWFLOWSEQ }
  WS: { new: NEWFLOWSEQ }

  ANCHOR:
    match: cb_anchor
    WS: { new: NEWFLOWSEQ_ANCHOR_SPC }
    EOL: { new: NEWFLOWSEQ_ANCHOR_SPC }
    DEFAULT: { new: NEWFLOWSEQ_ANCHOR }

  TAG:
    match: cb_tag
    WS: { new: NEWFLOWSEQ_TAG_SPC }
    EOL: { new: NEWFLOWSEQ_TAG_SPC }
    DEFAULT: { new: NEWFLOWSEQ_TAG }

  FLOWSEQ_END:
    match: cb_end_flowseq
    return: 1

  DEFAULT: { new: FLOWSEQ }

NODETYPE_FLOWSEQ:
  EOL: { new: NODETYPE_FLOWSEQ }
  WS: { new: NODETYPE_FLOWSEQ }
  FLOWSEQ_END:
    match: cb_end_flowseq
    return: 1
  DEFAULT: { new: NEWFLOWSEQ }
```

(continues on next page)

(continued from previous page)

```
NODETYPE_FLOWMAPVALUE:
  WS: { new: NODETYPE_FLOWMAPVALUE }
  EOL: { new: NODETYPE_FLOWMAPVALUE }
  COLON:
    match: cb_flow_colon
    WS: { new: RULE_FULLFLOWSCALAR }
    EOL: { new: RULE_FULLFLOWSCALAR }
    DEFAULT: { new: RULE_FULLFLOWSCALAR }
  FLOW_COMMA:
    match: cb_empty_flowmap_value
    return: 1
  FLOWMAP_END:
    match: cb_end_flowmap_empty
    return: 1

NEWFLOWSEQ_ANCHOR:
  DEFAULT: { new: FLOWSEQ_EMPTY }
NEWFLOWSEQ_TAG:
  DEFAULT: { new: FLOWSEQ_EMPTY }

NEWFLOWSEQ_ANCHOR_SPC:
  WS: { new: NEWFLOWSEQ_ANCHOR_SPC }
  EOL: { new: NEWFLOWSEQ_ANCHOR_SPC }
  TAG:
    match: cb_tag
    WS: { new: FLOWSEQ_PROPS }
    EOL: { new: FLOWSEQ_PROPS }
    DEFAULT: { new: FLOWSEQ_EMPTY }
  DEFAULT: { new: FLOWSEQ_PROPS }

NEWFLOWSEQ_TAG_SPC:
  WS: { new: NEWFLOWSEQ_TAG_SPC }
  EOL: { new: NEWFLOWSEQ_TAG_SPC }
  ANCHOR:
    match: cb_anchor
    WS: { new: FLOWSEQ_PROPS }
    EOL: { new: FLOWSEQ_PROPS }
    DEFAULT: { new: FLOWSEQ_EMPTY }
  DEFAULT: { new: FLOWSEQ_PROPS }


NEWFLOWMAP_ANCHOR:
  DEFAULT: { new: FLOWMAP_EMPTYKEY }
NEWFLOWMAP_TAG:
  DEFAULT: { new: FLOWMAP_EMPTYKEY }

NEWFLOWMAP_ANCHOR_SPC:
  WS: { new: NEWFLOWMAP_ANCHOR_SPC }
  EOL: { new: NEWFLOWMAP_ANCHOR_SPC }
```

(continues on next page)

(continued from previous page)

```
  TAG:
    match: cb_tag
    WS: { new: FLOWMAP_PROPS }
    EOL: { new: FLOWMAP_PROPS }
    DEFAULT: { new: FLOWMAP_EMPTYKEY }
  DEFAULT: { new: FLOWMAP_PROPS }

NEWFLOWMAP_TAG_SPC:
  WS: { new: NEWFLOWMAP_TAG_SPC }
  EOL: { new: NEWFLOWMAP_TAG_SPC }
  ANCHOR:
    match: cb_anchor
    WS: { new: FLOWMAP_PROPS }
    EOL: { new: FLOWMAP_PROPS }
    DEFAULT: { new: FLOWMAP_EMPTYKEY }
  DEFAULT: { new: FLOWMAP_PROPS }

NEWFLOWMAP:
  EOL: { new: NEWFLOWMAP }
  WS: { new: NEWFLOWMAP }
  QUESTION: { match: cb_flow_question, new: FLOWMAP_EXPLICIT_KEY }
  DEFAULT: { new: FLOWMAP }


FLOWMAP_EXPLICIT_KEY:
  WS: { new: FLOWMAP_EXPLICIT_KEY }
  EOL: { new: FLOWMAP_EXPLICIT_KEY }
  FLOWMAP_END:
    match: cb_end_empty_flowmap_key_value
    return: 1
  FLOW_COMMA:
    match: cb_empty_flowmap_key_value
    return: 1
  DEFAULT: { new: FLOWMAP }

FLOWMAP:
  EOL: { new: FLOWMAP }
  WS: { new: FLOWMAP }

  ANCHOR:
    match: cb_anchor
    WS: { new: NEWFLOWMAP_ANCHOR_SPC }
    EOL: { new: NEWFLOWMAP_ANCHOR_SPC }
    DEFAULT: { new: NEWFLOWMAP_ANCHOR }
  TAG:
    match: cb_tag
    WS: { new: NEWFLOWMAP_TAG_SPC }
    EOL: { new: NEWFLOWMAP_TAG_SPC }
    DEFAULT: { new: NEWFLOWMAP_TAG }
```

(continues on next page)

(continued from previous page)

```
  FLOWMAP_END:
    match: cb_end_flowmap
    return: 1

  COLON:
    WS:
      match: cb_empty_flow_mapkey
      new: RULE_FULLFLOWSCALAR
    EOL:
      match: cb_empty_flow_mapkey
      new: RULE_FULLFLOWSCALAR

  DEFAULT: { new: FLOWMAP_CONTENT }

NODETYPE_FLOWMAP:
  EOL: { new: NODETYPE_FLOWMAP }
  WS: { new: NODETYPE_FLOWMAP }
  FLOWMAP_END:
    match: cb_end_flowmap
    return: 1
  FLOW_COMMA: { match: cb_flow_comma, new: NEWFLOWMAP }
  DEFAULT: { new: NEWFLOWMAP }

END_FLOW:
  EOL:
    match: cb_end_outer_flow
    return: 1

RULE_MAPKEY:
  QUESTION:
    match: cb_question
    EOL: { new: FULLNODE }
    WS: { new: FULLNODE }
  ALIAS:
    match: cb_send_alias_key
    WS:
      COLON:
        EOL: { new: FULLNODE }
        WS: { new: FULLMAPVALUE_INLINE }

  QUOTED:
    match: cb_take_quoted_key
    WS:
      COLON:
        EOL: { new: FULLNODE }
        WS: { new: FULLMAPVALUE_INLINE }
    COLON:
      EOL: { new: FULLNODE }
```

(continues on next page)

```
      WS: { new: FULLMAPVALUE_INLINE }

  PLAIN:
    match: cb_mapkey
    WS:
      COLON:
        match: cb_send_mapkey
        EOL: { new: FULLNODE }
        WS: { new: FULLMAPVALUE_INLINE }
    COLON:
      match: cb_send_mapkey
      EOL: { new: FULLNODE }
      WS: { new: FULLMAPVALUE_INLINE }

  COLON:
    match: cb_empty_mapkey
    EOL: { new: FULLNODE }
    WS: { new: FULLMAPVALUE_INLINE }

  DOC_END:
    match: cb_end_document
    EOL: { }

  DOC_START:
    match: cb_end_doc_start_document
    EOL: { new: FULLNODE }
    WS: { new: FULLNODE }

  EOL:
    new: RULE_MAPKEY


NODETYPE_SEQ:
  DASH:
    match: cb_seqitem
    EOL: { new: FULLNODE }
    WS: { new: FULLNODE }
  DOC_END:
    match: cb_end_document
    EOL: { }
  DOC_START:
    match: cb_end_doc_start_document
    EOL: { new: FULLNODE }
    WS: { new: FULLNODE }

  EOL:
    new: NODETYPE_SEQ

NODETYPE_MAP:
  ANCHOR:
```

(continues on next page)

(continued from previous page)

```
    match: cb_anchor
    WS:
      TAG:
        match: cb_tag
        WS: { new: RULE_MAPKEY  }
      DEFAULT: { new: RULE_MAPKEY }
  TAG:
    match: cb_tag
    WS:
      ANCHOR:
        match: cb_anchor
        WS: { new: RULE_MAPKEY  }
      DEFAULT: { new: RULE_MAPKEY }
  DEFAULT: { new: RULE_MAPKEY }

FULLNODE_ANCHOR:
  TAG:
    match: cb_tag
    EOL: { match: cb_property_eol, new: FULLNODE_TAG_ANCHOR }
    WS:
      ANCHOR:
        match: cb_anchor
        WS: { new: NODETYPE_SCALAR_OR_MAP  }
      DEFAULT: { new: NODETYPE_SCALAR_OR_MAP }
  ANCHOR:
    match: cb_anchor
    WS:
      TAG:
        match: cb_tag
        WS: { new: NODETYPE_SCALAR_OR_MAP  }
      DEFAULT: { new: NODETYPE_SCALAR_OR_MAP }
  EOL: { new: FULLNODE_ANCHOR }
  DEFAULT: { new: NODETYPE_NODE }

FULLNODE_TAG:
  ANCHOR:
    match: cb_anchor
    EOL: { match: cb_property_eol, new: FULLNODE_TAG_ANCHOR }
    WS:
      TAG:
        match: cb_tag
        WS: { new: NODETYPE_SCALAR_OR_MAP  }
      DEFAULT: { new: NODETYPE_SCALAR_OR_MAP, }
  TAG:
    match: cb_tag
    WS:
      ANCHOR:
        match: cb_anchor
        WS: { new: NODETYPE_SCALAR_OR_MAP  }
```

(continues on next page)

(continued from previous page)

```
      DEFAULT: { new: NODETYPE_SCALAR_OR_MAP }
  EOL: { new: FULLNODE_TAG }
  DEFAULT: { new: NODETYPE_NODE }

FULLNODE_TAG_ANCHOR:
  ANCHOR:
    match: cb_anchor
    WS:
      TAG:
        match: cb_tag
        WS: { new: NODETYPE_SCALAR_OR_MAP  }
      DEFAULT: { new: NODETYPE_SCALAR_OR_MAP }
  TAG:
    match: cb_tag
    WS:
      ANCHOR:
        match: cb_anchor
        WS: { new: NODETYPE_SCALAR_OR_MAP  }
      DEFAULT: { new: NODETYPE_SCALAR_OR_MAP }
  EOL: { new: FULLNODE_TAG_ANCHOR }
  DEFAULT: { new: NODETYPE_NODE }

FULLNODE:
  ANCHOR:
    match: cb_anchor
    EOL: { match: cb_property_eol, new: FULLNODE_ANCHOR }
    WS:
      TAG:
        match: cb_tag
        EOL: { match: cb_property_eol, new: FULLNODE_TAG_ANCHOR }
        WS: { new: NODETYPE_SCALAR_OR_MAP  }
      DEFAULT: { new: NODETYPE_SCALAR_OR_MAP }
  TAG:
    match: cb_tag
    EOL: { match: cb_property_eol, new: FULLNODE_TAG }
    WS:
      ANCHOR:
        match: cb_anchor
        EOL: { match: cb_property_eol, new: FULLNODE_TAG_ANCHOR }
        WS: { new: NODETYPE_SCALAR_OR_MAP  }
      DEFAULT: { new: NODETYPE_SCALAR_OR_MAP }
  EOL: { new: FULLNODE }
  DEFAULT: { new: NODETYPE_NODE }

FULLMAPVALUE_INLINE:
  ANCHOR:
    match: cb_anchor
    EOL: { match: cb_property_eol, new: FULLNODE_ANCHOR }
    WS:
```

(continues on next page)

(continued from previous page)

```
      TAG:
        match: cb_tag
        EOL: { match: cb_property_eol, new: FULLNODE_TAG_ANCHOR }
        WS: { new: NODETYPE_MAPVALUE_INLINE  }
      DEFAULT: { new: NODETYPE_MAPVALUE_INLINE }
  TAG:
    match: cb_tag
    EOL: { match: cb_property_eol, new: FULLNODE_TAG }
    WS:
      ANCHOR:
        match: cb_anchor
        EOL: { match: cb_property_eol, new: FULLNODE_TAG_ANCHOR }
        WS: { new: NODETYPE_MAPVALUE_INLINE  }
      DEFAULT: { new: NODETYPE_MAPVALUE_INLINE }
  DEFAULT: { new: NODETYPE_MAPVALUE_INLINE }


NODETYPE_MAPVALUE_INLINE:
  ALIAS:
    match: cb_send_alias
    EOL: { }

  QUOTED:
    match: cb_take_quoted
    EOL: { match: cb_send_scalar }

  QUOTED_MULTILINE:
    match: cb_quoted_multiline
    EOL: { }

  PLAIN:
    match: cb_start_plain
    EOL:
      match: cb_send_scalar

  PLAIN_MULTI:
    match: cb_send_plain_multi
    EOL: { }

  BLOCK_SCALAR:
    match: cb_send_block_scalar
    EOL: { }

  FLOWSEQ_START:
    match: cb_start_flowseq
    new: NEWFLOWSEQ

  FLOWMAP_START:
    match: cb_start_flowmap
```

(continues on next page)

(continued from previous page)

```
    new: NEWFLOWMAP

  DOC_END:
    match: cb_end_document
    EOL: { }


DOCUMENT_END:
  DOC_END:
    match: cb_end_document
    EOL: { }
  DOC_START:
    match: cb_end_doc_start_document
    EOL: { new: FULLNODE }
    WS: { new: FULLNODE }

  EOL:
    new: DOCUMENT_END


STREAM:

  DOC_END:
    match: cb_end_document_empty
    EOL: {  }
  DOC_START:
    match: cb_doc_start_explicit
    EOL: { new: FULLNODE }
    WS: { new: FULLNODE }
  YAML_DIRECTIVE:
    match: cb_set_yaml_version_directive
    EOL: { new: DIRECTIVE }
    WS: { new: DIRECTIVE }
  RESERVED_DIRECTIVE:
    match: cb_reserved_directive
    EOL: { new: DIRECTIVE }
    WS: { new: DIRECTIVE }
  TAG_DIRECTIVE:
    match: cb_tag_directive
    EOL: { new: DIRECTIVE }
    WS: { new: DIRECTIVE }

  EOL:
    new: STREAM

  DEFAULT:
    match: cb_doc_start_implicit
    new: FULLNODE
```

(continues on next page)

(continued from previous page)

```
DIRECTIVE:
  DOC_START:
    match: cb_doc_start_explicit
    EOL: { new: FULLNODE }
    WS: { new: FULLNODE }

  YAML_DIRECTIVE:
    match: cb_set_yaml_version_directive
    EOL: { new: DIRECTIVE }
    WS: { new: DIRECTIVE }
  RESERVED_DIRECTIVE:
    match: cb_reserved_directive
    EOL: { new: DIRECTIVE }
    WS: { new: DIRECTIVE }
  TAG_DIRECTIVE:
    match: cb_tag_directive
    EOL: { new: DIRECTIVE }
    WS: { new: DIRECTIVE }

  EOL:
    new: DIRECTIVE


# END OF YAML INLINE
```

YAML::PP::Highlight

Syntax highlighting utilities

SYNOPSIS

```
use YAML::PP::Highlight qw/ Dump /;

my $highlighted = Dump $data;
```

FUNCTIONS

- Dump

```
use YAML::PP::Highlight qw/ Dump /;

my $highlighted = Dump $data;
my $highlighted = Dump @docs;
```

It will dump the given data, and then parse it again to create tokens, which are then highlighted with ansi colors.

The return value is ansi colored YAML.

YAML::PP::Perl

Convenience module for loading and dumping Perl objects

SYNOPSIS

```
use YAML::PP::Perl;
my @docs = YAML::PP::Perl->new->load_string($yaml);
my @docs = YAML::PP::Perl::Load($yaml);

# same as
use YAML::PP;
my $yp = YAML::PP->new( schema => [qw/ Core Perl /] );
my @docs = $yp->load_string($yaml);
```

DESCRIPTION

This is just for convenience. It will create a YAML::PP object using the default schema (`Core`) and the *YAML::PP::Schema::Perl* schema.

See *YAML::PP::Schema::Perl* for documentation.

METHODS

- Load, Dump, LoadFile, DumpFile

  These work like the functions in *YAML::PP*, just adding the `Perl` schema.

- new

  Constructor, works like in *YAML::PP*, just adds the `Perl` schema to the list of arguments.

YAML::PP::Schema

Schema for YAML::PP

Modules

- `YAML::PP::Schema::Binary` - Schema for loading and binary data
- `YAML::PP::Schema::Catchall` - YAML 1.2 JSON Schema
- `YAML::PP::Schema::Core` - YAML 1.2 Core Schema
- `YAML::PP::Schema::Failsafe` - YAML 1.2 Failsafe Schema
- `YAML::PP::Schema::Include` - Include YAML files
- `YAML::PP::Schema::JSON` - YAML 1.2 JSON Schema
- `YAML::PP::Schema::Merge` - Enabling YAML merge keys for mappings
- `YAML::PP::Schema::Perl` - Schema for serializing perl objects and special types
- `YAML::PP::Schema::Tie`

- `YAML::PP::Schema::YAML1_1` - YAML 1.1 Schema for YAML::PP

YAML::PP::Schema::Binary

Schema for loading and binary data

SYNOPSIS

```
use YAML::PP;
my $yp = YAML::PP->new( schema => [qw/ + Binary /] );
# or

my ($binary, $same_binary) = $yp->load_string(<<'EOM');
--- !!binary "\
  R0lGODlhDAAMAIQAAP//9/X17unp5WZmZgAAAOfn515eXvPz7Y6OjuDg4J+fn5\
  OTk6enp56enmlpaWNjY6Ojo4SEhP/++f/++f/++f/++f/++f/++f/++f/+\
  +f/++f/++f/++f/++f/++SH+Dk1hZGUgd2l0aCBHSU1QACwAAAAADAAMAAAFLC\
  AgjoEwnuNAFOhpEMTRiggcz4BNJHrv/zCFcLiwMWYNG84BwwEeECcgggoBADs="
--- !!binary |
  R0lGODlhDAAMAIQAAP//9/X17unp5WZmZgAAAOfn515eXvPz7Y6OjuDg4J+fn5
  OTk6enp56enmlpaWNjY6Ojo4SEhP/++f/++f/++f/++f/++f/++f/++f/+
  +f/++f/++f/++f/++f/++SH+Dk1hZGUgd2l0aCBHSU1QACwAAAAADAAMAAAFLC
  AgjoEwnuNAFOhpEMTRiggcz4BNJHrv/zCFcLiwMWYNG84BwwEeECcgggoBADs=
# The binary value above is a tiny arrow encoded as a gif image.
EOM
```

DESCRIPTION

See <https://yaml.org/type/binary.html>

By prepending a base64 encoded binary string with the `!!binary` tag, it can be automatically decoded when loading.

Note that the logic for dumping is probably broken, see https://github.com/perlpunk/YAML-PP-p5/issues/28.

Suggestions welcome.

METHODS

- register

  Called by *YAML::PP::Schema*

YAML::PP::Schema::Catchall

YAML 1.2 JSON Schema

SYNOPSIS

```
my $yp = YAML::PP->new( schema => ['JSON'] );
my $yp = YAML::PP->new( schema => [qw/ JSON empty=str /] );
my $yp = YAML::PP->new( schema => [qw/ JSON empty=null /] );
```

DESCRIPTION

YAML::PP::Schema::Core

YAML 1.2 Core Schema

SYNOPSIS

```
my $yp = YAML::PP->new( schema => ['Core'] );
```

DESCRIPTION

This schema is the official recommended Core Schema for YAML 1.2. It loads additional values to the JSON schema as special types, for example `TRUE` and `True` additional to `true`.

Official Schema: https://yaml.org/spec/1.2/spec.html#id2804923

Here you can see all Schemas and examples implemented by YAML::PP: https://perlpunk.github.io/YAML-PP-p5/schemas.html

METHODS

- register

  Called by YAML::PP::Schema

YAML::PP::Schema::Failsafe

YAML 1.2 Failsafe Schema

SYNOPSIS

```
my $yp = YAML::PP->new( schema => ['Failsafe'] );
```

DESCRIPTION

With this schema, everything will be treated as a string. There are no booleans, integers, floats or undefined values.

Here you can see all Schemas and examples implemented by YAML::PP: https://perlpunk.github.io/YAML-PP-p5/schemas.html

Official Schema: https://yaml.org/spec/1.2/spec.html#id2802346

METHODS

- register

  Called by YAML::PP::Schema

YAML::PP::Schema::Include

Include YAML files

SYNOPSIS

```
# /path/to/file.yaml
# ---
# included: !include include/file2.yaml

# /path/to/include/file2.yaml
# ---
# a: b

my $include = YAML::PP::Schema::Include->new;

my $yp = YAML::PP->new( schema => ['+', $include] );
# we need the original YAML::PP object for getting the current␣
 ↪filename
# and for loading another file
$include->yp($yp);

my ($data) = $yp->load_file("/path/to/file.yaml");

# The result will be:
$data = {
    included => { a => 'b' }
};
```

Allow absolute filenames and upwards '..':

```
my $include = YAML::PP::Schema::Include->new(
    allow_absolute => 1, # default: 0
);
```

Specify paths to search for includes:

```
my @include_paths = ("/path/to/include/yaml/1", "/path/to/include/
 ↪yaml/2");
my $include = YAML::PP::Schema::Include->new(
    paths => \@include_paths,
);
my $yp = YAML::PP->new( schema => ['+', $include] );
$include->yp($yp);
```

(continues on next page)

(continued from previous page)

```
# /path/to/include/yaml/1/file1.yaml
# ---
# a: b

my $yaml = <<'EOM';
- included: !include file1.yaml
EOM
my ($data) = $yp->load_string($yaml);
```

DESCRIPTION

This plugin allows you to split a large YAML file into smaller ones. You can then include these files with the `!include` tag.

It will search for the specified filename relative to the currently processed filename.

You can also specify the paths where to search for files to include. It iterates through the paths and returns the first filename that exists.

By default, only relative paths are allowed. Any `../` in the path will be removed. You can change that behaviour by setting the option `allow_absolute` to true.

If the included file contains more than one document, only the first one will be included.

I will probably add a possibility to return all documents as an arrayref.

The included YAML file will be loaded by creating a new *YAML::PP* object with the schema from the existing object. This way you can recursively include files.

You can even reuse the same include via an alias:

```
---
invoice:
    shipping address: &address !include address.yaml
    billing address: *address
```

Circular includes will be detected, and will be fatal.

It's possible to specify what to do with the included file:

```
my $include = YAML::PP::Schema::Include->new(
    loader => sub {
        my ($yp, $filename);
        if ($filename =~ m/\.txt$/) {
            # open file and just return text
        }
        else {
            # default behaviour
            return $yp->load_file($filename);
        }
    },
);
```

For example, RAML defines an `!include` tag which depends on the file content. If it contains a special RAML directive, it will be loaded as YAML, otherwise the content of the file will be included as a string.

So with this plugin you are able to read RAML specifications.

YAML::PP::Schema::JSON

YAML 1.2 JSON Schema

SYNOPSIS

```
my $yp = YAML::PP->new( schema => ['JSON'] );
my $yp = YAML::PP->new( schema => [qw/ JSON empty=str /] );
my $yp = YAML::PP->new( schema => [qw/ JSON empty=null /] );
```

DESCRIPTION

With this schema, the resolution of plain values will work like in JSON. Everything that matches a special value will be loaded as such, other plain scalars will be loaded as strings.

Note that this is different from the official YAML 1.2 JSON Schema, where all strings have to be quoted.

Here you can see all Schemas and examples implemented by YAML::PP: https://perlpunk.github.io/YAML-PP-p5/schemas.html

Official Schema: https://yaml.org/spec/1.2/spec.html#id2803231

CONFIGURATION

The official YAML 1.2 JSON Schema wants all strings to be quoted. YAML::PP currently does not require that (it might do this optionally in the future).

That means, there are no empty nodes allowed in the official schema. Example:

```
---
key:
```

The default behaviour of YAML::PP::Schema::JSON is to return an empty string, so it would be equivalent to:

```
---
key: ''
```

You can configure it to resolve this as `undef`:

```
my $yp = YAML::PP->new( schema => [qw/ JSON empty=null /] );
```

This way it is equivalent to:

```
---
key: null
```

The default is:

```
my $yp = YAML::PP->new( schema => [qw/ JSON empty=str /] );
```

METHODS

- register

  Called by YAML::PP::Schema

- represent_bool, represent_float, represent_int, represent_literal, represent_undef

  Functions to represent the several node types.

  ```
  represent_bool($representer, $node);
  ```

YAML::PP::Schema::Merge

Enabling YAML merge keys for mappings

SYNOPSIS

```
use YAML::PP;
my $yp = YAML::PP->new( schema => [qw/ + Merge /] );

my $yaml = <<'EOM';
---
- &CENTER { x: 1, y: 2 }
- &LEFT { x: 0, y: 2 }
- &BIG { r: 10 }
- &SMALL { r: 1 }

# All the following maps are equal:

- # Explicit keys
  x: 1
  y: 2
  r: 10
  label: center/big

- # Merge one map
  << : *CENTER
  r: 10
  label: center/big

- # Merge multiple maps
  << : [ *CENTER, *BIG ]
```

(continues on next page)

(continued from previous page)

```
  label: center/big

- # Override
  << : [ *BIG, *LEFT, *SMALL ]
  x: 1
  label: center/big
EOM
my $data = $yp->load_string($yaml);
# $data->[4] == $data->[5] == $data->[6] == $data->[7]
```

DESCRIPTION

See https://yaml.org/type/merge.html for the specification.

Quote:

"Specify one or more mappings to be merged with the current one.

The << merge key is used to indicate that all the keys of one or more specified maps should be inserted into the current map. If the value associated with the key is a single mapping node, each of its key/value pairs is inserted into the current mapping, unless the key already exists in it. If the value associated with the merge key is a sequence, then this sequence is expected to contain mapping nodes and each of these nodes is merged in turn according to its order in the sequence. Keys in mapping nodes earlier in the sequence override keys specified in later mapping nodes."

The implementation of this in a generic way is not trivial, because we also have to handle duplicate keys, and YAML::PP allows you to write your own handler for processing mappings.

So the inner API of that is not stable at this point.

Note that if you enable this schema, a plain scalar `<<` will be seen as special anywhere in your document, so if you want a literal `<<`, you have to put it in quotes.

Note that the performed merge is not a "deep merge". Only top-level keys are merged.

METHODS

- register

  Called by YAML::PP::Schema

YAML::PP::Schema::Perl

Schema for serializing perl objects and special types

SYNOPSIS

```
use YAML::PP;
# This can be dangerous when loading untrusted YAML!
my $yp = YAML::PP->new( schema => [qw/ + Perl /] );
```

(continues on next page)

(continued from previous page)

```
# or
my $yp = YAML::PP->new( schema => [qw/ Core Perl /] );
my $yaml = $yp->dump_string(sub { return 23 });

# loading code references
# This is very dangerous when loading untrusted YAML!!
my $yp = YAML::PP->new( schema => [qw/ + Perl +loadcode /] );
my $code = $yp->load_string(<<'EOM');
--- !perl/code |
    {
        use 5.010;
        my ($name) = @_;
        say "Hello $name!";
    }
EOM
$code->("Ingy");
```

DESCRIPTION

This schema allows you to load and dump perl objects and special types.

Please note that loading objects of arbitrary classes can be dangerous in Perl. You have to load the modules yourself, but if an exploitable module is loaded and an object is created, its DESTROY method will be called when the object falls out of scope. *File::Temp* is an example that can be exploitable and might remove arbitrary files.

Dumping code references is on by default, but not loading (because that is easily exploitable since it's using string `eval`).

Tag Styles

You can define the style of tags you want to support:

```
my $yp_perl_two_one = YAML::PP->new(
    schema => [qw/ + Perl tags=!!perl+!perl /],
);
```

- `!perl` (default)

  Only `!perl/type` tags are supported.

- `!!perl`

  Only `!!perl/type` tags are supported.

- `!perl+!!perl`

  Both `!perl/type` and `!!perl/tag` are supported when loading. When dumping, `!perl/type` is used.

- `!!perl+!perl`

  Both `!perl/type` and `!!perl/tag` are supported when loading. When dumping, `!!perl/type` is used.

YAML.pm, YAML::Syck and *YAML::XS* are using `!!perl/type` when dumping.

YAML.pm and YAML::Syck are supporting both `!perl/type` and `!!perl/type` when loading. *YAML::XS* currently only supports the latter.

Allow only certain classes

Since v0.017

Blessing arbitrary objects can be dangerous. Maybe you want to allow blessing only specific classes and ignore others. For this you have to instantiate a Perl Schema object first and use the `classes` option.

Currently it only allows a list of strings:

```
my $perl = YAML::PP::Schema::Perl->new(
    classes => ['Foo', 'Bar'],
);
my $yp = YAML::PP::Perl->new(
    schema => [qw/ + /, $perl],
);
```

Allowed classes will be loaded and dumped as usual. The others will be ignored.

If you want to allow no objects at all, pass an empty array ref.

EXAMPLES

This is a list of the currently supported types and how they are dumped into YAML:

- array

```
# Code
[
    qw/ one two three four /
]


# YAML
---
- one
- two
- three
- four
```

- array_blessed

```
# Code
bless [
    qw/ one two three four /
], "Just::An::Arrayref"
```

(continues on next page)

(continued from previous page)

```
# YAML
--- !perl/array:Just::An::Arrayref
- one
- two
- three
- four
```

- circular

```
# Code
my $circle = bless [ 1, 2 ], 'Circle';
push @$circle, $circle;
$circle;


# YAML
--- &1 !perl/array:Circle
- 1
- 2
- *1
```

- coderef

```
# Code
sub {
    my (%args) = @_;
    return $args{x} + $args{y};
}


# YAML
--- !perl/code |-
  {
      use warnings;
      use strict;
      (my(%args) = @_);
      (return ($args{'x'} + $args{'y'}));
  }
```

- coderef_blessed

```
# Code
bless sub {
    my (%args) = @_;
    return $args{x} - $args{y};
}, "I::Am::Code"


# YAML
--- !perl/code:I::Am::Code |-
```

(continues on next page)

(continued from previous page)

```
  {
      use warnings;
      use strict;
      (my(%args) = @_);
      (return ($args{'x'} - $args{'y'}));
  }
```

- hash

```
# Code
{
    U => 2,
    B => 52,
}


# YAML
---
B: 52
U: 2
```

- hash_blessed

```
# Code
bless {
    U => 2,
    B => 52,
}, 'A::Very::Exclusive::Class'


# YAML
--- !perl/hash:A::Very::Exclusive::Class
B: 52
U: 2
```

- refref

```
# Code
my $ref = { a => 'hash' };
my $refref = \$ref;
$refref;


# YAML
--- !perl/ref
=:
  a: hash
```

- refref_blessed

```
# Code
my $ref = { a => 'hash' };
my $refref = bless \$ref, 'Foo';
$refref;


# YAML
--- !perl/ref:Foo
=:
  a: hash
```

- regexp

```
# Code
my $string = 'unblessed';
qr{$string}


# YAML
--- !perl/regexp unblessed
```

- regexp_blessed

```
# Code
my $string = 'blessed';
bless qr{$string}, "Foo"


# YAML
--- !perl/regexp:Foo blessed
```

- scalarref

```
# Code
my $scalar = "some string";
my $scalarref = \$scalar;
$scalarref;


# YAML
--- !perl/scalar
=: some string
```

- scalarref_blessed

```
# Code
my $scalar = "some other string";
my $scalarref = bless \$scalar, 'Foo';
$scalarref;
```

(continues on next page)

(continued from previous page)

```
# YAML
--- !perl/scalar:Foo
=: some other string
```

METHODS

- new

```
my $perl = YAML::PP::Schema::Perl->new(
    tags => "!perl",
    classes => ['MyClass'],
    loadcode => 1,
    dumpcode => 1,
);
```

The constructor recognizes the following options:

- tags

  Default: '!perl'

  See "Tag Styles"

- classes

  Default: undef

  Since: v0.017

  Accepts an array ref of class names

- loadcode

  Default: 0

- dumpcode

  Default: 1

```
my $yp = YAML::PP->new( schema => [qw/ + Perl -dumpcode /] );
```

- register

  A class method called by *YAML::PP::Schema*

- construct_ref, represent_ref

  Perl variables of the type REF are represented in yaml like this:

```
--- !perl/ref
=:
  a: 1
```

  construct_ref returns the perl data:

```
my $data = YAML::PP::Schema::Perl->construct_ref([ '=', { some
 ↪=> 'data' } ];
my $data = \{ a => 1 };
```

represent_ref turns a REF variable into a YAML mapping:

```
my $data = YAML::PP::Schema::Perl->represent_ref(\{ a => 1 });
my $data = { '=' => { a => 1 } };
```

- construct_scalar, represent_scalar

  Perl variables of the type SCALAR are represented in yaml like this:

```
--- !perl/scalar
=: string
```

  construct_scalar returns the perl data:

```
my $data = YAML::PP::Schema::Perl->construct_ref([ '=', 'string
 ↪' ];
my $data = \'string';
```

  represent_scalar turns a SCALAR variable into a YAML mapping:

```
my $data = YAML::PP::Schema::Perl->represent_scalar(\'string');
my $data = { '=' => 'string' };
```

- construct_regex, represent_regex

  construct_regex returns a qr{} object from the YAML string:

```
my $qr = YAML::PP::Schema::Perl->construct_regex('foo.*');
```

  represent_regex returns a string representing the regex object:

```
my $string = YAML::PP::Schema::Perl->represent_regex(qr{...});
```

- evaluate_code, represent_code

  evaluate_code returns a code reference from a string. The string must start with a { and end with a }.

```
my $code = YAML::PP::Schema::Perl->evaluate_code('{ return 23 }
 ↪');
```

  represent_code returns a string representation of the code reference with the help of B::Deparse:

```
my $string = YAML::PP::Schema::Perl->represent_code(sub {
 ↪return 23 });
```

- construct_glob, represent_glob

  construct_glob returns a glob from a hash.

```
my $glob = YAML::PP::Schema::Perl->construct_glob($hash);
```

represent_glob returns a hash representation of the glob.

```
my $hash = YAML::PP::Schema::Perl->represent_glob($glob);
```

- object

  Does the same as bless:

```
my $object = YAML::PP::Schema::Perl->object($data, $class);
```

YAML::PP::Schema::Tie

Namespace containing the following modules:

- YAML::PP::Schema::Tie::IxHash - (Deprecated) Schema for serializing ordered hashes

YAML::PP::Schema::Tie::IxHash

(Deprecated) Schema for serializing ordered hashes

SYNOPSIS

```
use YAML::PP;
use Tie::IxHash;
my $yp = YAML::PP->new( schema => [qw/ + Tie::IxHash /] );

tie(my %ordered, 'Tie::IxHash');
%ordered = (
    U => 2,
    B => 52,
);

my $yaml = $yp->dump_string(\%ordered);


# Output:
---
U: 2
B: 52
```

DESCRIPTION

This is deprecated. See the new option preserve in *YAML::PP*.

This schema allows you to dump ordered hashes which are tied to Tie::IxHash.

This code is pretty new and experimental.

It is not yet implemented for loading yet, so for now you have to tie the hashes yourself.

Examples:

- order

```
# Code
tie(my %order, 'Tie::IxHash');
%order = (
    U => 2,
    B => 52,
    c => 64,
    19 => 84,
    Disco => 2000,
    Year => 2525,
    days_on_earth => 20_000,
);
\%order;


# YAML
---
U: 2
B: 52
c: 64
19: 84
Disco: 2000
Year: 2525
days_on_earth: 20000
```

- order_blessed

```
# Code
tie(my %order, 'Tie::IxHash');
%order = (
    U => 2,
    B => 52,
    c => 64,
    19 => 84,
    Disco => 2000,
    Year => 2525,
    days_on_earth => 20_000,
);
bless \%order, 'Order';


# YAML
--- !perl/hash:Order
U: 2
B: 52
c: 64
19: 84
Disco: 2000
```

(continues on next page)

(continued from previous page)

```
Year: 2525
days_on_earth: 20000
```

METHODS

- register

  Called by YAML::PP::Schema

YAML::PP::Schema::YAML1_1

YAML 1.1 Schema for YAML::PP

SYNOPSIS

```
use YAML::PP;

my $yp = YAML::PP->new( schema => ['YAML1_1'] );
my $yaml = <<'EOM';
---
booltrue: [ true, True, TRUE, y, Y, yes, Yes, YES, on, On, ON ]
EOM
my $data = $yp->load_string($yaml);
```

DESCRIPTION

This schema allows you to load the common YAML Types from YAML 1.1.

METHODS

- register

  Called by YAML::PP::Schema

SEE ALSO

- https://yaml.org/type/null.html
- https://yaml.org/type/float.html
- https://yaml.org/type/int.html
- https://yaml.org/type/bool.html

YAML::PP::Type

Namespace containing the following modules:

- `YAML::PP::Type::MergeKey` - A special node type for merge keys

YAML::PP::Type::MergeKey

A special node type for merge keys

DESCRIPTION

See *YAML::PP::Schema::Merge*

METHODS

- new

  Constructor

YAML::PP::Writer

Write YAML output

SYNOPSIS

```
my $writer = YAML::PP::Writer->new;
```

DESCRIPTION

The *YAML::PP::Emitter* sends its output to the writer.

You can use your own writer. if you want to send the YAML output to somewhere else. See t/44.writer.t for an example.

METHODS

- new

  ```
  my $writer = YAML::PP::Writer->new;
  ```

  Constructor.

- write

  ```
  $writer->write('- ');
  ```

- init

  ```
  $writer->init;
  ```

  Initialize

- finish

  ```
  $writer->finish;
  ```

  Gets called when the output ends.

- output, set_output

  Getter/setter for the YAML output

Modules

- `YAML::PP::Writer::File` - Write YAML output to file or file handle

YAML::PP::Writer::File

Write YAML output to file or file handle

SYNOPSIS

```
my $writer = YAML::PP::Writer::File->new(output => $file);
```

DESCRIPTION

The *YAML::PP::Emitter* sends its output to the writer.

You can use your own writer. if you want to send the YAML output to somewhere else. See t/44.writer.t for an example.

METHODS

- new

  ```
  my $writer = YAML::PP::Writer::File->new(output => $file);
  my $writer = YAML::PP::Writer::File->new(output => $filehandle);
  ```

  Constructor.

- write

  ```
  $writer->write('- ');
  ```

- init

  ```
  $writer->init;
  ```

  Initialize

- finish

  ```
  $writer->finish;
  ```

  Gets called when the output ends. If The argument was a filename, the filehandle will be closed. If the argument was a filehandle, the caller needs to close it.

- output, set_output

  Getter/setter for the YAML output

YAML::XS

SYNOPSIS

```
use YAML::XS;

# Classic functional interface
my $yaml = Dump [ 1..4 ];
my $array = Load $yaml;

# EXPERIMENTAL: Object Oriented interface for YAML 1.2
# Incompatible to functional interface!
my $xs = YAML::XS->new;
my $yaml = $xs->dump([ 1..4 ]);
my $array = $xs->load($yaml);
```

DESCRIPTION

Under perl this module binds Kirill Simonov's `libyaml`, a C library written to the YAML 1.1 specification. Under pperl its compiled part, *YAML::XS::LibYAML*, reads YAML with *YAML::PP* and writes it as libyaml writes it; that page lists where the two differ.

This module exports the functions `Dump`, `Load`, `DumpFile` and `LoadFile`. These functions are intended to work exactly like `YAML.pm`'s corresponding functions. Only `Load` and `Dump` are exported by default.

CONFIGURATION

The object oriented interface is described below: "OBJECT ORIENTED INTERFACE"

- `$YAML::XS::LoadBlessed` (since v0.69)

  Default: false.

  The default was changed in version 0.81.

  When set to false, it will not bless data into objects, which can be a security problem, when loading YAML from an untrusted source. It will silently ignore the tag and just load the data unblessed.

  In PyYAML, this is called SafeLoad.

  If set to true, it will load the following YAML as objects:

  ```
  ---
  local: !Foo::Bar [a]
  perl: !!perl/hash:Foo::Bar { a: 1 }
  regex: !!perl/regexp:Foo::Bar pattern
  ```

  You can create any kind of object with YAML. The creation itself is not the critical part. If the class has a `DESTROY` method, it will be called once the object is deleted. An example with File::Temp removing files can be found at https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=862373.

- `$YAML::XS::ForbidDuplicateKeys` (since 0.84)

Default: false

When set to true, `Load` will die when encountering a duplicate key in a hash, e.g.

```
key: value
key: another value
```

This can be useful for bigger YAML documents where it is not that obvious, and it is recommended to set it to true. That's also what a YAML loader should do by default according to the YAML specification.

- `$YAML::XS::UseCode`
- `$YAML::XS::DumpCode`
- `$YAML::XS::LoadCode`

  If enabled supports deparsing and evaling of code blocks.

  Note that support for loading code was added in version 0.75, although `$LoadCode` was documented already in earlier versions.

- `$YAML::XS::QuoteNumericStrings`

  When true (the default) strings that look like numbers but have not been numified will be quoted when dumping.

  This ensures leading that things like leading zeros and other formatting are preserved.

- `$YAML::XS::Boolean` (since v0.67)

  Default: undef

  Since YAML::XS 0.89: When used with perl 5.36 or later, builtin booleans will work out of the box. They will be created by `Load` and recognized by `Dump` automatically (since YAML::XS 0.89).

```
say Dump({ truth => builtin::true });
# truth: true
```

  Since YAML::XS v0.902: loaded booleans are not set to readonly anymore.

  For older perl versions you can use the following configuration to serialize data as YAML booleans:

  When set to `"JSON::PP"` or `"boolean"`, the plain (unquoted) strings `true` and `false` will be loaded as `JSON::PP::Boolean` or `boolean.pm` objects. Those objects will be dumped again as plain "true" or "false".

  It will try to load [JSON::PP] or [boolean] and die if it can't be loaded.

  With that it's possible to add new "real" booleans to a data structure:

```
local $YAML::XS::Boolean = "JSON::PP"; # or "boolean"
my $data = Load("booltrue: true");
$data->{boolfalse} = JSON::PP::false;
my $yaml = Dump($data);
# boolfalse: false
# booltrue: true
```

It also lets booleans survive when loading YAML via YAML::XS and encode it in JSON via one of the various JSON encoders, which mostly support JSON::PP booleans.

Please note that JSON::PP::Boolean and boolean.pm behave a bit differently. Ideally you should only use them in boolean context.

If not set, booleans are loaded as special perl variables `PL_sv_yes` and `PL_sv_no`, which have the disadvantage that they are readonly, and you can't add those to an existing data structure with pure perl.

If you simply need to load "perl booleans" that are true or false in boolean context, you will be fine with the default setting.

- `$YAML::XS::Indent` (since v0.76)

  Default is 2.

  Sets the number of spaces for indentation for `Dump`.

USING YAML::XS WITH UNICODE

Handling unicode properly in Perl can be a pain. YAML::XS only deals with streams of utf8 octets. Just remember this:

```
$perl = Load($utf8_octets);
$utf8_octets = Dump($perl);
```

There are many, many places where things can go wrong with unicode. If you are having problems, use Devel::Peek on all the possible data points.

OBJECT ORIENTED INTERFACE

Since version v0.904.0, EXPERIMENTAL

+++NOTE: This is incompatible with the functional interface and will treat YAML values in a different way.+++

This has two MAJOR differences to the old functional interface:

- Object with options

  This provides an interface where you create a YAML::XS object with options (instead of the old interface with global variables).

- YAML 1.2 Core Schema

  It implements the YAML 1.2 Core Schema.

  (Note that the functional interface does not implement YAML 1.1, when it comes to loading numbers, booleans, null etc. It implements its own set of rules.)

  Here is an (incomplete!) example of values that are treated differently than with the functional interface. YAML values that match a certain pattern, are not loaded as strings, but as other types:

```
# Functional interface: special values (not compatible to other
 ↪YAML modules)
- [true, false]                              # booleans
- [null, ~]                                  # undef
- [inf, INF, iNf, iNF, InF, INf, -inf, ...] # Inf
- [100_000]                                  # 100000
- # anything that looks_like_number()        # number

# OOP YAML 1.2 special values
- [true, True, TRUE, false, False, FALSE]   # booleans
- [null, Null, NULL, ~]                     # undef
- [.inf, .Inf, .INF, -.inf, -.Inf, -.INF]   # Inf
- [.nan, .NAN, .NaN]                        # nan
- [42, 0x10, 0o10]                          # dec 42, hex 16,
 ↪oct 8
```

For more subtle differences regarding numbers checkout the comprehensive data here:

- YAML 1.1 / 1.2 definitions: https://perlpunk.github.io/yaml-test-schema/schemas.html
- Test data: https://perlpunk.github.io/yaml-test-schema/data.html

This way the OOP interface is compatible to YAML::PP and YAML processors in other languages supporting YAML 1.2.

- `load` will return the first document in scalar context

  The functional interface returns the last.

- Numbers don't have string flags
- Booleans currently are only preserved for the new builtin booleans

  Adding a JSON::PP boolean option for older perls is planned

- References, objects, globs, regexes and coderefs cannot be loaded or dumped

  This is planned.

Usage

```
use YAML::XS ();
my $xs = YAML::XS->new;
my $yaml = "foo: bar";

# Load single (first) document:
my $data = $xs->load($yaml);
$yaml = $xs->dump($data);

# Or to load all documents:
my @data = $xs->load($yaml);
$yaml = $xs->dump(@data);
```

METHODS

new

```
use YAML::XS;
my $xs = YAML::XS->new(
    # load options
    # require_footer => 0,

    # dump options
    # indent => 2,
    # header => 1,
    # footer => 0,
    # width => 80,
    # anchor_prefix => '',
    # cyclic_refs => 0,

    # load and dump options
    # utf8 => 0,
    # boolean => 0,
);
```

Options:

- indent

  Default: 2

  Sets the number of spaces for indentation for dumping.

- utf8

  Default: false

  When false, the loader will accept unencoded character strings, and the dumper returns unencoded character strings.

  When true, the loader accepts a UTF-8 encoded string of bytes, and the dumper returns a UTF-8 encoded string of bytes. This typically makes sense when you read from / write to a file directly.

- header

  Default: 1

  Writes a `---` before every document

- footer

  Default: 0

  Writes a `...` at the end of every document.

- width

  Set the maximum number of colums for dumping. If a value has too many characters, it will be split into multiple lines.

- require_footer

Default: 0

Can be useful in a use case where you want to make sure you received the complete document from the sender.

- anchor_prefix

  Default: ''

  If set to a string like ANCHOR, anchors and aliases will be prefixed with this instead of just being numbers:

```
# my $xs = YAML::XS->new( anchor_prefix => 'ANCHOR' );
# my $hash = { some => "mapping" };
# say $xs->dump([$hash, $hash]);
- &ANCHOR1
  some: mapping
- *ANCHOR1
```

- cyclic_refs

  Since version v0.907

  Default: 0

  YAML can contain data structures where an alias points to a node containing the current node, creating a cyclic (circular) data structure. If you load untrusted YAML documents, that can lead to memory leaks. Since it's not a very common use case, YAML::XS defaults to not allowing it. It will just behave as if the anchor was never created in this case. You an set it to 1 to allow it.

- boolean

  Since version v0.909

  Default: 0

  Booleans will be loaded as real booleans since perl 5.36.0, and as 1|'' before.

  You can set the value to *JSON::PP* or boolean to load and dump booleans as instances of those classes. Mainly useful for older perl versions where booleans could not roundtrip.

load

```
my $yaml = <<'EOM';
---
- 23
---
some: mapping
EOM
my $array = $xs->load($yaml);
# [23]
my @documents = $xs->load($yaml);
# (
#   [23],
```

(continues on next page)

(continued from previous page)

```
#   { some => "mapping" }
# )
```

In scalar context, if the YAML string contains more than one document, it will return the first document.

dump

```
$yaml = $xs->dump($data);
$yaml = $xs->dump($data1, $data2, $data3);
$yaml = $xs->dump(@data);
```

LIBYAML

`libyaml_version` (since v0.79) answers the libyaml version the module was built with. Under pperl it answers 0.2.4, the libyaml release whose behaviour pperl's engine follows:

```
my $libyaml_version = YAML::XS::LibYAML::libyaml_version();
```

SEE ALSO

- YAML.pm
- YAML::Syck
- YAML::Tiny
- YAML::PP
- YAML::PP::LibYAML

AUTHORS

- Ingy döt Net ingy@ingy.net
- Tina Müller tinita@cpan.org

COPYRIGHT AND LICENSE

Copyright 2007-2025 - Ingy döt Net

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See http://www.perl.com/perl/misc/Artistic.html

Modules

- `YAML::XS::LibYAML` - the YAML engine behind pperl's YAML::XS

YAML::XS::LibYAML

the YAML engine behind pperl's YAML::XS

SYNOPSIS

```
use YAML::XS qw(Load Dump);

my $data = Load("---\nname: pperl\nlist: [1, 2]\n");
print Dump($data);

my $yaml = YAML::XS->new(indent => 4, footer => 1);
print $yaml->dump($data);
```

DESCRIPTION

`YAML::XS` is YAML-LibYAML 0.910.0's own module. Its compiled part, `YAML::XS::LibYAML`, which under perl wraps the libyaml `C` library, is this module: it reads YAML with *YAML::PP*, pperl's YAML engine, and writes it as libyaml writes it. `Load`, `Dump`, `LoadFile`, `DumpFile`, the object interface (`new`, `load`, `dump`) and the `$YAML::XS::...` settings work as they do under perl, and `Dump` writes the same text: the same quoting, escapes, line folding and indentation. `libyaml_version` answers 0.2.4, the libyaml release whose behaviour this follows.

DIFFERENCES FROM PERL

- Error messages

  A document that does not parse dies with YAML::PP's account of the problem, in libyaml's layout (the problem, the document, the line and the column). The document is nearly always the one libyaml names and the line usually the same, but the words differ and the place can differ by a few columns, since the two parsers notice an error at different points.

- YAML 1.2

  YAML is read as the YAML 1.2 specification reads it, where libyaml, a YAML 1.1 parser, reads it otherwise: `{?foo: bar}` has the key `?foo`, `[?x]` is a list of `?x`, an anchor name may contain `:`, and `{x: :x}` or a tab after an indicator load where libyaml dies. Documents the specification rejects, which libyaml loads (`[-]`, a comment touching a quoted string), die.

- LS and PS

  `Dump` writes the characters U+2028 and U+2029 as `\L` and `\P`. libyaml writes them unescaped, as YAML 1.1 line breaks, and a YAML 1.2 reader then loads a different string back.

- Not yet read

  A flow collection used as a mapping key (`[flow]: block`, `{ [a]: b }`, `[ [a]: b ]`) and the key `: ` do not load yet: YAML::PP does not read them, where libyaml does.

- Where perl fails

  The object interface loads a flow mapping inside a flow sequence (`{ a: [b, { c: d }] }`), on which perl aborts, and dumps strings of latin-1 bytes, which perl refuses ("Could not initialize scalar event").

SEE ALSO

YAML::XS, *YAML::PP*

5.2.116 _charnames

`_charnames` - the Unicode character name table behind `\N{NAME}`, `charnames::viacode`, `charnames::vianame` and `\p{name=...}`.

```
use charnames ();
printf "%s | U+%04X | %s\n", charnames::viacode(0x263A),
    charnames::vianame("GREEK SMALL LETTER ALPHA"),
    "\N{LATIN SMALL LETTER E WITH ACUTE}" eq "\x{e9}" ? "same" :
 ↪"differs";
# WHITE SMILING FACE | U+03B1 | same
```

Nothing uses `_charnames` directly: it is the internal half of `charnames`, loaded by the first `\N{...}` or `charnames` call. The names are those of Unicode 17.0.0, as in perl 5.44.

`_charnames.pm` 1.51 is upstream's. Its one large input, `unicore/Name.pl`, is stored in the binary when pperl is built rather than read from disk at the first lookup, so it costs no parsing or file access.

Differences from upstream

None in behaviour.

5.2.117 attributes

The machinery behind `sub name :attr` and `my $x :attr`: applying the built-in attributes, handing the others to the package that defines them, and reading attributes back.

Perl loads it by itself when it compiles an attribute; `use attributes` is needed only to call its functions. The built-in attributes are `lvalue`, `method`, `const` and `prototype(...)` on a sub, and `shared` on a variable. Any other attribute goes to the package's `MODIFY_CODE_ATTRIBUTES` (or `_SCALAR_`, `_ARRAY_`, `_HASH_`), called as `($package, $ref, @attributes)`; what it returns is the list it did not accept, and those die "Invalid CODE attribute". `FETCH_CODE_ATTRIBUTES` (and the others) add a package's own attributes to what `attributes::get` returns.

This is attributes 0.37; `attributes.pm` is upstream's, and its XS half is native.

Synopsis

```
use attributes;

our $count;
sub counter :lvalue { $count }
counter() = 5;                                  # $count is now 5

sub method_sub :method { 1 }
print join(',', attributes::get(\&counter)), "\n";    # lvalue
print attributes::reftype(\&counter), "\n";           # CODE

package My::Attrs;
my @seen;
sub MODIFY_CODE_ATTRIBUTES { my ($pkg, $ref, @attrs) = @_; push␣
↪@seen, @attrs; return }
sub handler :Cached :Log { 1 }                  # @seen is (Cached,␣
↪Log)
```

Functions

- `attributes::get($ref)` - the attributes of what `$ref` points to: the built-in ones, then those of the package's `FETCH_*_ATTRIBUTES`.
- `attributes::reftype($ref)` - the type `$ref` points to (`CODE`, `HASH`, ...), ignoring blessing, like `Scalar::Util::reftype`.
- `use attributes PACKAGE, \&sub, ATTRIBUTES` - apply attributes at run time, as `sub name :ATTRIBUTES` does at compile time.

Differences from upstream

None: upstream's tests pass unchanged (t/81-xs-native/attributes/100-upstream-op-attrs.t, 110-upstream-uni-attrs.t), and the rest of t/81-xs-native/attributes checks against perl 5.44.

5.2.118 autodie

Replace functions with ones that succeed or die with lexical scope

SYNOPSIS

```
use autodie;            # Recommended: implies 'use autodie␣
↪qw(:default)'

use autodie qw(:all);   # Recommended more: defaults and system/
↪exec.

use autodie qw(open close);   # open/close succeed or die

open(my $fh, "<", $filename); # No need to check!
```

(continues on next page)

(continued from previous page)

```
{
    no autodie qw(open);          # open failures won't die
    open(my $fh, "<", $filename); # Could fail silently!
    no autodie;                   # disable all autodies
}

print "Hello World" or die $!;    # autodie DOESN'T check print!
```

DESCRIPTION

```
bIlujDI' yIchegh()Qo'; yIHegh()!

It is better to die() than to return() in failure.

        -- Klingon programming proverb.
```

The `autodie` pragma provides a convenient way to replace functions that normally return false on failure with equivalents that throw an exception on failure.

The `autodie` pragma has *lexical scope*, meaning that functions and subroutines altered with `autodie` will only change their behaviour until the end of the enclosing block, file, or `eval`.

If `system` is specified as an argument to `autodie`, then it uses IPC::System::Simple to do the heavy lifting. See the description of that module for more information.

EXCEPTIONS

Exceptions produced by the `autodie` pragma are members of the *autodie::exception* class. The preferred way to work with these exceptions under Perl 5.10 is as follows:

```
eval {
    use autodie;

    open(my $fh, '<', $some_file);

    my @records = <$fh>;

    # Do things with @records...

    close($fh);
};

if ($@ and $@->isa('autodie::exception')) {
    if ($@->matches('open')) { print "Error from open\n";   }
    if ($@->matches(':io' )) { print "Non-open, IO error."; }
} elsif ($@) {
    # A non-autodie exception.
}
```

See *autodie::exception* for further information on interrogating exceptions.

CATEGORIES

Autodie uses a simple set of categories to group together similar built-ins. Requesting a category type (starting with a colon) will enable autodie for all built-ins beneath that category. For example, requesting `:file` will enable autodie for `close`, `fcntl`, `open` and `sysopen`.

The categories are currently:

```
:all
    :default
        :io
            read
            seek
            sysread
            sysseek
            syswrite
            :dbm
                dbmclose
                dbmopen
            :file
                binmode
                close
                chmod
                chown
                fcntl
                flock
                ioctl
                open
                sysopen
                truncate
            :filesys
                chdir
                closedir
                opendir
                link
                mkdir
                readlink
                rename
                rmdir
                symlink
                unlink
            :ipc
                kill
                pipe
                :msg
                    msgctl
                    msgget
                    msgrcv
```

(continues on next page)

(continued from previous page)

```
                msgsnd
            :semaphore
                semctl
                semget
                semop
            :shm
                shmctl
                shmget
                shmread
        :socket
            accept
            bind
            connect
            getsockopt
            listen
            recv
            send
            setsockopt
            shutdown
            socketpair
    :threads
        fork
:system
    system
    exec
```

Note that while the above category system is presently a strict hierarchy, this should not be assumed.

A plain `use autodie` implies `use autodie qw(:default)`. Note that `system` and `exec` are not enabled by default. `system` requires the optional IPC::System::Simple module to be installed, and enabling `system` or `exec` will invalidate their exotic forms. See "BUGS" below for more details.

The syntax:

```
use autodie qw(:1.994);
```

allows the `:default` list from a particular version to be used. This provides the convenience of using the default methods, but the surety that no behavioral changes will occur if the `autodie` module is upgraded.

`autodie` can be enabled for all of Perl's built-ins, including `system` and `exec` with:

```
use autodie qw(:all);
```

FUNCTION SPECIFIC NOTES

print

The autodie pragma **does not check calls to `print`**.

flock

It is not considered an error for `flock` to return false if it fails due to an `EWOULDBLOCK` (or equivalent) condition. This means one can still use the common convention of testing the return value of `flock` when called with the `LOCK_NB` option:

```
use autodie;

if ( flock($fh, LOCK_EX | LOCK_NB) ) {
    # We have a lock
}
```

Autodying `flock` will generate an exception if `flock` returns false with any other error.

system/exec

The `system` built-in is considered to have failed in the following circumstances:

- The command does not start.
- The command is killed by a signal.
- The command returns a non-zero exit value (but see below).

On success, the autodying form of `system` returns the *exit value* rather than the contents of `$?`.

Additional allowable exit values can be supplied as an optional first argument to autodying `system`:

```
system( [ 0, 1, 2 ], $cmd, @args);  # 0,1,2 are good exit values
```

`autodie` uses the IPC::System::Simple module to change `system`. See its documentation for further information.

Applying `autodie` to `system` or `exec` causes the exotic forms `system { $cmd } @args` or `exec { $cmd } @args` to be considered a syntax error until the end of the lexical scope. If you really need to use the exotic form, you can call `CORE::system` or `CORE::exec` instead, or use `no autodie qw(system exec)` before calling the exotic form.

GOTCHAS

Functions called in list context are assumed to have failed if they return an empty list, or a list consisting only of a single undef element.

Some builtins (e.g. `chdir` or `truncate`) has a call signature that cannot completely be represented with a Perl prototype. This means that some valid Perl code will be invalid under autodie. As an example:

```
chdir(BAREWORD);
```

Without autodie (and assuming BAREWORD is an open filehandle/dirhandle) this is a valid call to chdir. But under autodie, `chdir` will behave like it had the prototype ";$" and thus BAREWORD will be a syntax error (under "use strict". Without strict, it will interpreted as a filename).

DIAGNOSTICS

- :void cannot be used with lexical scope

  The `:void` option is supported in *Fatal*, but not `autodie`. To workaround this, `autodie` may be explicitly disabled until the end of the current block with `no autodie`. To disable autodie for only a single function (eg, open) use `no autodie qw(open)`.

  `autodie` performs no checking of called context to determine whether to throw an exception; the explicitness of error handling with `autodie` is a deliberate feature.

- No user hints defined for %s

  You've insisted on hints for user-subroutines, either by pre-pending a `!` to the subroutine name itself, or earlier in the list of arguments to `autodie`. However the subroutine in question does not have any hints available.

See also *"DIAGNOSTICS" in Fatal*.

Tips and Tricks

Importing autodie into another namespace than "caller"

It is possible to import autodie into a different namespace by using Import::Into. However, you have to pass a "caller depth" (rather than a package name) for this to work correctly.

BUGS

"Used only once" warnings can be generated when `autodie` or `Fatal` is used with package filehandles (eg, `FILE`). Scalar filehandles are strongly recommended instead.

When using `autodie` or `Fatal` with user subroutines, the declaration of those subroutines must appear before the first use of `Fatal` or `autodie`, or have been exported from a module. Attempting to use `Fatal` or `autodie` on other user subroutines will result in a compile-time error.

Due to a bug in Perl, `autodie` may "lose" any format which has the same name as an autodying built-in or function.

`autodie` may not work correctly if used inside a file with a name that looks like a string eval, such as `eval (3)`.

autodie and string eval

Due to the current implementation of `autodie`, unexpected results may be seen when used near or with the string version of eval. *None of these bugs exist when using block eval.*

Under Perl 5.8 only, `autodie` *does not* propagate into string `eval` statements, although it can be explicitly enabled inside a string `eval`.

Under Perl 5.10 only, using a string eval when `autodie` is in effect can cause the autodie behaviour to leak into the surrounding scope. This can be worked around by using a `no autodie` at the end of the scope to explicitly remove autodie's effects, or by avoiding the use of string eval.

None of these bugs exist when using block eval. The use of `autodie` with block eval is considered good practice.

REPORTING BUGS

Please report bugs via the GitHub Issue Tracker at https://github.com/pjf/autodie/issues.

FEEDBACK

If you find this module useful, please consider rating it on metacpan at https://metacpan.org/pod/autodie .

The module author loves to hear how `autodie` has made your life better (or worse). Feedback can be sent to pjf@perltraining.com.au.

AUTHOR

Copyright 2008-2009, Paul Fenwick pjf@perltraining.com.au

LICENSE

This module is free software. You may distribute it under the same terms as Perl itself.

SEE ALSO

Fatal, *autodie::exception*, *autodie::hints*, IPC::System::Simple

Perl tips, autodie at http://perltraining.com.au/tips/2008-08-20.html

ACKNOWLEDGEMENTS

Mark Reed and Roland Giersig – Klingon translators.

See the `AUTHORS` file for full credits. The latest version of this file can be found at https://github.com/pjf/autodie/tree/master/AUTHORS .

Modules

- `autodie::Scope`
- `autodie::Util` - Internal Utility subroutines for autodie and Fatal
- `autodie::exception` - Exceptions from autodying functions.
- `autodie::hints` - Provide hints about user subroutines to autodie
- `autodie::skip` - Skip a package when throwing autodie exceptions

autodie::Scope

Namespace containing the following modules:

- `autodie::Scope::Guard` - Wrapper class for calling subs at end of scope
- `autodie::Scope::GuardStack` - Hook stack for managing scopes via %^H

autodie::Scope::Guard

Wrapper class for calling subs at end of scope

SYNOPSIS

```
use autodie::Scope::Guard;
$^H{'my-key'} = autodie::Scope::Guard->new(sub {
    print "Hallo world\n";
});
```

DESCRIPTION

This class is used to bless perl subs so that they are invoked when they are destroyed. This is mostly useful for ensuring the code is invoked at end of scope. This module is not a part of autodie's public API.

This module is directly inspired by chocolateboy's excellent Scope::Guard module.

Methods

new

```
my $hook = autodie::Scope::Guard->new(sub {});
```

Creates a new `autodie::Scope::Guard`, which will invoke the given sub once it goes out of scope (i.e. its DESTROY handler is called).

AUTHOR

Copyright 2008-2009, Paul Fenwick pjf@perltraining.com.au

LICENSE

This module is free software. You may distribute it under the same terms as Perl itself.

autodie::Scope::GuardStack

Hook stack for managing scopes via %^H

SYNOPSIS

```
use autodie::Scope::GuardStack;
my $stack = autodie::Scope::GuardStack->new
$^H{'my-key'} = $stack;

$stack->push_hook(sub {});
```

DESCRIPTION

This class is a stack of hooks to be called in the right order as scopes go away. The stack is only useful when inserted into %^H and will pop hooks as their "scope" is popped. This is useful for uninstalling or reinstalling subs in a namespace as a pragma goes out of scope.

Due to how %^H works, this class is only useful during the compilation phase of a perl module and relies on the internals of how perl handles references in %^H. This module is not a part of autodie's public API.

Methods

new

```
my $stack = autodie::Scope::GuardStack->new;
```

Creates a new `autodie::Scope::GuardStack`. The stack is initially empty and must be inserted into %^H by the creator.

push_hook

```
$stack->push_hook(sub {});
```

Add a sub to the stack. The sub will be called once the current compile-time "scope" is left. Multiple hooks can be added per scope

AUTHOR

Copyright 2013, Niels Thykier niels@thykier.net

LICENSE

This module is free software. You may distribute it under the same terms as Perl itself.

autodie::Util

Internal Utility subroutines for autodie and Fatal

SYNOPSIS

```
# INTERNAL API for autodie and Fatal only!

use autodie::Util qw(on_end_of_compile_scope);
on_end_of_compile_scope(sub { print "Hallo world\n"; });
```

DESCRIPTION

Internal Utilities for autodie and Fatal! This module is not a part of autodie's public API.

This module contains utility subroutines for abstracting away the underlying magic of autodie and (ab)uses of %^H to call subs at the end of a (compile-time) scopes.

Note that due to how %^H works, some of these utilities are only useful during the compilation phase of a perl module and relies on the internals of how perl handles references in %^H.

Methods

on_end_of_compile_scope

```
on_end_of_compile_scope(sub { print "Hallo world\n"; });
```

Will invoke a sub at the end of a (compile-time) scope. The sub is called once with no arguments. Can be called multiple times (even in the same "compile-time" scope) to install multiple subs. Subs are called in a "first-in-last-out"-order (FILO or "stack"-order).

fill_protos

```
fill_protos('*$$;$@')
```

Given a Perl subroutine prototype, return a list of invocation specifications. Each specification is a listref, where the first member is the (minimum) number of arguments for this invocation specification. The remaining arguments are a string representation of how to pass the arguments correctly to a sub with the given prototype, when called with the given number of arguments.

The specifications are returned in increasing order of arguments starting at 0 (e.g. ';$') or 1 (e.g. '$@'). Note that if the prototype is "slurpy" (e.g. ends with a "@"), the number of arguments for the last specification is a "minimum" number rather than an exact number. This can be detected by the last member of the last specification matching m/[@#]_/.

make_core_trampoline

```
make_core_trampoline('CORE::open', 'main', prototype('CORE::open'))
```

Creates a trampoline for calling a core sub. Essentially, a tiny sub that figures out how we should be calling our core sub, puts in the arguments in the right way, and bounces our control over to it.

If we could reliably use `goto &` on core builtins, we wouldn't need this subroutine.

install_subs

```
install_subs('My::Module', { 'read' => sub { die("Hallo\n"), ... }})
```

Given a package name and a hashref mapping names to a subroutine reference (or undef), this subroutine will install said subroutines on their given name in that module.

If a name mapes to undef, any subroutine with that name in the target module will be removed (possibly "unshadowing" a CORE sub of same name).

AUTHOR

Copyright 2013-2014, Niels Thykier niels@thykier.net

LICENSE

This module is free software. You may distribute it under the same terms as Perl itself.

autodie::exception

Exceptions from autodying functions.

SYNOPSIS

```
eval {
    use autodie;

    open(my $fh, '<', 'some_file.txt');

    ...
};

if (my $E = $@) {
    say "Ooops!  ",$E->caller," had problems: $@";
}
```

DESCRIPTION

When an *autodie* enabled function fails, it generates an autodie::exception object. This can be interrogated to determine further information about the error that occurred.

This document is broken into two sections; those methods that are most useful to the end-developer, and those methods for anyone wishing to subclass or get very familiar with autodie::exception.

Common Methods

These methods are intended to be used in the everyday dealing of exceptions.

The following assume that the error has been copied into a separate scalar:

```
if ($E = $@) {
    ...
}
```

This is not required, but is recommended in case any code is called which may reset or alter $@.

args

```
my $array_ref = $E->args;
```

Provides a reference to the arguments passed to the subroutine that died.

function

```
my $sub = $E->function;
```

The subroutine (including package) that threw the exception.

file

```
my $file = $E->file;
```

The file in which the error occurred (eg, `myscript.pl` or `MyTest.pm`).

package

```
my $package = $E->package;
```

The package from which the exceptional subroutine was called.

caller

```
my $caller = $E->caller;
```

The subroutine that *called* the exceptional code.

line

```
my $line = $E->line;
```

The line in `$E->file` where the exceptional code was called.

context

```
my $context = $E->context;
```

The context in which the subroutine was called by autodie; usually the same as the context in which you called the autodying subroutine. This can be 'list', 'scalar', or undefined (unknown). It will never be 'void', as `autodie` always captures the return value in one way or another.

For some core functions that always return a scalar value regardless of their context (eg, `chown`), this may be 'scalar', even if you used a list context.

return

```
my $return_value = $E->return;
```

The value(s) returned by the failed subroutine. When the subroutine was called in a list context, this will always be a reference to an array containing the results. When the subroutine was called in a scalar context, this will be the actual scalar returned.

errno

```
my $errno = $E->errno;
```

The value of `$!` at the time when the exception occurred.

NOTE: This method will leave the main `autodie::exception` class and become part of a role in the future. You should only call `errno` for exceptions where `$!` would reasonably have been set on failure.

eval_error

```
my $old_eval_error = $E->eval_error;
```

The contents of `$@` immediately after autodie triggered an exception. This may be useful when dealing with modules such as *Text::Balanced* that set (but do not throw) `$@` on error.

matches

```
if ( $e->matches('open') ) { ... }

if ( 'open' ~~ $e ) { ... }
```

`matches` is used to determine whether a given exception matches a particular role.

An exception is considered to match a string if:

- For a string not starting with a colon, the string exactly matches the package and subroutine that threw the exception. For example, `MyModule::log`. If the string does not contain a package name, `CORE::` is assumed.
- For a string that does start with a colon, if the subroutine throwing the exception *does* that behaviour. For example, the `CORE::open` subroutine does `:file`, `:io` and `:all`.

  See *"CATEGORIES" in autodie* for further information.

  On Perl 5.10 and above, using smart-match (~~) with an `autodie::exception` object will use `matches` underneath. This module used to recommend using smart-match with the exception object on the left hand side, but in future Perls that is likely to stop working. The smart-match facility of this class should only be used with the exception object on the right hand side. Having the exception object on the right is both future-proof and portable to older Perls, back to 5.10. Beware that this facility can only be relied upon when it is certain that the exception object

actually is an `autodie::exception` object; it is no more capable than an explicit call to the `matches` method.

Advanced methods

The following methods, while usable from anywhere, are primarily intended for developers wishing to subclass `autodie::exception`, write code that registers custom error messages, or otherwise work closely with the `autodie::exception` model.

register

```
autodie::exception->register( 'CORE::open' => \&mysub );
```

The `register` method allows for the registration of a message handler for a given subroutine. The full subroutine name including the package should be used.

Registered message handlers will receive the `autodie::exception` object as the first parameter.

add_file_and_line

```
say "Problem occurred",$@->add_file_and_line;
```

Returns the string `at %s line %d`, where `%s` is replaced with the filename, and `%d` is replaced with the line number.

Primarily intended for use by format handlers.

stringify

```
say "The error was: ",$@->stringify;
```

Formats the error as a human readable string. Usually there's no reason to call this directly, as it is used automatically if an `autodie::exception` object is ever used as a string.

Child classes can override this method to change how they're stringified.

format_default

```
my $error_string = $E->format_default;
```

This produces the default error string for the given exception, *without using any registered message handlers*. It is primarily intended to be called from a message handler when they have been passed an exception they don't want to format.

Child classes can override this method to change how default messages are formatted.

new

```
my $error = autodie::exception->new(
    args => \@_,
    function => "CORE::open",
    errno => $!,
    context => 'scalar',
    return => undef,
);
```

Creates a new `autodie::exception` object. Normally called directly from an autodying function. The `function` argument is required, its the function we were trying to call that generated the exception. The `args` parameter is optional.

The `errno` value is optional. In versions of `autodie::exception` 1.99 and earlier the code would try to automatically use the current value of `$!`, but this was unreliable and is no longer supported.

Atrributes such as package, file, and caller are determined automatically, and cannot be specified.

SEE ALSO

autodie, *autodie::exception::system*

LICENSE

Copyright (C)2008 Paul Fenwick

This is free software. You may modify and/or redistribute this code under the same terms as Perl 5.10 itself, or, at your option, any later version of Perl 5.

AUTHOR

Paul Fenwick pjf@perltraining.com.au

Modules

- `autodie::exception::system` - Exceptions from autodying system().

autodie::exception::system

Exceptions from autodying system().

SYNOPSIS

```
eval {
    use autodie qw(system);

    system($cmd, @args);
```

(continues on next page)

(continued from previous page)

```
};

if (my $E = $@) {
    say "Ooops!  ",$E->caller," had problems: $@";
}
```

DESCRIPTION

This is a *autodie::exception* class for failures from the `system` command.

Presently there is no way to interrogate an `autodie::exception::system` object for the command, exit status, and other information you'd expect such an object to hold. The interface will be expanded to accommodate this in the future.

stringify

When stringified, `autodie::exception::system` objects currently use the message generated by IPC::System::Simple.

LICENSE

Copyright (C)2008 Paul Fenwick

This is free software. You may modify and/or redistribute this code under the same terms as Perl 5.10 itself, or, at your option, any later version of Perl 5.

AUTHOR

Paul Fenwick pjf@perltraining.com.au

autodie::hints

Provide hints about user subroutines to autodie

SYNOPSIS

```
package Your::Module;

our %DOES = ( 'autodie::hints::provider' => 1 );

sub AUTODIE_HINTS {
    return {
        foo => { scalar => HINTS, list => SOME_HINTS },
        bar => { scalar => HINTS, list => MORE_HINTS },
    }
}
```

(continues on next page)

(continued from previous page)

```
# Later, in your main program...

use Your::Module qw(foo bar);
use autodie      qw(:default foo bar);

foo();         # succeeds or dies based on scalar hints

# Alternatively, hints can be set on subroutines we've
# imported.

use autodie::hints;
use Some::Module qw(think_positive);

BEGIN {
    autodie::hints->set_hints_for(
        \&think_positive,
        {
            fail => sub { $_[0] <= 0 }
        }
    )
}
use autodie qw(think_positive);

think_positive(...);    # Returns positive or dies.
```

DESCRIPTION

Introduction

The *autodie* pragma is very smart when it comes to working with Perl's built-in functions. The behaviour for these functions are fixed, and `autodie` knows exactly how they try to signal failure.

But what about user-defined subroutines from modules? If you use `autodie` on a user-defined subroutine then it assumes the following behaviour to demonstrate failure:

- A false value, in scalar context
- An empty list, in list context
- A list containing a single undef, in list context

All other return values (including the list of the single zero, and the list containing a single empty string) are considered successful. However, real-world code isn't always that easy. Perhaps the code you're working with returns a string containing the word "FAIL" upon failure, or a two element list containing `(undef, "human error message")`. To make autodie work with these sorts of subroutines, we have the *hinting interface*.

The hinting interface allows *hints* to be provided to `autodie` on how it should detect failure from user-defined subroutines. While these *can* be provided by the end-user of `autodie`, they are ideally written into the module itself, or into a helper module or sub-class of `autodie` itself.

What are hints?

A *hint* is a subroutine or value that is checked against the return value of an autodying subroutine. If the match returns true, `autodie` considers the subroutine to have failed.

If the hint provided is a subroutine, then `autodie` will pass the complete return value to that subroutine. If the hint is a regexp object, then `autodie` will match it against the return value. If the hint is undef, the return value must be undef. On Perl versions 5.10 and newer, any other value can be provided and it will be smart matched against the value provided. However, smart matched values like this are deprecated.

Hints can be provided for both scalar and list contexts. Note that an autodying subroutine will never see a void context, as `autodie` always needs to capture the return value for examination. Autodying subroutines called in void context act as if they're called in a scalar context, but their return value is discarded after it has been checked.

Example hints

Hints may consist of subroutine references, objects overloading smart-match, regular expressions, and depending on Perl version possibly other things. You can specify different hints for how failure should be identified in scalar and list contexts.

These examples apply for use in the `AUTODIE_HINTS` subroutine and when calling `autodie::hints->set_hints_for()`.

The most common context-specific hints are:

```
# Scalar failures always return undef:
    {  scalar => sub { !defined($_[0]) }  }

# Scalar failures return any false value [default expectation]:
    {  scalar => sub { ! $_[0] }  }

# Scalar failures always return zero explicitly:
    {  scalar => sub { defined($_[0]) && $_[0] eq '0' }  }

# List failures always return an empty list:
    {  list => sub { !@_ }  }

# List failures return () or (undef) [default expectation]:
    {  list => sub { ! @_ || @_ == 1 && !defined $_[0] }  }

# List failures return () or a single false value:
    {  list => sub { ! @_ || @_ == 1 && !$_[0] }  }

# List failures return (undef, "some string")
    {  list => sub { @_ == 2 && !defined $_[0] }  }

# Unsuccessful foo() returns 'FAIL' or '_FAIL' in scalar context,
#                   returns (-1) in list context...
autodie::hints->set_hints_for(
    \&foo,
    {
```

(continues on next page)

(continued from previous page)

```
        scalar => qr/^ _? FAIL $/xms,
        list   => sub { @_ == 1 && $_[0] eq -1 },
    }
);

# Unsuccessful foo() returns 0 in all contexts...
autodie::hints->set_hints_for(
    \&foo,
    {
        scalar => sub { defined($_[0]) && $_[0] == 0 },
        list   => sub { @_ == 1 && defined($_[0]) && $_[0] == 0 },
    }
);
```

This “in all contexts” construction is very common, and can be abbreviated, using the ‘fail’ key. This sets both the `scalar` and `list` hints to the same value:

```
# Unsuccessful foo() returns 0 in all contexts...
autodie::hints->set_hints_for(
    \&foo,
    {
        fail => sub { @_ == 1 and defined $_[0] and $_[0] == 0 }
    }
);

# Unsuccessful think_positive() returns negative number on failure..
 ↪.
autodie::hints->set_hints_for(
    \&think_positive,
    {
        fail => sub { $_[0] < 0 }
    }
);

# Unsuccessful my_system() returns non-zero on failure...
autodie::hints->set_hints_for(
    \&my_system,
    {
        fail => sub { $_[0] != 0 }
    }
);
```

Manually setting hints from within your program

If you are using a module which returns something special on failure, then you can manually create hints for each of the desired subroutines. Once the hints are specified, they are available for all files and modules loaded thereafter, thus you can move this work into a module and it will still work.

```
use Some::Module qw(foo bar);
use autodie::hints;

autodie::hints->set_hints_for(
        \&foo,
        {
                scalar => SCALAR_HINT,
                list   => LIST_HINT,
        }
);
autodie::hints->set_hints_for(
        \&bar,
        { fail => SOME_HINT, }
);
```

It is possible to pass either a subroutine reference (recommended) or a fully qualified subroutine name as the first argument. This means you can set hints on modules that *might* get loaded:

```
use autodie::hints;
autodie::hints->set_hints_for(
        'Some::Module:bar', { fail => SCALAR_HINT, }
);
```

This technique is most useful when you have a project that uses a lot of third-party modules. You can define all your possible hints in one-place. This can even be in a sub-class of autodie. For example:

```
package my::autodie;

use parent qw(autodie);
use autodie::hints;

autodie::hints->set_hints_for(...);

1;
```

You can now `use my::autodie`, which will work just like the standard `autodie`, but is now aware of any hints that you've set.

Adding hints to your module

`autodie` provides a passive interface to allow you to declare hints for your module. These hints will be found and used by `autodie` if it is loaded, but otherwise have no effect (or dependencies) without autodie. To set these, your module needs to declare that it *does* the `autodie::hints::provider` role. This can be done by writing your own `DOES` method, using a system such as `Class::DOES` to handle the heavy-lifting for you, or declaring a `%DOES` package variable with a `autodie::hints::provider` key and a corresponding true value.

Note that checking for a `%DOES` hash is an `autodie`-only short-cut. Other modules do not use this mechanism for checking roles, although you can use the `Class::DOES` module

from the CPAN to allow it.

In addition, you must define a `AUTODIE_HINTS` subroutine that returns a hash-reference containing the hints for your subroutines:

```
package Your::Module;

# We can use the Class::DOES from the CPAN to declare adherence
# to a role.

use Class::DOES 'autodie::hints::provider' => 1;

# Alternatively, we can declare the role in %DOES.  Note that
# this is an autodie specific optimisation, although Class::DOES
# can be used to promote this to a true role declaration.

our %DOES = ( 'autodie::hints::provider' => 1 );

# Finally, we must define the hints themselves.

sub AUTODIE_HINTS {
    return {
        foo => { scalar => HINTS, list => SOME_HINTS },
        bar => { scalar => HINTS, list => MORE_HINTS },
        baz => { fail => HINTS },
    }
}
```

This allows your code to set hints without relying on `autodie` and `autodie::hints` being loaded, or even installed. In this way your code can do the right thing when `autodie` is installed, but does not need to depend upon it to function.

Insisting on hints

When a user-defined subroutine is wrapped by `autodie`, it will use hints if they are available, and otherwise reverts to the *default behaviour* described in the introduction of this document. This can be problematic if we expect a hint to exist, but (for whatever reason) it has not been loaded.

We can ask autodie to *insist* that a hint be used by prefixing an exclamation mark to the start of the subroutine name. A lone exclamation mark indicates that *all* subroutines after it must have hints declared.

```
# foo() and bar() must have their hints defined
use autodie qw( !foo !bar baz );

# Everything must have hints (recommended).
use autodie qw( ! foo bar baz );

# bar() and baz() must have their hints defined
use autodie qw( foo ! bar baz );
```

(continues on next page)

(continued from previous page)

```
# Enable autodie for all of Perl's supported built-ins,
# as well as for foo(), bar() and baz().  Everything must
# have hints.
use autodie qw( ! :all foo bar baz );
```

If hints are not available for the specified subroutines, this will cause a compile-time error. Insisting on hints for Perl's built-in functions (eg, open and `close`) is always successful.

Insisting on hints is *strongly* recommended.

Diagnostics

- Attempts to set_hints_for unidentifiable subroutine

  You've called `autodie::hints->set_hints_for()` using a subroutine reference, but that reference could not be resolved back to a subroutine name. It may be an anonymous subroutine (which can't be made autodying), or may lack a name for other reasons.

  If you receive this error with a subroutine that has a real name, then you may have found a bug in autodie. See *"BUGS" in autodie* for how to report this.

- fail hints cannot be provided with either scalar or list hints for %s

  When defining hints, you can either supply both `list` and `scalar` keywords, *or* you can provide a single `fail` keyword. You can't mix and match them.

- You've provided either a `scalar` hint without supplying a `list` hint, or vice-versa. You *must* supply both `scalar` and `list` hints, *or* a single `fail` hint.

ACKNOWLEDGEMENTS

- Dr Damian Conway for suggesting the hinting interface and providing the example usage.
- Jacinta Richardson for translating much of my ideas into this documentation.

AUTHOR

Copyright 2009, Paul Fenwick pjf@perltraining.com.au

LICENSE

This module is free software. You may distribute it under the same terms as Perl itself.

SEE ALSO

autodie, Class::DOES

autodie::skip

Skip a package when throwing autodie exceptions

SYNPOSIS

```
use parent qw(autodie::skip);
```

DESCRIPTION

This dummy class exists to signal that the class inheriting it should be skipped when reporting exceptions from autodie. This is useful for utility classes like Path::Tiny that wish to report the location of where they were called on failure.

If your class has a better way of doing roles, then you should not load this class and instead simply say that your class *DOES* `autodie::skip` instead.

AUTHOR

Copyright 2013, Paul Fenwick pjf@cpan.org

LICENSE

This module is free software. You may distribute it under the same terms as Perl itself.

SEE ALSO

autodie, *autodie::exception*

5.2.119 autouse

postpone load of modules until a function is used

SYNOPSIS

```
use autouse 'Carp' => qw(carp croak);
carp "this carp was predeclared and autoused ";
```

DESCRIPTION

If the module `Module` is already loaded, then the declaration

```
use autouse 'Module' => qw(func1 func2($;$));
```

is equivalent to

```
use Module qw(func1 func2);
```

if `Module` defines func2() with prototype `($;$)`, and func1() has no prototypes. (At least if `Module` uses `Exporter`'s `import`, otherwise it is a fatal error.)

If the module `Module` is not loaded yet, then the above declaration declares functions func1() and func2() in the current package. When these functions are called, they load the package `Module` if needed, and substitute themselves with the correct definitions.

WARNING

Using `autouse` will move important steps of your program's execution from compile time to runtime. This can

- Break the execution of your program if the module you `autoused` has some initialization which it expects to be done early.
- hide bugs in your code since important checks (like correctness of prototypes) is moved from compile time to runtime. In particular, if the prototype you specified on `autouse` line is wrong, you will not find it out until the corresponding function is executed. This will be very unfortunate for functions which are not always called (note that for such functions `autouse`ing gives biggest win, for a workaround see below).

To alleviate the second problem (partially) it is advised to write your scripts like this:

```
use Module;
use autouse Module => qw(carp($) croak(&$));
carp "this carp was predeclared and autoused ";
```

The first line ensures that the errors in your argument specification are found early. When you ship your application you should comment out the first line, since it makes the second one useless.

AUTHOR

Ilya Zakharevich (ilya@math.ohio-state.edu)

SEE ALSO

perl(1).

5.2.120 base

Establish an ISA relationship with base classes at compile time

SYNOPSIS

```
package Baz;
use base qw(Foo Bar);
```

DESCRIPTION

Unless you are using the `fields` pragma, consider this module discouraged in favor of the lighter-weight `parent`.

Allows you to both load one or more modules, while setting up inheritance from those modules at the same time. Roughly similar in effect to

```
package Baz;
BEGIN {
    require Foo;
    require Bar;
    push @ISA, qw(Foo Bar);
}
```

When `base` tries to `require` a module, it will not die if it cannot find the module's file, but will die on any other error. After all this, should your base class be empty, containing no symbols, `base` will die. This is useful for inheriting from classes in the same file as yourself but where the filename does not match the base module name, like so:

```
# in Bar.pm
package Foo;
sub exclaim { "I can have such a thing?!" }

package Bar;
use base "Foo";
```

There is no `Foo.pm`, but because `Foo` defines a symbol (the `exclaim` subroutine), `base` will not die when the `require` fails to load `Foo.pm`.

`base` will also initialize the fields if one of the base classes has it. Multiple inheritance of fields is **NOT** supported, if two or more base classes each have inheritable fields the 'base' pragma will croak. See *fields* for a description of this feature.

The base class' `import` method is **not** called.

DIAGNOSTICS

- Base class package "%s" is empty.

  base.pm was unable to require the base package, because it was not found in your path.

- Class 'Foo' tried to inherit from itself

  Attempting to inherit from yourself generates a warning.

  ```
  package Foo;
  use base 'Foo';
  ```

HISTORY

This module was introduced with Perl 5.004_04.

CAVEATS

Due to the limitations of the implementation, you must use base *before* you declare any of your own fields.

SEE ALSO

fields

5.2.121 bigfloat

transparent big floating point number support for Perl

SYNOPSIS

```
use bigfloat;

$x = 2 + 4.5;                       # Math::BigFloat 6.5
print 2 ** 512 * 0.1;               # Math::BigFloat 134...09.6
print inf + 42;                     # Math::BigFloat inf
print NaN * 7;                      # Math::BigFloat NaN
print hex("0x1234567890123490");    # Perl v5.10.0 or later

{
    no bigfloat;
    print 2 ** 256;                 # a normal Perl scalar now
}

# for older Perls, import into current package:
use bigfloat qw/hex oct/;
print hex("0x1234567890123490");
print oct("01234567890123490");
```

DESCRIPTION

All numeric literals in the given scope are converted to Math::BigFloat objects.

All operators (including basic math operations) except the range operator `..` are overloaded.

So, the following:

```
use bigfloat;
$x = 1234;
```

creates a Math::BigFloat and stores a reference to in $x. This happens transparently and behind your back, so to speak.

You can see this with the following:

```
perl -Mbigfloat -le 'print ref(1234)'
```

Since numbers are actually objects, you can call all the usual methods from Math::BigFloat on them. This even works to some extent on expressions:

```
perl -Mbigfloat -le '$x = 1234; print $x->bdec()'
perl -Mbigfloat -le 'print 1234->copy()->binc();'
perl -Mbigfloat -le 'print 1234->copy()->binc->badd(6);'
perl -Mbigfloat -le 'print +(1234)->copy()->binc()'
```

(Note that print doesn't do what you expect if the expression starts with '(' hence the +)

You can even chain the operations together as usual:

```
perl -Mbigfloat -le 'print 1234->copy()->binc->badd(6);'
1241
```

Please note the following does not work as expected (prints nothing), since overloading of '..' is not yet possible in Perl (as of v5.8.0):

```
perl -Mbigfloat -le 'for (1..2) { print ref($_); }'
```

Options

`bigfloat` recognizes some options that can be passed while loading it via via `use`. The following options exist:

- a or accuracy

  This sets the accuracy for all math operations. The argument must be greater than or equal to zero. See Math::BigInt's bround() method for details.

  ```
  perl -Mbigfloat=a,50 -le 'print sqrt(20)'
  ```

  Note that setting precision and accuracy at the same time is not possible.

- p or precision

  This sets the precision for all math operations. The argument can be any integer. Negative values mean a fixed number of digits after the dot, while a positive value rounds to this digit left from the dot. 0 means round to integer. See Math::BigInt's bfround() method for details.

  ```
  perl -Mbigfloat=p,-50 -le 'print sqrt(20)'
  ```

  Note that setting precision and accuracy at the same time is not possible.

- t or trace

  This enables a trace mode and is primarily for debugging.

- l, lib, try, or only

  Load a different math lib, see "Math Library".

```
perl -Mbigfloat=l,GMP -e 'print 2 ** 512'
perl -Mbigfloat=lib,GMP -e 'print 2 ** 512'
perl -Mbigfloat=try,GMP -e 'print 2 ** 512'
perl -Mbigfloat=only,GMP -e 'print 2 ** 512'
```

- hex

  Override the built-in hex() method with a version that can handle big numbers. This overrides it by exporting it to the current package. Under Perl v5.10.0 and higher, this is not so necessary, as hex() is lexically overridden in the current scope whenever the `bigfloat` pragma is active.

- oct

  Override the built-in oct() method with a version that can handle big numbers. This overrides it by exporting it to the current package. Under Perl v5.10.0 and higher, this is not so necessary, as oct() is lexically overridden in the current scope whenever the `bigfloat` pragma is active.

- v or version

  this prints out the name and version of the modules and then exits.

  ```
  perl -Mbigfloat=v
  ```

Math Library

Math with the numbers is done (by default) by a backend library module called Math::BigInt::Calc. The default is equivalent to saying:

```
use bigfloat lib => 'Calc';
```

you can change this by using:

```
use bigfloat lib => 'GMP';
```

The following would first try to find Math::BigInt::Foo, then Math::BigInt::Bar, and if this also fails, revert to Math::BigInt::Calc:

```
use bigfloat lib => 'Foo,Math::BigInt::Bar';
```

Using c<lib> warns if none of the specified libraries can be found and *Math::BigInt* fell back to one of the default libraries. To suppress this warning, use c<try> instead:

```
use bigfloat try => 'GMP';
```

If you want the code to die instead of falling back, use `only` instead:

```
use bigfloat only => 'GMP';
```

Please see respective module documentation for further details.

Method calls

Since all numbers are now objects, you can use all methods that are part of the Math::BigFloat API.

But a warning is in order. When using the following to make a copy of a number, only a shallow copy will be made.

```
$x = 9; $y = $x;
$x = $y = 7;
```

Using the copy or the original with overloaded math is okay, e.g., the following work:

```
$x = 9; $y = $x;
print $x + 1, " ", $y,"\n";     # prints 10 9
```

but calling any method that modifies the number directly will result in **both** the original and the copy being destroyed:

```
$x = 9; $y = $x;
print $x->badd(1), " ", $y,"\n";        # prints 10 10

$x = 9; $y = $x;
print $x->binc(1), " ", $y,"\n";        # prints 10 10

$x = 9; $y = $x;
print $x->bmul(2), " ", $y,"\n";        # prints 18 18
```

Using methods that do not modify, but test that the contents works:

```
$x = 9; $y = $x;
$z = 9 if $x->is_zero();                # works fine
```

See the documentation about the copy constructor and = in overload, as well as the documentation in Math::BigFloat for further details.

Methods

- inf()

  A shortcut to return Math::BigFloat->binf(). Useful because Perl does not always handle bareword `inf` properly.

- NaN()

  A shortcut to return Math::BigFloat->bnan(). Useful because Perl does not always handle bareword `NaN` properly.

- e

  ```
  # perl -Mbigfloat=e -wle 'print e'
  ```

  Returns Euler's number e, aka exp(1)

- PI

```
# perl -Mbigfloat=PI -wle 'print PI'
```

Returns PI.

- bexp()

```
bexp($power, $accuracy);
```

Returns Euler's number `e` raised to the appropriate power, to the wanted accuracy.

Example:

```
# perl -Mbigfloat=bexp -wle 'print bexp(1,80)'
```

- bpi()

```
bpi($accuracy);
```

Returns PI to the wanted accuracy.

Example:

```
# perl -Mbigfloat=bpi -wle 'print bpi(80)'
```

- accuracy()

  Set or get the accuracy.

- precision()

  Set or get the precision.

- round_mode()

  Set or get the rounding mode.

- div_scale()

  Set or get the division scale.

- upgrade()

  Set or get the class that the downgrade class upgrades to, if any. Set the upgrade class to `undef` to disable upgrading.

  Upgrading is disabled by default.

- downgrade()

  Set or get the class that the upgrade class downgrades to, if any. Set the downgrade class to `undef` to disable upgrading.

  Downgrading is disabled by default.

- in_effect()

```
use bigfloat;

print "in effect\n" if bigfloat::in_effect;       # true
```

(continues on next page)

(continued from previous page)

```
{
    no bigfloat;
    print "in effect\n" if bigfloat::in_effect;   # false
}
```

Returns true or false if `bigfloat` is in effect in the current scope.

This method only works on Perl v5.9.4 or later.

CAVEATS

- Hexadecimal, octal, and binary floating point literals

  Perl (and this module) accepts hexadecimal, octal, and binary floating point literals, but use them with care with Perl versions before v5.32.0, because some versions of Perl silently give the wrong result.

- Operator vs literal overloading

  `bigrat` works by overloading handling of integer and floating point literals, converting them to *Math::BigRat* objects.

  This means that arithmetic involving only string values or string literals are performed using Perl's built-in operators.

  For example:

```
use bigrat;
my $x = "900000000000000009";
my $y = "900000000000000007";
print $x - $y;
```

  outputs `0` on default 32-bit builds, since `bigfloat` never sees the string literals. To ensure the expression is all treated as `Math::BigFloat` objects, use a literal number in the expression:

```
print +(0+$x) - $y;
```

- Ranges

  Perl does not allow overloading of ranges, so you can neither safely use ranges with `bigfloat` endpoints, nor is the iterator variable a `Math::BigFloat`.

```
use 5.010;
for my $i (12..13) {
  for my $j (20..21) {
    say $i ** $j;  # produces a floating-point number,
                   # not an object
  }
}
```

- in_effect()

  This method only works on Perl v5.9.4 or later.

- hex()/oct()

  `bigfloat` overrides these routines with versions that can also handle big integer values. Under Perl prior to version v5.9.4, however, this will not happen unless you specifically ask for it with the two import tags "hex" and "oct" - and then it will be global and cannot be disabled inside a scope with `no bigfloat`:

```
use bigfloat qw/hex oct/;

print hex("0x1234567890123456");
{
    no bigfloat;
    print hex("0x1234567890123456");
}
```

  The second call to hex() will warn about a non-portable constant.

  Compare this to:

```
use bigfloat;

# will warn only under Perl older than v5.9.4
print hex("0x1234567890123456");
```

EXAMPLES

Some cool command line examples to impress the Python crowd ;)

```
perl -Mbigfloat -le 'print sqrt(33)'
perl -Mbigfloat -le 'print 2**255'
perl -Mbigfloat -le 'print 4.5+2**255'
perl -Mbigfloat -le 'print 3/7 + 5/7 + 8/3'
perl -Mbigfloat -le 'print 123->is_odd()'
perl -Mbigfloat -le 'print log(2)'
perl -Mbigfloat -le 'print exp(1)'
perl -Mbigfloat -le 'print 2 ** 0.5'
perl -Mbigfloat=a,65 -le 'print 2 ** 0.2'
perl -Mbigfloat=l,GMP -le 'print 7 ** 7777'
```

BUGS

Please report any bugs or feature requests to `bug-bignum at rt.cpan.org`, or through the web interface at https://rt.cpan.org/Ticket/Create.html?Queue=bignum (requires login). We will be notified, and then you'll automatically be notified of progress on your bug as I make changes.

SUPPORT

You can find documentation for this module with the perldoc command.

```
perldoc bigfloat
```

You can also look for information at:

- GitHub

  https://github.com/pjacklam/p5-bignum

- RT: CPAN's request tracker

  https://rt.cpan.org/Dist/Display.html?Name=bignum

- MetaCPAN

  https://metacpan.org/release/bignum

- CPAN Testers Matrix

  http://matrix.cpantesters.org/?dist=bignum

LICENSE

This program is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

SEE ALSO

bigint and *bigrat*.

Math::BigInt, *Math::BigFloat*, *Math::BigRat* and Math::Big as well as *Math::BigInt::FastCalc*, Math::BigInt::Pari and Math::BigInt::GMP.

AUTHORS

- (C) by Tels http://bloodgate.com/ in early 2002 - 2007.
- Maintained by Peter John Acklam pjacklam@gmail.com, 2014-.

5.2.122 bigint

transparent big integer support for Perl

SYNOPSIS

```
use bigint;

$x = 2 + 4.5;                       # Math::BigInt 6
print 2 ** 512;                     # Math::BigInt 134...096
print inf + 42;                     # Math::BigInt inf
print NaN * 7;                      # Math::BigInt NaN
print hex("0x1234567890123490");    # Perl v5.10.0 or later

{
    no bigint;
    print 2 ** 256;                 # a normal Perl scalar now
}
```

(continues on next page)

(continued from previous page)

```
# for older Perls, import into current package:
use bigint qw/hex oct/;
print hex("0x1234567890123490");
print oct("01234567890123490");
```

DESCRIPTION

All numeric literal in the given scope are converted to Math::BigInt objects. Numeric literal that represent non-integers are truncated to an integer. All results of expressions are also truncated to integer.

All operators (including basic math operations) except the range operator `..` are overloaded.

Unlike the *integer* pragma, the `bigint` pragma creates integers that are only limited in their size by the available memory.

So, the following:

```
use bigint;
$x = 1234;
```

creates a Math::BigInt and stores a reference to in $x. This happens transparently and behind your back, so to speak.

You can see this with the following:

```
perl -Mbigint -le 'print ref(1234)'
```

Since numbers are actually objects, you can call all the usual methods from Math::BigFloat on them. This even works to some extent on expressions:

```
perl -Mbigint -le '$x = 1234; print $x->bdec()'
perl -Mbigint -le 'print 1234->copy()->binc();'
perl -Mbigint -le 'print 1234->copy()->binc->badd(6);'
perl -Mbigint -le 'print +(1234)->copy()->binc()'
```

(Note that print doesn't do what you expect if the expression starts with '(' hence the +)

You can even chain the operations together as usual:

```
perl -Mbigint -le 'print 1234->copy()->binc->badd(6);'
1241
```

Please note the following does not work as expected (prints nothing), since overloading of '..' is not yet possible in Perl (as of v5.8.0):

```
perl -Mbigint -le 'for (1..2) { print ref($_); }'
```

use integer vs. use bigint

There are some difference between `use integer` and `use bigint`.

Whereas `use integer` is limited to what can be handled as a Perl scalar, `use bigint` can handle arbitrarily large integers.

Also, `use integer` does affect assignments to variables and the return value of some functions. `use bigint` truncates these results to integer:

```
# perl -Minteger -wle 'print 3.2'
3.2
# perl -Minteger -wle 'print 3.2 + 0'
3
# perl -Mbigint -wle 'print 3.2'
3
# perl -Mbigint -wle 'print 3.2 + 0'
3

# perl -Mbigint -wle 'print exp(1) + 0'
2
# perl -Mbigint -wle 'print exp(1)'
2
# perl -Minteger -wle 'print exp(1)'
2.71828182845905
# perl -Minteger -wle 'print exp(1) + 0'
2
```

In practice this seldom makes a difference for small integers as **parts and results** of expressions are truncated anyway, but this can, for instance, affect the return value of subroutines:

```
sub three_integer { use integer; return 3.2; }
sub three_bigint { use bigint; return 3.2; }

print three_integer(), " ", three_bigint(),"\n";    # prints "3.2 3"
```

Options

`bigint` recognizes some options that can be passed while loading it via `use`. The following options exist:

- a or accuracy

  This sets the accuracy for all math operations. The argument must be greater than or equal to zero. See Math::BigInt's bround() method for details.

  ```
  perl -Mbigint=a,2 -le 'print 12345+1'
  ```

  Note that setting precision and accuracy at the same time is not possible.

- p or precision

  This sets the precision for all math operations. The argument can be any integer. Negative values mean a fixed number of digits after the dot, and are ignored since

all operations happen in integer space. A positive value rounds to this digit left from the dot. 0 means round to integer. See Math::BigInt's bfround() method for details.

```
perl -mbigint=p,5 -le 'print 123456789+123'
```

Note that setting precision and accuracy at the same time is not possible.

- t or trace

  This enables a trace mode and is primarily for debugging.

- l, lib, try, or only

  Load a different math lib, see "Math Library".

```
perl -Mbigint=l,GMP -e 'print 2 ** 512'
perl -Mbigint=lib,GMP -e 'print 2 ** 512'
perl -Mbigint=try,GMP -e 'print 2 ** 512'
perl -Mbigint=only,GMP -e 'print 2 ** 512'
```

- hex

  Override the built-in hex() method with a version that can handle big numbers. This overrides it by exporting it to the current package. Under Perl v5.10.0 and higher, this is not necessary, as hex() is lexically overridden in the current scope whenever the `bigint` pragma is active.

- oct

  Override the built-in oct() method with a version that can handle big numbers. This overrides it by exporting it to the current package. Under Perl v5.10.0 and higher, this is not so necessary, as oct() is lexically overridden in the current scope whenever the `bigint` pragma is active.

- v or version

  this prints out the name and version of the modules and then exits.

```
perl -Mbigint=v
```

Math Library

Math with the numbers is done (by default) by a backend library module called Math::BigInt::Calc. The default is equivalent to saying:

```
use bigint lib => 'Calc';
```

you can change this by using:

```
use bigint lib => 'GMP';
```

The following would first try to find Math::BigInt::Foo, then Math::BigInt::Bar, and if this also fails, revert to Math::BigInt::Calc:

```
use bigint lib => 'Foo,Math::BigInt::Bar';
```

Using c<lib> warns if none of the specified libraries can be found and *Math::BigInt* fell back to one of the default libraries. To suppress this warning, use c<try> instead:

```
use bigint try => 'GMP';
```

If you want the code to die instead of falling back, use only instead:

```
use bigint only => 'GMP';
```

Please see the respective module documentation for further details.

Method calls

Since all numbers are now objects, you can use all methods that are part of the Math::BigInt API.

But a warning is in order. When using the following to make a copy of a number, only a shallow copy will be made.

```
$x = 9; $y = $x;
$x = $y = 7;
```

Using the copy or the original with overloaded math is okay, e.g., the following work:

```
$x = 9; $y = $x;
print $x + 1, " ", $y,"\n";     # prints 10 9
```

but calling any method that modifies the number directly will result in **both** the original and the copy being destroyed:

```
$x = 9; $y = $x;
print $x->badd(1), " ", $y,"\n";        # prints 10 10

$x = 9; $y = $x;
print $x->binc(1), " ", $y,"\n";        # prints 10 10

$x = 9; $y = $x;
print $x->bmul(2), " ", $y,"\n";        # prints 18 18
```

Using methods that do not modify, but test that the contents works:

```
$x = 9; $y = $x;
$z = 9 if $x->is_zero();                # works fine
```

See the documentation about the copy constructor and = in overload, as well as the documentation in Math::BigInt for further details.

Methods

- inf()

  A shortcut to return Math::BigInt->binf(). Useful because Perl does not always handle bareword `inf` properly.

- NaN()

  A shortcut to return Math::BigInt->bnan(). Useful because Perl does not always handle bareword `NaN` properly.

- e

```
# perl -Mbigint=e -wle 'print e'
```

  Returns Euler's number `e`, aka exp(1). Note that under `bigint`, this is truncated to an integer, i.e., 2.

- PI

```
# perl -Mbigint=PI -wle 'print PI'
```

  Returns PI. Note that under `bigint`, this is truncated to an integer, i.e., 3.

- bexp()

```
bexp($power, $accuracy);
```

  Returns Euler's number `e` raised to the appropriate power, to the wanted accuracy.

  Note that under `bigint`, the result is truncated to an integer.

  Example:

```
# perl -Mbigint=bexp -wle 'print bexp(1,80)'
```

- bpi()

```
bpi($accuracy);
```

  Returns PI to the wanted accuracy. Note that under `bigint`, this is truncated to an integer, i.e., 3.

  Example:

```
# perl -Mbigint=bpi -wle 'print bpi(80)'
```

- accuracy()

  Set or get the accuracy.

- precision()

  Set or get the precision.

- round_mode()

  Set or get the rounding mode.

- div_scale()

  Set or get the division scale.

- in_effect()

```
use bigint;

print "in effect\n" if bigint::in_effect;       # true
{
    no bigint;
    print "in effect\n" if bigint::in_effect;   # false
}
```

  Returns true or false if `bigint` is in effect in the current scope.

  This method only works on Perl v5.9.4 or later.

CAVEATS

- Hexadecimal, octal, and binary floating point literals

  Perl (and this module) accepts hexadecimal, octal, and binary floating point literals, but use them with care with Perl versions before v5.32.0, because some versions of Perl silently give the wrong result.

- Operator vs literal overloading

  `bigint` works by overloading handling of integer and floating point literals, converting them to *Math::BigInt* objects.

  This means that arithmetic involving only string values or string literals are performed using Perl's built-in operators.

  For example:

```
use bigint;
my $x = "900000000000000009";
my $y = "900000000000000007";
print $x - $y;
```

  outputs `0` on default 32-bit builds, since `bigint` never sees the string literals. To ensure the expression is all treated as `Math::BigInt` objects, use a literal number in the expression:

```
print +(0+$x) - $y;
```

- Ranges

  Perl does not allow overloading of ranges, so you can neither safely use ranges with `bigint` endpoints, nor is the iterator variable a `Math::BigInt`.

```
use 5.010;
for my $i (12..13) {
  for my $j (20..21) {
    say $i ** $j;  # produces a floating-point number,
```

(continues on next page)

(continued from previous page)

```
                    # not an object
  }
}
```

- in_effect()

  This method only works on Perl v5.9.4 or later.

- hex()/oct()

  `bigint` overrides these routines with versions that can also handle big integer values. Under Perl prior to version v5.9.4, however, this will not happen unless you specifically ask for it with the two import tags "hex" and "oct" - and then it will be global and cannot be disabled inside a scope with `no bigint`:

```
use bigint qw/hex oct/;

print hex("0x1234567890123456");
{
    no bigint;
    print hex("0x1234567890123456");
}
```

  The second call to hex() will warn about a non-portable constant.

  Compare this to:

```
use bigint;

# will warn only under Perl older than v5.9.4
print hex("0x1234567890123456");
```

EXAMPLES

Some cool command line examples to impress the Python crowd ;) You might want to compare them to the results under -Mbigfloat or -Mbigrat:

```
perl -Mbigint -le 'print sqrt(33)'
perl -Mbigint -le 'print 2**255'
perl -Mbigint -le 'print 4.5+2**255'
perl -Mbigint -le 'print 123->is_odd()'
perl -Mbigint=l,GMP -le 'print 7 ** 7777'
```

BUGS

Please report any bugs or feature requests to `bug-bignum at rt.cpan.org`, or through the web interface at https://rt.cpan.org/Ticket/Create.html?Queue=bignum (requires login). We will be notified, and then you'll automatically be notified of progress on your bug as I make changes.

SUPPORT

You can find documentation for this module with the perldoc command.

```
perldoc bigint
```

You can also look for information at:

- GitHub

  https://github.com/pjacklam/p5-bignum

- RT: CPAN's request tracker

  https://rt.cpan.org/Dist/Display.html?Name=bignum

- MetaCPAN

  https://metacpan.org/release/bignum

- CPAN Testers Matrix

  http://matrix.cpantesters.org/?dist=bignum

LICENSE

This program is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

SEE ALSO

bignum and *bigrat*.

Math::BigInt, *Math::BigFloat*, *Math::BigRat* and Math::Big as well as *Math::BigInt::FastCalc*, Math::BigInt::Pari and Math::BigInt::GMP.

AUTHORS

- (C) by Tels http://bloodgate.com/ in early 2002 - 2007.
- Maintained by Peter John Acklam pjacklam@gmail.com, 2014-.

5.2.123 bignum

transparent big number support for Perl

SYNOPSIS

```
use bignum;

$x = 2 + 4.5;                       # Math::BigFloat 6.5
print 2 ** 512 * 0.1;               # Math::BigFloat 134...09.6
print 2 ** 512;                     # Math::BigInt 134...096
print inf + 42;                     # Math::BigInt inf
print NaN * 7;                      # Math::BigInt NaN
```

(continues on next page)

(continued from previous page)

```
print hex("0x1234567890123490");    # Perl v5.10.0 or later

{
    no bignum;
    print 2 ** 256;                 # a normal Perl scalar now
}

# for older Perls, import into current package:
use bignum qw/hex oct/;
print hex("0x1234567890123490");
print oct("01234567890123490");
```

DESCRIPTION

Literal numeric constants

By default, every literal integer becomes a Math::BigInt object, and literal non-integer becomes a Math::BigFloat object. Whether a numeric literal is considered an integer or non-integers depends only on the value of the constant, not on how it is represented. For instance, the constants 3.14e2 and 0x1.3ap8 become Math::BigInt objects, because they both represent the integer value decimal 314.

The default `use bignum;` is equivalent to

```
use bignum downgrade => "Math::BigInt", upgrade => "Math::BigFloat";
```

The classes used for integers and non-integers can be set at compile time with the `downgrade` and `upgrade` options, for example

```
# use Math::BigInt for integers and Math::BigRat for non-integers
use bignum upgrade => "Math::BigRat";
```

Note that disabling downgrading and upgrading does not affect how numeric literals are converted to objects

```
# disable both downgrading and upgrading
use bignum downgrade => undef, upgrade => undef;
$x = 2.4;       # becomes 2.4 as a Math::BigFloat
$y = 2;         # becomes 2 as a Math::BigInt
```

Upgrading and downgrading

By default, when the result of a computation is an integer, an Inf, or a NaN, the result is downgraded even when all the operands are instances of the upgrade class.

```
use bignum;
$x = 2.4;       # becomes 2.4 as a Math::BigFloat
$y = 1.2;       # becomes 1.2 as a Math::BigFloat
$z = $x / $y;   # becomes 2 as a Math::BigInt due to downgrading
```

Equivalently, by default, when the result of a computation is a finite non-integer, the result is upgraded even when all the operands are instances of the downgrade class.

```
use bignum;
$x = 7;          # becomes 7 as a Math::BigInt
$y = 2;          # becomes 2 as a Math::BigInt
$z = $x / $y;    # becomes 3.5 as a Math::BigFloat due to upgrading
```

The classes used for downgrading and upgrading can be set at runtime with the "downgrade()" and "upgrade()" methods, but see "CAVEATS" below.

The upgrade and downgrade classes don't have to be Math::BigInt and Math::BigFloat. For example, to use Math::BigRat as the upgrade class, use

```
use bignum upgrade => "Math::BigRat";
$x = 2;          # becomes 2 as a Math::BigInt
$y = 3.6;        # becomes 18/5 as a Math::BigRat
```

The upgrade and downgrade classes can be modified at runtime

```
use bignum;
$x = 3;          # becomes 3 as a Math::BigInt
$y = 2;          # becomes 2 as a Math::BigInt
$z = $x / $y;    # becomes 1.5 as a Math::BigFlaot

bignum -> upgrade("Math::BigRat");
$w = $x / $y;    # becomes 3/2 as a Math::BigRat
```

Disabling downgrading doesn't change the fact that literal constant integers are converted to the downgrade class, it only prevents downgrading as a result of a computation. E.g.,

```
use bignum downgrade => undef;
$x = 2;          # becomes 2 as a Math::BigInt
$y = 2.4;        # becomes 2.4 as a Math::BigFloat
$z = 1.2;        # becomes 1.2 as a Math::BigFloat
$w = $x / $y;    # becomes 2 as a Math::BigFloat due to no␣
↪downgrading
```

If you want all numeric literals, both integers and non-integers, to become Math::BigFloat objects, use the *bigfloat* pragma.

Equivalently, disabling upgrading doesn't change the fact that literal constant non-integers are converted to the upgrade class, it only prevents upgrading as a result of a computation. E.g.,

```
use bignum upgrade => undef;
$x = 2.5;        # becomes 2.5 as a Math::BigFloat
$y = 7;          # becomes 7 as a Math::BigInt
$z = 2;          # becomes 2 as a Math::BigInt
$w = $x / $y;    # becomes 3 as a Math::BigInt due to no upgrading
```

If you want all numeric literals, both integers and non-integers, to become Math::BigInt objects, use the *bigint* pragma.

You can even do

```
use bignum upgrade => "Math::BigRat", upgrade => undef;
```

which converts all integer literals to Math::BigInt objects and all non-integer literals to Math::BigRat objects. However, when the result of a computation involving two Math::BigInt objects results in a non-integer (e.g., 7/2), the result will be truncted to a Math::BigInt rather than being upgraded to a Math::BigRat, since upgrading is disabled.

Overloading

Since all numeric literals become objects, you can call all the usual methods from Math::BigInt and Math::BigFloat on them. This even works to some extent on expressions:

```
perl -Mbignum -le '$x = 1234; print $x->bdec()'
perl -Mbignum -le 'print 1234->copy()->binc();'
perl -Mbignum -le 'print 1234->copy()->binc()->badd(6);'
```

Options

`bignum` recognizes some options that can be passed while loading it via via `use`. The following options exist:

- a or accuracy

  This sets the accuracy for all math operations. The argument must be greater than or equal to zero. See Math::BigInt's bround() method for details.

  ```
  perl -Mbignum=a,50 -le 'print sqrt(20)'
  ```

  Note that setting precision and accuracy at the same time is not possible.

- p or precision

  This sets the precision for all math operations. The argument can be any integer. Negative values mean a fixed number of digits after the dot, while a positive value rounds to this digit left from the dot. 0 means round to integer. See Math::BigInt's bfround() method for details.

  ```
  perl -Mbignum=p,-50 -le 'print sqrt(20)'
  ```

  Note that setting precision and accuracy at the same time is not possible.

- l, lib, try, or only

  Load a different math lib, see "Math Library".

  ```
  perl -Mbignum=l,GMP -e 'print 2 ** 512'
  perl -Mbignum=lib,GMP -e 'print 2 ** 512'
  perl -Mbignum=try,GMP -e 'print 2 ** 512'
  perl -Mbignum=only,GMP -e 'print 2 ** 512'
  ```

- hex

Override the built-in hex() method with a version that can handle big numbers. This overrides it by exporting it to the current package. Under Perl v5.10.0 and higher, this is not so necessary, as hex() is lexically overridden in the current scope whenever the `bignum` pragma is active.

- oct

  Override the built-in oct() method with a version that can handle big numbers. This overrides it by exporting it to the current package. Under Perl v5.10.0 and higher, this is not so necessary, as oct() is lexically overridden in the current scope whenever the `bignum` pragma is active.

- v or version

  this prints out the name and version of the modules and then exits.

```
perl -Mbignum=v
```

Math Library

Math with the numbers is done (by default) by a backend library module called Math::BigInt::Calc. The default is equivalent to saying:

```
use bignum lib => 'Calc';
```

you can change this by using:

```
use bignum lib => 'GMP';
```

The following would first try to find Math::BigInt::Foo, then Math::BigInt::Bar, and if this also fails, revert to Math::BigInt::Calc:

```
use bignum lib => 'Foo,Math::BigInt::Bar';
```

Using c<lib> warns if none of the specified libraries can be found and *Math::BigInt* and *Math::BigFloat* fell back to one of the default libraries. To suppress this warning, use `try` instead:

```
use bignum try => 'GMP';
```

If you want the code to die instead of falling back, use `only` instead:

```
use bignum only => 'GMP';
```

Please see respective module documentation for further details.

Method calls

Since all numbers are now objects, you can use the methods that are part of the Math::BigInt and Math::BigFloat API.

But a warning is in order. When using the following to make a copy of a number, only a shallow copy will be made.

```
$x = 9; $y = $x;
$x = $y = 7;
```

Using the copy or the original with overloaded math is okay, e.g., the following work:

```
$x = 9; $y = $x;
print $x + 1, " ", $y,"\n";     # prints 10 9
```

but calling any method that modifies the number directly will result in **both** the original and the copy being destroyed:

```
$x = 9; $y = $x;
print $x->badd(1), " ", $y,"\n";        # prints 10 10

$x = 9; $y = $x;
print $x->binc(1), " ", $y,"\n";        # prints 10 10

$x = 9; $y = $x;
print $x->bmul(2), " ", $y,"\n";        # prints 18 18
```

Using methods that do not modify, but test that the contents works:

```
$x = 9; $y = $x;
$z = 9 if $x->is_zero();                # works fine
```

See the documentation about the copy constructor and = in overload, as well as the documentation in Math::BigFloat for further details.

Methods

- inf()

  A shortcut to return `inf` as an object. Useful because Perl does not always handle bareword `inf` properly.

- NaN()

  A shortcut to return `NaN` as an object. Useful because Perl does not always handle bareword `NaN` properly.

- e

  ```
  # perl -Mbignum=e -wle 'print e'
  ```

  Returns Euler's number e, aka exp(1) (= 2.7182818284...).

- PI

  ```
  # perl -Mbignum=PI -wle 'print PI'
  ```

  Returns PI (= 3.1415926532..).

- bexp()

```
bexp($power, $accuracy);
```

Returns Euler's number e raised to the appropriate power, to the wanted accuracy.

Example:

```
# perl -Mbignum=bexp -wle 'print bexp(1,80)'
```

- bpi()

```
bpi($accuracy);
```

Returns PI to the wanted accuracy.

Example:

```
# perl -Mbignum=bpi -wle 'print bpi(80)'
```

- accuracy()

  Set or get the accuracy.

- precision()

  Set or get the precision.

- round_mode()

  Set or get the rounding mode.

- div_scale()

  Set or get the division scale.

- upgrade()

  Set or get the class that the downgrade class upgrades to, if any. Set the upgrade class to undef to disable upgrading. See /CAVEATS below.

- downgrade()

  Set or get the class that the upgrade class downgrades to, if any. Set the downgrade class to undef to disable upgrading. See "CAVEATS" below.

- in_effect()

```
use bignum;

print "in effect\n" if bignum::in_effect;       # true
{
    no bignum;
    print "in effect\n" if bignum::in_effect;   # false
}
```

Returns true or false if bignum is in effect in the current scope.

This method only works on Perl v5.9.4 or later.

CAVEATS

- The upgrade() and downgrade() methods

  Note that setting both the upgrade and downgrade classes at runtime with the "upgrade()" and "downgrade()" methods, might not do what you expect:

```
# Assuming that downgrading and upgrading hasn't been modified␣
 ↪so far, so
# the downgrade and upgrade classes are Math::BigInt and␣
 ↪Math::BigFloat,
# respectively, the following sets the upgrade class to␣
 ↪Math::BigRat, i.e.,
# makes Math::BigInt upgrade to Math::BigRat:

bignum -> upgrade("Math::BigRat");

# The following sets the downgrade class to Math::BigInt::Lite,␣
 ↪i.e., makes
# the new upgrade class Math::BigRat downgrade to␣
 ↪Math::BigInt::Lite

bignum -> downgrade("Math::BigInt::Lite");

# Note that at this point, it is still Math::BigInt, not␣
 ↪Math::BigInt::Lite,
# that upgrades to Math::BigRat, so to get Math::BigInt::Lite␣
 ↪to upgrade to
# Math::BigRat, we need to do the following (again):

bignum -> upgrade("Math::BigRat");
```

  A simpler way to do this at runtime is to use import(),

```
bignum -> import(upgrade => "Math::BigRat",
                 downgrade => "Math::BigInt::Lite");
```

- Hexadecimal, octal, and binary floating point literals

  Perl (and this module) accepts hexadecimal, octal, and binary floating point literals, but use them with care with Perl versions before v5.32.0, because some versions of Perl silently give the wrong result.

- Operator vs literal overloading

  `bigrat` works by overloading handling of integer and floating point literals, converting them to *Math::BigRat* objects.

  This means that arithmetic involving only string values or string literals are performed using Perl's built-in operators.

  For example:

```
use bigrat;
my $x = "900000000000000009";
```

(continues on next page)

(continued from previous page)

```
my $y = "900000000000000007";
print $x - $y;
```

outputs `0` on default 32-bit builds, since `bignum` never sees the string literals. To ensure the expression is all treated as `Math::BigFloat` objects, use a literal number in the expression:

```
print +(0+$x) - $y;
```

- Ranges

  Perl does not allow overloading of ranges, so you can neither safely use ranges with `bignum` endpoints, nor is the iterator variable a `Math::BigFloat`.

```
use 5.010;
for my $i (12..13) {
  for my $j (20..21) {
    say $i ** $j;  # produces a floating-point number,
                   # not an object
  }
}
```

- in_effect()

  This method only works on Perl v5.9.4 or later.

- hex()/oct()

  `bignum` overrides these routines with versions that can also handle big integer values. Under Perl prior to version v5.9.4, however, this will not happen unless you specifically ask for it with the two import tags “hex” and “oct” - and then it will be global and cannot be disabled inside a scope with `no bignum`:

```
use bignum qw/hex oct/;

print hex("0x1234567890123456");
{
    no bignum;
    print hex("0x1234567890123456");
}
```

  The second call to hex() will warn about a non-portable constant.

  Compare this to:

```
use bignum;

# will warn only under Perl older than v5.9.4
print hex("0x1234567890123456");
```

EXAMPLES

Some cool command line examples to impress the Python crowd ;)

```
perl -Mbignum -le 'print sqrt(33)'
perl -Mbignum -le 'print 2**255'
perl -Mbignum -le 'print 4.5+2**255'
perl -Mbignum -le 'print 3/7 + 5/7 + 8/3'
perl -Mbignum -le 'print 123->is_odd()'
perl -Mbignum -le 'print log(2)'
perl -Mbignum -le 'print exp(1)'
perl -Mbignum -le 'print 2 ** 0.5'
perl -Mbignum=a,65 -le 'print 2 ** 0.2'
perl -Mbignum=l,GMP -le 'print 7 ** 7777'
```

BUGS

Please report any bugs or feature requests to `bug-bignum at rt.cpan.org`, or through the web interface at https://rt.cpan.org/Ticket/Create.html?Queue=bignum (requires login). We will be notified, and then you'll automatically be notified of progress on your bug as I make changes.

SUPPORT

You can find documentation for this module with the perldoc command.

```
perldoc bignum
```

You can also look for information at:

- GitHub

  https://github.com/pjacklam/p5-bignum

- RT: CPAN's request tracker

  https://rt.cpan.org/Dist/Display.html?Name=bignum

- MetaCPAN

  https://metacpan.org/release/bignum

- CPAN Testers Matrix

  http://matrix.cpantesters.org/?dist=bignum

LICENSE

This program is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

SEE ALSO

bigint and *bigrat*.

Math::BigInt, *Math::BigFloat*, *Math::BigRat* and Math::Big as well as *Math::BigInt::FastCalc*, Math::BigInt::Pari and Math::BigInt::GMP.

AUTHORS

- (C) by Tels http://bloodgate.com/ in early 2002 - 2007.
- Maintained by Peter John Acklam pjacklam@gmail.com, 2014-.

5.2.124 bigrat

transparent big rational number support for Perl

SYNOPSIS

```
use bigrat;

print 2 + 4.5;                      # Math::BigRat 13/2
print 1/3 + 1/4;                    # Math::BigRat 7/12
print inf + 42;                     # Math::BigRat inf
print NaN * 7;                      # Math::BigRat NaN
print hex("0x1234567890123490");    # Perl v5.10.0 or later

{
    no bigrat;
    print 1/3;                      # 0.33333...
}

# for older Perls, import into current package:
use bigrat qw/hex oct/;
print hex("0x1234567890123490");
print oct("01234567890123490");
```

DESCRIPTION

All numeric literal in the given scope are converted to Math::BigRat objects.

All operators (including basic math operations) except the range operator `..` are overloaded.

So, the following:

```
use bigrat;
$x = 1234;
```

creates a Math::BigRat and stores a reference to in $x. This happens transparently and behind your back, so to speak.

You can see this with the following:

```
perl -Mbigrat -le 'print ref(1234)'
```

Since numbers are actually objects, you can call all the usual methods from Math::BigRat on them. This even works to some extent on expressions:

```
perl -Mbigrat -le '$x = 1234; print $x->bdec()'
perl -Mbigrat -le 'print 1234->copy()->binc();'
perl -Mbigrat -le 'print 1234->copy()->binc->badd(6);'
perl -Mbigrat -le 'print +(1234)->copy()->binc()'
```

(Note that print doesn't do what you expect if the expression starts with '(' hence the +)

You can even chain the operations together as usual:

```
perl -Mbigrat -le 'print 1234->copy()->binc->badd(6);'
1241
```

Please note the following does not work as expected (prints nothing), since overloading of '..' is not yet possible in Perl (as of v5.8.0):

```
perl -Mbigrat -le 'for (1..2) { print ref($_); }'
```

Options

`bigrat` recognizes some options that can be passed while loading it via `use`. The following options exist:

- a or accuracy

  This sets the accuracy for all math operations. The argument must be greater than or equal to zero. See Math::BigInt's bround() method for details.

  ```
  perl -Mbigrat=a,50 -le 'print sqrt(20)'
  ```

  Note that setting precision and accuracy at the same time is not possible.

- p or precision

  This sets the precision for all math operations. The argument can be any integer. Negative values mean a fixed number of digits after the dot, while a positive value rounds to this digit left from the dot. 0 means round to integer. See Math::BigInt's bfround() method for details.

  ```
  perl -Mbigrat=p,-50 -le 'print sqrt(20)'
  ```

  Note that setting precision and accuracy at the same time is not possible.

- t or trace

  This enables a trace mode and is primarily for debugging.

- l, lib, try, or only

  Load a different math lib, see "Math Library".

```
perl -Mbigrat=l,GMP -e 'print 2 ** 512'
perl -Mbigrat=lib,GMP -e 'print 2 ** 512'
perl -Mbigrat=try,GMP -e 'print 2 ** 512'
perl -Mbigrat=only,GMP -e 'print 2 ** 512'
```

- hex

  Override the built-in hex() method with a version that can handle big numbers. This overrides it by exporting it to the current package. Under Perl v5.10.0 and higher, this is not so necessary, as hex() is lexically overridden in the current scope whenever the `bigrat` pragma is active.

- oct

  Override the built-in oct() method with a version that can handle big numbers. This overrides it by exporting it to the current package. Under Perl v5.10.0 and higher, this is not so necessary, as oct() is lexically overridden in the current scope whenever the `bigrat` pragma is active.

- v or version

  this prints out the name and version of the modules and then exits.

```
perl -Mbigrat=v
```

Math Library

Math with the numbers is done (by default) by a backend library module called Math::BigInt::Calc. The default is equivalent to saying:

```
use bigrat lib => 'Calc';
```

you can change this by using:

```
use bigrat lib => 'GMP';
```

The following would first try to find Math::BigInt::Foo, then Math::BigInt::Bar, and if this also fails, revert to Math::BigInt::Calc:

```
use bigrat lib => 'Foo,Math::BigInt::Bar';
```

Using c<lib> warns if none of the specified libraries can be found and *Math::BigInt* fell back to one of the default libraries. To suppress this warning, use c<try> instead:

```
use bigrat try => 'GMP';
```

If you want the code to die instead of falling back, use `only` instead:

```
use bigrat only => 'GMP';
```

Please see the respective module documentation for further details.

Method calls

Since all numbers are now objects, you can use all methods that are part of the Math::BigRat API.

But a warning is in order. When using the following to make a copy of a number, only a shallow copy will be made.

```
$x = 9; $y = $x;
$x = $y = 7;
```

Using the copy or the original with overloaded math is okay, e.g., the following work:

```
$x = 9; $y = $x;
print $x + 1, " ", $y,"\n";     # prints 10 9
```

but calling any method that modifies the number directly will result in **both** the original and the copy being destroyed:

```
$x = 9; $y = $x;
print $x->badd(1), " ", $y,"\n";        # prints 10 10

$x = 9; $y = $x;
print $x->binc(1), " ", $y,"\n";        # prints 10 10

$x = 9; $y = $x;
print $x->bmul(2), " ", $y,"\n";        # prints 18 18
```

Using methods that do not modify, but test that the contents works:

```
$x = 9; $y = $x;
$z = 9 if $x->is_zero();                # works fine
```

See the documentation about the copy constructor and = in overload, as well as the documentation in Math::BigFloat for further details.

Methods

- inf()

  A shortcut to return Math::BigRat->binf(). Useful because Perl does not always handle bareword `inf` properly.

- NaN()

  A shortcut to return Math::BigRat->bnan(). Useful because Perl does not always handle bareword `NaN` properly.

- e

  ```
  # perl -Mbigrat=e -wle 'print e'
  ```

  Returns Euler's number e, aka exp(1).

- PI

```
# perl -Mbigrat=PI -wle 'print PI'
```

Returns PI.

- bexp()

```
bexp($power, $accuracy);
```

Returns Euler's number e raised to the appropriate power, to the wanted accuracy.

Example:

```
# perl -Mbigrat=bexp -wle 'print bexp(1,80)'
```

- bpi()

```
bpi($accuracy);
```

Returns PI to the wanted accuracy.

Example:

```
# perl -Mbigrat=bpi -wle 'print bpi(80)'
```

- accuracy()

  Set or get the accuracy.

- precision()

  Set or get the precision.

- round_mode()

  Set or get the rounding mode.

- div_scale()

  Set or get the division scale.

- in_effect()

```
use bigrat;

print "in effect\n" if bigrat::in_effect;       # true
{
    no bigrat;
    print "in effect\n" if bigrat::in_effect;   # false
}
```

Returns true or false if `bigrat` is in effect in the current scope.

This method only works on Perl v5.9.4 or later.

CAVEATS

- Hexadecimal, octal, and binary floating point literals

  Perl (and this module) accepts hexadecimal, octal, and binary floating point literals, but use them with care with Perl versions before v5.32.0, because some versions of Perl silently give the wrong result.

- Operator vs literal overloading

  `bigrat` works by overloading handling of integer and floating point literals, converting them to *Math::BigRat* objects.

  This means that arithmetic involving only string values or string literals are performed using Perl's built-in operators.

  For example:

```
use bigrat;
my $x = "900000000000000009";
my $y = "900000000000000007";
print $x - $y;
```

  outputs `0` on default 32-bit builds, since `bigrat` never sees the string literals. To ensure the expression is all treated as `Math::BigRat` objects, use a literal number in the expression:

```
print +(0+$x) - $y;
```

- Ranges

  Perl does not allow overloading of ranges, so you can neither safely use ranges with `bigrat` endpoints, nor is the iterator variable a `Math::BigRat`.

```
use 5.010;
for my $i (12..13) {
  for my $j (20..21) {
    say $i ** $j;  # produces a floating-point number,
                   # not an object
  }
}
```

- in_effect()

  This method only works on Perl v5.9.4 or later.

- hex()/oct()

  `bigrat` overrides these routines with versions that can also handle big integer values. Under Perl prior to version v5.9.4, however, this will not happen unless you specifically ask for it with the two import tags "hex" and "oct" - and then it will be global and cannot be disabled inside a scope with `no bigrat`:

```
use bigrat qw/hex oct/;

print hex("0x1234567890123456");
```

(continues on next page)

(continued from previous page)

```
{
    no bigrat;
    print hex("0x1234567890123456");
}
```

The second call to hex() will warn about a non-portable constant.

Compare this to:

```
use bigrat;

# will warn only under Perl older than v5.9.4
print hex("0x1234567890123456");
```

EXAMPLES

```
perl -Mbigrat -le 'print sqrt(33)'
perl -Mbigrat -le 'print 2**255'
perl -Mbigrat -le 'print 4.5+2**255'
perl -Mbigrat -le 'print 3/7 + 5/7 + 8/3'
perl -Mbigrat -le 'print 12->is_odd()';
perl -Mbigrat=l,GMP -le 'print 7 ** 7777'
```

BUGS

Please report any bugs or feature requests to `bug-bignum at rt.cpan.org`, or through the web interface at https://rt.cpan.org/Ticket/Create.html?Queue=bignum (requires login). We will be notified, and then you'll automatically be notified of progress on your bug as I make changes.

SUPPORT

You can find documentation for this module with the perldoc command.

```
perldoc bigrat
```

You can also look for information at:

- GitHub

  https://github.com/pjacklam/p5-bignum

- RT: CPAN's request tracker

  https://rt.cpan.org/Dist/Display.html?Name=bignum

- MetaCPAN

  https://metacpan.org/release/bignum

- CPAN Testers Matrix

  http://matrix.cpantesters.org/?dist=bignum

LICENSE

This program is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

SEE ALSO

bignum and *bigint*.

Math::BigInt, *Math::BigFloat*, *Math::BigRat* and Math::Big as well as *Math::BigInt::FastCalc*, Math::BigInt::Pari and Math::BigInt::GMP.

AUTHORS

- (C) by Tels http://bloodgate.com/ in early 2002 - 2007.
- Maintained by Peter John Acklam pjacklam@gmail.com, 2014-.

5.2.125 blib

Use MakeMaker's uninstalled version of a package

SYNOPSIS

```
perl -Mblib script [args...]

perl -Mblib=dir script [args...]
```

DESCRIPTION

Looks for MakeMaker-like *'blib'* directory structure starting in *dir* (or current directory) and working back up to five levels of '..'.

Intended for use on command line with **-M** option as a way of testing arbitrary scripts against an uninstalled version of a package.

However it is possible to :

```
use blib;
or
use blib '..';
```

etc. if you really must.

BUGS

Pollutes global name space for development only task.

AUTHOR

Nick Ing-Simmons nik@tiuk.ti.com

5.2.126 builtin

Perl pragma to import built-in utility functions

SYNOPSIS

```
use builtin qw(
    true false is_bool
    inf nan
    weaken unweaken is_weak
    blessed refaddr reftype
    created_as_string created_as_number
    stringify
    ceil floor
    indexed
    trim
    is_tainted
    export_lexically
    load_module
);

use builtin ':5.40';  # most of the above
```

DESCRIPTION

Perl provides several utility functions in the `builtin` package. These are plain functions, and look and behave just like regular user-defined functions do. They do not provide new syntax or require special parsing. These functions are always present in the interpreter and can be called at any time by their fully-qualified names. By default they are not available as short names, but can be requested for convenience.

Individual named functions can be imported by listing them as import parameters on the `use` statement for this pragma.

The builtin::compat module from CPAN provides versions of many of these functions that can be used on Perl versions where `builtin` or specific functions are not yet available.

Warning: At present, many of the functions in the `builtin` namespace are experimental. Calling them will trigger warnings of the `experimental::builtin` category.

Lexical Import

This pragma module creates *lexical* aliases in the currently-compiling scope to these builtin functions. This is similar to the lexical effect of other pragmas such as *strict* and *feature*.

```
sub classify
{
    my $val = shift;

    use builtin 'is_bool';
    return is_bool($val) ? "boolean" : "not a boolean";
}

# the is_bool() function is no longer visible here
# but may still be called by builtin::is_bool()
```

Because these functions are imported lexically, rather than by package symbols, the user does not need to take any special measures to ensure they don't accidentally appear as object methods from a class.

```
package An::Object::Class {
    use builtin 'true', 'false';
    ...
}

# does not appear as a method
An::Object::Class->true;

# Can't locate object method "true" via package "An::Object::Class"
#   at ...
```

Once imported, a lexical function is much like any other lexical symbol (such as a variable) in that it cannot be removed again. If you wish to limit the visiblity of an imported `builtin` function, put it inside its own scope:

```
{
  use builtin 'refaddr';
  ...
}
```

Version Bundles

The entire set of builtin functions that were considered non-experimental by a version of perl can be imported all at once, by requesting a version bundle. This is done by giving the perl release version (without its subversion suffix) after a colon character:

```
use builtin ':5.40';
```

The following bundles currently exist:

```
Version    Includes
-------    --------

:5.40      true false weaken unweaken is_weak blessed refaddr
 ↪reftype
           ceil floor is_tainted trim indexed
```

Function Optimisations

There are a number of optimisations that apply to functions in the *builtin* package. If you replace or override these functions (such as by assignment into glob references) the optimisations may not take effect. Do so with caution.

FUNCTIONS

true

```
$val = true;
```

Returns the boolean truth value. While any scalar value can be tested for truth and most defined, non-empty and non-zero values are considered "true" by perl, this one is special in that "is_bool" considers it to be a distinguished boolean value.

This gives an equivalent value to expressions like `!!1` or `!0`.

Available starting with Perl 5.36. Since Perl 5.40, it is no longer experimental and it is included in the 5.40 and higher builtin version bundles.

false

```
$val = false;
```

Returns the boolean false value. While any non-true scalar value is considered "false" by perl, this one is special in that "is_bool" considers it to be a distinguished boolean value.

This gives an equivalent value to expressions like `!!0` or `!1`.

Available starting with Perl 5.36. Since Perl 5.40, it is no longer experimental and it is included in the 5.40 and higher builtin version bundles.

is_bool

```
$bool = is_bool($val);
```

This function is currently **experimental**.

Returns true when given a distinguished boolean value, or false if not. A distinguished boolean value is the result of any boolean-returning builtin function (such as `true` or `is_bool` itself), boolean-returning operator (such as the `eq` or `==` comparison tests or the `!` negation operator), or any variable containing one of these results.

This function used to be named `isbool`. A compatibility alias is provided currently but will be removed in a later version.

Available starting with Perl 5.36.

inf

```
$num = inf;
```

This function is currently **experimental**.

Returns the floating-point infinity value. If the underlying numeric C type does not support such a value, it throws a runtime error instead.

Available starting with Perl 5.40.

nan

```
$num = nan;
```

This function is currently **experimental**.

Returns the floating-point "Not-a-Number" value. If the underlying numeric C type does not support such a value, it throws a runtime error instead.

Available starting with Perl 5.40.

weaken

```
weaken($ref);
```

Weakens a reference. A weakened reference does not contribute to the reference count of its referent. If only weakened references to a referent remain, it will be disposed of, and all remaining weak references to it will have their value set to `undef`.

Available starting with Perl 5.36. Since Perl 5.40, it is no longer experimental and it is included in the 5.40 and higher builtin version bundles.

unweaken

```
unweaken($ref);
```

Strengthens a reference, undoing the effects of a previous call to "weaken".

Available starting with Perl 5.36. Since Perl 5.40, it is no longer experimental and it is included in the 5.40 and higher builtin version bundles.

is_weak

```
$bool = is_weak($ref);
```

Returns true when given a weakened reference, or false if not a reference or not weak.

This function used to be named `isweak`. A compatibility alias is provided currently but will be removed in a later version.

Available starting with Perl 5.36. Since Perl 5.40, it is no longer experimental and it is included in the 5.40 and higher builtin version bundles.

blessed

```
$str = blessed($ref);
```

Returns the package name for an object reference, or `undef` for a non-reference or reference that is not an object.

Available starting with Perl 5.36. Since Perl 5.40, it is no longer experimental and it is included in the 5.40 and higher builtin version bundles.

refaddr

```
$num = refaddr($ref);
```

Returns the memory address for a reference, or `undef` for a non-reference. This value is not likely to be very useful for pure Perl code, but is handy as a means to test for referential identity or uniqueness.

Available starting with Perl 5.36. Since Perl 5.40, it is no longer experimental and it is included in the 5.40 and higher builtin version bundles.

reftype

```
$str = reftype($ref);
```

Returns the basic container type of the referent of a reference, or `undef` for a non-reference. This is returned as a string in all-capitals, such as `ARRAY` for array references, or `HASH` for hash references.

Available starting with Perl 5.36. Since Perl 5.40, it is no longer experimental and it is included in the 5.40 and higher builtin version bundles.

created_as_string

```
$bool = created_as_string($val);
```

This function is currently **experimental**.

Returns a boolean representing if the argument value was originally created as a string. It will return true for any scalar expression whose most recent assignment or modification was of a string-like nature - such as assignment from a string literal, or the result of a string operation such as concatenation or regexp. It will return false for references (including any object), numbers, booleans and undef.

It is unlikely that you will want to use this for regular data validation within Perl, as it will not return true for regular numbers that are still perfectly usable as strings, nor for any object reference - especially objects that overload the stringification operator in an attempt to behave more like strings. For example

```
my $val = URI->new( "https://metacpan.org/" );

if( created_as_string $val ) { ... }    # this will not execute
```

Available starting with Perl 5.36.

created_as_number

```
$bool = created_as_number($val);
```

This function is currently **experimental**.

Returns a boolean representing if the argument value was originally created as a number. It will return true for any scalar expression whose most recent assignment or modification was of a numerical nature - such as assignment from a number literal, or the result of a numerical operation such as addition. It will return false for references (including any object), strings, booleans and undef.

It is unlikely that you will want to use this for regular data validation within Perl, as it will not return true for regular strings of decimal digits that are still perfectly usable as numbers, nor for any object reference - especially objects that overload the numification operator in an attempt to behave more like numbers. For example

```
my $val = Math::BigInt->new( 123 );

if( created_as_number $val ) { ... }    # this will not execute
```

While most Perl code should operate on scalar values without needing to know their creation history, these two functions are intended to be used by data serialisation modules such as JSON encoders or similar situations, where language interoperability concerns require making a distinction between values that are fundamentally stringlike versus numberlike in nature.

Available starting with Perl 5.36.

stringify

```
$str = stringify($val);
```

This function is currently **experimental**.

Returns a new plain perl string that represents the given argument.

When given a value that is already a string, a copy of this value is returned unchanged. False booleans are treated like the empty string.

Numbers are turned into a decimal representation. True booleans are treated like the number 1.

References to objects in classes that have *overload* and define the "" overload entry will use the delegated method to provide a value here.

Non-object references, or references to objects in classes without a "" overload will return a string that names the underlying container type of the reference, its memory address, and possibly its class name if it is an object.

Available starting with Perl 5.40.

ceil

```
$num = ceil($num);
```

Returns the smallest integer value greater than or equal to the given numerical argument.

Available starting with Perl 5.36. Since Perl 5.40, it is no longer experimental and it is included in the 5.40 and higher builtin version bundles.

floor

```
$num = floor($num);
```

Returns the largest integer value less than or equal to the given numerical argument.

Available starting with Perl 5.36. Since Perl 5.40, it is no longer experimental and it is included in the 5.40 and higher builtin version bundles.

indexed

```
@ivpairs = indexed(@items)
```

Returns an even-sized list of number/value pairs, where each pair is formed of a number giving an index in the original list followed by the value at that position in it. I.e. returns a list twice the size of the original, being equal to

```
(0, $items[0], 1, $items[1], 2, $items[2], ...)
```

Note that unlike the core `values` function, this function returns copies of its original arguments, not aliases to them. Any modifications of these copies are *not* reflected in modifications to the original.

```
my @x = ...;
$_++ for indexed @x;  # The @x array remains unaffected
```

This function is primarily intended to be useful combined with multi-variable `foreach` loop syntax; as

```
foreach my ($index, $value) (indexed LIST) {
    ...
}
```

In scalar context this function returns the size of the list that it would otherwise have returned, and provokes a warning in the `scalar` category.

Available starting with Perl 5.36. Since Perl 5.40, it is no longer experimental and it is included in the 5.40 and higher builtin version bundles.

trim

```
$stripped = trim($string);
```

Returns the input string with whitespace stripped from the beginning and end. trim() will remove these characters:

" ", an ordinary space.

"\t", a tab.

"\n", a new line (line feed).

"\r", a carriage return.

and all other Unicode characters that are flagged as whitespace. A complete list is in "Whitespace" in perlrecharclass.

```
$var = "  Hello world   ";            # "Hello world"
$var = "\t\t\tHello world";           # "Hello world"
$var = "Hello world\n";               # "Hello world"
$var = "\x{2028}Hello world\x{3000}"; # "Hello world"
```

`trim` is equivalent to:

```
my $trimmed = $str =~ s/\A\s+//ur =~ s/\s+\z//ur;
```

Available starting with Perl 5.36. Since Perl 5.40, it is no longer experimental and it is included in the 5.40 and higher builtin version bundles.

is_tainted

```
$bool = is_tainted($var);
```

Returns true when given a tainted variable.

Available starting with Perl 5.38.

export_lexically

```
export_lexically($name1, $ref1, $name2, $ref2, ...)
```

This function is currently **experimental**.

Exports new lexical names into the scope currently being compiled. Names given by the first of each pair of values will refer to the corresponding item whose reference is given by the second. Types of item that are permitted are subroutines, and scalar, array, and hash variables. If the item is a subroutine, the name may optionally be prefixed with the & sigil, but for convenience it doesn't have to. For items that are variables the sigil is required, and must match the type of the variable.

```
export_lexically func    => \&func,
                 '&func' => \&func;  # same as above
```

(continues on next page)

(continued from previous page)

```
export_lexically '$scalar' => \my $var;
```

```
# The following are not permitted
export_lexically '$var' => \@arr;   # sigil does not match
export_lexically name => \$scalar;  # implied '&' sigil does not
↪match

export_lexically '*name' => \*globref;  # globrefs are not supported
```

This must be called at compile time; which typically means during a `BEGIN` block. Usually this would be used as part of an `import` method of a module, when invoked as part of a `use ...` statement.

Available starting with Perl 5.38.

load_module

```
load_module($module_name);
```

This function is currently **experimental**.

Loads a named module from the inclusion paths (`@INC`). `$module_name` must be a string that provides a module name. It cannot be omitted, and providing an invalid module name will result in an exception. Not providing any argument results in a compilation error. Returns the loaded module's name on success.

The effect of `load_module`-ing a module is mostly the same as `require`-ing, down to the same error conditions when the module does not exist, does not compile, or does not evaluate to a true value. See also *the* `module_true` *feature*.

`load_module` can't be used to require a particular version of Perl, nor can it be given a bareword module name as an argument.

Available starting with Perl 5.40.

SEE ALSO

perlop, *perlfunc*, *Scalar::Util*, builtin::compat

5.2.127 bytes

Perl pragma to expose the individual bytes of characters

NOTICE

Because the bytes pragma breaks encapsulation (i.e. it exposes the innards of how the perl executable currently happens to store a string), the byte values that result are in an unspecified encoding.

Use of this module for anything other than debugging purposes is strongly discouraged. If you feel that the functions here within might be useful for your application, this

possibly indicates a mismatch between your mental model of Perl Unicode and the current reality. In that case, you may wish to read some of the perl Unicode documentation: perluniintro, perlunitut, perlunifaq and perlunicode.

SYNOPSIS

```
use bytes;
... chr(...);       # or bytes::chr
... index(...);     # or bytes::index
... length(...);    # or bytes::length
... ord(...);       # or bytes::ord
... rindex(...);    # or bytes::rindex
... substr(...);    # or bytes::substr
no bytes;
```

DESCRIPTION

Perl's characters are stored internally as sequences of one or more bytes. This pragma allows for the examination of the individual bytes that together comprise a character.

Originally the pragma was designed for the loftier goal of helping incorporate Unicode into Perl, but the approach that used it was found to be defective, and the one remaining legitimate use is for debugging when you need to non-destructively examine characters' individual bytes. Just insert this pragma temporarily, and remove it after the debugging is finished.

The original usage can be accomplished by explicit (rather than this pragma's implicit) encoding using the *Encode* module:

```
use Encode qw/encode/;

my $utf8_byte_string   = encode "UTF8",   $string;
my $latin1_byte_string = encode "Latin1", $string;
```

Or, if performance is needed and you are only interested in the UTF-8 representation:

```
utf8::encode(my $utf8_byte_string = $string);
```

`no bytes` can be used to reverse the effect of `use bytes` within the current lexical scope.

As an example, when Perl sees `$x = chr(400)`, it encodes the character in UTF-8 and stores it in `$x`. Then it is marked as character data, so, for instance, `length $x` returns 1. However, in the scope of the `bytes` pragma, `$x` is treated as a series of bytes - the bytes that make up the UTF8 encoding - and `length $x` returns 2:

```
$x = chr(400);
print "Length is ", length $x, "\n";     # "Length is 1"
printf "Contents are %vd\n", $x;         # "Contents are 400"
{
    use bytes; # or "require bytes; bytes::length()"
    print "Length is ", length $x, "\n"; # "Length is 2"
```

(continues on next page)

(continued from previous page)

```
    printf "Contents are %vd\n", $x;     # "Contents are 198.144 (on
                                         # ASCII platforms)"
}
```

`chr()`, `ord()`, `substr()`, `index()` and `rindex()` behave similarly.

For more on the implications, see perluniintro and perlunicode.

`bytes::length()` is admittedly handy if you need to know the **byte length** of a Perl scalar. But a more modern way is:

```
use Encode 'encode';
length(encode('UTF-8', $scalar))
```

LIMITATIONS

`bytes::substr()` does not work as an *lvalue()*.

SEE ALSO

perluniintro, perlunicode, *utf8*, *Encode*

5.2.128 charnames

access to Unicode character names and named character sequences; also define character names

SYNOPSIS

```
use charnames ':full';
print "\N{GREEK SMALL LETTER SIGMA} is called sigma.\n";
print "\N{LATIN CAPITAL LETTER E WITH VERTICAL LINE BELOW}",
      " is an officially named sequence of two Unicode characters\n
 ↪";

use charnames ':loose';
print "\N{Greek small-letter  sigma}",
       "can be used to ignore case, underscores, most blanks,"
       "and when you aren't sure if the official name has hyphens\n
 ↪";

use charnames ':short';
print "\N{greek:Sigma} is an upper-case sigma.\n";

use charnames qw(cyrillic greek);
print "\N{sigma} is Greek sigma, and \N{be} is Cyrillic b.\n";

use utf8;
use charnames ":full", ":alias" => {
```

(continues on next page)

(continued from previous page)

```
  e_ACUTE => "LATIN SMALL LETTER E WITH ACUTE",
  mychar => 0xE8000,  # Private use area
  "自転車に乗る人" => "BICYCLIST"
};
print "\N{e_ACUTE} is a small letter e with an acute.\n";
print "\N{mychar} allows me to name private use characters.\n";
print "And I can create synonyms in other languages,",
      " such as \N{自転車に乗る人} for "BICYCLIST (U+1F6B4)\n";

use charnames ();
print charnames::viacode(0x1234); # prints "ETHIOPIC SYLLABLE SEE"
printf "%04X", charnames::vianame("GOTHIC LETTER AHSA"); # prints
                                                         # "10330"
print charnames::vianame("LATIN CAPITAL LETTER A"); # prints 65 on
                                                    # ASCII␣
 ↪platforms;
                                                    # 193 on EBCDIC
print charnames::string_vianame("LATIN CAPITAL LETTER A"); # prints
 ↪"A"
```

DESCRIPTION

Pragma `use charnames` is used to gain access to the names of the Unicode characters and named character sequences, and to allow you to define your own character and character sequence names.

All forms of the pragma enable use of the following 3 functions:

- "charnames::string_vianame(*name*)" for run-time lookup of a either a character name or a named character sequence, returning its string representation
- "charnames::vianame(*name*)" for run-time lookup of a character name (but not a named character sequence) to get its ordinal value (code point)
- "charnames::viacode(*code*)" for run-time lookup of a code point to get its Unicode name.

Starting in Perl v5.16, any occurrence of `\N{_CHARNAME_}` sequences in a double-quotish string automatically loads this module with arguments `:full` and `:short` (described below) if it hasn't already been loaded with different arguments, in order to compile the named Unicode character into position in the string. Prior to v5.16, an explicit `use charnames` was required to enable this usage. (However, prior to v5.16, the form `"use charnames ();"` did not enable `\N{_CHARNAME_}`.)

Note that `\N{U+_..._}`, where the ... is a hexadecimal number, also inserts a character into a string. The character it inserts is the one whose Unicode code point (ordinal value) is equal to the number. For example, `"\N{U+263a}"` is the Unicode (white background, black foreground) smiley face equivalent to `"\N{WHITE SMILING FACE}"`. Also note, `\N{_..._}` can mean a regex quantifier instead of a character name, when the ... is a number (or comma separated pair of numbers (see "QUANTIFIERS" in perlreref), and is not related to this pragma.

The charnames pragma supports arguments :full, :loose, :short, script names and customized aliases.

If :full is present, for expansion of \N{_CHARNAME_}, the string *CHARNAME* is first looked up in the list of standard Unicode character names.

:loose is a variant of :full which allows *CHARNAME* to be less precisely specified. Details are in "LOOSE MATCHES".

If :short is present, and *CHARNAME* has the form _SCRIPT_:_CNAME_, then *CNAME* is looked up as a letter in script *SCRIPT*, as described in the next paragraph. Or, if use charnames is used with script name arguments, then for \N{_CHARNAME_} the name *CHARNAME* is looked up as a letter in the given scripts (in the specified order). Customized aliases can override these, and are explained in "CUSTOM ALIASES".

For lookup of *CHARNAME* inside a given script *SCRIPTNAME*, this pragma looks in the table of standard Unicode names for the names

```
SCRIPTNAME CAPITAL LETTER CHARNAME
SCRIPTNAME SMALL LETTER CHARNAME
SCRIPTNAME LETTER CHARNAME
```

If *CHARNAME* is all lowercase, then the CAPITAL variant is ignored, otherwise the SMALL variant is ignored, and both *CHARNAME* and *SCRIPTNAME* are converted to all uppercase for look-up. Other than that, both of them follow loose rules if :loose is also specified; strict otherwise.

Note that \N{...} is compile-time; it's a special form of string constant used inside double-quotish strings; this means that you cannot use variables inside the \N{...}. If you want similar run-time functionality, use charnames::string_vianame().

Note, starting in Perl 5.18, the name BELL refers to the Unicode character U+1F514, instead of the traditional U+0007. For the latter, use ALERT or BEL.

It is a syntax error to use \N{NAME} where NAME is unknown.

For \N{NAME}, it is a fatal error if use bytes is in effect and the input name is that of a character that won't fit into a byte (i.e., whose ordinal is above 255).

Otherwise, any string that includes a \N{_charname_} or \N{U+_code point_} will automatically have Unicode rules (see "Byte and Character Semantics" in perlunicode).

LOOSE MATCHES

By specifying :loose, Unicode's loose character name matching rules are selected instead of the strict exact match used otherwise. That means that *CHARNAME* doesn't have to be so precisely specified. Upper/lower case doesn't matter (except with scripts as mentioned above), nor do any underscores, and the only hyphens that matter are those at the beginning or end of a word in the name (with one exception: the hyphen in U+1180 HANGUL JUNGSEONG O-E does matter). Also, blanks not adjacent to hyphens don't matter. The official Unicode names are quite variable as to where they use hyphens versus spaces to separate word-like units, and this option allows you to not have to care as much. The reason non-medial hyphens matter is because of cases like U+0F60 TIBETAN LETTER -A versus U+0F68 TIBETAN LETTER A. The hyphen here is significant, as is the space before it, and so both must be included.

`:loose` slows down look-ups by a factor of 2 to 3 versus `:full`, but the trade-off may be worth it to you. Each individual look-up takes very little time, and the results are cached, so the speed difference would become a factor only in programs that do look-ups of many different spellings, and probably only when those look-ups are through `vianame()` and `string_vianame()`, since `\N{...}` look-ups are done at compile time.

ALIASES

Starting in Unicode 6.1 and Perl v5.16, Unicode defines many abbreviations and names that were formerly Perl extensions, and some additional ones that Perl did not previously accept. The list is getting too long to reproduce here, but you can get the complete list from the Unicode web site: http://www.unicode.org/Public/UNIDATA/NameAliases.txt.

Earlier versions of Perl accepted almost all the 6.1 names. These were most extensively documented in the v5.14 version of this pod: http://perldoc.perl.org/5.14.0/charnames.html#ALIASES.

CUSTOM ALIASES

You can add customized aliases to standard (`:full`) Unicode naming conventions. The aliases override any standard definitions, so, if you're twisted enough, you can change `"\N{LATIN CAPITAL LETTER A}"` to mean `"B"`, etc.

Aliases must begin with a character that is alphabetic. After that, each may contain any combination of word (`\w`) characters, SPACE (U+0020), HYPHEN-MINUS (U+002D), LEFT PARENTHESIS (U+0028), and RIGHT PARENTHESIS (U+0029). These last two should never have been allowed in names, and are retained for backwards compatibility only, and may be deprecated and removed in future releases of Perl, so don't use them for new names. (More precisely, the first character of a name you specify must be something that matches all of `\p{ID_Start}`, `\p{Alphabetic}`, and `\p{Gc=Letter}`. This makes sure it is what any reasonable person would view as an alphabetic character. And, the continuation characters that match `\w` must also match `\p{ID_Continue}`.) Starting with Perl v5.18, any Unicode characters meeting the above criteria may be used; prior to that only Latin1-range characters were acceptable.

An alias can map to either an official Unicode character name (not a loose matched name) or to a numeric code point (ordinal). The latter is useful for assigning names to code points in Unicode private use areas such as U+E800 through U+F8FF. A numeric code point must be a non-negative integer, or a string beginning with `"U+"` or `"0x"` with the remainder considered to be a hexadecimal integer. A literal numeric constant must be unsigned; it will be interpreted as hex if it has a leading zero or contains non-decimal hex digits; otherwise it will be interpreted as decimal. If it begins with `"U+"`, it is interpreted as the Unicode code point; otherwise it is interpreted as native. (Only code points below 256 can differ between Unicode and native.) Thus `U+41` is always the Latin letter "A"; but `0x41` can be "NO-BREAK SPACE" on EBCDIC platforms.

Aliases are added either by the use of anonymous hashes:

```
use charnames ":alias" => {
    e_ACUTE => "LATIN SMALL LETTER E WITH ACUTE",
    mychar1 => 0xE8000,
```

(continues on next page)

(continued from previous page)

```
    };
my $str = "\N{e_ACUTE}";
```

or by using a file containing aliases:

```
use charnames ":alias" => "pro";
```

This will try to read "unicore/pro_alias.pl" from the @INC path. This file should return a list in plain perl:

```
(
A_GRAVE         => "LATIN CAPITAL LETTER A WITH GRAVE",
A_CIRCUM        => "LATIN CAPITAL LETTER A WITH CIRCUMFLEX",
A_DIAERES       => "LATIN CAPITAL LETTER A WITH DIAERESIS",
A_TILDE         => "LATIN CAPITAL LETTER A WITH TILDE",
A_BREVE         => "LATIN CAPITAL LETTER A WITH BREVE",
A_RING          => "LATIN CAPITAL LETTER A WITH RING ABOVE",
A_MACRON        => "LATIN CAPITAL LETTER A WITH MACRON",
mychar2         => "U+E8001",
);
```

Both these methods insert ":full" automatically as the first argument (if no other argument is given), and you can give the ":full" explicitly as well, like

```
use charnames ":full", ":alias" => "pro";
```

":loose" has no effect with these. Input names must match exactly, using ":full" rules.

Also, both these methods currently allow only single characters to be named. To name a sequence of characters, use a custom translator (described below).

charnames::string_vianame(*name*)

This is a runtime equivalent to \N{...}. *name* can be any expression that evaluates to a name accepted by \N{...} under the :full option to charnames. In addition, any other options for the controlling "use charnames" in the same scope apply, like :loose or any script list, :short option, or custom aliases you may have defined.

The only differences are due to the fact that string_vianame is run-time and \N{} is compile time. You can't interpolate inside a \N{}, (so \N{$variable} doesn't work); and if the input name is unknown, string_vianame returns undef instead of it being a syntax error.

charnames::vianame(*name*)

This is similar to string_vianame. The main difference is that under most circumstances, vianame returns an ordinal code point, whereas string_vianame returns a string. For example,

```
printf "U+%04X", charnames::vianame("FOUR TEARDROP-SPOKED ASTERISK
↪");
```

prints “U+2722”.

This leads to the other two differences. Since a single code point is returned, the function can’t handle named character sequences, as these are composed of multiple characters (it returns `undef` for these. And, the code point can be that of any character, even ones that aren’t legal under the `use bytes` pragma,

See “BUGS” for the circumstances in which the behavior differs from that described above.

charnames::viacode(*code*)

Returns the full name of the character indicated by the numeric code. For example,

```
print charnames::viacode(0x2722);
```

prints “FOUR TEARDROP-SPOKED ASTERISK”.

The name returned is the “best” (defined below) official name or alias for the code point, if available; otherwise your custom alias for it, if defined; otherwise `undef`. This means that your alias will only be returned for code points that don’t have an official Unicode name (nor alias) such as private use code points.

If you define more than one name for the code point, it is indeterminate which one will be returned.

As mentioned, the function returns `undef` if no name is known for the code point. In Unicode the proper name for these is the empty string, which `undef` stringifies to. (If you ask for a code point past the legal Unicode maximum of U+10FFFF that you haven’t assigned an alias to, you get `undef` plus a warning.)

The input number must be a non-negative integer, or a string beginning with `"U+"` or `"0x"` with the remainder considered to be a hexadecimal integer. A literal numeric constant must be unsigned; it will be interpreted as hex if it has a leading zero or contains non-decimal hex digits; otherwise it will be interpreted as decimal. If it begins with `"U+"`, it is interpreted as the Unicode code point; otherwise it is interpreted as native. (Only code points below 256 can differ between Unicode and native.) Thus `U+41` is always the Latin letter “A”; but `0x41` can be “NO-BREAK SPACE” on EBCDIC platforms.

As mentioned above under “ALIASES”, Unicode 6.1 defines extra names (synonyms or aliases) for some code points, most of which were already available as Perl extensions. All these are accepted by `\N{...}` and the other functions in this module, but `viacode` has to choose which one name to return for a given input code point, so it returns the “best” name. To understand how this works, it is helpful to know more about the Unicode name properties. All code points actually have only a single name, which (starting in Unicode 2.0) can never change once a character has been assigned to the code point. But mistakes have been made in assigning names, for example sometimes a clerical error was made during the publishing of the Standard which caused words to be misspelled, and there was no way to correct those. The Name_Alias property was eventually created to handle these situations. If a name was wrong, a corrected synonym would be published for it, using Name_Alias. `viacode` will return that corrected synonym as the “best” name for a code point. (It is even possible, though it hasn’t happened yet, that the correction itself will need to be corrected, and so another Name_Alias can be created for that code point; `viacode` will return the most recent correction.)

The Unicode name for each of the control characters (such as LINE FEED) is the empty string. However almost all had names assigned by other standards, such as the ASCII Standard, or were in common use. `viacode` returns these names as the "best" ones available. Unicode 6.1 has created Name_Aliases for each of them, including alternate names, like NEW LINE. `viacode` uses the original name, "LINE FEED" in preference to the alternate. Similarly the name returned for U+FEFF is "ZERO WIDTH NO-BREAK SPACE", not "BYTE ORDER MARK".

Until Unicode 6.1, the 4 control characters U+0080, U+0081, U+0084, and U+0099 did not have names nor aliases. To preserve backwards compatibility, any alias you define for these code points will be returned by this function, in preference to the official name.

Some code points also have abbreviated names, such as "LF" or "NL". `viacode` never returns these.

Because a name correction may be added in future Unicode releases, the name that `viacode` returns may change as a result. This is a rare event, but it does happen.

CUSTOM TRANSLATORS

The mechanism of translation of `\N{...}` escapes is general and not hardwired into `charnames.pm`. A module can install custom translations (inside the scope which `uses` the module) with the following magic incantation:

```
sub import {
    shift;
    $^H{charnames} = \&translator;
}
```

Here translator() is a subroutine which takes *CHARNAME* as an argument, and returns text to insert into the string instead of the `\N{_CHARNAME_}` escape.

This is the only way you can create a custom named sequence of code points.

Since the text to insert should be different in `bytes` mode and out of it, the function should check the current state of `bytes`-flag as in:

```
use bytes ();                       # for $bytes::hint_bits
sub translator {
    if ($^H & $bytes::hint_bits) {
        return bytes_translator(@_);
    }
    else {
        return utf8_translator(@_);
    }
}
```

See "CUSTOM ALIASES" above for restrictions on *CHARNAME*.

Of course, `vianame`, `viacode`, and `string_vianame` would need to be overridden as well.

BUGS

vianame() normally returns an ordinal code point, but when the input name is of the form `U+...`, it returns a chr instead. In this case, if `use bytes` is in effect and the character won't fit into a byte, it returns `undef` and raises a warning.

Since evaluation of the translation function (see "CUSTOM TRANSLATORS") happens in the middle of compilation (of a string literal), the translation function should not do any `evals` or `requires`. This restriction should be lifted (but is low priority) in a future version of Perl.

5.2.129 constant

Perl pragma to declare constants

SYNOPSIS

```
use constant PI    => 4 * atan2(1, 1);
use constant DEBUG => 0;

print "Pi equals ", PI, "...\n" if DEBUG;

use constant {
    SEC   => 0,
    MIN   => 1,
    HOUR  => 2,
    MDAY  => 3,
    MON   => 4,
    YEAR  => 5,
    WDAY  => 6,
    YDAY  => 7,
    ISDST => 8,
};

use constant WEEKDAYS => qw(
    Sunday Monday Tuesday Wednesday Thursday Friday Saturday
);

print "Today is ", (WEEKDAYS)[ (localtime)[WDAY] ], ".\n";
```

DESCRIPTION

This pragma allows you to declare constants at compile-time.

When you declare a constant such as `PI` using the method shown above, each machine your script runs upon can have as many digits of accuracy as it can use. Also, your program will be easier to read, more likely to be maintained (and maintained correctly), and far less likely to send a space probe to the wrong planet because nobody noticed the one equation in which you wrote `3.14195`.

When a constant is used in an expression, Perl replaces it with its value at compile time,

and may then optimize the expression further. In particular, any code in an `if (CONSTANT)` block will be optimized away if the constant is false.

NOTES

As with all `use` directives, defining a constant happens at compile time. Thus, it's probably not correct to put a constant declaration inside of a conditional statement (like `if ($foo) { use constant ... }`).

Constants defined using this module cannot be interpolated into strings like variables. However, concatenation works just fine:

```
print "Pi equals PI...\n";        # WRONG: does not expand "PI"
print "Pi equals ".PI."...\n";    # right
```

Even though a reference may be declared as a constant, the reference may point to data which may be changed, as this code shows.

```
use constant ARRAY => [ 1,2,3,4 ];
print ARRAY->[1];
ARRAY->[1] = " be changed";
print ARRAY->[1];
```

Constants belong to the package they are defined in. To refer to a constant defined in another package, specify the full package name, as in `Some::Package::CONSTANT`. Constants may be exported by modules, and may also be called as either class or instance methods, that is, as `Some::Package->CONSTANT` or as `$obj->CONSTANT` where `$obj` is an instance of `Some::Package`. Subclasses may define their own constants to override those in their base class.

As of version 1.32 of this module, constants can be defined in packages other than the caller, by including the package name in the name of the constant:

```
use constant "OtherPackage::FWIBBLE" => 7865;
constant->import("Other::FWOBBLE",$value); # dynamically at run time
```

The use of all caps for constant names is merely a convention, although it is recommended in order to make constants stand out and to help avoid collisions with other barewords, keywords, and subroutine names. Constant names must begin with a letter or underscore. Names beginning with a double underscore are reserved. Some poor choices for names will generate warnings, if warnings are enabled at compile time.

List constants

Constants may be lists of more (or less) than one value. A constant with no values evaluates to `undef` in scalar context. Note that constants with more than one value do *not* return their last value in scalar context as one might expect. They currently return the number of values, but **this may change in the future**. Do not use constants with multiple values in scalar context.

NOTE: This implies that the expression defining the value of a constant is evaluated in list context. This may produce surprises:

```
use constant TIMESTAMP => localtime;                # WRONG!
use constant TIMESTAMP => scalar localtime;         # right
```

The first line above defines `TIMESTAMP` as a 9-element list, as returned by `localtime()` in list context. To set it to the string returned by `localtime()` in scalar context, an explicit `scalar` keyword is required.

List constants are lists, not arrays. To index or slice them, they must be placed in parentheses.

```
my @workdays = WEEKDAYS[1 .. 5];            # WRONG!
my @workdays = (WEEKDAYS)[1 .. 5];          # right
```

Defining multiple constants at once

Instead of writing multiple `use constant` statements, you may define multiple constants in a single statement by giving, instead of the constant name, a reference to a hash where the keys are the names of the constants to be defined. Obviously, all constants defined using this method must have a single value.

```
use constant {
    FOO => "A single value",
    BAR => "This", "won't", "work!",        # Error!
};
```

This is a fundamental limitation of the way hashes are constructed in Perl. The error messages produced when this happens will often be quite cryptic – in the worst case there may be none at all, and you'll only later find that something is broken.

When defining multiple constants, you cannot use the values of other constants defined in the same declaration. This is because the calling package doesn't know about any constant within that group until *after* the `use` statement is finished.

```
use constant {
    BITMASK => 0xAFBAEBA8,
    NEGMASK => ~BITMASK,                    # Error!
};
```

Magic constants

Magical values and references can be made into constants at compile time, allowing for way cool stuff like this. (These error numbers aren't totally portable, alas.)

```
use constant E2BIG => ($! = 7);
print   E2BIG, "\n";        # something like "Arg list too long"
print 0+E2BIG, "\n";        # "7"
```

You can't produce a tied constant by giving a tied scalar as the value. References to tied variables, however, can be used as constants without any problems.

TECHNICAL NOTES

In the current implementation, scalar constants are actually inlinable subroutines. As of version 5.004 of Perl, the appropriate scalar constant is inserted directly in place of some subroutine calls, thereby saving the overhead of a subroutine call. See *"Constant Functions" in perlsub* for details about how and when this happens.

In the rare case in which you need to discover at run time whether a particular constant has been declared via this module, you may use this function to examine the hash `%constant::declared`. If the given constant name does not include a package name, the current package is used.

```
sub declared ($) {
    use constant 1.01;              # don't omit this!
    my $name = shift;
    $name =~ s/^::/main::/;
    my $pkg = caller;
    my $full_name = $name =~ /::/ ? $name : "${pkg}::$name";
    $constant::declared{$full_name};
}
```

CAVEATS

List constants are not inlined unless you are using Perl v5.20 or higher. In v5.20 or higher, they are still not read-only, but that may change in future versions.

It is not possible to have a subroutine or a keyword with the same name as a constant in the same package. This is probably a Good Thing.

A constant with a name in the list `STDIN STDOUT STDERR ARGV ARGVOUT ENV INC SIG` is not allowed anywhere but in package `main::`, for technical reasons.

Unlike constants in some languages, these cannot be overridden on the command line or via environment variables.

You can get into trouble if you use constants in a context which automatically quotes barewords (as is true for any subroutine call). For example, you can't say `$hash{CONSTANT}` because `CONSTANT` will be interpreted as a string. Use `$hash{CONSTANT()}` or `$hash{+CONSTANT}` to prevent the bareword quoting mechanism from kicking in. Similarly, since the `=>` operator quotes a bareword immediately to its left, you have to say `CONSTANT() => 'value'` (or simply use a comma in place of the big arrow) instead of `CONSTANT => 'value'`.

SEE ALSO

Readonly - Facility for creating read-only scalars, arrays, hashes.

Attribute::Constant - Make read-only variables via attribute

Scalar::Readonly - Perl extension to the `SvREADONLY` scalar flag

Hash::Util - A selection of general-utility hash subroutines (mostly to lock/unlock keys and values)

BUGS

Please report any bugs or feature requests via the perlbug(1) utility.

AUTHORS

Tom Phoenix, <rootbeer@redcat.com>, with help from many other folks.

Multiple constant declarations at once added by Casey West, <casey@geeknest.com>.

Documentation mostly rewritten by Ilmari Karonen, <perl@itz.pp.sci.fi>.

This program is maintained by the Perl 5 Porters. The CPAN distribution is maintained by Sébastien Aperghis-Tramoni <sebastien@aperghis.net>.

COPYRIGHT & LICENSE

Copyright (C) 1997, 1999 Tom Phoenix

This module is free software; you can redistribute it or modify it under the same terms as Perl itself.

5.2.130 deprecate

Perl pragma for deprecating the inclusion of a module in core

SYNOPSIS

```
use deprecate;  # warn about future absence if loaded from core
```

DESCRIPTION

This pragma simplifies the maintenance of dual-life modules that will no longer be included in the Perl core in a future Perl release, but are still included currently.

The purpose of the pragma is to alert users to the status of such a module by issuing a warning that encourages them to install the module from CPAN, so that a future upgrade to a perl which omits the module will not break their code.

This warning will only be issued if the module was loaded from a core library directory, which allows the `use deprecate` line to be included in the CPAN version of the module. Because the pragma remains silent when the module is run from a non-core library directory, the pragma call does not need to be patched into or out of either the core or CPAN version of the module. The exact same code can be shipped for either purpose.

Important Caveat

Note that when a module installs from CPAN to a core library directory rather than the site library directories, the user gains no protection from having installed it.

At the same time, this pragma cannot detect when such a module has installed from CPAN to the core library, and so it would endlessly and uselessly exhort the user to upgrade.

Therefore modules that can install from CPAN to the core library must make sure not to call this pragma when they have done so. Generally this means that the exact logic from the installer must be mirrored inside the module. E.g.:

```
# Makefile.PL
WriteMakefile(
    # ...
    INSTALLDIRS => ( "$]" >= 5.011 ? 'site' : 'perl' ),
);

# lib/Foo/Bar.pm
use if "$]" >= 5.011, 'deprecate';
```

(The above example shows the most important case of this: when the target is a Perl older than 5.12 (where the core library directories take precedence over the site library directories) and the module being installed was included in core in that Perl version. Under those circumstances, an upgrade of the module from CPAN is only possible by installing to the core library.)

EXPORT

None by default. The only method is `import`, called by `use deprecate;`.

SEE ALSO

First example to `use deprecate;` was Switch.

AUTHOR

Original version by Nicholas Clark

COPYRIGHT AND LICENSE

Copyright (C) 2009, 2011

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself, either Perl version 5.10.0 or, at your option, any later version of Perl 5 you may have available.

5.2.131 diagnostics

produce verbose warning diagnostics

SYNOPSIS

Using the `diagnostics` pragma:

```
use diagnostics;
use diagnostics -verbose;

diagnostics->enable;
diagnostics->disable;
```

Using the `splain` standalone filter program:

```
perl program 2>diag.out
splain [-v] [-p] diag.out
```

Using diagnostics to get stack traces from a misbehaving script:

```
perl -Mdiagnostics=-traceonly my_script.pl
```

DESCRIPTION

The `diagnostics` Pragma

This module extends the terse diagnostics normally emitted by both the perl compiler and the perl interpreter (from running perl with a -w switch or `use warnings`), augmenting them with the more explicative and endearing descriptions found in *perldiag*. Like the other pragmata, it affects the compilation phase of your program rather than merely the execution phase.

To use in your program as a pragma, merely invoke

```
use diagnostics;
```

at the start (or near the start) of your program. (Note that this *does* enable perl's **-w** flag.) Your whole compilation will then be subject(ed :-) to the enhanced diagnostics. These still go out **STDERR**.

Due to the interaction between runtime and compiletime issues, and because it's probably not a very good idea anyway, you may not use `no diagnostics` to turn them off at compiletime. However, you may control their behaviour at runtime using the `disable()` and `enable()` methods to turn them off and on respectively.

The **-verbose** flag first prints out the *perldiag* introduction before any other diagnostics. The $diagnostics::PRETTY variable can generate nicer escape sequences for pagers.

Warnings dispatched from perl itself (or more accurately, those that match descriptions found in *perldiag*) are only displayed once (no duplicate descriptions). User code generated warnings a la warn() are unaffected, allowing duplicate user messages to be displayed.

This module also adds a stack trace to the error message when perl dies. This is useful for pinpointing what caused the death. The **-traceonly** (or just **-t**) flag turns off the explanations of warning messages leaving just the stack traces. So if your script is dieing, run it again with

```
perl -Mdiagnostics=-traceonly my_bad_script
```

to see the call stack at the time of death. By supplying the **-warntrace** (or just **-w**) flag, any warnings emitted will also come with a stack trace.

The *splain* Program

Another program, *splain* is actually nothing more than a link to the (executable) `diagnostics.pm` module, as well as a link to the `diagnostics.pod` documentation. The **-v** flag is like the `use diagnostics -verbose` directive. The **-p** flag is like the $diagnostics::PRETTY variable. Since you're post-processing with *splain*, there's no sense in being able to enable() or disable() processing.

Output from *splain* is directed to **STDOUT**, unlike the pragma.

EXAMPLES

The following file is certain to trigger a few errors at both runtime and compiletime:

```
use diagnostics;
print NOWHERE "nothing\n";
print STDERR "\n\tThis message should be unadorned.\n";
warn "\tThis is a user warning";
print "\nDIAGNOSTIC TESTER: Please enter a <CR> here: ";
my $a, $b = scalar <STDIN>;
print "\n";
print $x/$y;
```

If you prefer to run your program first and look at its problem afterwards, do this:

```
perl -w test.pl 2>test.out
./splain < test.out
```

Note that this is not in general possible in shells of more dubious heritage, as the theoretical

```
(perl -w test.pl >/dev/tty) >& test.out
./splain < test.out
```

Because you just moved the existing **stdout** to somewhere else.

If you don't want to modify your source code, but still have on-the-fly warnings, do this:

```
exec 3>&1; perl -w test.pl 2>&1 1>&3 3>&- | splain 1>&2 3>&-
```

Nifty, eh?

If you want to control warnings on the fly, do something like this. Make sure you do the `use` first, or you won't be able to get at the enable() or disable() methods.

```
use diagnostics; # checks entire compilation phase
    print "\ntime for 1st bogus diags: SQUAWKINGS\n";
    print BOGUS1 'nada';
    print "done with 1st bogus\n";

diagnostics->disable; # only turns off runtime warnings
    print "\ntime for 2nd bogus: (squelched)\n";
    print BOGUS2 'nada';
```

(continues on next page)

(continued from previous page)

```
    print "done with 2nd bogus\n";

diagnostics->enable; # turns back on runtime warnings
    print "\ntime for 3rd bogus: SQUAWKINGS\n";
    print BOGUS3 'nada';
    print "done with 3rd bogus\n";

diagnostics->disable;
    print "\ntime for 4th bogus: (squelched)\n";
    print BOGUS4 'nada';
    print "done with 4th bogus\n";
```

INTERNALS

Diagnostic messages derive from the `perldiag.pod` file when available at runtime. Otherwise, they may be embedded in the file itself when the splain package is built. See the `Makefile` for details.

If an extant $SIG{__WARN__} handler is discovered, it will continue to be honored, but only after the diagnostics::splainthis() function (the module's $SIG{__WARN__} interceptor) has had its way with your warnings.

There is a $diagnostics::DEBUG variable you may set if you're desperately curious what sorts of things are being intercepted.

```
BEGIN { $diagnostics::DEBUG = 1 }
```

BUGS

Not being able to say "no diagnostics" is annoying, but may not be insurmountable.

The `-pretty` directive is called too late to affect matters. You have to do this instead, and *before* you load the module.

```
BEGIN { $diagnostics::PRETTY = 1 }
```

I could start up faster by delaying compilation until it should be needed, but this gets a "panic: top_level" when using the pragma form in Perl 5.001e.

While it's true that this documentation is somewhat subserious, if you use a program named *splain*, you should expect a bit of whimsy.

AUTHOR

Tom Christiansen <`tchrist@mox.perl.com`>, 25 June 1995.

5.2.132 encoding

allows you to write your script in non-ASCII and non-UTF-8

WARNING

This module has been deprecated since perl v5.18. See “DESCRIPTION” and “BUGS”.

SYNOPSIS

```
use encoding "greek";  # Perl like Greek to you?
use encoding "euc-jp"; # Jperl!

# or you can even do this if your shell supports your native␣
↪encoding

perl -Mencoding=latin2 -e'...' # Feeling centrally European?
perl -Mencoding=euc-kr -e'...' # Or Korean?

# more control

# A simple euc-cn => utf-8 converter
use encoding "euc-cn", STDOUT => "utf8";  while(<>){print};

# "no encoding;" supported
no encoding;

# an alternate way, Filter
use encoding "euc-jp", Filter=>1;
# now you can use kanji identifiers -- in euc-jp!

# encode based on the current locale - specialized purposes only;
# fraught with danger!!
use encoding ':locale';
```

DESCRIPTION

This pragma is used to enable a Perl script to be written in encodings that aren’t strictly ASCII nor UTF-8. It translates all or portions of the Perl program script from a given encoding into UTF-8, and changes the PerlIO layers of `STDIN` and `STDOUT` to the encoding specified.

This pragma dates from the days when UTF-8-enabled editors were uncommon. But that was long ago, and the need for it is greatly diminished. That, coupled with the fact that it doesn’t work with threads, along with other problems, (see “BUGS”) have led to its being deprecated. It is planned to remove this pragma in a future Perl version. New code should be written in UTF-8, and the `use utf8` pragma used instead (see perluniintro and *utf8* for details). Old code should be converted to UTF-8, via something like the recipe in the “SYNOPSIS” (though this simple approach may require manual adjustments afterwards).

If UTF-8 is not an option, it is recommended that one use a simple source filter, such as that provided by Filter::Encoding on CPAN or this pragma’s own `Filter` option (see below).

The only legitimate use of this pragma is almost certainly just one per file, near the top,

with file scope, as the file is likely going to only be written in one encoding. Further restrictions apply in Perls before v5.22 (see "Prior to Perl v5.22").

There are two basic modes of operation (plus turning if off):

- `use encoding ['_ENCNAME_'] ;`

  Please note: This mode of operation is no longer supported as of Perl v5.26.

  This is the normal operation. It translates various literals encountered in the Perl source file from the encoding *ENCNAME* into UTF-8, and similarly converts character code points. This is used when the script is a combination of ASCII (for the variable names and punctuation, *etc*), but the literal data is in the specified encoding.

  ENCNAME is optional. If omitted, the encoding specified in the environment variable `PERL_ENCODING` is used. If this isn't set, or the resolved-to encoding is not known to `Encode`, the error `Unknown encoding '_ENCNAME_'` will be thrown.

  Starting in Perl v5.8.6 (`Encode` version 2.0.1), *ENCNAME* may be the name `:locale`. This is for very specialized applications, and is documented in "The `:locale` sub-pragma" below.

  The literals that are converted are `q//`, `qq//`, `qr//`, `qw///`, `qx//`, and starting in v5.8.1, `tr///`. Operations that do conversions include `chr`, `ord`, `utf8::upgrade` (but not `utf8::downgrade`), and `chomp`.

  Also starting in v5.8.1, the `DATA` pseudo-filehandle is translated from the encoding into UTF-8.

  For example, you can write code in EUC-JP as follows:

```
my $Rakuda = "\xF1\xD1\xF1\xCC"; # Camel in Kanji
             #<-char-><-char->   # 4 octets
s/\bCamel\b/$Rakuda/;
```

  And with `use encoding "euc-jp"` in effect, it is the same thing as that code in UTF-8:

```
my $Rakuda = "\x{99F1}\x{99DD}"; # two Unicode Characters
s/\bCamel\b/$Rakuda/;
```

  See "EXAMPLE" below for a more complete example.

  Unless `${^UNICODE}` (available starting in v5.8.2) exists and is non-zero, the PerlIO layers of `STDIN` and `STDOUT` are set to "`:encoding(_ENCNAME_)`". Therefore,

```
use encoding "euc-jp";
my $message = "Camel is the symbol of perl.\n";
my $Rakuda = "\xF1\xD1\xF1\xCC"; # Camel in Kanji
$message =~ s/\bCamel\b/$Rakuda/;
print $message;
```

  will print

```
"\xF1\xD1\xF1\xCC is the symbol of perl.\n"
```

not

```
"\x{99F1}\x{99DD} is the symbol of perl.\n"
```

You can override this by giving extra arguments; see below.

Note that STDERR WILL NOT be changed, regardless.

Also note that non-STD file handles remain unaffected. Use `use open` or `binmode` to change the layers of those.

- `use encoding _ENCNAME_, Filter=>1;`

  This operates as above, but the `Filter` argument with a non-zero value causes the entire script, and not just literals, to be translated from the encoding into UTF-8. This allows identifiers in the source to be in that encoding as well. (Problems may occur if the encoding is not a superset of ASCII; imagine all your semi-colons being translated into something different.) One can use this form to make

  ```
  ${"\x{4eba}"}++
  ```

  work. (This is equivalent to `$_human_++`, where *human* is a single Han ideograph).

  This effectively means that your source code behaves as if it were written in UTF-8 with `'use utf8'` in effect. So even if your editor only supports Shift_JIS, for example, you can still try examples in Chapter 15 of `Programming Perl, 3rd Ed.`.

  This option is significantly slower than the other one.

- `no encoding;`

  Unsets the script encoding. The layers of `STDIN`, `STDOUT` are reset to "`:raw`" (the default unprocessed raw stream of bytes).

OPTIONS

Setting STDIN and/or STDOUT individually

The encodings of `STDIN` and `STDOUT` are individually settable by parameters to the pragma:

```
use encoding 'euc-tw', STDIN => 'greek'  ...;
```

In this case, you cannot omit the first *ENCNAME*. `STDIN => undef` turns the I/O transcoding completely off for that filehandle.

When `${^UNICODE}` (available starting in v5.8.2) exists and is non-zero, these options will be completely ignored. See "`${^UNICODE}`" *in perlvar* and "`-C`" in perlrun for details.

The `:locale` sub-pragma

Starting in v5.8.6, the encoding name may be `:locale`. This means that the encoding is taken from the current locale, and not hard-coded by the pragma. Since a script really can only be encoded in exactly one encoding, this option is dangerous. It makes sense only if the script itself is written in ASCII, and all the possible locales that will be in

use when the script is executed are supersets of ASCII. That means that the script itself doesn't get changed, but the I/O handles have the specified encoding added, and the operations like `chr` and `ord` use that encoding.

The logic of finding which locale `:locale` uses is as follows:

1. If the platform supports the `langinfo(CODESET)` interface, the codeset returned is used as the default encoding for the open pragma.
2. If 1. didn't work but we are under the locale pragma, the environment variables `LC_ALL` and `LANG` (in that order) are matched for encodings (the part after "`.`", if any), and if any found, that is used as the default encoding for the open pragma.
3. If 1. and 2. didn't work, the environment variables `LC_ALL` and `LANG` (in that order) are matched for anything looking like UTF-8, and if any found, `:utf8` is used as the default encoding for the open pragma.

If your locale environment variables (`LC_ALL`, `LC_CTYPE`, `LANG`) contain the strings 'UTF-8' or 'UTF8' (case-insensitive matching), the default encoding of your `STDIN`, `STDOUT`, and `STDERR`, and of **any subsequent file open**, is UTF-8.

CAVEATS

SIDE EFFECTS

- If the `encoding` pragma is in scope then the lengths returned are calculated from the length of `$/` in Unicode characters, which is not always the same as the length of `$/` in the native encoding.
- Without this pragma, if strings operating under byte semantics and strings with Unicode character data are concatenated, the new string will be created by decoding the byte strings as *ISO 8859-1 (Latin-1)*.

  The **encoding** pragma changes this to use the specified encoding instead. For example:

```
use encoding 'utf8';
my $string = chr(20000); # a Unicode string
utf8::encode($string);   # now it's a UTF-8 encoded byte string
# concatenate with another Unicode string
print length($string . chr(20000));
```

  Will print 2, because `$string` is upgraded as UTF-8. Without `use encoding 'utf8';`, it will print 4 instead, since `$string` is three octets when interpreted as Latin-1.

DO NOT MIX MULTIPLE ENCODINGS

Notice that only literals (string or regular expression) having only legacy code points are affected: if you mix data like this

```
\x{100}\xDF
\xDF\x{100}
```

the data is assumed to be in (Latin 1 and) Unicode, not in your native encoding. In other words, this will match in "greek":

```
"\xDF" =~ /\x{3af}/
```

but this will not

```
"\xDF\x{100}" =~ /\x{3af}\x{100}/
```

since the `\xDF` (ISO 8859-7 GREEK SMALL LETTER IOTA WITH TONOS) on the left will **not** be upgraded to `\x{3af}` (Unicode GREEK SMALL LETTER IOTA WITH TONOS) because of the `\x{100}` on the left. You should not be mixing your legacy data and Unicode in the same string.

This pragma also affects encoding of the 0x80..0xFF code point range: normally characters in that range are left as eight-bit bytes (unless they are combined with characters with code points 0x100 or larger, in which case all characters need to become UTF-8 encoded), but if the `encoding` pragma is present, even the 0x80..0xFF range always gets UTF-8 encoded.

After all, the best thing about this pragma is that you don't have to resort to \x{....} just to spell your name in a native encoding. So feel free to put your strings in your encoding in quotes and regexes.

Prior to Perl v5.22

The pragma was a per script, not a per block lexical. Only the last `use encoding` or `no encoding` mattered, and it affected **the whole script**. However, the `no encoding` pragma was supported and `use encoding` could appear as many times as you want in a given script (though only the last was effective).

Since the scope wasn't lexical, other modules' use of `chr`, `ord`, *etc.* were affected. This leads to spooky, incorrect action at a distance that is hard to debug.

This means you would have to be very careful of the load order:

```
# called module
package Module_IN_BAR;
use encoding "bar";
# stuff in "bar" encoding here
1;

# caller script
use encoding "foo"
use Module_IN_BAR;
# surprise! use encoding "bar" is in effect.
```

The best way to avoid this oddity is to use this pragma RIGHT AFTER other modules are loaded. i.e.

```
use Module_IN_BAR;
use encoding "foo";
```

Prior to Encode version 1.87

- `STDIN` and `STDOUT` were not set under the filter option. And `STDIN=>_ENCODING_` and `STDOUT=>_ENCODING_` didn't work like non-filter version.
- `use utf8` wasn't implicitly declared so you have to `use utf8` to do

```
${"\x{4eba}"}++
```

Prior to Perl v5.8.1

- "NON-EUC" doublebyte encodings

  Because perl needs to parse the script before applying this pragma, such encodings as Shift_JIS and Big-5 that may contain `'\'` (BACKSLASH; `\x5c`) in the second byte fail because the second byte may accidentally escape the quoting character that follows.

- `tr///`

  The **encoding** pragma works by decoding string literals in `q//`, `qq//`, `qr//`, `qw//`, `qx//` and so forth. In perl v5.8.0, this does not apply to `tr///`. Therefore,

```
use encoding 'euc-jp';
#....
$kana =~ tr/\xA4\xA1-\xA4\xF3/\xA5\xA1-\xA5\xF3/;
#           -------- -------- -------- --------
```

  Does not work as

```
$kana =~ tr/\x{3041}-\x{3093}/\x{30a1}-\x{30f3}/;
```

  – Legend of characters above

```
utf8     euc-jp   charnames::viacode()
----------------------------------------
\x{3041} \xA4\xA1 HIRAGANA LETTER SMALL A
\x{3093} \xA4\xF3 HIRAGANA LETTER N
\x{30a1} \xA5\xA1 KATAKANA LETTER SMALL A
\x{30f3} \xA5\xF3 KATAKANA LETTER N
```

  This counterintuitive behavior has been fixed in perl v5.8.1.

  In perl v5.8.0, you can work around this as follows;

```
use encoding 'euc-jp';
#  ....
eval qq{ \$kana =~ tr/\xA4\xA1-\xA4\xF3/\xA5\xA1-\xA5\xF3/ };
```

  Note the `tr//` expression is surrounded by `qq{}`. The idea behind this is the same as the classic idiom that makes `tr///` 'interpolate':

```
tr/$from/$to/;             # wrong!
eval qq{ tr/$from/$to/ }; # workaround.
```

EXAMPLE - Greekperl

```
use encoding "iso 8859-7";

# \xDF in ISO 8859-7 (Greek) is \x{3af} in Unicode.

$a = "\xDF";
$b = "\x{100}";

printf "%#x\n", ord($a); # will print 0x3af, not 0xdf

$c = $a . $b;

# $c will be "\x{3af}\x{100}", not "\x{df}\x{100}".

# chr() is affected, and ...

print "mega\n"  if ord(chr(0xdf)) == 0x3af;

# ... ord() is affected by the encoding pragma ...

print "tera\n" if ord(pack("C", 0xdf)) == 0x3af;

# ... as are eq and cmp ...

print "peta\n" if "\x{3af}" eq  pack("C", 0xdf);
print "exa\n"  if "\x{3af}" cmp pack("C", 0xdf) == 0;

# ... but pack/unpack C are not affected, in case you still
# want to go back to your native encoding

print "zetta\n" if unpack("C", (pack("C", 0xdf))) == 0xdf;
```

BUGS

- Thread safety

  `use encoding ...` is not thread-safe (i.e., do not use in threaded applications).

- Can't be used by more than one module in a single program.

  Only one encoding is allowed. If you combine modules in a program that have different encodings, only one will be actually used.

- Other modules using `STDIN` and `STDOUT` get the encoded stream

  They may be expecting something completely different.

- literals in regex that are longer than 127 bytes

  For native multibyte encodings (either fixed or variable length), the current implementation of the regular expressions may introduce recoding errors for regular expression literals longer than 127 bytes.

- EBCDIC

The encoding pragma is not supported on EBCDIC platforms.

- `format`

  This pragma doesn't work well with `format` because PerlIO does not get along very well with it. When `format` contains non-ASCII characters it prints funny or gets "wide character warnings". To understand it, try the code below.

```
# Save this one in utf8
# replace *non-ascii* with a non-ascii string
my $camel;
format STDOUT =
*non-ascii*@>>>>>>>
$camel
.
$camel = "*non-ascii*";
binmode(STDOUT=>':encoding(utf8)'); # bang!
write;              # funny
print $camel, "\n"; # fine
```

  Without binmode this happens to work but without binmode, print() fails instead of write().

  At any rate, the very use of `format` is questionable when it comes to unicode characters since you have to consider such things as character width (i.e. double-width for ideographs) and directions (i.e. BIDI for Arabic and Hebrew).

- See also "CAVEATS"

HISTORY

This pragma first appeared in Perl v5.8.0. It has been enhanced in later releases as specified above.

SEE ALSO

perlunicode, *Encode*, *open*, *Filter::Util::Call*,

Ch. 15 of `Programming Perl (3rd Edition)` by Larry Wall, Tom Christiansen, Jon Orwant; O'Reilly & Associates; ISBN 0-596-00027-8

Modules

- `encoding::warnings` - Warn on implicit encoding conversions

encoding::warnings

Warn on implicit encoding conversions

VERSION

This document describes version 0.13 of encoding::warnings, released June 20, 2016.

NOTICE

As of Perl 5.26.0, this module has no effect. The internal Perl feature that was used to implement this module has been removed. In recent years, much work has been done on the Perl core to eliminate discrepancies in the treatment of upgraded versus downgraded strings. In addition, the *encoding* pragma, which caused many of the problems, is no longer supported. Thus, the warnings this module produced are no longer necessary.

Hence, if you load this module on Perl 5.26.0, you will get one warning that the module is no longer supported; and the module will do nothing thereafter.

SYNOPSIS

```
use encoding::warnings; # or 'FATAL' to raise fatal exceptions

utf8::encode($a = chr(20000));  # a byte-string (raw bytes)
$b = chr(20000);                # a unicode-string (wide characters)

# "Bytes implicitly upgraded into wide characters as iso-8859-1"
$c = $a . $b;
```

DESCRIPTION

Overview of the problem

By default, there is a fundamental asymmetry in Perl's unicode model: implicit upgrading from byte-strings to unicode-strings assumes that they were encoded in *ISO 8859-1 (Latin-1)*, but unicode-strings are downgraded with UTF-8 encoding. This happens because the first 256 codepoints in Unicode happens to agree with Latin-1.

However, this silent upgrading can easily cause problems, if you happen to mix unicode strings with non-Latin1 data – i.e. byte-strings encoded in UTF-8 or other encodings. The error will not manifest until the combined string is written to output, at which time it would be impossible to see where did the silent upgrading occur.

Detecting the problem

This module simplifies the process of diagnosing such problems. Just put this line on top of your main program:

```
use encoding::warnings;
```

Afterwards, implicit upgrading of high-bit bytes will raise a warning. Ex.: `Bytes implicitly upgraded into wide characters as iso-8859-1 at \- line 7.`

However, strings composed purely of ASCII code points (`0x00..0x7F`) will *not* trigger this warning.

You can also make the warnings fatal by importing this module as:

```
use encoding::warnings 'FATAL';
```

Solving the problem

Most of the time, this warning occurs when a byte-string is concatenated with a unicode-string. There are a number of ways to solve it:

- Upgrade both sides to unicode-strings

  If your program does not need compatibility for Perl 5.6 and earlier, the recommended approach is to apply appropriate IO disciplines, so all data in your program become unicode-strings. See *encoding*, *open* and *"binmode" in perlfunc* for how.

- Downgrade both sides to byte-strings

  The other way works too, especially if you are sure that all your data are under the same encoding, or if compatibility with older versions of Perl is desired.

  You may downgrade strings with `Encode::encode` and `utf8::encode`. See *Encode* and *utf8* for details.

- Specify the encoding for implicit byte-string upgrading

  If you are confident that all byte-strings will be in a specific encoding like UTF-8, *and* need not support older versions of Perl, use the `encoding` pragma:

  ```
  use encoding 'utf8';
  ```

  Similarly, this will silence warnings from this module, and preserve the default behaviour:

  ```
  use encoding 'iso-8859-1';
  ```

  However, note that `use encoding` actually had three distinct effects:

  – PerlIO layers for **STDIN** and **STDOUT**

    This is similar to what *open* pragma does.

  – Literal conversions

    This turns *all* literal string in your program into unicode-strings (equivalent to a `use utf8`), by decoding them using the specified encoding.

  – Implicit upgrading for byte-strings

    This will silence warnings from this module, as shown above.

  Because literal conversions also work on empty strings, it may surprise some people:

  ```
  use encoding 'big5';

  my $byte_string = pack("C*", 0xA4, 0x40);
  print length $a;    # 2 here.
  ```

(continues on next page)

(continued from previous page)

```
$a .= "";           # concatenating with a unicode string...
print length $a;    # 1 here!
```

In other words, do not `use encoding` unless you are certain that the program will not deal with any raw, 8-bit binary data at all.

However, the `Filter => 1` flavor of `use encoding` will *not* affect implicit upgrading for byte-strings, and is thus incapable of silencing warnings from this module. See *encoding* for more details.

CAVEATS

For Perl 5.9.4 or later, this module's effect is lexical.

For Perl versions prior to 5.9.4, this module affects the whole script, instead of inside its lexical block.

SEE ALSO

perlunicode, perluniintro

open, *utf8*, *encoding*, *Encode*

AUTHORS

Audrey Tang

COPYRIGHT

Copyright 2004, 2005, 2006, 2007 by Audrey Tang cpan@audreyt.org.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

See http://www.perl.com/perl/misc/Artistic.html

5.2.133 experimental

Experimental features made easy

VERSION

version 0.036

SYNOPSIS

```
use experimental 'lexical_subs', 'signatures';
my sub plus_one($value) { $value + 1 }
```

DESCRIPTION

This pragma provides an easy and convenient way to enable or disable experimental features.

Every version of perl has some number of features present but considered "experimental." For much of the life of Perl 5, this was only a designation found in the documentation. Starting in Perl v5.10.0, and more aggressively in v5.18.0, experimental features were placed behind pragmata used to enable the feature and disable associated warnings.

The `experimental` pragma exists to combine the required incantations into a single interface stable across releases of perl. For every experimental feature, this should enable the feature and silence warnings for the enclosing lexical scope:

```
use experimental 'feature-name';
```

To disable the feature and, if applicable, re-enable any warnings, use:

```
no experimental 'feature-name';
```

The supported features, documented further below, are:

- `args_array_with_signatures` - allow `@_` to be used in signatured subs.

  This is supported on perl 5.20.0 and above, but is likely to be removed in the future.

- `array_base` - allow the use of `$[` to change the starting index of `@array`.

  This was removed in perl 5.30.0.

- `autoderef` - allow push, each, keys, and other built-ins on references.

  This was added in perl 5.14.0 and removed in perl 5.24.0.

- `bitwise` - allow the new stringwise bit operators

  This was added in perl 5.22.0.

- `builtin` - allow the use of the functions in the builtin:: namespace

  This was added in perl 5.36.0

- `const_attr` - allow the :const attribute on subs

  This was added in perl 5.22.0.

- `class` - enables the `class`, `field`, and `method` syntax

  This was added in perl 5.38.0.

- `declared_refs` - enables aliasing via assignment to references

  This was added in perl 5.26.0.

- `defer` - enables the use of defer blocks

  This was added in perl 5.36.0

- `extra_paired_delimiters` - enables the use of more paired string delimiters than the traditional four, `< >`, `( )`, `{ }`, and `[ ]`.

  This was added in perl 5.36.

- `for_list` - allows iterating over multiple values at a time with `for`

  This was added in perl 5.36.0

- `isa` - allow the use of the `isa` infix operator

  This was added in perl 5.32.0.

- `keyword_all` - enables the `all` operator

  This was added in perl 5.42.0.

- `keyword_any` - enables the `any` operator

  This was added in perl 5.42.0.

- `lexical_topic` - allow the use of lexical `$_` via `my $_`.

  This was added in perl 5.10.0 and removed in perl 5.24.0.

- `lexical_subs` - allow the use of lexical subroutines.

  This was added in 5.18.0, and became non-experimental (and always enabled) in 5.26.0.

- `postderef` - allow the use of postfix dereferencing expressions

  This was added in perl 5.20.0, and became non-experimental (and always enabled) in 5.24.0.

- `postderef_qq` - allow the use of postfix dereferencing expressions inside interpolating strings

  This was added in perl 5.20.0, and became non-experimental (and always enabled) in 5.24.0.

- `re_strict` - enables strict mode in regular expressions

  This was added in perl 5.22.0.

- `refaliasing` - allow aliasing via `\$x = \$y`

  This was added in perl 5.22.0.

- `regex_sets` - allow extended bracketed character classes in regexps

  This was added in perl 5.18.0, and became non-experimental (and always enabled) in 5.36.0.

  This is documented at "Extended Bracketed Character Classes" in perlrecharclass.

- `signatures` - allow subroutine signatures (for named arguments)

  This was added in perl 5.20.0.

- `smartmatch` - allow the use of ~~

  This was added in perl 5.10.0, but it should be noted there are significant incompatibilities between 5.10.0 and 5.10.1.

  The feature was deprecated in perl 5.38.0, and undeprecated in 5.42.0.

- `switch` - allow the use of ~~, given, and when

  This was added in perl 5.10.0.

The feature was deprecated in perl 5.38.0, and undeprecated in 5.42.0.

- `try` - allow the use of `try` and `catch`

  This was added in perl 5.34.0

- `win32_perlio` - allows the use of the :win32 IO layer.

  This was added on perl 5.22.0.

Ordering matters

Using this pragma to 'enable an experimental feature' is another way of saying that this pragma will disable the warnings which would result from using that feature. Therefore, the order in which pragmas are applied is important. In particular, you probably want to enable experimental features *after* you enable warnings:

```
use warnings;
use experimental 'smartmatch';
```

You also need to take care with modules that enable warnings for you. A common example being Moose. In this example, warnings for the 'smartmatch' feature are first turned on by the warnings pragma, off by the experimental pragma and back on again by the Moose module (fix is to switch the last two lines):

```
use warnings;
use experimental 'smartmatch';
use Moose;
```

Disclaimer

Because of the nature of the features it enables, forward compatibility can not be guaranteed in any way.

SEE ALSO

perlexperiment contains more information about experimental features.

AUTHOR

Leon Timmermans fawaka@gmail.com

COPYRIGHT AND LICENSE

This software is copyright (c) 2013 by Leon Timmermans.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

5.2.134 feature

Perl pragma to enable new features

SYNOPSIS

```
use feature qw(fc say);

# Without the "use feature" above, this code would not be able to␣
 ↪find
# the built-ins "say" or "fc":
say "The case-folded version of $x is: " . fc $x;


# set features to match the :5.36 bundle, which may turn off or on
# multiple features (see "FEATURE BUNDLES" below)
use feature ':5.36';


# implicitly loads :5.36 feature bundle
use v5.36;
```

DESCRIPTION

It is usually impossible to add new syntax to Perl without breaking some existing programs. This pragma provides a way to minimize that risk. New syntactic constructs, or new semantic meanings to older constructs, can be enabled by `use feature 'foo'`, and will be parsed only when the appropriate feature pragma is in scope. (Nevertheless, the `CORE::` prefix provides access to all Perl keywords, regardless of this pragma.)

Lexical effect

Like other pragmas (`use strict`, for example), features have a lexical effect. `use feature qw(foo)` will only make the feature "foo" available from that point to the end of the enclosing block.

```
{
    use feature 'say';
    say "say is available here";
}
print "But not here.\n";
```

no feature

Features can also be turned off by using `no feature "foo"`. This too has lexical effect.

```
use feature 'say';
say "say is available here";
{
    no feature 'say';
```

(continues on next page)

(continued from previous page)

```
    print "But not here.\n";
}
say "Yet it is here.";
```

`no feature` with no features specified will reset to the default group. To disable *all* features (an unusual request!) use `no feature ':all'`.

AVAILABLE FEATURES

Read "FEATURE BUNDLES" for the feature cheat sheet summary.

The 'apostrophe_as_package_separator' feature

This feature enables use `'` (apostrophe) as an alternative to using `::` as a separate in package and other global names.

This is enabled by default, but disabled from the 5.42 feature bundle onwards. In previous versions it was enabled all the time.

This only disables `'` in symbols in your source code, the internal conversion from `'` to `::`, including for symbolic references, is always enabled.

The 'array_base' feature

This feature supported the legacy `$[` variable. See *"$[" in perlvar*. It was on by default but disabled under `use v5.16` (see "IMPLICIT LOADING", below) and unavailable since perl 5.30.

This feature is available under this name starting with Perl 5.16. In previous versions, it was simply on all the time, and this pragma knew nothing about it.

The 'bareword_filehandles' feature

This feature enables bareword filehandles for builtin functions operations, a generally discouraged practice. It is enabled by default, but can be turned off to disable bareword filehandles, except for the exceptions listed below.

The perl built-in filehandles `STDIN`, `STDOUT`, `STDERR`, `DATA`, `ARGV`, `ARGVOUT` and the special `_` are always enabled.

This feature is available under this name from Perl 5.34 onwards. In previous versions it was simply on all the time. It is disabled from the 5.38 feature bundle onwards.

You can use the bareword::filehandles module on CPAN to disable bareword filehandles for older versions of perl.

The 'bitwise' feature

This makes the four standard bitwise operators (`& | ^ ~`) treat their operands consistently as numbers, and introduces four new dotted operators (`&. |. ^. ~.`) that treat their operands consistently as strings. The same applies to the assignment variants (`&= |= ^= &.= |.= ^.=`).

See *"Bitwise String Operators" in perlop* for details.

This feature is available from Perl 5.22 onwards. Starting in Perl 5.28, `use v5.28` will enable the feature. Before 5.28, it was still experimental and would emit a warning in the "experimental::bitwise" category.

The 'class' feature

WARNING: This feature is still experimental and the implementation may change or be removed in future versions of Perl. For this reason, Perl will warn when you use the feature, unless you have explicitly disabled the warning:

```
no warnings "experimental::class";
```

This feature enables the `class` block syntax and other associated keywords which implement the "new" object system, previously codenamed "Corinna".

This feature is available starting in Perl 5.38.

The 'current_sub' feature

This provides the `__SUB__` token that returns a reference to the current subroutine or `undef` outside of a subroutine.

This feature is available starting with Perl 5.16.

The 'declared_refs' feature

WARNING: This feature is still experimental and the implementation may change or be removed in future versions of Perl. For this reason, Perl will warn when you use the feature, unless you have explicitly disabled the warning:

```
no warnings "experimental::declared_refs";
```

This allows a reference to a variable to be declared with `my`, `state`, or `our`, or localized with `local`. It is intended mainly for use in conjunction with the "refaliasing" feature. See "Declaring a Reference to a Variable" in perlref for examples.

This feature is available from Perl 5.26 onwards.

The 'defer' feature

WARNING: This feature is still experimental and the implementation may change or be removed in future versions of Perl. For this reason, Perl will warn when you use the feature, unless you have explicitly disabled the warning:

```
no warnings "experimental::defer";
```

This feature enables the `defer` block syntax, which allows a block of code to be deferred until when the flow of control leaves the block which contained it. For more details, see "defer" in perlsyn.

This feature is available starting in Perl 5.36.

The 'enhanced_xx' feature

WARNING: This feature is still experimental and the implementation may change or be removed in future versions of Perl. For this reason, Perl will warn when you use the feature, unless you have explicitly disabled the warning:

```
no warnings "experimental::eenhanced_xx";
```

This feature changes the meaning of the `/xx` regular expression pattern modifier to allow bracketed character classes to be split across lines, and to catch cases where a literal # was meant to be matched, but instead that and the rest of the line was taken to be a comment.

This feature is available starting in Perl 5.44.

The 'evalbytes' feature

See "The 'unicode_eval' and 'evalbytes' features"

The 'extra_paired_delimiters' feature

WARNING: This feature is still experimental and the implementation may change or be removed in future versions of Perl. For this reason, Perl will warn when you use the feature, unless you have explicitly disabled the warning:

```
no warnings "experimental::extra_paired_delimiters";
```

This feature enables the use of more paired string delimiters than the traditional four, `< >`, `( )`, `{ }`, and `[ ]`. When this feature is on, for example, you can say `qr«pat»`.

As with any usage of non-ASCII delimiters in a UTF-8-encoded source file, you will want to ensure the parser will decode the source code from UTF-8 bytes with a declaration such as `use utf8`.

This feature is available starting in Perl 5.36.

For a full list of the available characters, see *"List of Extra Paired Delimiters" in perlop*.

The 'fc' feature

`use feature 'fc'` tells the compiler to enable the `fc` function, which implements Unicode casefolding.

See *"fc" in perlfunc* for details.

This feature is available from Perl 5.16 onwards.

The 'indirect' feature

This feature allows the use of indirect object syntax for method calls, e.g. `new Foo 1, 2;`. It is enabled by default, but can be turned off to disallow indirect object syntax.

This feature is available under this name from Perl 5.32 onwards. In previous versions, it was simply on all the time. To disallow (or warn on) indirect object syntax on older Perls, see the indirect CPAN module. It is disabled from the 5.36 feature bundle onwards.

The 'isa' feature

This allows the use of the `isa` infix operator, which tests whether the scalar given by the left operand is an object of the class given by the right operand. See *"Class Instance Operator" in perlop* for more details.

This feature is available from Perl 5.32 onwards. From Perl 5.32 to 5.34, it was classed as experimental, and Perl emitted a warning for its usage, except when explicitly disabled:

```
no warnings "experimental::isa";
```

As of Perl 5.36, use of this feature no longer triggers a warning (though the `experimental::isa` warning category still exists for compatibility with code that disables it). This feature is now considered stable, and is enabled automatically by `use v5.36` (or higher).

The 'keyword_all' feature

WARNING: This feature is still experimental and the implementation may change or be removed in future versions of Perl. For this reason, Perl will warn when you use the feature, unless you have explicitly disabled the warning:

```
no warnings "experimental::keyword_all";
```

This feature enables the `all` operator keyword. This allow testing whether all of the values in a list satisfy a given condition, with short-circuiting behaviour as soon as it finds one that does not.

This feature is available starting in Perl 5.42.

The 'keyword_any' feature

WARNING: This feature is still experimental and the implementation may change or be removed in future versions of Perl. For this reason, Perl will warn when you use the feature, unless you have explicitly disabled the warning:

```
no warnings "experimental::keyword_any";
```

This feature enables the `any` operator keyword. This allow testing whether any of the values in a list satisfy a given condition, with short-circuiting behaviour as soon as it finds one.

This feature is available starting in Perl 5.42.

The 'lexical_subs' feature

In Perl versions prior to 5.26, this feature enabled declaration of subroutines via `my sub foo`, `state sub foo` and `our sub foo` syntax. See *"Lexical Subroutines" in perlsub* for details.

This feature is available from Perl 5.18 onwards. From Perl 5.18 to 5.24, it was classed as experimental, and Perl emitted a warning for its usage, except when explicitly disabled:

```
no warnings "experimental::lexical_subs";
```

As of Perl 5.26, use of this feature no longer triggers a warning, though the `experimental::lexical_subs` warning category still exists (for compatibility with code that disables it). In addition, this syntax is not only no longer experimental, but it is enabled for all Perl code, regardless of what feature declarations are in scope.

The 'module_true' feature

This feature removes the need to return a true value at the end of a module loaded with `require` or `use`. Any errors during compilation will cause failures, but reaching the end of the module when this feature is in effect will prevent `perl` from throwing an exception that the module "did not return a true value".

This feature is available from Perl 5.38 onwards.

The 'multidimensional' feature

This feature enables multidimensional array emulation, a perl 4 (or earlier) feature that was used to emulate multidimensional arrays with hashes. This works by converting code like `$foo{$x, $y}` into `$foo{join($;, $x, $y)}`. It is enabled by default, but can be turned off to disable multidimensional array emulation.

When this feature is disabled the syntax that is normally replaced will report a compilation error.

This feature is available under this name from Perl 5.34 onwards. In previous versions, it was simply on all the time. It is disabled from the 5.36 feature bundle onwards.

You can use the multidimensional module on CPAN to disable multidimensional array emulation for older versions of Perl.

The 'postderef' and 'postderef_qq' features

The 'postderef_qq' feature extends the applicability of postfix dereference syntax so that postfix array dereference, postfix scalar dereference, and postfix array highest index access are available in double-quotish interpolations. For example, it makes the following two statements equivalent:

```
my $s = "[@{ $h->{a} }]";
my $s = "[$h->{a}->@*]";
```

This feature is available from Perl 5.20 onwards. In Perl 5.20 and 5.22, it was classed as experimental, and Perl emitted a warning for its usage, except when explicitly disabled:

```
no warnings "experimental::postderef";
```

As of Perl 5.24, use of this feature no longer triggers a warning, though the `experimental::postderef` warning category still exists (for compatibility with code that disables it).

The 'postderef' feature was used in Perl 5.20 and Perl 5.22 to enable postfix dereference syntax outside double-quotish interpolations. In those versions, using it triggered the

`experimental::postderef` warning in the same way as the 'postderef_qq' feature did. As of Perl 5.24, this syntax is not only no longer experimental, but it is enabled for all Perl code, regardless of what feature declarations are in scope.

The 'refaliasing' feature

WARNING: This feature is still experimental and the implementation may change or be removed in future versions of Perl. For this reason, Perl will warn when you use the feature, unless you have explicitly disabled the warning:

```
no warnings "experimental::refaliasing";
```

This enables aliasing via assignment to references:

```
\$a = \$b; # $a and $b now point to the same scalar
\@a = \@b; #                    to the same array
\%a = \%b;
\&a = \&b;
foreach \%hash (@array_of_hash_refs) {
    ...
}
```

See "Assigning to References" in perlref for details.

This feature is available from Perl 5.22 onwards.

The 'say' feature

`use feature 'say'` tells the compiler to enable the Raku-inspired `say` function.

See *"say" in perlfunc* for details.

This feature is available starting with Perl 5.10.

The 'signatures' feature

This enables syntax for declaring subroutine arguments as lexical variables. For example, for this subroutine:

```
sub foo ($left, $right) {
    return $left + $right;
}
```

Calling `foo(3, 7)` will assign 3 into `$left` and 7 into `$right`.

See *"Signatures" in perlsub* for details.

This feature is available from Perl 5.20 onwards. From Perl 5.20 to 5.34, it was classed as experimental, and Perl emitted a warning for its usage, except when explicitly disabled:

```
no warnings "experimental::signatures";
```

As of Perl 5.36, use of this feature no longer triggers a warning, though the `experimental::signatures` warning category still exists (for compatibility with code that disables it). This feature is now considered stable, and is enabled automatically by `use v5.36` (or higher).

The 'smartmatch' feature

`use feature 'smartmatch'` tells the compiler to enable the smartmatch operator `~~`. It is enabled by default, but can be turned off to disallow the `~~` operator.

This feature is disabled by default in the 5.42 feature bundle onwards:

```
$x ~~ $y; # fine
use v5.42;
$x ~~ $y; # error
```

This has no effect on the implicit smartmatches done by `when`.

See *"Smartmatch Operator" in perlop* for details.

The 'state' feature

`use feature 'state'` tells the compiler to enable `state` variables.

See *"Persistent Private Variables" in perlsub* for details.

This feature is available starting with Perl 5.10.

The 'switch' feature

`use feature 'switch'` tells the compiler to enable the Raku given/when construct.

See "Switch Statements" in perlsyn for details.

This feature is available starting with Perl 5.10. It is enabled by feature bundles 5.10 through 5.34, and disabled from the 5.36 feature bundle onwards.

The 'try' feature

WARNING: This feature is still partly experimental, and the implementation may change or be removed in future versions of Perl.

This feature enables the `try` and `catch` syntax, which allows exception handling, where exceptions thrown from the body of the block introduced with `try` are caught by executing the body of the `catch` block.

This feature is available starting in Perl 5.34. Before Perl 5.40 it was classed as experimental, and Perl emitted a warning for its usage, except when explicitly disabled:

```
no warnings "experimental::try";
```

As of Perl 5.40, use of this feature without a `finally` block no longer triggers a warning. The optional `finally` block is still considered experimental and emits a warning, except when explicitly disabled as above.

For more information, see "Try Catch Exception Handling" in perlsyn.

The 'unicode_eval' and 'evalbytes' features

Together, these two features are intended to replace the legacy string `eval` function, which behaves problematically in some instances. They are available starting with Perl 5.16, and are enabled by default by a `use 5.16` or higher declaration.

`unicode_eval` changes the behavior of plain string `eval` to work more consistently, especially in the Unicode world. Certain (mis)behaviors couldn't be changed without breaking some things that had come to rely on them, so the feature can be enabled and disabled. Details are at *"Under the "unicode_eval" feature" in perlfunc*.

`evalbytes` is like string `eval`, but it treats its argument as a byte string. Details are at *"evalbytes EXPR" in perlfunc*. Without a `use feature 'evalbytes'` nor a `use v5.16` (or higher) declaration in the current scope, you can still access it by instead writing `CORE::evalbytes`.

The 'unicode_strings' feature

`use feature 'unicode_strings'` tells the compiler to use Unicode rules in all string operations executed within its scope (unless they are also within the scope of either `use locale` or `use bytes`). The same applies to all regular expressions compiled within the scope, even if executed outside it. It does not change the internal representation of strings, but only how they are interpreted.

`no feature 'unicode_strings'` tells the compiler to use the traditional Perl rules wherein the native character set rules is used unless it is clear to Perl that Unicode is desired. This can lead to some surprises when the behavior suddenly changes. (See "The "Unicode Bug"" in perlunicode for details.) For this reason, if you are potentially using Unicode in your program, the `use feature 'unicode_strings'` subpragma is **strongly** recommended.

This feature is available starting with Perl 5.12; was almost fully implemented in Perl 5.14; and extended in Perl 5.16 to cover `quotemeta`; was extended further in Perl 5.26 to cover *the range operator*; and was extended again in Perl 5.28 to cover *special-cased whitespace splitting*.

FEATURE BUNDLES

It's possible to load multiple features together, using a *feature bundle*. The name of a feature bundle is prefixed with a colon, to distinguish it from an actual feature.

```
use feature ":5.10";
```

The following feature bundles are available:

```
bundle    features included
--------- -----------------
:default  indirect multidimensional
          bareword_filehandles
          apostrophe_as_package_separator smartmatch

:5.10     apostrophe_as_package_separator
          bareword_filehandles indirect
```

(continues on next page)

(continued from previous page)

```
          multidimensional say smartmatch state switch

:5.12     apostrophe_as_package_separator
          bareword_filehandles indirect
          multidimensional say smartmatch state switch
          unicode_strings

:5.14     apostrophe_as_package_separator
          bareword_filehandles indirect
          multidimensional say smartmatch state switch
          unicode_strings

:5.16     apostrophe_as_package_separator
          bareword_filehandles current_sub evalbytes
          fc indirect multidimensional say smartmatch
          state switch unicode_eval unicode_strings

:5.18     apostrophe_as_package_separator
          bareword_filehandles current_sub evalbytes
          fc indirect multidimensional say smartmatch
          state switch unicode_eval unicode_strings

:5.20     apostrophe_as_package_separator
          bareword_filehandles current_sub evalbytes
          fc indirect multidimensional say smartmatch
          state switch unicode_eval unicode_strings

:5.22     apostrophe_as_package_separator
          bareword_filehandles current_sub evalbytes
          fc indirect multidimensional say smartmatch
          state switch unicode_eval unicode_strings

:5.24     apostrophe_as_package_separator
          bareword_filehandles current_sub evalbytes
          fc indirect multidimensional postderef_qq
          say smartmatch state switch unicode_eval
          unicode_strings

:5.26     apostrophe_as_package_separator
          bareword_filehandles current_sub evalbytes
          fc indirect multidimensional postderef_qq
          say smartmatch state switch unicode_eval
          unicode_strings

:5.28     apostrophe_as_package_separator
          bareword_filehandles bitwise current_sub
          evalbytes fc indirect multidimensional
          postderef_qq say smartmatch state switch
          unicode_eval unicode_strings
```

(continues on next page)

(continued from previous page)

```
:5.30     apostrophe_as_package_separator
          bareword_filehandles bitwise current_sub
          evalbytes fc indirect multidimensional
          postderef_qq say smartmatch state switch
          unicode_eval unicode_strings

:5.32     apostrophe_as_package_separator
          bareword_filehandles bitwise current_sub
          evalbytes fc indirect multidimensional
          postderef_qq say smartmatch state switch
          unicode_eval unicode_strings

:5.34     apostrophe_as_package_separator
          bareword_filehandles bitwise current_sub
          evalbytes fc indirect multidimensional
          postderef_qq say smartmatch state switch
          unicode_eval unicode_strings

:5.36     apostrophe_as_package_separator
          bareword_filehandles bitwise current_sub
          evalbytes fc isa postderef_qq say signatures
          smartmatch state unicode_eval
          unicode_strings

:5.38     apostrophe_as_package_separator bitwise
          current_sub evalbytes fc isa module_true
          postderef_qq say signatures smartmatch state
          unicode_eval unicode_strings

:5.40     apostrophe_as_package_separator bitwise
          current_sub evalbytes fc isa module_true
          postderef_qq say signatures smartmatch state
          try unicode_eval unicode_strings

:5.42     bitwise current_sub evalbytes fc isa
          module_true postderef_qq say signatures
          state try unicode_eval unicode_strings

:5.44     bitwise current_sub evalbytes fc isa
          module_true postderef_qq say signatures
          state try unicode_eval unicode_strings
```

The `:default` bundle represents the feature set that is enabled before any `use feature` or `no feature` declaration.

Specifying sub-versions such as the `0` in `5.14.0` in feature bundles has no effect. Feature bundles are guaranteed to be the same for all sub-versions.

```
use feature ":5.14.0";    # same as ":5.14"
```

(continues on next page)

(continued from previous page)

```
use feature ":5.14.1";    # same as ":5.14"
```

You can also do:

```
use feature ":all";
```

or

```
no feature ":all";
```

but the first may enable features in a later version of Perl that change the meaning of your code, and the second may disable mechanisms that are part of Perl's current behavior that have been turned into features, just as `indirect` and `bareword_filehandles` were.

IMPLICIT LOADING

Instead of loading feature bundles by name, it is easier to let Perl do implicit loading of a feature bundle for you.

There are two ways to load the `feature` pragma implicitly:

- By using the `-E` switch on the Perl command-line instead of `-e`. That will enable the feature bundle for that version of Perl in the main compilation unit (that is, the one-liner that follows `-E`).
- By explicitly requiring a minimum Perl version number for your program, with the `use VERSION` construct. That is,

  ```
  use v5.36.0;
  ```

  will do an implicit

  ```
  no feature ':all';
  use feature ':5.36';
  ```

  and so on. Note how the trailing sub-version is automatically stripped from the version.

  But to avoid portability warnings (see *"use" in perlfunc*), you may prefer:

  ```
  use 5.036;
  ```

  with the same effect.

  If the required version is older than Perl 5.10, the ":default" feature bundle is automatically loaded instead.

  Unlike `use feature ":5.12"`, saying `use v5.12` (or any higher version) also does the equivalent of `use strict`; see *"use" in perlfunc* for details.

CHECKING FEATURES

`feature` provides some simple APIs to check which features are enabled.

These functions cannot be imported and must be called by their fully qualified names. If you don't otherwise need to set a feature you will need to ensure `feature` is loaded with:

```
use feature ();
```

- feature_enabled($feature)
- feature_enabled($feature, $depth)

  ```
  package MyStandardEnforcer;
  use feature ();
  use Carp "croak";
  sub import {
    croak "disable indirect!" if feature::feature_enabled(
  ↪"indirect");
  }
  ```

  Test whether a named feature is enabled at a given level in the call stack, returning a true value if it is. `$depth` defaults to 1, which checks the scope that called the scope calling feature::feature_enabled().

  croaks for an unknown feature name.

- features_enabled()
- features_enabled($depth)

  ```
  package ReportEnabledFeatures;
  use feature "say";
  sub import {
    say STDERR join " ", feature::features_enabled();
  }
  ```

  Returns a list of the features enabled at a given level in the call stack. `$depth` defaults to 1, which checks the scope that called the scope calling feature::features_enabled().

- feature_bundle()
- feature_bundle($depth)

  Returns the feature bundle, if any, selected at a given level in the call stack. `$depth` defaults to 1, which checks the scope that called the scope calling feature::feature_bundle().

  Returns an undefined value if no feature bundle is selected in the scope.

  The bundle name returned will be for the earliest bundle matching the selected bundle, so:

```
use feature ();
use v5.12;
BEGIN { print feature::feature_bundle(0); }
```

will print `5.11`.

This returns internal state, at this point `use v5.12;` sets the feature bundle, but `use feature ":5.12";` does not set the feature bundle. This may change in a future release of perl.

5.2.135 fields

compile-time class fields

SYNOPSIS

```
{
    package Foo;
    use fields qw(foo bar _Foo_private);
    sub new {
        my Foo $self = shift;
        unless (ref $self) {
            $self = fields::new($self);
            $self->{_Foo_private} = "this is Foo's secret";
        }
        $self->{foo} = 10;
        $self->{bar} = 20;
        return $self;
    }
}

my $var = Foo->new;
$var->{foo} = 42;

# this will generate a run-time error
$var->{zap} = 42;

# this will generate a compile-time error
my Foo $foo = Foo->new;
$foo->{zap} = 24;

# subclassing
{
    package Bar;
    use base 'Foo';
    use fields qw(baz _Bar_private);        # not shared with Foo
    sub new {
        my $class = shift;
        my $self = fields::new($class);
        $self->SUPER::new();                # init base fields
```

(continues on next page)

(continued from previous page)

```
        $self->{baz} = 10;                  # init own fields
        $self->{_Bar_private} = "this is Bar's secret";
        return $self;
    }
}
```

DESCRIPTION

The `fields` pragma enables compile-time and run-time verified class fields.

NOTE: The current implementation keeps the declared fields in the %FIELDS hash of the calling package, but this may change in future versions. Do **not** update the %FIELDS hash directly, because it must be created at compile-time for it to be fully useful, as is done by this pragma.

If a typed lexical variable (`my Class $var`) holding a reference is used to access a hash element and a package with the same name as the type has declared class fields using this pragma, then the hash key is verified at compile time. If the variables are not typed, access is only checked at run time.

The related `base` pragma will combine fields from base classes and any fields declared using the `fields` pragma. This enables field inheritance to work properly. Inherited fields can be overridden but will generate a warning if warnings are enabled.

Only valid for Perl 5.8.x and earlier: Field names that start with an underscore character are made private to the class and are not visible to subclasses.

Also, **in Perl 5.8.x and earlier**, this pragma uses pseudo-hashes, the effect being that you can have objects with named fields which are as compact and fast as arrays to access, as long as the objects are accessed through properly typed variables.

The following functions are supported:

- new

  fields::new() creates and blesses a hash of the fields declared using the `fields` pragma into the specified class. It is the recommended way to construct a fields-based object.

  This makes it possible to write a constructor like this:

  ```
  package Critter::Sounds;
  use fields qw(cat dog bird);

  sub new {
      my $self = shift;
      $self = fields::new($self) unless ref $self;
      $self->{cat} = 'meow';                    # scalar element
      @$self{'dog','bird'} = ('bark','tweet');  # slice
      return $self;
  }
  ```

- phash

This function only works in Perl 5.8.x and earlier. Pseudo-hashes were removed from Perl as of 5.10. Consider using restricted hashes or fields::new() instead (which itself uses restricted hashes under 5.10+). See *Hash::Util*. Using fields::phash() under 5.10 or higher will cause an error.

fields::phash() can be used to create and initialize a plain (unblessed) pseudo-hash. This function should always be used instead of creating pseudo-hashes directly.

If the first argument is a reference to an array, the pseudo-hash will be created with keys from that array. If a second argument is supplied, it must also be a reference to an array whose elements will be used as the values. If the second array contains fewer elements than the first, the trailing elements of the pseudo-hash will not be initialized. This makes it particularly useful for creating a pseudo-hash from subroutine arguments:

```
sub dogtag {
   my $tag = fields::phash([qw(name rank ser_num)], [@_]);
}
```

fields::phash() also accepts a list of key-value pairs that will be used to construct the pseudo hash. Examples:

```
my $tag = fields::phash(name => "Joe",
                        rank => "captain",
                        ser_num => 42);

my $pseudohash = fields::phash(%args);
```

SEE ALSO

base, *Hash::Util*

5.2.136 filetest

Perl pragma to control the filetest permission operators

SYNOPSIS

```
$can_perhaps_read = -r "file";       # use the mode bits
{
    use filetest 'access';           # intuit harder
    $can_really_read = -r "file";
}
$can_perhaps_read = -r "file";       # use the mode bits again
```

DESCRIPTION

This pragma tells the compiler to change the behaviour of the filetest permission operators, `-r -w -x -R -W -X` (see *perlfunc*).

The default behaviour of file test operators is to use the simple mode bits as returned by the stat() family of system calls. However, many operating systems have additional

features to define more complex access rights, for example ACLs (Access Control Lists). For such environments, `use filetest` may help the permission operators to return results more consistent with other tools.

The `use filetest` or `no filetest` statements affect file tests defined in their block, up to the end of the closest enclosing block (they are lexically block-scoped).

Currently, only the `access` sub-pragma is implemented. It enables (or disables) the use of access() when available, that is, on most UNIX systems and other POSIX environments. See details below.

Consider this carefully

The stat() mode bits are probably right for most of the files and directories found on your system, because few people want to use the additional features offered by access(). But you may encounter surprises if your program runs on a system that uses ACLs, since the stat() information won't reflect the actual permissions.

There may be a slight performance decrease in the filetest operations when the filetest pragma is in effect, because checking bits is very cheap.

Also, note that using the file tests for security purposes is a lost cause from the start: there is a window open for race conditions (who is to say that the permissions will not change between the test and the real operation?). Therefore if you are serious about security, just try the real operation and test for its success - think in terms of atomic operations. Filetests are more useful for filesystem administrative tasks, when you have no need for the content of the elements on disk.

The "access" sub-pragma

UNIX and POSIX systems provide an abstract access() operating system call, which should be used to query the read, write, and execute rights. This function hides various distinct approaches in additional operating system specific security features, like Access Control Lists (ACLs)

The extended filetest functionality is used by Perl only when the argument of the operators is a filename, not when it is a filehandle.

Limitation with regard to _

Because access() does not invoke stat() (at least not in a way visible to Perl), **the stat result cache "_" is not set**. This means that the outcome of the following two tests is different. The first has the stat bits of `/etc/passwd` in _, and in the second case this still contains the bits of `/etc`.

```
{ -d '/etc';
  -w '/etc/passwd';
  print -f _ ? 'Yes' : 'No';   # Yes
}

{ use filetest 'access';
  -d '/etc';
  -w '/etc/passwd';
```

(continues on next page)

(continued from previous page)

```
  print -f _ ? 'Yes' : 'No';   # No
}
```

Of course, unless your OS does not implement access(), in which case the pragma is simply ignored. Best not to use _ at all in a file where the filetest pragma is active!

As a side effect, as _ doesn't work, stacked filetest operators (`-f -w $file`) won't work either.

This limitation might be removed in a future version of perl.

5.2.137 if

`use` a Perl module if a condition holds

SYNOPSIS

```
use if CONDITION, "MODULE", ARGUMENTS;
no  if CONDITION, "MODULE", ARGUMENTS;
```

DESCRIPTION

use if

The `if` module is used to conditionally load another module. The construct:

```
use if CONDITION, "MODULE", ARGUMENTS;
```

... will load `MODULE` only if `CONDITION` evaluates to true; it has no effect if `CONDITION` evaluates to false. (The module name, assuming it contains at least one `::`, must be quoted when `'use strict "subs";'` is in effect.) If the CONDITION does evaluate to true, then the above line has the same effect as:

```
use MODULE ARGUMENTS;
```

For example, the `Unicode::UCD` module's `charinfo` function will use two functions from `Unicode::Normalize` only if a certain condition is met:

```
use if defined &DynaLoader::boot_DynaLoader,
    "Unicode::Normalize" => qw(getCombinClass NFD);
```

Suppose you wanted `ARGUMENTS` to be an empty list, *i.e.*, to have the effect of:

```
use MODULE ();
```

You can't do this with the `if` pragma; however, you can achieve exactly this effect, at compile time, with:

```
BEGIN { require MODULE if CONDITION }
```

no if

The `no if` construct is mainly used to deactivate categories of warnings when those categories would produce superfluous output under specified versions of `perl`.

For example, the `redundant` category of warnings was introduced in Perl-5.22. This warning flags certain instances of superfluous arguments to `printf` and `sprintf`. But if your code was running warnings-free on earlier versions of `perl` and you don't care about `redundant` warnings in more recent versions, you can call:

```
use warnings;
no if $] >= 5.022, q|warnings|, qw(redundant);

my $test   = { fmt  => "%s", args => [ qw( x y ) ] };
my $result  = sprintf $test->{fmt}, @{$test->{args}};
```

The `no if` construct assumes that a module or pragma has correctly implemented an `unimport()` method – but most modules and pragmata have not. That explains why the `no if` construct is of limited applicability.

BUGS

The current implementation does not allow specification of the required version of the module.

SEE ALSO

Module::Requires can be used to conditionally load one or more modules, with constraints based on the version of the module. Unlike `if` though, Module::Requires is not a core module.

Module::Load::Conditional provides a number of functions you can use to query what modules are available, and then load one or more of them at runtime.

The provide module from CPAN can be used to select one of several possible modules to load based on the version of Perl that is running.

AUTHOR

Ilya Zakharevich mailto:ilyaz@cpan.org.

COPYRIGHT AND LICENCE

This software is copyright (c) 2002 by Ilya Zakharevich.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

5.2.138 integer

Perl pragma to use integer arithmetic instead of floating point

SYNOPSIS

```
use integer;
$x = 10/3;
# $x is now 3, not 3.33333333333333333
```

DESCRIPTION

This tells the compiler to use integer operations from here to the end of the enclosing BLOCK. On many machines, this doesn't matter a great deal for most computations, but on those without floating point hardware, it can make a big difference in performance.

Note that this only affects how most of the arithmetic and relational **operators** handle their operands and results, and **not** how all numbers everywhere are treated. Specifically, `use integer;` has the effect that before computing the results of the arithmetic operators (+, -, *, /, %, +=, -=, *=, /=, %=, and unary minus), the comparison operators (<, <=, >, >=, ==, !=, <=>), and the bitwise operators (|, &, ^, <<, >>, |=, &=, ^=, <<=, >>=), the operands have their fractional portions truncated (or floored), and the result will have its fractional portion truncated as well. In addition, the range of operands and results is restricted to that of familiar two's complement integers, i.e., -(2**31) .. (2**31-1) on 32-bit architectures, and -(2**63) .. (2**63-1) on 64-bit architectures. For example, this code

```
use integer;
$x = 5.8;
$y = 2.5;
$z = 2.7;
$a = 2**31 - 1;  # Largest positive integer on 32-bit machines
$, = ", ";
print $x, -$x, $x+$y, $x-$y, $x/$y, $x*$y, $y==$z, $a, $a+1;
```

will print: 5.8, -5, 7, 3, 2, 10, 1, 2147483647, -2147483648

Note that $x is still printed as having its true non-integer value of 5.8 since it wasn't operated on. And note too the wrap-around from the largest positive integer to the largest negative one. Also, arguments passed to functions and the values returned by them are **not** affected by `use integer;`. E.g.,

```
srand(1.5);
$, = ", ";
print sin(.5), cos(.5), atan2(1,2), sqrt(2), rand(10);
```

will give the same result with or without `use integer;` The power operator ** is also not affected, so that 2 ** .5 is always the square root of 2. Now, it so happens that the pre- and post- increment and decrement operators, ++ and –, are not affected by `use integer;` either. Some may rightly consider this to be a bug – but at least it's a long-standing one.

Finally, `use integer;` also has an additional affect on the bitwise operators. Normally, the operands and results are treated as **unsigned** integers, but with `use integer;` the operands and results are **signed**. This means, among other things, that ~0 is -1, and -2 & -5 is -6.

Internally, native integer arithmetic (as provided by your C compiler) is used. This means that Perl's own semantics for arithmetic operations may not be preserved. One common source of trouble is the modulus of negative numbers, which Perl does one way, but your hardware may do another.

```
% perl -le 'print (4 % -3)'
-2
% perl -Minteger -le 'print (4 % -3)'
1
```

See "Pragmatic Modules" in perlmodlib, *"Integer Arithmetic" in perlop*

5.2.139 less

perl pragma to request less of something

SYNOPSIS

```
use less 'CPU';
```

DESCRIPTION

This is a user-pragma. If you're very lucky some code you're using will know that you asked for less CPU usage or ram or fat or... we just can't know. Consult your documentation on everything you're currently using.

For general suggestions, try requesting `CPU` or `memory`.

```
use less 'memory';
use less 'CPU';
use less 'fat';
```

If you ask for nothing in particular, you'll be asking for `less 'please'`.

```
use less 'please';
```

FOR MODULE AUTHORS

less has been in the core as a "joke" module for ages now and it hasn't had any real way to communicating any information to anything. Thanks to Nicholas Clark we have user pragmas (see perlpragma) and now `less` can do something.

You can probably expect your users to be able to guess that they can request less CPU or memory or just "less" overall.

If the user didn't specify anything, it's interpreted as having used the `please` tag. It's up to you to make this useful.

```
# equivalent
use less;
use less 'please';
```

BOOLEAN = less->of(FEATURE)

The class method `less->of( NAME )` returns a boolean to tell you whether your user requested less of something.

```
if ( less->of( 'CPU' ) ) {
    ...
}
elsif ( less->of( 'memory' ) ) {

}
```

FEATURES = less->of()

If you don't ask for any feature, you get the list of features that the user requested you to be nice to. This has the nice side effect that if you don't respect anything in particular then you can just ask for it and use it like a boolean.

```
if ( less->of ) {
    ...
}
else {
    ...
}
```

CAVEATS

- This probably does nothing.
- This works only on 5.10+

  At least it's backwards compatible in not doing much.

5.2.140 lib

manipulate @INC at compile time

SYNOPSIS

```
use lib LIST;

no lib LIST;
```

DESCRIPTION

This is a small simple module which simplifies the manipulation of @INC at compile time.

It is typically used to add extra directories to perl's search path so that later `use` or `require` statements will find modules which are not located on perl's default search path.

Adding directories to @INC

The parameters to `use lib` are added to the start of the perl search path. Saying

```
use lib LIST;
```

is *almost* the same as saying

```
BEGIN { unshift(@INC, LIST) }
```

For each directory in LIST (called $dir here) the lib module also checks to see if a directory called $dir/$archname/auto exists. If so the $dir/$archname directory is assumed to be a corresponding architecture specific directory and is added to @INC in front of $dir. lib.pm also checks if directories called $dir/$version and $dir/$version/$archname exist and adds these directories to @INC.

The current value of `$archname` can be found with this command:

```
perl -V:archname
```

The corresponding command to get the current value of `$version` is:

```
perl -V:version
```

To avoid memory leaks, all trailing duplicate entries in @INC are removed.

Deleting directories from @INC

You should normally only add directories to @INC. If you need to delete directories from @INC take care to only delete those which you added yourself or which you are certain are not needed by other modules in your script. Other modules may have added directories which they need for correct operation.

The `no lib` statement deletes all instances of each named directory from @INC.

For each directory in LIST (called $dir here) the lib module also checks to see if a directory called $dir/$archname/auto exists. If so the $dir/$archname directory is assumed to be a corresponding architecture specific directory and is also deleted from @INC. lib.pm also checks if directories called $dir/$version and $dir/$version/$archname exist and deletes these directories from @INC.

Restoring original @INC

When the lib module is first loaded it records the current value of @INC in an array `@lib::ORIG_INC`. To restore @INC to that value you can say

```
@INC = @lib::ORIG_INC;
```

CAVEATS

In order to keep lib.pm small and simple, it only works with Unix filepaths. This doesn't mean it only works on Unix, but non-Unix users must first translate their file paths to Unix conventions.

```
# VMS users wanting to put [.stuff.moo] into
# their @INC would write
use lib 'stuff/moo';
```

NOTES

In the future, this module will likely use File::Spec for determining paths, as it does now for Mac OS (where Unix-style or Mac-style paths work, and Unix-style paths are converted properly to Mac-style paths before being added to @INC).

If you try to add a file to @INC as follows:

```
use lib 'this_is_a_file.txt';
```

`lib` will warn about this. The sole exceptions are files with the `.par` extension which are intended to be used as libraries.

SEE ALSO

FindBin - optional module which deals with paths relative to the source file.

PAR - optional module which can treat `.par` files as Perl libraries.

AUTHOR

Tim Bunce, 2nd June 1995.

`lib` is maintained by the perl5-porters. Please direct any questions to the canonical mailing list. Anything that is applicable to the CPAN release can be sent to its maintainer, though.

Maintainer: The Perl5-Porters perl5-porters@perl.org

Maintainer of the CPAN release: Steffen Mueller smueller@cpan.org

COPYRIGHT AND LICENSE

This package has been part of the perl core since perl 5.001. It has been released separately to CPAN so older installations can benefit from bug fixes.

This package has the same copyright and license as the perl core.

5.2.141 locale

Perl pragma to use or avoid POSIX locales for built-in operations

SYNOPSIS

```
my @x1 = sort @y;      # Native-platform/Unicode code point sort␣
↪order
{
    use locale;
```

(continues on next page)

(continued from previous page)

```
    my @x2 = sort @y;  # Locale-defined sort order
}
my @x3 = sort @y;      # Native-platform/Unicode code point sort␣
↪order
                       # again

# Parameters to the pragma are to work around deficiencies in locale
# handling that have since been fixed, and hence these are likely no
# longer useful
use locale qw(:ctype :collate);    # Only use the locale for␣
↪character
                                   # classification (\w, \d, etc.),␣
↪and
                                   # for string comparison␣
↪operations
                                   # like '$a le $b' and sorting.
use locale ':not_characters';      # Use the locale for everything␣
↪but
                                   # character classification and␣
↪string
                                   # comparison operations

use locale ':!numeric';            # Use the locale for everything␣
↪but
                                   # numeric-related operations
use locale ':not_numeric';         # Same

no locale;             # Turn off locale handling for the remainder␣
↪of
                       # the scope.
```

DESCRIPTION

This pragma tells the compiler to enable (or disable) the use of POSIX locales for built-in operations (for example, `LC_CTYPE` for regular expressions, `LC_COLLATE` for string comparison, and `LC_NUMERIC` for number formatting). Each `use locale` or `no locale` affects statements to the end of the enclosing BLOCK.

The pragma is documented as part of *perllocale*.

5.2.142 mro

Method Resolution Order - how Perl decides which parent class's method a call inherits.

When a method is called on an object, Perl walks a list of classes looking for a matching subroutine. That list - the *linearized MRO* - is derived from `@ISA` and from the algorithm currently in effect for the class. The `mro` module lets you pick the algorithm, inspect the resulting class list, query the reverse `@ISA` relation, and manage the method cache.

Two algorithms are available. `dfs` (depth-first search) is the default and matches Perl's historical behaviour. `c3` produces the *C3 linearization* familiar from Python, Dylan, and

Raku, which keeps subclasses ahead of their ancestors even under diamond inheritance. Pick `c3` when multiple inheritance makes the `dfs` order surprising; otherwise leave the default alone.

Loading the module also enables `next::method`, `next::can`, and `maybe::next::method` - the C3-aware cousins of `SUPER::`. They always walk the C3 linearization regardless of which MRO the class itself uses.

Synopsis

```
use mro;                  # enables next::method, next::can,␣
↪maybe::next::method

use mro 'c3';             # set C3 linearization for the current␣
↪package
use mro 'dfs';            # set depth-first (Perl's default)␣
↪explicitly

my $order = mro::get_linear_isa('MyClass');   # arrayref of class␣
↪names
my $algo  = mro::get_mro('MyClass');          # "dfs" or "c3"
mro::set_mro('MyClass', 'c3');                # set per-class at␣
↪runtime
```

next:: dispatch

These are mro.pm's own Perl subs, unchanged:

- `$self->next::method(@args)` - jumps (as `goto &sub`) to the next implementation of the current method in the C3 linearization of `$self`'s class, in the caller's context; dies with `No next::method 'NAME' found for CLASS` when there is none.
- `$self->maybe::next::method(@args)` - the same, but returns empty when there is no next method.
- `$self->next::can` - a code reference to the next method, or undef.

All three must be called from inside a named method; elsewhere they die with `next::method/next::can/maybe::next::method must be used in method context`.

This is a native implementation of the XS half of mro 1.30; its `mro.pm` is used unchanged.

Differences from upstream

None.

Functions

MRO selection

set_mro

Set the method-resolution algorithm for `$class` to either `"dfs"` or `"c3"`.

MRO introspection

get_linear_isa

Return the full ordered list of classes Perl would search for a method call on `$class`, starting with `$class` itself.

get_mro

Report which method-resolution algorithm is currently in effect for `$class`.

get_isarev

List every class that inherits from `$class`, directly or transitively, via `@ISA`.

is_universal

Report whether `$class` is `UNIVERSAL` itself or one of its ancestors via `@ISA`.

get_pkg_gen

Return a counter that increments every time a local method in `$class` changes or its local `@ISA` is modified.

Method cache

invalidate_all_method_caches

Invalidate the method cache for every package in the interpreter.

Other Functions

next::method

Tail-call the next implementation of the currently executing method, passing the original arguments.

next::can

Return a code reference to the next implementation of the currently executing method, or `undef` if there is none.

maybe::next::method

Tail-call the next implementation of the currently executing method if one exists; otherwise return quietly.

get_linear_isa

Return the full ordered list of classes Perl would search for a method call on `$class`, starting with `$class` itself.

Synopsis

```
my $list = mro::get_linear_isa($class);
my $list = mro::get_linear_isa($class, $type);
```

The result is an array reference of package names. By default the list is produced using whichever algorithm is active for the class (`dfs` or `c3`). Pass an explicit `"dfs"` or `"c3"` as `$type` to force one without changing the class's setting.

The list always begins with `$class`, even if `$class` has no stash yet - in that case the result is `[ $class ]`. `UNIVERSAL` and its ancestors are not included; they are searched implicitly after the returned list is exhausted. For a class that exists the array is the cached linearization itself, and it is read-only.

Inconsistent C3 hierarchies (conflicting parent orderings across a diamond) croak with `Inconsistent hierarchy during C3 merge of class '...'`. Recursive `@ISA` chains deeper than 100 levels croak with `Recursive inheritance detected in package '...'`. An unknown `$type` croaks with `Invalid mro name: '...'`.

set_mro

Set the method-resolution algorithm for `$class` to either `"dfs"` or `"c3"`.

Synopsis

```
mro::set_mro($class, 'c3');
mro::set_mro($class, 'dfs');
```

After the call, subsequent method dispatches on `$class` - and subclasses that inherit the setting - walk the chosen linearization. `get_linear_isa($class)` reflects the new choice on the next invocation.

A name that is not a registered algorithm croaks with `Invalid mro name: '...'`. Usage with any argument count other than two croaks with `Usage: mro::set_mro(classname, type)`. The stash for `$class` is created on demand if it does not yet exist, so selecting an MRO before the package is declared works.

get_mro

Report which method-resolution algorithm is currently in effect for `$class`.

Synopsis

```
my $algo = mro::get_mro($class);   # "dfs" or "c3"
```

Returns the string `"dfs"` or `"c3"`. Classes that have never called `set_mro` or `use mro '...'` report `"dfs"` (Perl's default). Calling on a non-existent class also reports `"dfs"`; the call does not force the stash into existence.

get_isarev

List every class that inherits from `$class`, directly or transitively, via `@ISA`.

Synopsis

```
my $subclasses = mro::get_isarev($class);   # arrayref
```

Returns an array reference of package names: each class whose linearized MRO contains `$class`. This is the inverse of `get_linear_isa` - given a parent, find its descendants.

`UNIVERSAL` is treated specially: its `isarev` list is not populated with every class in the program, even though every class implicitly inherits from it. Check with `is_universal` rather than `get_isarev` when you need that relationship.

Classes with no known subclasses produce an empty array reference.

is_universal

Report whether `$class` is `UNIVERSAL` itself or one of its ancestors via `@ISA`.

Synopsis

```
if (mro::is_universal($class)) { ... }
```

Returns a true value exactly when every class in the program can inherit methods from `$class`. That happens in two cases: the class literally *is* `UNIVERSAL`, or something has pushed it into `@UNIVERSAL::ISA` (directly or indirectly). Most programs never see the second case, but frameworks that mix a trait into `UNIVERSAL` rely on it.

A single argument is required; calling with any other arity croaks with `Usage: mro::is_universal(classname)`.

invalidate_all_method_caches

Invalidate the method cache for every package in the interpreter.

Synopsis

```
mro::invalidate_all_method_caches();
```

The next method dispatch after the call walks `@ISA` from scratch instead of reusing a cached resolution. Use this when you have globally disturbed the class graph - rebuilt `@ISA` wholesale across many packages, or hot-patched method tables - and cannot name every affected class individually. For a targeted invalidation, prefer `method_changed_in`.

Takes no arguments. Any non-empty call croaks with `Usage: mro::invalidate_all_method_caches()`.

get_pkg_gen

Return a counter that increments every time a local method in `$class` changes or its local `@ISA` is modified.

Synopsis

```
my $gen = mro::get_pkg_gen($class);
```

Useful to class-introspection code that wants a cheap "has anything I care about changed since last time?" check. Store the value, do your work, compare the next value against the stored one. Equal values mean nothing local changed; unequal values mean *something* did, but you still have to investigate what.

Changes in superclasses do **not** bump this number - only edits inside `$class` itself. A stash instantiated for a normal package starts at `1`; `$class` with no stash at all returns `0`. Wiping a stash (`undef %Pkg::`) can reset the counter to `0` or `1` depending on how thorough the wipe was.

next::method

Tail-call the next implementation of the currently executing method, passing the original arguments.

Synopsis

```
sub greet {
    my $self = shift;
    $self->next::method(@_);   # call the parent implementation
}
```

Resolves the next method via the C3 linearization of `$self` and jumps to it as if with `goto &sub`. Control does not return to the caller of `next::method`; the resolved method's return value is the one the original caller sees.

Croaks with `No next::method 'NAME' found for CLASS` when nothing further in the linearization implements the method. Use `maybe::next::method` or `next::can` if the absence of a further implementation is expected.

next::can

Return a code reference to the next implementation of the currently executing method, or `undef` if there is none.

Synopsis

```
my $cv = $self->next::can;
$cv->($self, @args) if $cv;
```

Looks at the method currently on the call stack, walks the C3 linearization of `$self` past its defining package, and returns the first same-named method found further down the list as a CV reference. Unlike `next::method`, finding no such method is not fatal - it returns `undef` so you can decide how to handle the absence.

Must be called inside a method body. Top-level or non-method callers croak with `next::method/next::can/maybe::next::method must be used in method context`.

maybe::next::method

Tail-call the next implementation of the currently executing method if one exists; otherwise return quietly.

Synopsis

```
sub DESTROY {
    my $self = shift;
    ...                         # cleanup
    $self->maybe::next::method;
}
```

Equivalent to `$self->next::method(@_) if $self->next::can` in the common case. The usual reason to reach for it is a `DESTROY` or hook method where missing parent implementations are normal.

It does not work as the target of `goto &maybe::next::method`: after the `goto` the method that did it is no longer on the call stack, so the lookup starts from its caller instead - at top level that croaks with `next::method/next::can/maybe::next::method must be used in method context`.

5.2.143 ok

Alternative to Test::More::use_ok

SYNOPSIS

```
use ok 'Some::Module';
```

DESCRIPTION

With this module, simply change all `use_ok` in test scripts to `use ok`, and they will be executed at `BEGIN` time.

Please see *Test::use::ok* for the full description.

CC0 1.0 Universal

To the extent possible under law, 唐鳳 has waived all copyright and related or neighboring rights to *Test::use::ok*.

This work is published from Taiwan.

https://creativecommons.org/publicdomain/zero/1.0/

5.2.144 open

perl pragma to set default PerlIO layers for input and output

SYNOPSIS

```
use open IN  => ':crlf', OUT => ':raw';
open my $in, '<', 'foo.txt' or die "open failed: $!";
my $line = <$in>; # CRLF translated
close $in;
open my $out, '>', 'bar.txt' or die "open failed: $!";
print $out $line; # no translation of bytes
close $out;

use open OUT => ':encoding(UTF-8)';
use open IN  => ':encoding(iso-8859-7)';

use open IO  => ':locale';

# IO implicit only for :utf8, :encoding, :locale
use open ':encoding(UTF-8)';
use open ':encoding(iso-8859-7)';
use open ':locale';

# with :std, also affect global standard handles
use open ':std', ':encoding(UTF-8)';
use open ':std', OUT => ':encoding(cp1252)';
use open ':std', IO => ':raw :encoding(UTF-16LE)';
```

DESCRIPTION

Full-fledged support for I/O layers is now implemented provided Perl is configured to use PerlIO as its IO system (which has been the default since 5.8, and the only supported configuration since 5.16).

The open pragma serves as one of the interfaces to declare default “layers” (previously known as “disciplines”) for all I/O. Any open(), readpipe() (aka qx//) and similar opera-

tors found within the lexical scope of this pragma will use the declared defaults via the `${^OPEN}` variable.

Layers are specified with a leading colon by convention. You can specify a stack of multiple layers as a space-separated string. See *PerlIO* for more information on the available layers.

With the `IN` subpragma you can declare the default layers of input streams, and with the `OUT` subpragma you can declare the default layers of output streams. With the `IO` subpragma (may be omitted for `:utf8`, `:locale`, or `:encoding`) you can control both input and output streams simultaneously.

When open() is given an explicit list of layers (with the three-arg syntax), they override the list declared using this pragma. open() can also be given a single colon (:) for a layer name, to override this pragma and use the default as detailed in *"Defaults and how to override them" in PerlIO*.

To translate from and to an arbitrary text encoding, use the `:encoding` layer. The matching of encoding names in `:encoding` is loose: case does not matter, and many encodings have several aliases. See Encode::Supported for details and the list of supported locales.

If you want to set your encoding layers based on your locale environment variables, you can use the `:locale` pseudo-layer. For example:

```
$ENV{LANG} = 'ru_RU.KOI8-R';
# the :locale will probe the locale environment variables like LANG
use open OUT => ':locale';
open(my $out, '>', 'koi8') or die "open failed: $!";
print $out chr(0x430); # CYRILLIC SMALL LETTER A = KOI8-R 0xc1
close $out;
open(my $in, '<', 'koi8') or die "open failed: $!";
printf "%#x\n", ord(<$in>); # this should print 0xc1
close $in;
```

The logic of `:locale` is described in full in *"The `:locale` sub-pragma" in encoding*, but in short it is first trying nl_langinfo(CODESET) and then guessing from the LC_ALL and LANG locale environment variables. `:locale` also implicitly turns on `:std`.

`:std` is not a layer but an additional subpragma. When specified in the import list, it activates an additional functionality of pushing the layers selected for input/output handles to the standard filehandles (STDIN, STDOUT, STDERR). If the new layers and existing layer stack both end with an `:encoding` layer, the existing `:encoding` layer will also be removed.

For example, if both input and out are chosen to be `:encoding(UTF-8)`, a `:std` will mean that STDIN, STDOUT, and STDERR will also have `:encoding(UTF-8)` set. On the other hand, if only output is chosen to be in `:encoding(koi8r)`, a `:std` will cause only the STDOUT and STDERR to be in `koi8r`.

The effect of `:std` is not lexical as it modifies the layer stack of the global handles. If you wish to apply only this global effect and not the effect on handles that are opened in that scope, you can isolate the call to this pragma in its own lexical scope.

```
{ use open ':std', IO => ':encoding(UTF-8)' }
```

Before Perl 5.34, `:std` would only apply the first layer provided that is either `:utf8` or has a layer argument, e.g. `:encoding(UTF-8)`. Since Perl 5.34 it will apply the same layer stack it provides to `${^OPEN}`.

IMPLEMENTATION DETAILS

It is called by `import` to validate the layers:

```
PerlIO::Layer::->find("perlio")
```

The return value (if defined) is a Perl object, of class `PerlIO::Layer` which is created by the C code in `perlio.c`. As yet there is nothing useful you can do with the object at the perl level.

SEE ALSO

"binmode" in perlfunc, *"open" in perlfunc*, perlunicode, *PerlIO*, *encoding*

5.2.145 ops

Perl pragma to restrict unsafe operations when compiling

SYNOPSIS

```
perl -Mops=:default ...    # only allow reasonably safe operations

perl -M-ops=system ...     # disable the 'system' opcode
```

DESCRIPTION

Since the ops pragma currently has an irreversible global effect, it is only of significant practical use with the `-M` option on the command line.

See the *Opcode* module for information about opcodes, optags, opmasks and important information about safety.

SEE ALSO

Opcode, *Safe*, perlrun

5.2.146 overload

Package for overloading Perl operations

SYNOPSIS

```
package SomeThing;

use overload
    '+' => \&myadd,
```

(continues on next page)

(continued from previous page)

```
    '-' => \&mysub;
    # etc
...

package main;
$a = SomeThing->new( 57 );
$b = 5 + $a;
...
if (overload::Overloaded $b) {...}
...
$strval = overload::StrVal $b;
```

DESCRIPTION

This pragma allows overloading of Perl's operators for a class. To overload built-in functions, see *"Overriding Built-in Functions" in perlsub* instead.

Fundamentals

Declaration

Arguments of the `use overload` directive are (key, value) pairs. For the full set of legal keys, see "Overloadable Operations" below.

Operator implementations (the values) can be subroutines, references to subroutines, or anonymous subroutines - in other words, anything legal inside a `&{ ... }` call. Values specified as strings are interpreted as method names. Thus

```
package Number;
use overload
    "-" => "minus",
    "*=" => \&muas,
    '""' => sub { ...; };
```

declares that subtraction is to be implemented by method `minus()` in the class `Number` (or one of its base classes), and that the function `Number::muas()` is to be used for the assignment form of multiplication, `*=`. It also defines an anonymous subroutine to implement stringification: this is called whenever an object blessed into the package `Number` is used in a string context (this subroutine might, for example, return the number as a Roman numeral).

Calling Conventions and Magic Autogeneration

The following sample implementation of `minus()` (which assumes that `Number` objects are simply blessed references to scalars) illustrates the calling conventions:

```
package Number;
sub minus {
    my ($self, $other, $swap) = @_;
```

(continues on next page)

(continued from previous page)

```
    my $result = $$self - $other;         # *
    $result = -$result if $swap;
    ref $result ? $result : bless \$result;
}
# * may recurse once - see table below
```

Three arguments are passed to all subroutines specified in the `use overload` directive (with exceptions - see below, particularly "nomethod").

The first of these is the operand providing the overloaded operator implementation - in this case, the object whose `minus()` method is being called.

The second argument is the other operand, or `undef` in the case of a unary operator.

The third argument is set to TRUE if (and only if) the two operands have been swapped. Perl may do this to ensure that the first argument (`$self`) is an object implementing the overloaded operation, in line with general object calling conventions. For example, if `$x` and `$y` are `Numbers`:

```
operation    |   generates a call to
=============|======================
$x - $y      |   minus($x, $y, '')
$x - 7       |   minus($x, 7, '')
7 - $x       |   minus($x, 7, 1)
```

Perl may also use `minus()` to implement other operators which have not been specified in the `use overload` directive, according to the rules for "Magic Autogeneration" described later. For example, the `use overload` above declared no subroutine for any of the operators `--`, neg (the overload key for unary minus), or `-=`. Thus

```
operation    |   generates a call to
=============|======================
-$x          |   minus($x, 0, 1)
$x--         |   minus($x, 1, undef)
$x -= 3      |   minus($x, 3, undef)
```

Note the `undefs`: where autogeneration results in the method for a standard operator which does not change either of its operands, such as `-`, being used to implement an operator which changes the operand ("mutators": here, `--` and `-=`), Perl passes undef as the third argument. This still evaluates as FALSE, consistent with the fact that the operands have not been swapped, but gives the subroutine a chance to alter its behaviour in these cases.

In all the above examples, `minus()` is required only to return the result of the subtraction: Perl takes care of the assignment to $x. In fact, such methods should *not* modify their operands, even if `undef` is passed as the third argument (see "Overloadable Operations").

The same is not true of implementations of `++` and `--`: these are expected to modify their operand. An appropriate implementation of `--` might look like

```
use overload '--' => "decr",
    # ...
sub decr { --${$_[0]}; }
```

If the "bitwise" feature is enabled (see *feature*), a fifth TRUE argument is passed to subroutines handling &, |, ^ and ~. This indicates that the caller is expecting numeric behaviour. The fourth argument will be `undef`, as that position (`$_[3]`) is reserved for use by "nomethod".

Mathemagic, Mutators, and Copy Constructors

The term 'mathemagic' describes the overloaded implementation of mathematical operators. Mathemagical operations raise an issue. Consider the code:

```
$a = $b;
--$a;
```

If `$a` and `$b` are scalars then after these statements

```
$a == $b - 1
```

An object, however, is a reference to blessed data, so if `$a` and `$b` are objects then the assignment `$a = $b` copies only the reference, leaving `$a` and `$b` referring to the same object data. One might therefore expect the operation `--$a` to decrement `$b` as well as `$a`. However, this would not be consistent with how we expect the mathematical operators to work.

Perl resolves this dilemma by transparently calling a copy constructor before calling a method defined to implement a mutator (`--`, `+=`, and so on.). In the above example, when Perl reaches the decrement statement, it makes a copy of the object data in `$a` and assigns to `$a` a reference to the copied data. Only then does it call `decr()`, which alters the copied data, leaving `$b` unchanged. Thus the object metaphor is preserved as far as possible, while mathemagical operations still work according to the arithmetic metaphor.

Note: the preceding paragraph describes what happens when Perl autogenerates the copy constructor for an object based on a scalar. For other cases, see "Copy Constructor".

Overloadable Operations

The complete list of keys that can be specified in the `use overload` directive are given, separated by spaces, in the values of the hash `%overload::ops`:

```
with_assign          => '+ - * / % ** << >> x .',
assign               => '+= -= *= /= %= **= <<= >>= x= .=',
num_comparison       => '< <= > >= == !=',
'3way_comparison'    => '<=> cmp',
str_comparison       => 'lt le gt ge eq ne',
binary               => '& &= | |= ^ ^= &. &.= |. |.= ^. ^.=',
unary                => 'neg ! ~ ~.',
mutators             => '++ --',
```

(continues on next page)

(continued from previous page)

```
func                => 'atan2 cos sin exp abs log sqrt int',
conversion          => 'bool "" 0+ qr',
iterators           => '<>',
filetest            => '-X',
dereferencing       => '${} @{} %{} &{} *{}',
matching            => '~~',
special             => 'nomethod fallback =',
```

Most of the overloadable operators map one-to-one to these keys. Exceptions, including additional overloadable operations not apparent from this hash, are included in the notes which follow. This list is subject to growth over time.

A warning is issued if an attempt is made to register an operator not found above.

- `not`

  The operator `not` is not a valid key for `use overload`. However, if the operator `!` is overloaded then the same implementation will be used for `not` (since the two operators differ only in precedence).

- `neg`

  The key `neg` is used for unary minus to disambiguate it from binary `-`.

- `++`, `--`

  Assuming they are to behave analogously to Perl's `++` and `--`, overloaded implementations of these operators are required to mutate their operands.

  No distinction is made between prefix and postfix forms of the increment and decrement operators: these differ only in the point at which Perl calls the associated subroutine when evaluating an expression.

- *Assignments*

  ```
  +=   -=   *=   /=   %=   **=   <<=   >>=   x=   .=
  &=   |=   ^=   &.=   |.=   ^.=
  ```

  Simple assignment is not overloadable (the `'='` key is used for the "Copy Constructor"). Perl does have a way to make assignments to an object do whatever you want, but this involves using tie(), not overload - see *"tie" in perlfunc* and the "COOKBOOK" examples below.

  The subroutine for the assignment variant of an operator is required only to return the result of the operation. It is permitted to change the value of its operand (this is safe because Perl calls the copy constructor first), but this is optional since Perl assigns the returned value to the left-hand operand anyway.

  An object that overloads an assignment operator does so only in respect of assignments to that object. In other words, Perl never calls the corresponding methods with the third argument (the "swap" argument) set to TRUE. For example, the operation

  ```
  $a *= $b
  ```

cannot lead to $b's implementation of *= being called, even if $a is a scalar. (It can, however, generate a call to $b's method for *).

- *Non-mutators with a mutator variant*

```
+  -  *  /  %  **  <<  >>  x  .
&  |  ^  &.  |.  ^.
```

As described above, Perl may call methods for operators like + and & in the course of implementing missing operations like ++, +=, and &=. While these methods may detect this usage by testing the definedness of the third argument, they should in all cases avoid changing their operands. This is because Perl does not call the copy constructor before invoking these methods.

- `int`

Traditionally, the Perl function `int` rounds to 0 (see *"int" in perlfunc*), and so for floating-point-like types one should follow the same semantic.

- *String, numeric, boolean, and regexp conversions*

```
""  0+  bool
```

These conversions are invoked according to context as necessary. For example, the subroutine for `'""'` (stringify) may be used where the overloaded object is passed as an argument to `print`, and that for `'bool'` where it is tested in the condition of a flow control statement (like `while`) or the ternary `?:` operation.

Of course, in contexts like, for example, `$obj + 1`, Perl will invoke `$obj`'s implementation of + rather than (in this example) converting `$obj` to a number using the numify method `'0+'` (an exception to this is when no method has been provided for `'+'` and "fallback" is set to TRUE).

The subroutines for `'""'`, `'0+'`, and `'bool'` can return any arbitrary Perl value. If the corresponding operation for this value is overloaded too, the operation will be called again with this value.

As a special case if the overload returns the object itself then it will be used directly. An overloaded conversion returning the object is probably a bug, because you're likely to get something that looks like `YourPackage=HASH(0x8172b34)`.

```
qr
```

The subroutine for `'qr'` is used wherever the object is interpolated into or used as a regexp, including when it appears on the RHS of a `=~` or `!~` operator.

`qr` must return a compiled regexp, or a ref to a compiled regexp (such as `qr//` returns), and any further overloading on the return value will be ignored.

- *Iteration*

If `<>` is overloaded then the same implementation is used for both the *read-filehandle* syntax `<$var>` and *globbing* syntax `<${var}>`.

- *File tests*

The key `'-X'` is used to specify a subroutine to handle all the filetest operators (`-f`, `-x`, and so on: see *"-X" in perlfunc* for the full list); it is not possible to overload

any filetest operator individually. To distinguish them, the letter following the '-' is passed as the second argument (that is, in the slot that for binary operators is used to pass the second operand).

Calling an overloaded filetest operator does not affect the stat value associated with the special filehandle _. It still refers to the result of the last `stat`, `lstat` or unoverloaded filetest.

This overload was introduced in Perl 5.12.

- *Matching*

  The key `"~~"` allows you to override the smart matching logic used by the ~~ operator and the switch construct (`given`/`when`). See "Switch Statements" in perlsyn and *feature*.

  Unusually, the overloaded implementation of the smart match operator does not get full control of the smart match behaviour. In particular, in the following code:

```
package Foo;
use overload '~~' => 'match';

my $obj =  Foo->new();
$obj ~~ [ 1,2,3 ];
```

  the smart match does *not* invoke the method call like this:

```
$obj->match([1,2,3],0);
```

  rather, the smart match distributive rule takes precedence, so $obj is smart matched against each array element in turn until a match is found, so you may see between one and three of these calls instead:

```
$obj->match(1,0);
$obj->match(2,0);
$obj->match(3,0);
```

  Consult the match table in *"Smartmatch Operator" in perlop* for details of when overloading is invoked.

- *Dereferencing*

```
${}  @{}  %{}  &{}  *{}
```

  If these operators are not explicitly overloaded then they work in the normal way, yielding the underlying scalar, array, or whatever stores the object data (or the appropriate error message if the dereference operator doesn't match it). Defining a catch-all `'nomethod'` (see below) makes no difference to this as the catch-all function will not be called to implement a missing dereference operator.

  If a dereference operator is overloaded then it must return a *reference* of the appropriate type (for example, the subroutine for key `'${}'` should return a reference to a scalar, not a scalar), or another object which overloads the operator: that is, the subroutine only determines what is dereferenced and the actual dereferencing is left to Perl. As a special case, if the subroutine returns the object itself then it will not be called again - avoiding infinite recursion.

- *Special*

```
nomethod  fallback  =
```

  See “Special Keys for `use overload`”.

Magic Autogeneration

If a method for an operation is not found then Perl tries to autogenerate a substitute implementation from the operations that have been defined.

Note: the behaviour described in this section can be disabled by setting `fallback` to FALSE (see “fallback”).

In the following tables, numbers indicate priority. For example, the table below states that, if no implementation for `'!'` has been defined then Perl will implement it using `'bool'` (that is, by inverting the value returned by the method for `'bool'`); if boolean conversion is also unimplemented then Perl will use `'0+'` or, failing that, `'""'`.

```
operator | can be autogenerated from
         |
         | 0+   ""   bool   .   x
=========|========================
   0+    |       1     2
   ""    |  1          2
   bool  |  1    2
   int   |  1    2     3
   !     |  2    3     1
   qr    |  2    1     3
   .     |  2    1     3
   x     |  2    1     3
   .=    |  3    2     4    1
   x=    |  3    2     4         1
   <>    |  2    1     3
   -X    |  2    1     3
```

Note: The iterator (`'<>'`) and file test (`'-X'`) operators work as normal: if the operand is not a blessed glob or IO reference then it is converted to a string (using the method for `'""'`, `'0+'`, or `'bool'`) to be interpreted as a glob or filename.

```
operator | can be autogenerated from
         |
         |  <   <=>   neg   -=    -
=========|==========================
   neg   |                        1
   -=    |                        1
   --    |                    1   2
   abs   | a1    a2    b1        b2    [*]
   <     |        1
   <=    |        1
   >     |        1
   >=    |        1
```

(continues on next page)

(continued from previous page)

```
   ==     |         1
   !=     |         1

* one from [a1, a2] and one from [b1, b2]
```

Just as numeric comparisons can be autogenerated from the method for `'<=>'`, string comparisons can be autogenerated from that for `'cmp'`:

```
 operators           |  can be autogenerated from
=====================|============================
 lt gt le ge eq ne   |  cmp
```

Similarly, autogeneration for keys `'+='` and `'++'` is analogous to `'-='` and `'--'` above:

```
operator | can be autogenerated from
         |
         |  +=    +
=========|==========================
    +=   |         1
    ++   |   1     2
```

And other assignment variations are analogous to `'+='` and `'-='` (and similar to `'.='` and `'x='` above):

```
          operator ||  *= /= %= **= <<= >>= &= ^= |= &.= ^.= |.=
-------------------||-----------------------------------------
autogenerated from ||  *  /  %  **  <<  >>  &  ^  |  &.  ^.  |.
```

Note also that the copy constructor (key `'='`) may be autogenerated, but only for objects based on scalars. See “Copy Constructor”.

Minimal Set of Overloaded Operations

Since some operations can be automatically generated from others, there is a minimal set of operations that need to be overloaded in order to have the complete set of overloaded operations at one’s disposal. Of course, the autogenerated operations may not do exactly what the user expects. The minimal set is:

```
+ - * / % ** << >> x
<=> cmp
& | ^ ~ &. |. ^. ~.
atan2 cos sin exp log sqrt int
"" 0+ bool
~~
```

Of the conversions, only one of string, boolean or numeric is needed because each can be generated from either of the other two.

Special Keys for `use overload`

`nomethod`

The `'nomethod'` key is used to specify a catch-all function to be called for any operator that is not individually overloaded. The specified function will be passed four parameters. The first three arguments coincide with those that would have been passed to the corresponding method if it had been defined. The fourth argument is the `use overload` key for that missing method. If the "bitwise" feature is enabled (see *feature*), a fifth TRUE argument is passed to subroutines handling `&`, `|`, `^` and `~` to indicate that the caller is expecting numeric behaviour.

For example, if `$a` is an object blessed into a package declaring

```
use overload 'nomethod' => 'catch_all', # ...
```

then the operation

```
3 + $a
```

could (unless a method is specifically declared for the key `'+'`) result in a call

```
catch_all($a, 3, 1, '+')
```

See "How Perl Chooses an Operator Implementation".

`fallback`

The value assigned to the key `'fallback'` tells Perl how hard it should try to find an alternative way to implement a missing operator.

- defined, but FALSE

  ```
  use overload "fallback" => 0, # ... ;
  ```

  This disables "Magic Autogeneration".

- `undef`

  In the default case where no value is explicitly assigned to `fallback`, magic autogeneration is enabled.

- TRUE

  The same as for `undef`, but if a missing operator cannot be autogenerated then, instead of issuing an error message, Perl is allowed to revert to what it would have done for that operator if there had been no `use overload` directive.

  Note: in most cases, particularly the "Copy Constructor", this is unlikely to be appropriate behaviour.

See "How Perl Chooses an Operator Implementation".

Copy Constructor

As mentioned above, this operation is called when a mutator is applied to a reference that shares its object with some other reference. For example, if `$b` is mathemagical, and `'++'` is overloaded with `'incr'`, and `'='` is overloaded with `'clone'`, then the code

```
$a = $b;
# ... (other code which does not modify $a or $b) ...
++$b;
```

would be executed in a manner equivalent to

```
$a = $b;
# ...
$b = $b->clone(undef, "");
$b->incr(undef, "");
```

Note:

- The subroutine for `'='` does not overload the Perl assignment operator: it is used only to allow mutators to work as described here. (See "Assignments" above.)
- As for other operations, the subroutine implementing '=' is passed three arguments, though the last two are always `undef` and `''`.
- The copy constructor is called only before a call to a function declared to implement a mutator, for example, if `++$b;` in the code above is effected via a method declared for key `'++'` (or 'nomethod', passed `'++'` as the fourth argument) or, by autogeneration, `'+='`. It is not called if the increment operation is effected by a call to the method for `'+'` since, in the equivalent code,

  ```
  $a = $b;
  $b = $b + 1;
  ```

  the data referred to by `$a` is unchanged by the assignment to `$b` of a reference to new object data.
- The copy constructor is not called if Perl determines that it is unnecessary because there is no other reference to the data being modified.
- If `'fallback'` is undefined or TRUE then a copy constructor can be autogenerated, but only for objects based on scalars. In other cases it needs to be defined explicitly. Where an object's data is stored as, for example, an array of scalars, the following might be appropriate:

  ```
  use overload '=' => sub { bless [ @{$_[0]} ] },  # ...
  ```

- If `'fallback'` is TRUE and no copy constructor is defined then, for objects not based on scalars, Perl may silently fall back on simple assignment - that is, assignment of the object reference. In effect, this disables the copy constructor mechanism since no new copy of the object data is created. This is almost certainly not what you want. (It is, however, consistent: for example, Perl's fallback for the `++` operator is to increment the reference itself.)

How Perl Chooses an Operator Implementation

Which is checked first, `nomethod` or `fallback`? If the two operands of an operator are of different types and both overload the operator, which implementation is used? The following are the precedence rules:

1. If the first operand has declared a subroutine to overload the operator then use that implementation.
2. Otherwise, if fallback is TRUE or undefined for the first operand then see if the rules for autogeneration allows another of its operators to be used instead.
3. Unless the operator is an assignment (`+=`, `-=`, etc.), repeat step (1) in respect of the second operand.
4. Repeat Step (2) in respect of the second operand.
5. If the first operand has a “nomethod” method then use that.
6. If the second operand has a “nomethod” method then use that.
7. If `fallback` is TRUE for both operands then perform the usual operation for the operator, treating the operands as numbers, strings, or booleans as appropriate for the operator (see note).
8. Nothing worked - die.

Where there is only one operand (or only one operand with overloading) the checks in respect of the other operand above are skipped.

There are exceptions to the above rules for dereference operations (which, if Step 1 fails, always fall back to the normal, built-in implementations - see Dereferencing), and for ~~ (which has its own set of rules - see `Matching` under “Overloadable Operations” above).

Note on Step 7: some operators have a different semantic depending on the type of their operands. As there is no way to instruct Perl to treat the operands as, e.g., numbers instead of strings, the result here may not be what you expect. See “BUGS AND PITFALLS”.

Losing Overloading

The restriction for the comparison operation is that even if, for example, `cmp` should return a blessed reference, the autogenerated `lt` function will produce only a standard logical value based on the numerical value of the result of `cmp`. In particular, a working numeric conversion is needed in this case (possibly expressed in terms of other conversions).

Similarly, `.=` and `x=` operators lose their mathemagical properties if the string conversion substitution is applied.

When you chop() a mathemagical object it is promoted to a string and its mathemagical properties are lost. The same can happen with other operations as well.

Inheritance and Overloading

Overloading respects inheritance via the @ISA hierarchy. Inheritance interacts with overloading in two ways.

- Method names in the `use overload` directive

  If `value` in

  ```
  use overload key => value;
  ```

  is a string, it is interpreted as a method name - which may (in the usual way) be inherited from another class.

- Overloading of an operation is inherited by derived classes

  Any class derived from an overloaded class is also overloaded and inherits its operator implementations. If the same operator is overloaded in more than one ancestor then the implementation is determined by the usual inheritance rules.

  For example, if `A` inherits from `B` and `C` (in that order), `B` overloads `+` with `\&D::plus_sub`, and `C` overloads `+` by `"plus_meth"`, then the subroutine `D::plus_sub` will be called to implement operation `+` for an object in package `A`.

Note that in Perl version prior to 5.18 inheritance of the `fallback` key was not governed by the above rules. The value of `fallback` in the first overloaded ancestor was used. This was fixed in 5.18 to follow the usual rules of inheritance.

Run-time Overloading

Since all `use` directives are executed at compile-time, the only way to change overloading during run-time is to

```
eval 'use overload "+" => \&addmethod';
```

You can also use

```
eval 'no overload "+", "--", "<="';
```

though the use of these constructs during run-time is questionable.

Public Functions

Package `overload.pm` provides the following public functions:

- overload::StrVal(arg)

  Gives the string value of `arg` as in the absence of stringify overloading. If you are using this to get the address of a reference (useful for checking if two references point to the same thing) then you may be better off using `builtin::refaddr()` or `Scalar::Util::refaddr()`, which are faster.

- overload::Overloaded(arg)

  Returns true if `arg` is subject to overloading of some operations.

- overload::Method(obj,op)

  Returns `undef` or a reference to the method that implements op.

Overloading Constants

For some applications, the Perl parser mangles constants too much. It is possible to hook into this process via `overload::constant()` and `overload::remove_constant()` functions.

These functions take a hash as an argument. The recognized keys of this hash are:

- integer

  to overload integer constants,

- float

  to overload floating point constants,

- binary

  to overload octal and hexadecimal constants,

- q

  to overload q-quoted strings, constant pieces of qq- and qx-quoted strings and here-documents,

- qr

  to overload constant pieces of regular expressions.

The corresponding values are references to functions which take three arguments: the first one is the *initial* string form of the constant, the second one is how Perl interprets this constant, the third one is how the constant is used. Note that the initial string form does not contain string delimiters, and has backslashes in backslash-delimiter combinations stripped (thus the value of delimiter is not relevant for processing of this string). The return value of this function is how this constant is going to be interpreted by Perl. The third argument is undefined unless for overloaded `q`- and `qr`- constants, it is `q` in single-quote context (comes from strings, regular expressions, and single-quote HERE documents), it is `tr` for arguments of `tr`/`y` operators, it is `s` for right-hand side of `s`-operator, and it is `qq` otherwise.

Since an expression `"ab$cd,,"` is just a shortcut for `'ab' . $cd . ',,'`, it is expected that overloaded constant strings are equipped with reasonable overloaded catenation operator, otherwise absurd results will result. Similarly, negative numbers are considered as negations of positive constants.

Note that it is probably meaningless to call the functions overload::constant() and overload::remove_constant() from anywhere but import() and unimport() methods. From these methods they may be called as

```
sub import {
    shift;
    return unless @_;
    die "unknown import: @_" unless @_ == 1 and $_[0] eq ':constant
↪';
    overload::constant integer => sub {Math::BigInt->new(shift)};
}
```

IMPLEMENTATION

What follows is subject to change RSN.

The table of methods for all operations is cached in magic for the symbol table hash for the package. The cache is invalidated during processing of `use overload`, `no overload`, new function definitions, and changes in @ISA.

(Every SVish thing has a magic queue, and magic is an entry in that queue. This is how a single variable may participate in multiple forms of magic simultaneously. For instance, environment variables regularly have two forms at once: their %ENV magic and their taint magic. However, the magic which implements overloading is applied to the stashes, which are rarely used directly, thus should not slow down Perl.)

If a package uses overload, it carries a special flag. This flag is also set when new functions are defined or @ISA is modified. There will be a slight speed penalty on the very first operation thereafter that supports overloading, while the overload tables are updated. If there is no overloading present, the flag is turned off. Thus the only speed penalty thereafter is the checking of this flag.

It is expected that arguments to methods that are not explicitly supposed to be changed are constant (but this is not enforced).

COOKBOOK

Please add examples to what follows!

Two-face Scalars

Put this in `two_face.pm` in your Perl library directory:

```
package two_face;               # Scalars with separate string and
                                # numeric values.
sub new { my $p = shift; bless [@_], $p }
use overload '""' => \&str, '0+' => \&num, fallback => 1;
sub num {shift->[1]}
sub str {shift->[0]}
```

Use it as follows:

```
require two_face;
my $seven = two_face->new("vii", 7);
printf "seven=$seven, seven=%d, eight=%d\n", $seven, $seven+1;
print "seven contains 'i'\n" if $seven =~ /i/;
```

(The second line creates a scalar which has both a string value, and a numeric value.) This prints:

```
seven=vii, seven=7, eight=8
seven contains 'i'
```

Two-face References

Suppose you want to create an object which is accessible as both an array reference and a hash reference.

```
package two_refs;
use overload '%{}' => \&gethash, '@{}' => sub { $ {shift()} };
sub new {
    my $p = shift;
    bless \ [@_], $p;
}
sub gethash {
    my %h;
    my $self = shift;
    tie %h, ref $self, $self;
    \%h;
}

sub TIEHASH { my $p = shift; bless \ shift, $p }
my %fields;
my $i = 0;
$fields{$_} = $i++ foreach qw{zero one two three};
sub STORE {
    my $self = ${shift()};
    my $key = $fields{shift()};
    defined $key or die "Out of band access";
    $$self->[$key] = shift;
}
sub FETCH {
    my $self = ${shift()};
    my $key = $fields{shift()};
    defined $key or die "Out of band access";
    $$self->[$key];
}
```

Now one can access an object using both the array and hash syntax:

```
my $bar = two_refs->new(3,4,5,6);
$bar->[2] = 11;
$bar->{two} == 11 or die 'bad hash fetch';
```

Note several important features of this example. First of all, the *actual* type of $bar is a scalar reference, and we do not overload the scalar dereference. Thus we can get the *actual* non-overloaded contents of $bar by just using `$$bar` (what we do in functions which overload dereference). Similarly, the object returned by the TIEHASH() method is a scalar reference.

Second, we create a new tied hash each time the hash syntax is used. This allows us not to worry about a possibility of a reference loop, which would lead to a memory leak.

Both these problems can be cured. Say, if we want to overload hash dereference on a reference to an object which is *implemented* as a hash itself, the only problem one has to circumvent is how to access this *actual* hash (as opposed to the *virtual* hash exhibited

by the overloaded dereference operator). Here is one possible fetching routine:

```
sub access_hash {
    my ($self, $key) = (shift, shift);
    my $class = ref $self;
    bless $self, 'overload::dummy'; # Disable overloading of %{}
    my $out = $self->{$key};
    bless $self, $class;            # Restore overloading
    $out;
}
```

To remove creation of the tied hash on each access, one may an extra level of indirection which allows a non-circular structure of references:

```
package two_refs1;
use overload
    '%{}' => sub { ${shift()}->[1] },
    '@{}' => sub { ${shift()}->[0] };

sub new {
    my $p = shift;
    my $a = [@_];
    my %h;
    tie %h, $p, $a;
    bless \ [$a, \%h], $p;
}
sub gethash {
    my %h;
    my $self = shift;
    tie %h, ref $self, $self;
    \%h;
}

sub TIEHASH { my $p = shift; bless \ shift, $p }
my %fields;
my $i = 0;
$fields{$_} = $i++ foreach qw{zero one two three};
sub STORE {
    my $a = ${shift()};
    my $key = $fields{shift()};
    defined $key or die "Out of band access";
    $a->[$key] = shift;
}
sub FETCH {
    my $a = ${shift()};
    my $key = $fields{shift()};
    defined $key or die "Out of band access";
    $a->[$key];
}
```

Now if $baz is overloaded like this, then `$baz` is a reference to a reference to the intermediate array, which keeps a reference to an actual array, and the access hash. The

tie()ing object for the access hash is a reference to a reference to the actual array, so

- There are no loops of references.
- Both "objects" which are blessed into the class `two_refs1` are references to a reference to an array, thus references to a *scalar*. Thus the accessor expression `$$foo->[$ind]` involves no overloaded operations.

Symbolic Calculator

Put this in `symbolic.pm` in your Perl library directory:

```
package symbolic;            # Primitive symbolic calculator
use overload nomethod => \&wrap;

sub new { shift; bless ['n', @_] }
sub wrap {
    my ($obj, $other, $inv, $meth) = @_;
    ($obj, $other) = ($other, $obj) if $inv;
    bless [$meth, $obj, $other];
}
```

This module is very unusual as overloaded modules go: it does not provide any usual overloaded operators, instead it provides an implementation for `"nomethod"`. In this example the `nomethod` subroutine returns an object which encapsulates operations done over the objects: `symbolic->new(3)` contains `['n', 3]`, `2 + symbolic->new(3)` contains `['+', 2, ['n', 3]]`.

Here is an example of the script which "calculates" the side of circumscribed octagon using the above package:

```
require symbolic;
my $iter = 1;                   # 2**($iter+2) = 8
my $side = symbolic->new(1);
my $cnt = $iter;

while ($cnt--) {
    $side = (sqrt(1 + $side**2) - 1)/$side;
}
print "OK\n";
```

The value of $side is

```
['/', ['-', ['sqrt', ['+', 1, ['**', ['n', 1], 2]],
                     undef], 1], ['n', 1]]
```

Note that while we obtained this value using a nice little script, there is no simple way to *use* this value. In fact this value may be inspected in debugger (see perldebug), but only if `bareStringify` **O**ption is set, and not via p command.

If one attempts to print this value, then the overloaded operator `""` will be called, which will call `nomethod` operator. The result of this operator will be stringified again, but this result is again of type `symbolic`, which will lead to an infinite loop.

Add a pretty-printer method to the module `symbolic.pm`:

```
sub pretty {
    my ($meth, $a, $b) = @{+shift};
    $a = 'u' unless defined $a;
    $b = 'u' unless defined $b;
    $a = $a->pretty if ref $a;
    $b = $b->pretty if ref $b;
    "[$meth $a $b]";
}
```

Now one can finish the script by

```
print "side = ", $side->pretty, "\n";
```

The method `pretty` is doing object-to-string conversion, so it is natural to overload the operator `""` using this method. However, inside such a method it is not necessary to pretty-print the *components* `$a` and `$b` of an object. In the above subroutine `"[$meth $a $b]"` is a catenation of some strings and components `$a` and `$b`. If these components use overloading, the catenation operator will look for an overloaded operator `.`; if not present, it will look for an overloaded operator `""`. Thus it is enough to use

```
use overload nomethod => \&wrap, '""' => \&str;
sub str {
    my ($meth, $a, $b) = @{+shift};
    $a = 'u' unless defined $a;
    $b = 'u' unless defined $b;
    "[$meth $a $b]";
}
```

Now one can change the last line of the script to

```
print "side = $side\n";
```

which outputs

```
side = [/ [- [sqrt [+ 1 [** [n 1 u] 2]] u] 1] [n 1 u]]
```

and one can inspect the value in debugger using all the possible methods.

Something is still amiss: consider the loop variable `$cnt` of the script. It was a number, not an object. We cannot make this value of type `symbolic`, since then the loop will not terminate.

Indeed, to terminate the cycle, the `$cnt` should become false. However, the operator `bool` for checking falsity is overloaded (this time via overloaded `""`), and returns a long string, thus any object of type `symbolic` is true. To overcome this, we need a way to compare an object to 0. In fact, it is easier to write a numeric conversion routine.

Here is the text of `symbolic.pm` with such a routine added (and slightly modified str()):

```
package symbolic;           # Primitive symbolic calculator
use overload
```

(continues on next page)

(continued from previous page)

```
    nomethod => \&wrap, '"' => \&str, '0+' => \&num;

sub new { shift; bless ['n', @_] }
sub wrap {
    my ($obj, $other, $inv, $meth) = @_;
    ($obj, $other) = ($other, $obj) if $inv;
    bless [$meth, $obj, $other];
}
sub str {
    my ($meth, $a, $b) = @{+shift};
    $a = 'u' unless defined $a;
    if (defined $b) {
        "[$meth $a $b]";
    } else {
        "[$meth $a]";
    }
}
my %subr = (
    n => sub {$_[0]},
    sqrt => sub {sqrt $_[0]},
    '-' => sub {shift() - shift()},
    '+' => sub {shift() + shift()},
    '/' => sub {shift() / shift()},
    '*' => sub {shift() * shift()},
    '**' => sub {shift() ** shift()},
);
sub num {
    my ($meth, $a, $b) = @{+shift};
    my $subr = $subr{$meth}
    or die "Do not know how to ($meth) in symbolic";
    $a = $a->num if ref $a eq __PACKAGE__;
    $b = $b->num if ref $b eq __PACKAGE__;
    $subr->($a,$b);
}
```

All the work of numeric conversion is done in %subr and num(). Of course, %subr is not complete, it contains only operators used in the example below. Here is the extra-credit question: why do we need an explicit recursion in num()? (Answer is at the end of this section.)

Use this module like this:

```
require symbolic;
my $iter = symbolic->new(2);        # 16-gon
my $side = symbolic->new(1);
my $cnt = $iter;

while ($cnt) {
    $cnt = $cnt - 1;                # Mutator '--' not implemented
    $side = (sqrt(1 + $side**2) - 1)/$side;
```

(continues on next page)

(continued from previous page)

```
}
printf "%s=%f\n", $side, $side;
printf "pi=%f\n", $side*(2**($iter+2));
```

It prints (without so many line breaks)

```
[/ [- [sqrt [+ 1 [** [/ [- [sqrt [+ 1 [** [n 1] 2]]] 1]
                        [n 1]] 2]]] 1]
[/ [- [sqrt [+ 1 [** [n 1] 2]]] 1] [n 1]]]=0.198912
pi=3.182598
```

The above module is very primitive. It does not implement mutator methods (++, -= and so on), does not do deep copying (not required without mutators!), and implements only those arithmetic operations which are used in the example.

To implement most arithmetic operations is easy; one should just use the tables of operations, and change the code which fills %subr to

```
my %subr = ( 'n' => sub {$_[0]} );
foreach my $op (split " ", $overload::ops{with_assign}) {
    $subr{$op} = $subr{"$op="} = eval "sub {shift() $op shift()}";
}
my @bins = qw(binary 3way_comparison num_comparison str_comparison);
foreach my $op (split " ", "@overload::ops{ @bins }") {
    $subr{$op} = eval "sub {shift() $op shift()}";
}
foreach my $op (split " ", "@overload::ops{qw(unary func)}") {
    print "defining '$op'\n";
    $subr{$op} = eval "sub {$op shift()}";
}
```

Since subroutines implementing assignment operators are not required to modify their operands (see “Overloadable Operations” above), we do not need anything special to make += and friends work, besides adding these operators to %subr and defining a copy constructor (needed since Perl has no way to know that the implementation of '+=' does not mutate the argument - see “Copy Constructor”).

To implement a copy constructor, add '=' => \&cpy to use overload line, and code (this code assumes that mutators change things one level deep only, so recursive copying is not needed):

```
sub cpy {
    my $self = shift;
    bless [@$self], ref $self;
}
```

To make ++ and -- work, we need to implement actual mutators, either directly, or in nomethod. We continue to do things inside nomethod, thus add

```
if ($meth eq '++' or $meth eq '--') {
    @$obj = ($meth, (bless [@$obj]), 1); # Avoid circular reference
```

(continues on next page)

(continued from previous page)

```
    return $obj;
}
```

after the first line of wrap(). This is not a most effective implementation, one may consider

```
sub inc { $_[0] = bless ['++', shift, 1]; }
```

instead.

As a final remark, note that one can fill %subr by

```
my %subr = ( 'n' => sub {$_[0]} );
foreach my $op (split " ", $overload::ops{with_assign}) {
    $subr{$op} = $subr{"$op="} = eval "sub {shift() $op shift()}";
}
my @bins = qw(binary 3way_comparison num_comparison str_comparison);
foreach my $op (split " ", "@overload::ops{ @bins }") {
    $subr{$op} = eval "sub {shift() $op shift()}";
}
foreach my $op (split " ", "@overload::ops{qw(unary func)}") {
    $subr{$op} = eval "sub {$op shift()}";
}
$subr{'++'} = $subr{'+'};
$subr{'--'} = $subr{'-'};
```

This finishes implementation of a primitive symbolic calculator in 50 lines of Perl code. Since the numeric values of subexpressions are not cached, the calculator is very slow.

Here is the answer for the exercise: In the case of str(), we need no explicit recursion since the overloaded `.`-operator will fall back to an existing overloaded operator `""`. Overloaded arithmetic operators *do not* fall back to numeric conversion if `fallback` is not explicitly requested. Thus without an explicit recursion num() would convert `['+', $a, $b]` to `$a + $b`, which would just rebuild the argument of num().

If you wonder why defaults for conversion are different for str() and num(), note how easy it was to write the symbolic calculator. This simplicity is due to an appropriate choice of defaults. One extra note: due to the explicit recursion num() is more fragile than sym(): we need to explicitly check for the type of $a and $b. If components $a and $b happen to be of some related type, this may lead to problems.

Really Symbolic Calculator

One may wonder why we call the above calculator symbolic. The reason is that the actual calculation of the value of expression is postponed until the value is *used*.

To see it in action, add a method

```
sub STORE {
    my $obj = shift;
    $#$obj = 1;
```

(continues on next page)

(continued from previous page)

```
    @$obj->[0,1] = ('=', shift);
}
```

to the package `symbolic`. After this change one can do

```
my $a = symbolic->new(3);
my $b = symbolic->new(4);
my $c = sqrt($a**2 + $b**2);
```

and the numeric value of $c becomes 5. However, after calling

```
$a->STORE(12);  $b->STORE(5);
```

the numeric value of $c becomes 13. There is no doubt now that the module symbolic provides a *symbolic* calculator indeed.

To hide the rough edges under the hood, provide a tie()d interface to the package `symbolic`. Add methods

```
sub TIESCALAR { my $pack = shift; $pack->new(@_) }
sub FETCH { shift }
sub nop {  }                    # Around a bug
```

(the bug, fixed in Perl 5.14, is described in "BUGS"). One can use this new interface as

```
tie $a, 'symbolic', 3;
tie $b, 'symbolic', 4;
$a->nop;  $b->nop;              # Around a bug

my $c = sqrt($a**2 + $b**2);
```

Now numeric value of $c is 5. After `$a = 12; $b = 5` the numeric value of $c becomes 13. To insulate the user of the module add a method

```
sub vars { my $p = shift; tie($_, $p), $_->nop foreach @_; }
```

Now

```
my ($a, $b);
symbolic->vars($a, $b);
my $c = sqrt($a**2 + $b**2);

$a = 3; $b = 4;
printf "c5  %s=%f\n", $c, $c;

$a = 12; $b = 5;
printf "c13  %s=%f\n", $c, $c;
```

shows that the numeric value of $c follows changes to the values of $a and $b.

AUTHOR

Ilya Zakharevich <ilya@math.mps.ohio-state.edu>.

SEE ALSO

The `overloading` pragma can be used to enable or disable overloaded operations within a lexical scope - see *overloading*.

DIAGNOSTICS

When Perl is run with the **-Do** switch or its equivalent, overloading induces diagnostic messages.

Using the `m` command of Perl debugger (see perldebug) one can deduce which operations are overloaded (and which ancestor triggers this overloading). Say, if `eq` is overloaded, then the method `(eq` is shown by debugger. The method `()` corresponds to the `fallback` key (in fact a presence of this method shows that this package has overloading enabled, and it is what is used by the `Overloaded` function of module `overload`).

The module might issue the following warnings:

- Odd number of arguments for overload::constant

  (W) The call to overload::constant contained an odd number of arguments. The arguments should come in pairs.

- '%s' is not an overloadable type

  (W) You tried to overload a constant type the overload package is unaware of.

- '%s' is not a code reference

  (W) The second (fourth, sixth, ...) argument of overload::constant needs to be a code reference. Either an anonymous subroutine, or a reference to a subroutine.

- overload arg '%s' is invalid

  (W) `use overload` was passed an argument it did not recognize. Did you mistype an operator?

BUGS AND PITFALLS

- A pitfall when fallback is TRUE and Perl resorts to a built-in implementation of an operator is that some operators have more than one semantic, for example `|`:

```
use overload '0+' => sub { $_[0]->{n}; },
    fallback => 1;
my $x = bless { n => 4 }, "main";
my $y = bless { n => 8 }, "main";
print $x | $y, "\n";
```

  You might expect this to output "12". In fact, it prints "<": the ASCII result of treating "|" as a bitwise string operator - that is, the result of treating the operands as the strings "4" and "8" rather than numbers. The fact that numify (`0+`) is implemented but stringify (`""`) isn't makes no difference since the latter is simply autogenerated from the former.

The only way to change this is to provide your own subroutine for `'|'`.

- Magic autogeneration increases the potential for inadvertently creating self-referential structures. Currently Perl will not free self-referential structures until cycles are explicitly broken. For example,

```
use overload '+' => 'add';
sub add { bless [ \$_[0], \$_[1] ] };
```

is asking for trouble, since

```
$obj += $y;
```

will effectively become

```
$obj = add($obj, $y, undef);
```

with the same result as

```
$obj = [\$obj, \$foo];
```

Even if no *explicit* assignment-variants of operators are present in the script, they may be generated by the optimizer. For example,

```
"obj = $obj\n"
```

may be optimized to

```
my $tmp = 'obj = ' . $obj;  $tmp .= "\n";
```

- The symbol table is filled with names looking like line-noise.
- This bug was fixed in Perl 5.18, but may still trip you up if you are using older versions:

  For the purpose of inheritance every overloaded package behaves as if `fallback` is present (possibly undefined). This may create interesting effects if some package is not overloaded, but inherits from two overloaded packages.

- Before Perl 5.14, the relation between overloading and tie()ing was broken. Overloading was triggered or not based on the *previous* class of the tie()d variable.

  This happened because the presence of overloading was checked too early, before any tie()d access was attempted. If the class of the value FETCH()ed from the tied variable does not change, a simple workaround for code that is to run on older Perl versions is to access the value (via `() = $foo` or some such) immediately after tie()ing, so that after this call the *previous* class coincides with the current one.

- Barewords are not covered by overloaded string constants.
- The range operator `..` cannot be overloaded.

5.2.147 overloading

perl pragma to lexically control overloading

SYNOPSIS

```
{
    no overloading;
    my $str = "$object"; # doesn't call stringification overload
}

# it's lexical, so this stringifies:
warn "$object";

# it can be enabled per op
no overloading qw("");
warn "$object";

# and also reenabled
use overloading;
```

DESCRIPTION

This pragma allows you to lexically disable or enable overloading.

- `no overloading`

  Disables overloading entirely in the current lexical scope.

- `no overloading @ops`

  Disables only specific overloads in the current lexical scope.

- `use overloading`

  Reenables overloading in the current lexical scope.

- `use overloading @ops`

  Reenables overloading only for specific ops in the current lexical scope.

5.2.148 parent

Establish an ISA relationship with base classes at compile time

SYNOPSIS

```
package Baz;
use parent qw(Foo Bar);
```

DESCRIPTION

Allows you to both load one or more modules, while setting up inheritance from those modules at the same time. Mostly similar in effect to

```
package Baz;
BEGIN {
    require Foo;
    require Bar;
    push @ISA, qw(Foo Bar);
}
```

By default, every base class needs to live in a file of its own. If you want to have a subclass and its parent class in the same file, you can tell `parent` not to load any modules by using the `-norequire` switch:

```
package Foo;
sub exclaim { "I CAN HAS PERL" }

package DoesNotLoadFooBar;
use parent -norequire, 'Foo', 'Bar';
# will not go looking for Foo.pm or Bar.pm
```

This is equivalent to the following code:

```
package Foo;
sub exclaim { "I CAN HAS PERL" }

package DoesNotLoadFooBar;
push @DoesNotLoadFooBar::ISA, 'Foo', 'Bar';
```

This is also helpful for the case where a package lives within a differently named file:

```
package MyHash;
use Tie::Hash;
use parent -norequire, 'Tie::StdHash';
```

This is equivalent to the following code:

```
package MyHash;
require Tie::Hash;
push @ISA, 'Tie::StdHash';
```

If you want to load a subclass from a file that `require` would not consider an eligible filename (that is, it does not end in either `.pm` or `.pmc`), use the following code:

```
package MySecondPlugin;
require './plugins/custom.plugin'; # contains Plugin::Custom
use parent -norequire, 'Plugin::Custom';
```

HISTORY

This module was forked from *base* to remove the cruft that had accumulated in it.

CAVEATS

SEE ALSO

- *base*
- parent::versioned

  A fork of *parent* that provides version checking in parent class modules.

AUTHORS AND CONTRIBUTORS

Rafaël Garcia-Suarez, Bart Lateur, Max Maischein, Anno Siegel, Michael Schwern

MAINTAINER

Max Maischein `corion@cpan.org`

Copyright (c) 2007-2017 Max Maischein <`corion@cpan.org`> Based on the idea of `base.pm`, which was introduced with Perl 5.004_04.

LICENSE

This module is released under the same terms as Perl itself.

5.2.149 perlfaq

Frequently asked questions about Perl

VERSION

version 5.20250619

DESCRIPTION

The perlfaq comprises several documents that answer the most commonly asked questions about Perl and Perl programming. It's divided by topic into nine major sections outlined in this document.

Where to find the perlfaq

The perlfaq is an evolving document. Read the latest version at https://perldoc.perl.org/perlfaq. It is also included in the standard Perl distribution.

How to contribute to the perlfaq

Review https://github.com/perl-doc-cats/perlfaq/wiki. If you don't find your suggestion create an issue or pull request against https://github.com/perl-doc-cats/perlfaq.

Once approved, changes will be distributed with the next Perl release and subsequently appear at https://perldoc.perl.org/perlfaq.

What if my question isn't answered in the FAQ?

Try the resources in perlfaq2.

TABLE OF CONTENTS

- perlfaq1 - General Questions About Perl
- perlfaq2 - Obtaining and Learning about Perl
- perlfaq3 - Programming Tools
- perlfaq4 - Data Manipulation
- perlfaq5 - Files and Formats
- perlfaq6 - Regular Expressions
- perlfaq7 - General Perl Language Issues
- perlfaq8 - System Interaction
- perlfaq9 - Web, Email and Networking

THE QUESTIONS

perlfaq1: General Questions About Perl

This section of the FAQ answers very general, high-level questions about Perl.

- What is Perl?
- Who supports Perl? Who develops it? Why is it free?
- Which version of Perl should I use?
- What is Raku (Perl 6)?
- How stable is Perl?
- How often are new versions of Perl released?
- Is Perl difficult to learn?
- How does Perl compare with other languages like Java, Python, REXX, Scheme, or Tcl?
- Can I do [task] in Perl?
- When shouldn't I program in Perl?
- What's the difference between "perl" and "Perl"?
- What is a JAPH?

- How can I convince others to use Perl?

perlfaq2: Obtaining and Learning about Perl

This section of the FAQ answers questions about where to find source and documentation for Perl, support, and related matters.

- What machines support Perl? Where do I get it?
- How can I get a binary version of Perl?
- I don't have a C compiler. How can I build my own Perl interpreter?
- I copied the Perl binary from one machine to another, but scripts don't work.
- I grabbed the sources and tried to compile but gdbm/dynamic loading/malloc/linking/... failed. How do I make it work?
- What modules and extensions are available for Perl? What is CPAN?
- Where can I get information on Perl?
- What is perl.com? Perl Mongers? pm.org? perl.org? cpan.org?
- Where can I post questions?
- Perl Books
- Which magazines have Perl content?
- Which Perl blogs should I read?
- What mailing lists are there for Perl?
- Where can I buy a commercial version of Perl?
- Where do I send bug reports?

perlfaq3: Programming Tools

This section of the FAQ answers questions related to programmer tools and programming support.

- How do I do (anything)?
- How can I use Perl interactively?
- How do I find which modules are installed on my system?
- How do I debug my Perl programs?
- How do I profile my Perl programs?
- How do I cross-reference my Perl programs?
- Is there a pretty-printer (formatter) for Perl?
- Is there an IDE or Windows Perl Editor?
- Where can I get Perl macros for vi?
- Where can I get perl-mode or cperl-mode for emacs?
- How can I use curses with Perl?

- How can I write a GUI (X, Tk, Gtk, etc.) in Perl?
- How can I make my Perl program run faster?
- How can I make my Perl program take less memory?
- Is it safe to return a reference to local or lexical data?
- How can I free an array or hash so my program shrinks?
- How can I make my CGI script more efficient?
- How can I hide the source for my Perl program?
- How can I compile my Perl program into byte code or C?
- How can I get `#!perl` to work on [MS-DOS,NT,...]?
- Can I write useful Perl programs on the command line?
- Why don't Perl one-liners work on my DOS/Mac/VMS system?
- Where can I learn about CGI or Web programming in Perl?
- Where can I learn about object-oriented Perl programming?
- Where can I learn about linking C with Perl?
- I've read perlembed, perlguts, etc., but I can't embed perl in my C program; what am I doing wrong?
- When I tried to run my script, I got this message. What does it mean?
- What's MakeMaker?

perlfaq4: Data Manipulation

This section of the FAQ answers questions related to manipulating numbers, dates, strings, arrays, hashes, and miscellaneous data issues.

- Why am I getting long decimals (eg, 19.9499999999999) instead of the numbers I should be getting (eg, 19.95)? - Why is int() broken? - Why isn't my octal data interpreted correctly? - Does Perl have a round() function? What about ceil() and floor()? Trig functions? - How do I convert between numeric representations/bases/radixes? - Why doesn't & work the way I want it to? - How do I multiply matrices? - How do I perform an operation on a series of integers? - How can I output Roman numerals? - Why aren't my random numbers random? - How do I get a random number between X and Y? - How do I find the day or week of the year? - How do I find the current century or millennium? - How can I compare two dates and find the difference? - How can I take a string and turn it into epoch seconds? - How can I find the Julian Day? - How do I find yesterday's date? - Does Perl have a Year 2000 or 2038 problem? Is Perl Y2K compliant? - How do I validate input? - How do I unescape a string? - How do I remove consecutive pairs of characters? - How do I expand function calls in a string? - How do I find matching/nesting anything? - How do I reverse a string? - How do I expand tabs in a string? - How do I reformat a paragraph? - How can I access or change N characters of a string? - How do I change the Nth occurrence of something? - How can I count the number of occurrences of a substring within a string? - How do I capitalize all the words on one line? - How can I split a [character]-delimited string except when inside [character]? - How do I strip blank space from the beginning/end of a string? -

How do I pad a string with blanks or pad a number with zeroes? - How do I extract selected columns from a string? - How do I find the soundex value of a string? - How can I expand variables in text strings? - Does Perl have anything like Ruby's #{} or Python's f string? - What's wrong with always quoting "$vars"? - Why don't my <<HERE documents work? - What is the difference between a list and an array? - What is the difference between $array[1] and @array[1]? - How can I remove duplicate elements from a list or array? - How can I tell whether a certain element is contained in a list or array? - How do I compute the difference of two arrays? How do I compute the intersection of two arrays? - How do I test whether two arrays or hashes are equal? - How do I find the first array element for which a condition is true? - How do I handle linked lists? - How do I handle circular lists? - How do I shuffle an array randomly? - How do I process/modify each element of an array? - How do I select a random element from an array? - How do I permute N elements of a list? - How do I sort an array by (anything)? - How do I manipulate arrays of bits? - Why does defined() return true on empty arrays and hashes? - How do I process an entire hash? - How do I merge two hashes? - What happens if I add or remove keys from a hash while iterating over it? - How do I look up a hash element by value? - How can I know how many entries are in a hash? - How do I sort a hash (optionally by value instead of key)? - How can I always keep my hash sorted? - What's the difference between "delete" and "undef" with hashes? - Why don't my tied hashes make the defined/exists distinction? - How do I reset an each() operation part-way through? - How can I get the unique keys from two hashes? - How can I store a multidimensional array in a DBM file? - How can I make my hash remember the order I put elements into it? - Why does passing a subroutine an undefined element in a hash create it? - How can I make the Perl equivalent of a C structure/C++ class/hash or array of hashes or arrays? - How can I use a reference as a hash key? - How can I check if a key exists in a multilevel hash? - How can I prevent addition of unwanted keys into a hash? - How do I handle binary data correctly? - How do I determine whether a scalar is a number/whole/integer/float? - How do I keep persistent data across program calls? - How do I print out or copy a recursive data structure? - How do I define methods for every class/object? - How do I verify a credit card checksum?

perlfaq5: Files and Formats

This section deals with I/O and the "f" issues: filehandles, flushing, formats, and footers.

- How do I flush/unbuffer an output filehandle? Why must I do this?
- How do I change, delete, or insert a line in a file, or append to the beginning of a file?
- How do I count the number of lines in a file?
- How do I delete the last N lines from a file?
- How can I use Perl's `-i` option from within a program?
- How can I copy a file?
- How do I make a temporary file name?
- How can I manipulate fixed-record-length files?
- How can I make a filehandle local to a subroutine? How do I pass filehandles between subroutines? How do I make an array of filehandles?

- How can I use a filehandle indirectly?
- How can I open a filehandle to a string?
- How can I set up a footer format to be used with write()?
- How can I write() into a string?
- How can I output my numbers with commas added?
- How can I translate tildes (~) in a filename?
- When I open a file read-write, why does it wipe it out?
- Why do I sometimes get an “Argument list too long” when I use <*>?
- How can I open a file named with a leading “>” or trailing blanks?
- How can I reliably rename a file?
- How can I lock a file?
- Why can’t I just open(FH, “>file.lock”)?
- I still don’t get locking. I just want to increment the number in the file. How can I do this?
- All I want to do is append a small amount of text to the end of a file. Do I still have to use locking?
- How do I randomly update a binary file?
- How do I get a file’s timestamp in perl?
- How do I set a file’s timestamp in perl?
- How do I print to more than one file at once?
- How can I read in an entire file all at once?
- How can I read in a file by paragraphs?
- How can I read a single character from a file? From the keyboard?
- How can I tell whether there’s a character waiting on a filehandle?
- How do I do a `tail -f` in perl?
- How do I dup() a filehandle in Perl?
- How do I close a file descriptor by number?
- Why can’t I use “C:\temp\foo” in DOS paths? Why doesn’t `C:\temp\foo.exe` work?
- Why doesn’t glob(“*.*”) get all the files?
- Why does Perl let me delete read-only files? Why does `-i` clobber protected files? Isn’t this a bug in Perl?
- How do I select a random line from a file?
- Why do I get weird spaces when I print an array of lines?
- How do I traverse a directory tree?
- How do I delete a directory tree?

- How do I copy an entire directory?

perlfaq6: Regular Expressions

This section is surprisingly small because the rest of the FAQ is littered with answers involving regular expressions. For example, decoding a URL and checking whether something is a number can be handled with regular expressions, but those answers are found elsewhere in this document (in perlfaq9 : "How do I decode or create those %-encodings on the web" and perlfaq4 : "How do I determine whether a scalar is a number/whole/integer/float", to be precise).

- How can I hope to use regular expressions without creating illegible and unmaintainable code?
- I'm having trouble matching over more than one line. What's wrong?
- How can I pull out lines between two patterns that are themselves on different lines?
- How do I match XML, HTML, or other nasty, ugly things with a regex?
- I put a regular expression into $/ but it didn't work. What's wrong?
- How do I substitute case-insensitively on the LHS while preserving case on the RHS?
- How can I make `\w` match national character sets?
- How can I match a locale-smart version of `/[a-zA-Z]/` ?
- How can I quote a variable to use in a regex?
- What is `/o` really for?
- How do I use a regular expression to strip C-style comments from a file?
- Can I use Perl regular expressions to match balanced text?
- What does it mean that regexes are greedy? How can I get around it?
- How do I process each word on each line?
- How can I print out a word-frequency or line-frequency summary?
- How can I do approximate matching?
- How do I efficiently match many regular expressions at once?
- Why don't word-boundary searches with `\b` work for me?
- Why does using $&, $`, or $' slow my program down?
- What good is `\G` in a regular expression?
- Are Perl regexes DFAs or NFAs? Are they POSIX compliant?
- What's wrong with using grep in a void context?
- How can I match strings with multibyte characters?
- How do I match a regular expression that's in a variable?

perlfaq7: General Perl Language Issues

This section deals with general Perl language issues that don't clearly fit into any of the other sections.

- Can I get a BNF/yacc/RE for the Perl language? - What are all these $@%&* punctuation signs, and how do I know when to use them? - Do I always/never have to quote my strings or use semicolons and commas? - How do I skip some return values? - How do I temporarily block warnings? - What's an extension? - Why do Perl operators have different precedence than C operators? - How do I declare/create a structure? - How do I create a module? - How do I adopt or take over a module already on CPAN? - How do I create a class? - How can I tell if a variable is tainted? - What's a closure? - What is variable suicide and how can I prevent it? - How can I pass/return a {Function, FileHandle, Array, Hash, Method, Regex}? - How do I create a static variable? - What's the difference between dynamic and lexical (static) scoping? Between local() and my()? - How can I access a dynamic variable while a similarly named lexical is in scope? - What's the difference between deep and shallow binding? - Why doesn't "my($foo) = <$fh>;" work right? - How do I redefine a builtin function, operator, or method? - What's the difference between calling a function as &foo and foo()? - How do I create a switch or case statement? - How can I catch accesses to undefined variables, functions, or methods? - Why can't a method included in this same file be found? - How can I find out my current or calling package? - How can I comment out a large block of Perl code? - How do I clear a package? - How can I use a variable as a variable name? - What does "bad interpreter" mean?

perlfaq8: System Interaction

This section of the Perl FAQ covers questions involving operating system interaction. Topics include interprocess communication (IPC), control over the user-interface (keyboard, screen and pointing devices), and most anything else not related to data manipulation.

- How do I find out which operating system I'm running under?
- Why does exec() not return?
- How do I do fancy stuff with the keyboard/screen/mouse?
- How do I print something out in color?
- How do I read just one key without waiting for a return key?
- How do I check whether input is ready on the keyboard?
- How do I clear the screen?
- How do I get the screen size?
- How do I ask the user for a password?
- How do I read and write the serial port?
- How do I decode encrypted password files?
- How do I start a process in the background?
- How do I trap control characters/signals?

- How do I modify the shadow password file on a Unix system?
- How do I set the time and date?
- How can I sleep() or alarm() for under a second?
- How can I measure time under a second?
- How can I do an atexit() or setjmp()/longjmp()? (Exception handling)
- Why doesn't my sockets program work under System V (Solaris)? What does the error message "Protocol not supported" mean?
- How can I call my system's unique C functions from Perl?
- Where do I get the include files to do ioctl() or syscall()?
- Why do setuid perl scripts complain about kernel problems?
- How can I open a pipe both to and from a command?
- Why can't I get the output of a command with system()?
- How can I capture STDERR from an external command?
- Why doesn't open() return an error when a pipe open fails?
- What's wrong with using backticks in a void context?
- How can I call backticks without shell processing?
- Why can't my script read from STDIN after I gave it EOF (^D on Unix, ^Z on MS-DOS)?
- How can I convert my shell script to perl?
- Can I use perl to run a telnet or ftp session?
- How can I write expect in Perl?
- Is there a way to hide perl's command line from programs such as "ps"?
- I {changed directory, modified my environment} in a perl script. How come the change disappeared when I exited the script? How do I get my changes to be visible?
- How do I close a process's filehandle without waiting for it to complete?
- How do I fork a daemon process?
- How do I find out if I'm running interactively or not?
- How do I timeout a slow event?
- How do I set CPU limits?
- How do I avoid zombies on a Unix system?
- How do I use an SQL database?
- How do I make a system() exit on control-C?
- How do I open a file without blocking?
- How do I tell the difference between errors from the shell and perl?
- How do I install a module from CPAN?

- What's the difference between require and use?
- How do I keep my own module/library directory?
- How do I add the directory my program lives in to the module/library search path?
- How do I add a directory to my include path (@INC) at runtime?
- Where are modules installed?
- What is socket.ph and where do I get it?

perlfaq9: Web, Email and Networking

This section deals with questions related to running web sites, sending and receiving email as well as general networking.

- Should I use a web framework?
- Which web framework should I use?
- What is Plack and PSGI?
- How do I remove HTML from a string?
- How do I extract URLs?
- How do I fetch an HTML file?
- How do I automate an HTML form submission?
- How do I decode or create those %-encodings on the web?
- How do I redirect to another page?
- How do I put a password on my web pages?
- How do I make sure users can't enter values into a form that causes my CGI script to do bad things?
- How do I parse a mail header?
- How do I check a valid mail address?
- How do I decode a MIME/BASE64 string?
- How do I find the user's mail address?
- How do I send email?
- How do I use MIME to make an attachment to a mail message?
- How do I read email?
- How do I find out my hostname, domainname, or IP address?
- How do I fetch/put an (S)FTP file?
- How can I do RPC in Perl?

CREDITS

Tom Christiansen wrote the original perlfaq then expanded it with the help of Nat Torkington. brian d foy substantially edited and expanded the perlfaq. perlfaq-workers and others have also supplied feedback, patches and corrections over the years.

AUTHOR AND COPYRIGHT

Tom Christiansen wrote the original version of this document. brian d foy <briandfoy@pobox.com> wrote this version. See the individual perlfaq documents for additional copyright information.

This document is available under the same terms as Perl itself. Code examples in all the perlfaq documents are in the public domain. Use them as you see fit (and at your own risk with no warranty from anyone).

5.2.150 re

The `re` pragma and regular-expression introspection module.

Two jobs, one module:

- **As a pragma** - `use re` turns on lexically scoped regex behaviour: compile-time and run-time debug output (`use re 'debug'`, `use re 'debugcolor'`, `use re qw(Debug ...)`), permission to run code inside patterns derived from interpolated variables (`use re 'eval'`), taint propagation (`use re 'taint'`), stricter pattern checking (`use re 'strict'`), and default pattern flags (`use re '/ix'`).
- **As an introspection API** - when imported, it exposes a small set of utility functions for inspecting compiled patterns (`qr//` objects) and the named-capture groups of the most recent successful match: `is_regexp`, `regexp_pattern`, `regname`, `regnames`, `regnames_count`, `regmust`, `optimization`.

Synopsis

```
use re 'debug';                         # dump compile + match info
use re qw(Debug PARSE DUMP);            # pick specific debug categories
use re 'eval';                          # allow (?{...}) from variables
use re '/ix';                           # add /i and /x to every regex

use re qw(is_regexp regexp_pattern regnames);
if (is_regexp($qr)) {
    my ($pat, $mods) = regexp_pattern($qr);   # list: pattern,␣
↪modifiers
    my $str          = regexp_pattern($qr);   # scalar: "(?^
↪flags:pattern)"
}
"foobar" =~ /(?<first>foo)(?<rest>bar)/;
my @names = regnames();                        # ('first', 'rest')
my $cap   = regname('first');                  # 'foo'
```

What you can do with it

- Turn on regex debug output during one lexical scope and turn it back off outside - handy when a single pattern misbehaves in a large program.
- Ask whether an arbitrary value is really a compiled regex (even if it's blessed or overloaded) without having it stringify on you.
- Recover the original pattern text and modifier letters from a `qr//`, in either list or scalar form.
- Walk the named-capture groups of the last successful match by name, enumerate all names, or just count them.
- Peek at what the regex optimiser found - the longest anchored and floating substrings, the start class, the start-of-string anchors - to understand why a pattern is fast or slow.

This is a native implementation of the XS half of re 0.48 (`regmust`, `optimization`, and the hook that switches a scope's regex engine); `re.pm` is used unchanged. `is_regexp`, `regexp_pattern`, `regname`, `regnames` and `regnames_count` are built into perl itself.

Differences from upstream

One gap, not a difference in what works: `use re 'debug'` and `use re 'debugcolor'` are accepted but print nothing, because the debugging regex engine they switch to is not built in.

Functions

Pragma behaviour

`install`

Return an integer handle to the regex engine and reset the debug-colour flag.

Pattern introspection

`is_regexp`

Returns true if the argument is a compiled regular expression as returned by `qr//`, false if it is not.

`regexp_pattern`

Returns the pattern of a compiled regular expression, and in list context its modifiers as well.

Named captures

`regname`

Returns the contents of a named capture group of the last successful match.

regnames

Returns the names of the named capture groups of the last successful match.

regnames_count

Returns the number of distinct names defined in the pattern of the last successful match.

Optimiser introspection

regmust

Return the longest anchored and longest floating fixed strings the optimiser extracted from a compiled pattern.

optimization

Return a hashref of the optimiser's compile-time findings for a compiled pattern.

install

Return an integer handle to the regex engine and reset the debug-colour flag.

Not intended for user code. This is the hook `use re` itself calls to wire the debug engine into `%^H{regcomp}`. End users should write `use re 'debug'`, `use re 'eval'`, `use re '/ix'` and friends instead of touching `install` directly.

Synopsis

```
my $handle = re::install();   # pragma plumbing; not for application
↪code
```

What you get back

An integer that, when reinterpreted as a C pointer, resolves to the compile-time regex engine. Any other use of the value produces undefined behaviour.

Differences from upstream

- The returned integer points to the core regex engine, not to the debug engine. The debug engine is not available in this build; `use re 'debug'` therefore cannot produce upstream-style compile-time dumps. Truthiness of the return value - which is all the non-debug subpragmas (`taint`, `eval`) rely on - is preserved.

regmust

Return the longest anchored and longest floating fixed strings the optimiser extracted from a compiled pattern.

A *fixed string* is a substring that must appear in any string that matches. An *anchored* fixed string is one whose offset from the start of the match is known; a *floating* fixed

string can appear anywhere in a range of positions. The optimiser uses whichever it finds - preferring the longer, or the floating one if they are the same length - to skip positions that cannot match.

Synopsis

```
use re 'regmust';
my ($anchored, $floating) = regmust(qr/here .* there/x);
# $anchored = 'here', $floating = 'there'
```

What you get back

A two-element list `($anchored, $floating)`. Each element is either the fixed-string text as an SV or `&PL_sv_no` (the false-but-defined sentinel) when the optimiser has no fixed string of that kind. Returns `undef` if the argument is not a compiled pattern or its engine is unknown.

Examples

```
my ($a, $f) = regmust(qr/foo\s+bar/);    # $a = 'foo', $f = 'bar'
my ($a, $f) = regmust(qr/\d+/);          # no fixed strings - both␣
↪false
my @got     = regmust("not a regex");    # empty list - ref wasn't a␣
↪qr//
```

Edge cases

- Argument is not a `qr//` - returns `undef`.
- Pattern's engine is not the core engine - returns `undef`.
- Pattern has only one kind of fixed string - the other slot is `&PL_sv_no` (false but defined).
- UTF-8 substrings are returned as-is; the optimiser stores the UTF-8 form in a separate slot and this function prefers the byte form when both are present.

Differences from upstream

- The returned fixed strings are the values picked by this build's optimiser; upstream's POD notes that the exact result is optimiser-dependent and may change between Perl versions. The same caveat applies here.

optimization

Return a hashref of the optimiser's compile-time findings for a compiled pattern.

Intended for writing tests that pin optimiser behaviour, not for application logic. The set of keys and the meaning of their values are not a stable API - they change as the optimiser changes, even between minor perl releases.

Synopsis

```
use re 'optimization';
my $info = optimization(qr/^\d+foo/);
print $info->{anchored};            # 'foo'
print $info->{'anchor SBOL'};       # 1
print $info->{minlen};              # minimum match length
```

What you get back

A hashref, or undef if the argument is not a compiled pattern or its engine is unknown. Keys include minlen, minlenret, gofs, anchored, anchored utf8, anchored min offset, anchored max offset, anchored end shift, floating, floating utf8, floating min offset, floating max offset, floating end shift, checking ("anchored", "floating", or "none"), the booleans noscan, isall, anchor SBOL, anchor MBOL, anchor GPOS, skip, implicit, and stclass.

Edge cases

- Argument is not a qr// - returns undef.
- Pattern has no extracted substrings - the substring-related keys are undef or 0 and checking is "none".

Differences from upstream

- stclass is always undef. Upstream returns a textual rendering of the start class produced by my_regprop, which is compiled only with PERL_EXT_RE_DEBUG and is not available in this build.

is_regexp

Returns true if the argument is a compiled regular expression as returned by qr//, false if it is not.

Synopsis

```
use re qw(is_regexp);
is_regexp(qr/x/);               # 1
is_regexp("x");                 # false
is_regexp(bless {}, "Regexp");  # false
```

Overloading and blessing do not fool it: it looks at the compiled regexp inside the value, not at the package the value is blessed into. A hash blessed into Regexp is therefore not a regexp, and a qr// object reblessed into another class still is.

regexp_pattern

Returns the pattern of a compiled regular expression, and in list context its modifiers as well.

Synopsis

```
use re qw(regexp_pattern);
my $qr = qr/(?<y>\d{4})-(?<m>\d\d)/i;
my ($pat, $mods) = regexp_pattern($qr);
# $pat  is '(?<y>\d{4})-(?<m>\d\d)', $mods is 'i'
my $str = regexp_pattern($qr);
# $str  is '(?^i:(?<y>\d{4})-(?<m>\d\d))'
```

In list context it returns two elements: the pattern and the modifiers it was compiled with. In scalar context it returns what perl gives when it stringifies the `qr//`.

For an argument that is not a compiled regexp it returns the empty list in list context and a false but defined value in scalar context, so `regexp_pattern($ref) eq '(?^i:foo)'` never warns, whatever `$ref` holds. Like `is_regexp`, it is not fooled by overloading or blessing.

regname

Returns the contents of a named capture group of the last successful match.

Synopsis

```
use re qw(regname);
if ("2026-10" =~ /(?<y>\d{4})-(?<m>\d\d)/) {
    my $year = regname("y");          # '2026'
}
"a1" =~ /(?<d>a)(?<d>\d)?/;
my $all = regname("d", 1);           # ['a', '1']
```

A name can label more than one group. Without the second argument, or with it false, the result is the first of those groups that matched. With it true, the result is an array reference holding one entry per group of that name, in pattern order, with `undef` for a group that did not take part in the match.

regnames

Returns the names of the named capture groups of the last successful match.

Synopsis

```
use re qw(regnames);
"ab" =~ /(?<a>a)|(?<b>x)/;
my @used = regnames();      # ('a')
my @all  = regnames(1);     # ('a', 'b')
```

Without an argument, or with a false one, only the names of groups that took part in the match are returned. With a true argument, every name the pattern defines is returned, matched or not. The order of the names is not defined; sort them if it matters.

regnames_count

Returns the number of distinct names defined in the pattern of the last successful match.

Synopsis

```
use re qw(regnames_count);
"ab" =~ /(?<a>a)|(?<b>x)/;
my $n = regnames_count();   # 2
```

The count is of the names the pattern defines, whether or not their groups matched, so it can be larger than the number of names `regnames()` returns without its argument.

5.2.151 sigtrap

Perl pragma to enable simple signal handling

SYNOPSIS

```
use sigtrap;
use sigtrap qw(stack-trace old-interface-signals);  # equivalent
use sigtrap qw(BUS SEGV PIPE ABRT);
use sigtrap qw(die INT QUIT);
use sigtrap qw(die normal-signals);
use sigtrap qw(die untrapped normal-signals);
use sigtrap qw(die untrapped normal-signals
               stack-trace any error-signals);
use sigtrap 'handler' => \&my_handler, 'normal-signals';
use sigtrap qw(handler my_handler normal-signals
               stack-trace error-signals);
```

DESCRIPTION

The **sigtrap** pragma is a simple interface to installing signal handlers. You can have it install one of two handlers supplied by **sigtrap** itself (one which provides a Perl stack trace and one which simply `die()`s), or alternately you can supply your own handler for it to install. It can be told only to install a handler for signals which are either untrapped or ignored. It has three lists of signals to trap, plus you can supply your own list of signals.

The arguments passed to the `use` statement which invokes **sigtrap** are processed in order. When a signal name or the name of one of **sigtrap**'s signal lists is encountered a handler is immediately installed, when an option is encountered it affects subsequently installed handlers.

OPTIONS

SIGNAL HANDLERS

These options affect which handler will be used for subsequently installed signals.

- **stack-trace**

  The handler used for subsequently installed signals outputs a Perl stack trace to STDERR and then tries to dump core. This is the default signal handler.

- **die**

  The handler used for subsequently installed signals calls `die` (actually `croak`) with a message indicating which signal was caught.

- **handler** *your-handler*

  your-handler will be used as the handler for subsequently installed signals. *your-handler* can be any value which is valid as an assignment to an element of `%SIG`. See *perlvar* for examples of handler functions.

SIGNAL LISTS

sigtrap has a few built-in lists of signals to trap. They are:

- **normal-signals**

  These are the signals which a program might normally expect to encounter and which by default cause it to terminate. They are HUP, INT, PIPE and TERM.

- **error-signals**

  These signals usually indicate a serious problem with the Perl interpreter or with your script. They are ABRT, BUS, EMT, FPE, ILL, QUIT, SEGV, SYS and TRAP.

- **old-interface-signals**

  These are the signals which were trapped by default by the old **sigtrap** interface, they are ABRT, BUS, EMT, FPE, ILL, PIPE, QUIT, SEGV, SYS, TERM, and TRAP. If no signals or signals lists are passed to **sigtrap**, this list is used.

For each of these three lists, the collection of signals set to be trapped is checked before trapping; if your architecture does not implement a particular signal, it will not be trapped but rather silently ignored.

OTHER

- **untrapped**

  This token tells **sigtrap** to install handlers only for subsequently listed signals which aren't already trapped or ignored.

- **any**

  This token tells **sigtrap** to install handlers for all subsequently listed signals. This is the default behavior.

- *signal*

Any argument which looks like a signal name (that is, `/^[A-Z][A-Z0-9]*$/`) indicates that **sigtrap** should install a handler for that name.

- *number*

  Require that at least version *number* of **sigtrap** is being used.

EXAMPLES

Provide a stack trace for the old-interface-signals:

```
use sigtrap;
```

Ditto:

```
use sigtrap qw(stack-trace old-interface-signals);
```

Provide a stack trace on the 4 listed signals only:

```
use sigtrap qw(BUS SEGV PIPE ABRT);
```

Die on INT or QUIT:

```
use sigtrap qw(die INT QUIT);
```

Die on HUP, INT, PIPE or TERM:

```
use sigtrap qw(die normal-signals);
```

Die on HUP, INT, PIPE or TERM, except don't change the behavior for signals which are already trapped or ignored:

```
use sigtrap qw(die untrapped normal-signals);
```

Die on receipt one of any of the **normal-signals** which is currently **untrapped**, provide a stack trace on receipt of **any** of the **error-signals**:

```
use sigtrap qw(die untrapped normal-signals
               stack-trace any error-signals);
```

Install my_handler() as the handler for the **normal-signals**:

```
use sigtrap 'handler', \&my_handler, 'normal-signals';
```

Install my_handler() as the handler for the normal-signals, provide a Perl stack trace on receipt of one of the error-signals:

```
use sigtrap qw(handler my_handler normal-signals
               stack-trace error-signals);
```

5.2.152 sort

perl pragma to control sort() behaviour

SYNOPSIS

The sort pragma is now a no-op, and its use is discouraged. These three operations are valid, but have no effect:

```
use sort 'stable';          # guarantee stability
use sort 'defaults';        # revert to default behavior
no  sort 'stable';          # stability not important
```

DESCRIPTION

Historically the `sort` pragma you can control the behaviour of the builtin `sort()` function.

Prior to v5.28.0 there were two other options:

```
use sort '_mergesort';
use sort '_qsort';          # or '_quicksort'
```

If you try and specify either of these in v5.28+ it will croak.

The default sort has been stable since v5.8.0, and given this consistent behaviour for almost two decades, everyone has come to assume stability.

Stability will remain the default - hence there is no need for a pragma for code to opt into stability "just in case" this changes - it won't.

We do not foresee going back to offering multiple implementations of general purpose sorting - hence there is no future need to offer a pragma to choose between them.

If you know that you care that much about performance of your sorting, and that for your use case and your data, it was worth investigating alternatives, possible to identify an alternative from our default that was better, and the cost of switching was worth it, then you know more than we do. Likely whatever choices we can give are not as good as implementing your own. (For example, a Radix sort can be faster than O(n log n), but can't be used for all keys and has larger overheads.)

We are not averse to **changing** the sort algorithm, but we don't see the benefit in offering the choice of two general purpose implementations.

CAVEATS

The function `sort::current()` was provided to report the current state of the sort pragmata. This function was not exported, and there is no code to call it on CPAN. It is now deprecated, and will warn by default.

As we no longer store any sort "state", it can no longer return the correct value, so it will always return the string `stable`, as this is consistent with what we actually have implemented.

5.2.153 source

Namespace containing the following modules:

- `source::encoding` - Declare Perl source code encoding

source::encoding

Declare Perl source code encoding

SYNOPSIS

```
use source::encoding 'ascii';
use source::encoding 'utf8';
no source::encoding;
```

DESCRIPTION

These days, Perl code either generally contains only ASCII characters with `\x{}` and similar escapes to represent non-ASCII, or `use utf8` is used to indicate that the source code itself contains characters encoded as UTF-8.

That means that a character in the source code not meeting these criteria is often a typographical error. This pragma is used to tell Perl to raise an error when this happens.

`use source::encoding 'utf8'` is a synonym for `use utf8`. They may be used interchangeably.

`use source::encoding 'ascii'` turns off any UTF-8 expectations, and raises a fatal error if any character within its scope in the input source code is not ASCII.

`no source::encoding` turns off any UTF-8/ASCII expectations for the remainder of its scope. The meaning of non-ASCII characters is then undefined.

`use source::encoding 'ascii'` is automatically enabled within the lexical scope of a `use v5.41.0` or higher.

SEE ALSO

utf8

5.2.154 stable

Experimental features made easy, once we know they're stable

VERSION

version 0.036

SYNOPSIS

```
use stable 'lexical_subs', 'bitwise';
my sub is_odd($value) { $value & 1 }
```

DESCRIPTION

The *experimental* pragma makes it easy to turn on experiments while turning off associated warnings. You should read about it, if you don't already know what it does.

Seeing `use experimental` in code might be scary. In fact, it probably should be! Code that uses experimental features might break in the future if the perl development team decides that the experiment needs to be altered. When experiments become stable, because the developers decide they're a success, the warnings associated with them go away. When that happens, they can generally be turned on with `use feature`.

This is great, if you are using a version of perl where the feature you want is already stable. If you're using an older perl, though, it might be the case that you want to use an experimental feature that still warns, even though there's no risk in using it, because subsequent versions of perl have that feature unchanged and now stable.

Here's an example: The `postderef` feature was added in perl 5.20.0. In perl 5.24.0, it was marked stable. Using it would no longer trigger a warning. The behavior of the feature didn't change between 5.20.0 and 5.24.0. That means that it's perfectly safe to use the feature on 5.20 or 5.22, even though there's a warning.

In that case, you could very justifiably add `use experimental 'postderef'` but the casual reader may still be worried at seeing that. The `stable` pragma exists to turn on experimental features only when it's known that their behavior in the running perl is their stable behavior.

If you try to use an experimental feature that isn't stable or available on the running version of perl, an exception will be thrown. You should also take care that you've required the version of `stable` that you need!

If it's not immediately obvious why, here's a bit of explanation:

- `stable` comes with perl, starting with perl v5.38.
- Imagine that v5.38 adds a feature called "florps". It will stop being experimental in v5.42.
- The version of `stable` that comes with perl v5.38 can't know that the *florps* experiment will succeed, so you can't `use stable 'florps'` on the version of stable ships with v5.38, because it can't see the future!
- You'll need to write `use stable 1.234 'florps'` to say that you need version 1.234 of stable, which is when *florps* became known to stable.

Sure, it's a little weird, but it's worth it! The documentation of this pragma will tell you what version of `stable` you need to require in order to use various features. See below.

At present there are only a few "stable" features:

- `bitwise` - stable as of perl 5.22, available via stable 0.031
- `isa` - stable as of perl 5.32, available via stable 0.031

- `lexical_subs` - stable as of perl 5.22, available via stable 0.031

  Lexical subroutines were actually added in 5.18, and their design did not change, but significant bugs makes them unsafe to use before 5.22.

- `postderef` - stable as of perl 5.20, available via stable 0.031
- `const_attr` - stable as of perl 5.22, available via stable 0.032
- `for_list` - stable as of perl 5.36, available via stable 0.032

SEE ALSO

perlexperiment contains more information about experimental features.

AUTHOR

Leon Timmermans fawaka@gmail.com

COPYRIGHT AND LICENSE

This software is copyright (c) 2013 by Leon Timmermans.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.

5.2.155 strict

Perl pragma to restrict unsafe constructs

SYNOPSIS

```
use strict;

use strict "vars";
use strict "refs";
use strict "subs";

use strict;
no strict "vars";
```

DESCRIPTION

The `strict` pragma disables certain Perl expressions that could behave unexpectedly or are difficult to debug, turning them into errors. The effect of this pragma is limited to the current file or scope block.

If no import list is supplied, all possible restrictions are assumed. (This is the safest mode to operate in, but is sometimes too strict for casual programming.) Currently, there are three possible things to be strict about: “subs”, “vars”, and “refs”.

- `strict refs`

  This generates a runtime error if you use symbolic references (see perlref).

```
use strict 'refs';
$ref = \$foo;
print $$ref;        # ok
$ref = "foo";
print $$ref;        # runtime error; normally ok
$file = "STDOUT";
print $file "Hi!";  # error; note: no comma after $file
```

There is one exception to this rule:

```
$bar = \&{'foo'};
&$bar;
```

is allowed so that `goto &$AUTOLOAD` would not break under stricture.

- `strict vars`

  This generates a compile-time error if you access a variable that was neither explicitly declared (using any of `my`, `our`, `state`, or `use vars`) nor fully qualified. (Because this is to avoid variable suicide problems and subtle dynamic scoping issues, a merely `local` variable isn't good enough.) See *"my" in perlfunc*, *"our" in perlfunc*, *"state" in perlfunc*, *"local" in perlfunc*, and *vars*.

```
use strict 'vars';
$X::foo = 1;          # ok, fully qualified
my $foo = 10;         # ok, my() var
local $baz = 9;       # blows up, $baz not declared before

package Cinna;
our $bar;                    # Declares $bar in current package
$bar = 'HgS';                # ok, global declared via pragma
```

The local() generated a compile-time error because you just touched a global name without fully qualifying it.

Because of their special use by sort(), the variables $a and $b are exempted from this check.

- `strict subs`

  This disables the poetry optimization, generating a compile-time error if you try to use a bareword identifier that's not a subroutine, unless it is a simple identifier (no colons) and that it appears in curly braces, on the left hand side of the => symbol, or has the unary minus operator applied to it.

```
use strict 'subs';
$SIG{PIPE} = Plumber;   # blows up
$SIG{PIPE} = "Plumber"; # fine: quoted string is always ok
$SIG{PIPE} = \&Plumber; # preferred form
```

See "Pragmatic Modules" in perlmodlib.

HISTORY

`strict 'subs'`, with Perl 5.6.1, erroneously permitted to use an unquoted compound identifier (e.g. `Foo::Bar`) as a hash key (before => or inside curlies), but without forcing it always to a literal string.

Starting with Perl 5.8.1 strict is strict about its restrictions: if unknown restrictions are used, the strict pragma will abort with

```
Unknown 'strict' tag(s) '...'
```

As of version 1.04 (Perl 5.10), strict verifies that it is used as "strict" to avoid the dreaded Strict trap on case insensitive file systems.

Beginning with Perl 5.12, use of "use VERSION" (where VERSION >= 5.11.0) now lexically enables strictures just like "use strict" (in addition to the normal "use VERSION" effects and features.) In other words, "use v5.011" or higher now implies "use strict" automatically, as noted in "Implicit strictures" in perl5120delta and `use VERSION`.

5.2.156 subs

Perl pragma to predeclare subroutine names

SYNOPSIS

```
use subs qw(frob);
frob 3..10;
```

DESCRIPTION

This will predeclare all the subroutines whose names are in the list, allowing you to use them without parentheses (as list operators) even before they're declared.

Unlike pragmas that affect the `$^H` hints variable, the `use vars` and `use subs` declarations are not lexically scoped to the block they appear in: they affect the entire package in which they appear. It is not possible to rescind these declarations with `no vars` or `no subs`.

See "Pragmatic Modules" in perlmodlib and *"strict subs" in strict*.

5.2.157 threads

Perl interpreter-based threads

VERSION

This document describes threads version 2.45

WARNING

The “interpreter-based threads” provided by Perl are not the fast, lightweight system for multitasking that one might expect or hope for. Threads are implemented in a way that makes them easy to misuse. Few people know how to use them correctly or will be able to provide help.

The use of interpreter-based threads in perl is officially discouraged.

SYNOPSIS

```
use threads ('yield',
             'stack_size' => 64*4096,
             'exit' => 'threads_only',
             'stringify');

sub start_thread {
    my @args = @_;
    print('Thread started: ', join(' ', @args), "\n");
}
my $thr = threads->create('start_thread', 'argument');
$thr->join();

threads->create(sub { print("I am a thread\n"); })->join();

my $thr2 = async { foreach (@files) { ... } };
$thr2->join();
if (my $err = $thr2->error()) {
    warn("Thread error: $err\n");
}

# Invoke thread in list context (implicit) so it can return a list
my ($thr) = threads->create(sub { return (qw/a b c/); });
# or specify list context explicitly
my $thr = threads->create({'context' => 'list'},
                          sub { return (qw/a b c/); });
my @results = $thr->join();

$thr->detach();

# Get a thread's object
$thr = threads->self();
$thr = threads->object($tid);

# Get a thread's ID
$tid = threads->tid();
$tid = $thr->tid();
$tid = "$thr";

# Give other threads a chance to run
threads->yield();
```

(continues on next page)

(continued from previous page)

```
yield();

# Lists of non-detached threads
my @threads = threads->list();
my $thread_count = threads->list();

my @running = threads->list(threads::running);
my @joinable = threads->list(threads::joinable);

# Test thread objects
if ($thr1 == $thr2) {
    ...
}

# Manage thread stack size
$stack_size = threads->get_stack_size();
$old_size = threads->set_stack_size(32*4096);

# Create a thread with a specific context and stack size
my $thr = threads->create({ 'context'    => 'list',
                            'stack_size' => 32*4096,
                            'exit'       => 'thread_only' },
                          \&foo);

# Get thread's context
my $wantarray = $thr->wantarray();

# Check thread's state
if ($thr->is_running()) {
    sleep(1);
}
if ($thr->is_joinable()) {
    $thr->join();
}

# Send a signal to a thread
$thr->kill('SIGUSR1');

# Exit a thread
threads->exit();
```

DESCRIPTION

Since Perl 5.8, thread programming has been available using a model called *interpreter threads* which provides a new Perl interpreter for each thread, and, by default, results in no data or state information being shared between threads.

(Prior to Perl 5.8, *5005threads* was available through the `Thread.pm` API. This threading model has been deprecated, and was removed as of Perl 5.10.0.)

As just mentioned, all variables are, by default, thread local. To use shared variables, you need to also load *threads::shared*:

```
use threads;
use threads::shared;
```

When loading *threads::shared*, you must `use threads` before you `use threads::shared`. (`threads` will emit a warning if you do it the other way around.)

It is strongly recommended that you enable threads via `use threads` as early as possible in your script.

If needed, scripts can be written so as to run on both threaded and non-threaded Perls:

```
my $can_use_threads = eval 'use threads; 1';
if ($can_use_threads) {
    # Do processing using threads
    ...
} else {
    # Do it without using threads
    ...
}
```

- $thr = threads->create(FUNCTION, ARGS)

  This will create a new thread that will begin execution with the specified entry point function, and give it the *ARGS* list as parameters. It will return the corresponding threads object, or `undef` if thread creation failed.

  FUNCTION may either be the name of a function, an anonymous subroutine, or a code ref.

  ```
  my $thr = threads->create('func_name', ...);
      # or
  my $thr = threads->create(sub { ... }, ...);
      # or
  my $thr = threads->create(\&func, ...);
  ```

  The `->new()` method is an alias for `->create()`.

- $thr->join()

  This will wait for the corresponding thread to complete its execution. When the thread finishes, `->join()` will return the return value(s) of the entry point function.

  The context (void, scalar or list) for the return value(s) for `->join()` is determined at the time of thread creation.

  ```
  # Create thread in list context (implicit)
  my ($thr1) = threads->create(sub {
                                  my @results = qw(a b c);
                                  return (@results);
                               });
  #   or (explicit)
  ```

(continues on next page)

(continued from previous page)

```
my $thr1 = threads->create({'context' => 'list'},
                           sub {
                                my @results = qw(a b c);
                                return (@results);
                           });
# Retrieve list results from thread
my @res1 = $thr1->join();

# Create thread in scalar context (implicit)
my $thr2 = threads->create(sub {
                                my $result = 42;
                                return ($result);
                             });
# Retrieve scalar result from thread
my $res2 = $thr2->join();

# Create a thread in void context (explicit)
my $thr3 = threads->create({'void' => 1},
                           sub { print("Hello, world\n"); });
# Join the thread in void context (i.e., no return value)
$thr3->join();
```

See “THREAD CONTEXT” for more details.

If the program exits without all threads having either been joined or detached, then a warning will be issued.

Calling `->join()` or `->detach()` on an already joined thread will cause an error to be thrown.

- $thr->detach()

  Makes the thread unjoinable, and causes any eventual return value to be discarded. When the program exits, any detached threads that are still running are silently terminated.

  If the program exits without all threads having either been joined or detached, then a warning will be issued.

  Calling `->join()` or `->detach()` on an already detached thread will cause an error to be thrown.

- threads->detach()

  Class method that allows a thread to detach itself.

- threads->self()

  Class method that allows a thread to obtain its own *threads* object.

- $thr->tid()

  Returns the ID of the thread. Thread IDs are unique integers with the main thread in a program being 0, and incrementing by 1 for every thread created.

- threads->tid()

Class method that allows a thread to obtain its own ID.

- "$thr"

  If you add the `stringify` import option to your `use threads` declaration, then using a threads object in a string or a string context (e.g., as a hash key) will cause its ID to be used as the value:

```
use threads qw(stringify);

my $thr = threads->create(...);
print("Thread $thr started\n");  # Prints: Thread 1 started
```

- threads->object($tid)

  This will return the *threads* object for the *active* thread associated with the specified thread ID. If `$tid` is the value for the current thread, then this call works the same as `->self()`. Otherwise, returns `undef` if there is no thread associated with the TID, if the thread is joined or detached, if no TID is specified or if the specified TID is undef.

- threads->yield()

  This is a suggestion to the OS to let this thread yield CPU time to other threads. What actually happens is highly dependent upon the underlying thread implementation.

  You may do `use threads qw(yield)`, and then just use `yield()` in your code.

- threads->list()
- threads->list(threads::all)
- threads->list(threads::running)
- threads->list(threads::joinable)

  With no arguments (or using `threads::all`) and in a list context, returns a list of all non-joined, non-detached *threads* objects. In a scalar context, returns a count of the same.

  With a *true* argument (using `threads::running`), returns a list of all non-joined, non-detached *threads* objects that are still running.

  With a *false* argument (using `threads::joinable`), returns a list of all non-joined, non-detached *threads* objects that have finished running (i.e., for which `->join()` will not *block*).

- $thr1->equal($thr2)

  Tests if two threads objects are the same thread or not. This is overloaded to the more natural forms:

```
if ($thr1 == $thr2) {
    print("Threads are the same\n");
}
# or
if ($thr1 != $thr2) {
    print("Threads differ\n");
}
```

(Thread comparison is based on thread IDs.)

- async BLOCK;

  `async` creates a thread to execute the block immediately following it. This block is treated as an anonymous subroutine, and so must have a semicolon after the closing brace. Like `threads->create()`, `async` returns a *threads* object.

- $thr->error()

  Threads are executed in an `eval` context. This method will return `undef` if the thread terminates *normally*. Otherwise, it returns the value of `$@` associated with the thread's execution status in its `eval` context.

- $thr->_handle()

  This *private* method returns a pointer (i.e., the memory location expressed as an unsigned integer) to the internal thread structure associated with a threads object. For Win32, this is a pointer to the `HANDLE` value returned by `CreateThread` (i.e., `HANDLE *`); for other platforms, it is a pointer to the `pthread_t` structure used in the `pthread_create` call (i.e., `pthread_t *`).

  This method is of no use for general Perl threads programming.

- threads->_handle()

  Class method that allows a thread to obtain its own *handle*.

EXITING A THREAD

The usual method for terminating a thread is to *return()* from the entry point function with the appropriate return value(s).

- threads->exit()

  If needed, a thread can be exited at any time by calling `threads->exit()`. This will cause the thread to return `undef` in a scalar context, or the empty list in a list context.

  When called from the *main* thread, this behaves the same as `exit(0)`.

- threads->exit(status)

  When called from a thread, this behaves like `threads->exit()` (i.e., the exit status code is ignored).

  When called from the *main* thread, this behaves the same as `exit(status)`.

- die()

  Calling `die()` in a thread indicates an abnormal exit for the thread. Any `$SIG{__DIE__}` handler in the thread will be called first, and then the thread will exit with a warning message that will contain any arguments passed in the `die()` call.

- exit(status)

  Calling *exit()* inside a thread causes the whole application to terminate. Because of this, the use of `exit()` inside threaded code, or in modules that might be used in threaded applications, is strongly discouraged.

If `exit()` really is needed, then consider using the following:

```
threads->exit() if threads->can('exit');   # Thread friendly
exit(status);
```

- use threads 'exit' => 'threads_only'

  This globally overrides the default behavior of calling `exit()` inside a thread, and effectively causes such calls to behave the same as `threads->exit()`. In other words, with this setting, calling `exit()` causes only the thread to terminate.

  Because of its global effect, this setting should not be used inside modules or the like.

  The *main* thread is unaffected by this setting.

- threads->create({'exit' => 'thread_only'}, ...)

  This overrides the default behavior of `exit()` inside the newly created thread only.

- $thr->set_thread_exit_only(boolean)

  This can be used to change the *exit thread only* behavior for a thread after it has been created. With a *true* argument, `exit()` will cause only the thread to exit. With a *false* argument, `exit()` will terminate the application.

  The *main* thread is unaffected by this call.

- threads->set_thread_exit_only(boolean)

  Class method for use inside a thread to change its own behavior for `exit()`.

  The *main* thread is unaffected by this call.

THREAD STATE

The following boolean methods are useful in determining the *state* of a thread.

- $thr->is_running()

  Returns true if a thread is still running (i.e., if its entry point function has not yet finished or exited).

- $thr->is_joinable()

  Returns true if the thread has finished running, is not detached and has not yet been joined. In other words, the thread is ready to be joined, and a call to `$thr->join()` will not *block*.

- $thr->is_detached()

  Returns true if the thread has been detached.

- threads->is_detached()

  Class method that allows a thread to determine whether or not it is detached.

THREAD CONTEXT

As with subroutines, the type of value returned from a thread's entry point function may be determined by the thread's *context*: list, scalar or void. The thread's context is determined at thread creation. This is necessary so that the context is available to the entry point function via *wantarray()*. The thread may then specify a value of the appropriate type to be returned from `->join()`.

Explicit context

Because thread creation and thread joining may occur in different contexts, it may be desirable to state the context explicitly to the thread's entry point function. This may be done by calling `->create()` with a hash reference as the first argument:

```
my $thr = threads->create({'context' => 'list'}, \&foo);
...
my @results = $thr->join();
```

In the above, the threads object is returned to the parent thread in scalar context, and the thread's entry point function `foo` will be called in list (array) context such that the parent thread can receive a list (array) from the `->join()` call. (`'array'` is synonymous with `'list'`.)

Similarly, if you need the threads object, but your thread will not be returning a value (i.e., *void* context), you would do the following:

```
my $thr = threads->create({'context' => 'void'}, \&foo);
...
$thr->join();
```

The context type may also be used as the *key* in the hash reference followed by a *true* value:

```
threads->create({'scalar' => 1}, \&foo);
...
my ($thr) = threads->list();
my $result = $thr->join();
```

Implicit context

If not explicitly stated, the thread's context is implied from the context of the `->create()` call:

```
# Create thread in list context
my ($thr) = threads->create(...);

# Create thread in scalar context
my $thr = threads->create(...);

# Create thread in void context
threads->create(...);
```

$thr->wantarray()

This returns the thread's context in the same manner as *wantarray()*.

threads->wantarray()

Class method to return the current thread's context. This returns the same value as running *wantarray()* inside the current thread's entry point function.

THREAD STACK SIZE

The default per-thread stack size for different platforms varies significantly, and is almost always far more than is needed for most applications. On Win32, Perl's makefile explicitly sets the default stack to 16 MB; on most other platforms, the system default is used, which again may be much larger than is needed.

By tuning the stack size to more accurately reflect your application's needs, you may significantly reduce your application's memory usage, and increase the number of simultaneously running threads.

Note that on Windows, address space allocation granularity is 64 KB, therefore, setting the stack smaller than that on Win32 Perl will not save any more memory.

- threads->get_stack_size();

  Returns the current default per-thread stack size. The default is zero, which means the system default stack size is currently in use.

- $size = $thr->get_stack_size();

  Returns the stack size for a particular thread. A return value of zero indicates the system default stack size was used for the thread.

- $old_size = threads->set_stack_size($new_size);

  Sets a new default per-thread stack size, and returns the previous setting.

  Some platforms have a minimum thread stack size. Trying to set the stack size below this value will result in a warning, and the minimum stack size will be used.

  Some Linux platforms have a maximum stack size. Setting too large of a stack size will cause thread creation to fail.

  If needed, `$new_size` will be rounded up to the next multiple of the memory page size (usually 4096 or 8192).

  Threads created after the stack size is set will then either call `pthread_attr_setstacksize()` *(for pthreads platforms)*, or supply the stack size to `CreateThread()` *(for Win32 Perl)*.

  (Obviously, this call does not affect any currently extant threads.)

- use threads ('stack_size' => VALUE);

  This sets the default per-thread stack size at the start of the application.

- $ENV{'PERL5_ITHREADS_STACK_SIZE'}

  The default per-thread stack size may be set at the start of the application through the use of the environment variable `PERL5_ITHREADS_STACK_SIZE`:

```
PERL5_ITHREADS_STACK_SIZE=1048576
export PERL5_ITHREADS_STACK_SIZE
perl -e'use threads; print(threads->get_stack_size(), "\n")'
```

This value overrides any `stack_size` parameter given to `use threads`. Its primary purpose is to permit setting the per-thread stack size for legacy threaded applications.

- threads->create({'stack_size' => VALUE}, FUNCTION, ARGS)

  To specify a particular stack size for any individual thread, call `->create()` with a hash reference as the first argument:

```
my $thr = threads->create({'stack_size' => 32*4096},
                          \&foo, @args);
```

- $thr2 = $thr1->create(FUNCTION, ARGS)

  This creates a new thread (`$thr2`) that inherits the stack size from an existing thread (`$thr1`). This is shorthand for the following:

```
my $stack_size = $thr1->get_stack_size();
my $thr2 = threads->create({'stack_size' => $stack_size},
                           FUNCTION, ARGS);
```

THREAD SIGNALLING

When safe signals is in effect (the default behavior - see "Unsafe signals" for more details), then signals may be sent and acted upon by individual threads.

- $thr->kill('SIG...');

  Sends the specified signal to the thread. Signal names and (positive) signal numbers are the same as those supported by *kill()*. For example, 'SIGTERM', 'TERM' and (depending on the OS) 15 are all valid arguments to `->kill()`.

  Returns the thread object to allow for method chaining:

```
$thr->kill('SIG...')->join();
```

Signal handlers need to be set up in the threads for the signals they are expected to act upon. Here's an example for *cancelling* a thread:

```
use threads;

sub thr_func
{
    # Thread 'cancellation' signal handler
    $SIG{'KILL'} = sub { threads->exit(); };

    ...
}
```

(continues on next page)

(continued from previous page)

```
# Create a thread
my $thr = threads->create('thr_func');

...

# Signal the thread to terminate, and then detach
# it so that it will get cleaned up automatically
$thr->kill('KILL')->detach();
```

Here's another simplistic example that illustrates the use of thread signalling in conjunction with a semaphore to provide rudimentary *suspend* and *resume* capabilities:

```
use threads;
use Thread::Semaphore;

sub thr_func
{
    my $sema = shift;

    # Thread 'suspend/resume' signal handler
    $SIG{'STOP'} = sub {
        $sema->down();      # Thread suspended
        $sema->up();        # Thread resumes
    };

    ...
}

# Create a semaphore and pass it to a thread
my $sema = Thread::Semaphore->new();
my $thr = threads->create('thr_func', $sema);

# Suspend the thread
$sema->down();
$thr->kill('STOP');

...

# Allow the thread to continue
$sema->up();
```

CAVEAT: The thread signalling capability provided by this module does not actually send signals via the OS. It *emulates* signals at the Perl-level such that signal handlers are called in the appropriate thread. For example, sending `$thr->kill('STOP')` does not actually suspend a thread (or the whole process), but does cause a `$SIG{'STOP'}` handler to be called in that thread (as illustrated above).

As such, signals that would normally not be appropriate to use in the `kill()` command (e.g., `kill('KILL', $$)`) are okay to use with the `->kill()` method (again, as illustrated above).

Correspondingly, sending a signal to a thread does not disrupt the operation the thread is currently working on: The signal will be acted upon after the current operation has completed. For instance, if the thread is *stuck* on an I/O call, sending it a signal will not cause the I/O call to be interrupted such that the signal is acted up immediately.

Sending a signal to a terminated/finished thread is ignored.

WARNINGS

- Perl exited with active threads:

  If the program exits without all threads having either been joined or detached, then this warning will be issued.

  NOTE: If the *main* thread exits, then this warning cannot be suppressed using no `warnings 'threads';` as suggested below.

- Thread creation failed: pthread_create returned #

  See the appropriate *man* page for `pthread_create` to determine the actual cause for the failure.

- Thread # terminated abnormally: ...

  A thread terminated in some manner other than just returning from its entry point function, or by using `threads->exit()`. For example, the thread may have terminated because of an error, or by using `die`.

- Using minimum thread stack size of #

  Some platforms have a minimum thread stack size. Trying to set the stack size below this value will result in the above warning, and the stack size will be set to the minimum.

- Thread creation failed: pthread_attr_setstacksize(*SIZE*) returned 22

  The specified *SIZE* exceeds the system's maximum stack size. Use a smaller value for the stack size.

If needed, thread warnings can be suppressed by using:

```
no warnings 'threads';
```

in the appropriate scope.

ERRORS

- This Perl not built to support threads

  The particular copy of Perl that you're trying to use was not built using the `useithreads` configuration option.

- Cannot change stack size of an existing thread

  The stack size of currently extant threads cannot be changed, therefore, the following results in the above error:

  ```
  $thr->set_stack_size($size);
  ```

- Cannot signal threads without safe signals

  Safe signals must be in effect to use the `->kill()` signalling method. See "Unsafe signals" for more details.

- Unrecognized signal name: ...

  The particular copy of Perl that you're trying to use does not support the specified signal being used in a `->kill()` call.

BUGS AND LIMITATIONS

Before you consider posting a bug report, please consult, and possibly post a message to the discussion forum to see if what you've encountered is a known problem.

- Thread-safe modules
- Using non-thread-safe modules

  Unfortunately, you may encounter Perl modules that are not *thread-safe*. For example, they may crash the Perl interpreter during execution, or may dump core on termination. Depending on the module and the requirements of your application, it may be possible to work around such difficulties.

  If the module will only be used inside a thread, you can try loading the module from inside the thread entry point function using `require` (and `import` if needed):

```
sub thr_func
{
    require Unsafe::Module
    # Unsafe::Module->import(...);

    ....
}
```

  If the module is needed inside the *main* thread, try modifying your application so that the module is loaded (again using `require` and `->import()`) after any threads are started, and in such a way that no other threads are started afterwards.

  If the above does not work, or is not adequate for your application, then file a bug report on https://rt.cpan.org/Public/ against the problematic module.

- Memory consumption

  On most systems, frequent and continual creation and destruction of threads can lead to ever-increasing growth in the memory footprint of the Perl interpreter. While it is simple to just launch threads and then `->join()` or `->detach()` them, for long-lived applications, it is better to maintain a pool of threads, and to reuse them for the work needed, using *queues* to notify threads of pending work. The CPAN distribution of this module contains a simple example (`examples/pool_reuse.pl`) illustrating the creation, use and monitoring of a pool of *reusable* threads.

- Current working directory

  On all platforms except MSWin32, the setting for the current working directory is shared among all threads such that changing it in one thread (e.g., using `chdir()`) will affect all the threads in the application.

On MSWin32, each thread maintains its own the current working directory setting.

- Locales

  Prior to Perl 5.28, locales could not be used with threads, due to various race conditions. Starting in that release, on systems that implement thread-safe locale functions, threads can be used, with some caveats. This includes Windows starting with Visual Studio 2005, and systems compatible with POSIX 2008. See *"Multi-threaded operation" in perllocale*.

  Each thread (except the main thread) is started using the C locale. The main thread is started like all other Perl programs; see *"ENVIRONMENT" in perllocale*. You can switch locales in any thread as often as you like.

  If you want to inherit the parent thread's locale, you can, in the parent, set a variable like so:

```
$foo = POSIX::setlocale(LC_ALL, NULL);
```

  and then pass to threads->create() a sub that closes over `$foo`. Then, in the child, you say

```
POSIX::setlocale(LC_ALL, $foo);
```

  Or you can use the facilities in *threads::shared* to pass `$foo`; or if the environment hasn't changed, in the child, do

```
POSIX::setlocale(LC_ALL, "");
```

- Environment variables

  Currently, on all platforms except MSWin32, all *system* calls (e.g., using `system()` or back-ticks) made from threads use the environment variable settings from the *main* thread. In other words, changes made to `%ENV` in a thread will not be visible in *system* calls made by that thread.

  To work around this, set environment variables as part of the *system* call. For example:

```
my $msg = 'hello';
system("FOO=$msg; echo \$FOO");   # Outputs 'hello' to STDOUT
```

  On MSWin32, each thread maintains its own set of environment variables.

- Catching signals

  Signals are *caught* by the main thread (thread ID = 0) of a script. Therefore, setting up signal handlers in threads for purposes other than "THREAD SIGNALLING" as documented above will not accomplish what is intended.

  This is especially true if trying to catch `SIGALRM` in a thread. To handle alarms in threads, set up a signal handler in the main thread, and then use "THREAD SIGNALLING" to relay the signal to the thread:

```
# Create thread with a task that may time out
my $thr = threads->create(sub {
```

(continues on next page)

(continued from previous page)

```
    threads->yield();
    eval {
        $SIG{ALRM} = sub { die("Timeout\n"); };
        alarm(10);
        ...  # Do work here
        alarm(0);
    };
    if ($@ =~ /Timeout/) {
        warn("Task in thread timed out\n");
    }
};

# Set signal handler to relay SIGALRM to thread
$SIG{ALRM} = sub { $thr->kill('ALRM') };

... # Main thread continues working
```

- Parent-child threads

  On some platforms, it might not be possible to destroy *parent* threads while there are still existing *child* threads.

- Unsafe signals

  Since Perl 5.8.0, signals have been made safer in Perl by postponing their handling until the interpreter is in a *safe* state. See "Safe Signals" in perl58delta and "Deferred Signals (Safe Signals)" in perlipc for more details.

  Safe signals is the default behavior, and the old, immediate, unsafe signalling behavior is only in effect in the following situations:

  – Perl has been built with `PERL_OLD_SIGNALS` (see `perl -V`).

  – The environment variable `PERL_SIGNALS` is set to `unsafe` (see "PERL_SIGNALS" in perlrun).

  – The module Perl::Unsafe::Signals is used.

  If unsafe signals is in effect, then signal handling is not thread-safe, and the `->kill()` signalling method cannot be used.

- Identity of objects returned from threads

  When a value is returned from a thread through a `join` operation, the value and everything that it references is copied across to the joining thread, in much the same way that values are copied upon thread creation. This works fine for most kinds of value, including arrays, hashes, and subroutines. The copying recurses through array elements, reference scalars, variables closed over by subroutines, and other kinds of reference.

  However, everything referenced by the returned value is a fresh copy in the joining thread, even if a returned object had in the child thread been a copy of something that previously existed in the parent thread. After joining, the parent will therefore have a duplicate of each such object. This sometimes matters, especially if the

object gets mutated; this can especially matter for private data to which a returned subroutine provides access.

- Returning blessed objects from threads

  Returning blessed objects from threads does not work. Depending on the classes involved, you may be able to work around this by returning a serialized version of the object (e.g., using *Data::Dumper* or *Storable*), and then reconstituting it in the joining thread. If you're using Perl 5.10.0 or later, and if the class supports *shared objects*, you can pass them via *shared queues*.

- END blocks in threads

  It is possible to add END blocks to threads by using *require* or *eval* with the appropriate code. These `END` blocks will then be executed when the thread's interpreter is destroyed (i.e., either during a `->join()` call, or at program termination).

  However, calling any *threads* methods in such an `END` block will most likely *fail* (e.g., the application may hang, or generate an error) due to mutexes that are needed to control functionality within the *threads* module.

  For this reason, the use of `END` blocks in threads is **strongly** discouraged.

- Open directory handles

  In perl 5.14 and higher, on systems other than Windows that do not support the `fchdir` C function, directory handles (see *opendir*) will not be copied to new threads. You can use the `d_fchdir` variable in *Config.pm* to determine whether your system supports it.

  In prior perl versions, spawning threads with open directory handles would crash the interpreter. [perl #75154]

- Detached threads and global destruction

  If the main thread exits while there are detached threads which are still running, then Perl's global destruction phase is not executed because otherwise certain global structures that control the operation of threads and that are allocated in the main thread's memory may get destroyed before the detached thread is destroyed.

  If you are using any code that requires the execution of the global destruction phase for clean up (e.g., removing temp files), then do not use detached threads, but rather join all threads before exiting the program.

- Perl Bugs and the CPAN Version of *threads*

  Support for threads extends beyond the code in this module (i.e., `threads.pm` and `threads.xs`), and into the Perl interpreter itself. Older versions of Perl contain bugs that may manifest themselves despite using the latest version of *threads* from CPAN. There is no workaround for this other than upgrading to the latest version of Perl.

  Even with the latest version of Perl, it is known that certain constructs with threads may result in warning messages concerning leaked scalars or unreferenced scalars. However, such warnings are harmless, and may safely be ignored.

  You can search for *threads* related bug reports at https://rt.cpan.org/Public/. If needed submit any new bugs, problems, patches, etc. to: https://rt.cpan.org/Public/Dist/Display.html?Name=threads

REQUIREMENTS

Perl 5.8.0 or later

SEE ALSO

threads on MetaCPAN: https://metacpan.org/release/threads

Code repository for CPAN distribution: https://github.com/Dual-Life/threads

threads::shared, perlthrtut

https://www.perl.com/pub/a/2002/06/11/threads.html and https://www.perl.com/pub/a/2002/09/04/threads.html

Perl threads mailing list: https://lists.perl.org/list/ithreads.html

Stack size discussion: https://www.perlmonks.org/?node_id=532956

Sample code in the *examples* directory of this distribution on CPAN.

AUTHOR

Artur Bergman <sky AT crucially DOT net>

CPAN version produced by Jerry D. Hedden <jdhedden AT cpan DOT org>

LICENSE

threads is released under the same license as Perl.

ACKNOWLEDGEMENTS

Richard Soderberg <perl AT crystalflame DOT net> - Helping me out tons, trying to find reasons for races and other weird bugs!

Simon Cozens <simon AT brecon DOT co DOT uk> - Being there to answer zillions of annoying questions

Rocco Caputo <troc AT netrus DOT net>

Vipul Ved Prakash <mail AT vipul DOT net> - Helping with debugging

Dean Arnold <darnold AT presicient DOT com> - Stack size API

Modules

- `threads::shared` - Perl extension for sharing data structures between threads

threads::shared

Perl extension for sharing data structures between threads

VERSION

This document describes threads::shared version 1.73

SYNOPSIS

```
use threads;
use threads::shared;

my $var :shared;
my %hsh :shared;
my @ary :shared;

my ($scalar, @array, %hash);
share($scalar);
share(@array);
share(%hash);

$var = $scalar_value;
$var = $shared_ref_value;
$var = shared_clone($non_shared_ref_value);
$var = shared_clone({'foo' => [qw/foo bar baz/]});

$hsh{'foo'} = $scalar_value;
$hsh{'bar'} = $shared_ref_value;
$hsh{'baz'} = shared_clone($non_shared_ref_value);
$hsh{'quz'} = shared_clone([1..3]);

$ary[0] = $scalar_value;
$ary[1] = $shared_ref_value;
$ary[2] = shared_clone($non_shared_ref_value);
$ary[3] = shared_clone([ {}, [] ]);

{ lock(%hash); ...  }

cond_wait($scalar);
cond_timedwait($scalar, time() + 30);
cond_broadcast(@array);
cond_signal(%hash);

my $lockvar :shared;
# condition var != lock var
cond_wait($var, $lockvar);
cond_timedwait($var, time()+30, $lockvar);
```

DESCRIPTION

By default, variables are private to each thread, and each newly created thread gets a private copy of each existing variable. This module allows you to share variables across different threads (and pseudo-forks on Win32). It is used together with the *threads* mod-

ule.

This module supports the sharing of the following data types only: scalars and scalar refs, arrays and array refs, and hashes and hash refs.

EXPORT

The following functions are exported by this module: `share`, `shared_clone`, `is_shared`, `cond_wait`, `cond_timedwait`, `cond_signal` and `cond_broadcast`

Note that if this module is imported when *threads* has not yet been loaded, then these functions all become no-ops. This makes it possible to write modules that will work in both threaded and non-threaded environments.

FUNCTIONS

- share VARIABLE

  `share` takes a variable and marks it as shared:

```
my ($scalar, @array, %hash);
share($scalar);
share(@array);
share(%hash);
```

  `share` will return the shared rvalue, but always as a reference.

  Variables can also be marked as shared at compile time by using the `:shared` attribute:

```
my ($var, %hash, @array) :shared;
```

  Shared variables can only store scalars, refs of shared variables, or refs of shared data (discussed in next section):

```
my ($var, %hash, @array) :shared;
my $bork;

# Storing scalars
$var = 1;
$hash{'foo'} = 'bar';
$array[0] = 1.5;

# Storing shared refs
$var = \%hash;
$hash{'ary'} = \@array;
$array[1] = \$var;

# The following are errors:
#   $var = \$bork;                  # ref of non-shared
 ↪variable
#   $hash{'bork'} = [];             # non-shared array ref
#   push(@array, { 'x' => 1 });     # non-shared hash ref
```

- shared_clone REF

  `shared_clone` takes a reference, and returns a shared version of its argument, performing a deep copy on any non-shared elements. Any shared elements in the argument are used as is (i.e., they are not cloned).

```
my $cpy = shared_clone({'foo' => [qw/foo bar baz/]});
```

  Object status (i.e., the class an object is blessed into) is also cloned.

```
my $obj = {'foo' => [qw/foo bar baz/]};
bless($obj, 'Foo');
my $cpy = shared_clone($obj);
print(ref($cpy), "\n");          # Outputs 'Foo'
```

  For cloning empty array or hash refs, the following may also be used:

```
$var = &share([]);   # Same as $var = shared_clone([]);
$var = &share({});   # Same as $var = shared_clone({});
```

  Not all Perl data types can be cloned (e.g., globs, code refs). By default, `shared_clone` will *croak* if it encounters such items. To change this behaviour to a warning, then set the following:

```
$threads::shared::clone_warn = 1;
```

  In this case, `undef` will be substituted for the item to be cloned. If set to zero:

```
$threads::shared::clone_warn = 0;
```

  then the `undef` substitution will be performed silently.

- is_shared VARIABLE

  `is_shared` checks if the specified variable is shared or not. If shared, returns the variable's internal ID (similar to `refaddr()` (see *Scalar::Util*). Otherwise, returns `undef`.

```
if (is_shared($var)) {
    print("\$var is shared\n");
} else {
    print("\$var is not shared\n");
}
```

  When used on an element of an array or hash, `is_shared` checks if the specified element belongs to a shared array or hash. (It does not check the contents of that element.)

```
my %hash :shared;
if (is_shared(%hash)) {
    print("\%hash is shared\n");
}

$hash{'elem'} = 1;
```

(continues on next page)

(continued from previous page)

```
if (is_shared($hash{'elem'})) {
    print("\$hash{'elem'} is in a shared hash\n");
}
```

- lock VARIABLE

  `lock` places a **advisory** lock on a variable until the lock goes out of scope. If the variable is locked by another thread, the `lock` call will block until it's available. Multiple calls to `lock` by the same thread from within dynamically nested scopes are safe – the variable will remain locked until the outermost lock on the variable goes out of scope.

  `lock` follows references exactly *one* level:

```
my %hash :shared;
my $ref = \%hash;
lock($ref);           # This is equivalent to lock(%hash)
```

  Note that you cannot explicitly unlock a variable; you can only wait for the lock to go out of scope. This is most easily accomplished by locking the variable inside a block.

```
my $var :shared;
{
    lock($var);
    # $var is locked from here to the end of the block
    ...
}
# $var is now unlocked
```

  As locks are advisory, they do not prevent data access or modification by another thread that does not itself attempt to obtain a lock on the variable.

  You cannot lock the individual elements of a container variable:

```
my %hash :shared;
$hash{'foo'} = 'bar';
#lock($hash{'foo'});          # Error
lock(%hash);                  # Works
```

  If you need more fine-grained control over shared variable access, see *Thread::Semaphore*.

- cond_wait VARIABLE
- cond_wait CONDVAR, LOCKVAR

  The `cond_wait` function takes a **locked** variable as a parameter, unlocks the variable, and blocks until another thread does a `cond_signal` or `cond_broadcast` for that same locked variable. The variable that `cond_wait` blocked on is re-locked after the `cond_wait` is satisfied. If there are multiple threads `cond_waiting` on the same variable, all but one will re-block waiting to reacquire the lock on the variable. (So if you're only using `cond_wait` for synchronization, give up the lock as soon as possible). The two actions of unlocking the variable and entering the

blocked wait state are atomic, the two actions of exiting from the blocked wait state and re-locking the variable are not.

In its second form, `cond_wait` takes a shared, **unlocked** variable followed by a shared, **locked** variable. The second variable is unlocked and thread execution suspended until another thread signals the first variable.

It is important to note that the variable can be notified even if no thread `cond_signal` or `cond_broadcast` on the variable. It is therefore important to check the value of the variable and go back to waiting if the requirement is not fulfilled. For example, to pause until a shared counter drops to zero:

```
{ lock($counter); cond_wait($counter) until $counter == 0; }
```

- cond_timedwait VARIABLE, ABS_TIMEOUT
- cond_timedwait CONDVAR, ABS_TIMEOUT, LOCKVAR

  In its two-argument form, `cond_timedwait` takes a **locked** variable and an absolute timeout in *epoch* seconds (see *time() in perlfunc* for more) as parameters, unlocks the variable, and blocks until the timeout is reached or another thread signals the variable. A false value is returned if the timeout is reached, and a true value otherwise. In either case, the variable is re-locked upon return.

  Like `cond_wait`, this function may take a shared, **locked** variable as an additional parameter; in this case the first parameter is an **unlocked** condition variable protected by a distinct lock variable.

  Again like `cond_wait`, waking up and reacquiring the lock are not atomic, and you should always check your desired condition after this function returns. Since the timeout is an absolute value, however, it does not have to be recalculated with each pass:

```
lock($var);
my $abs = time() + 15;
until ($ok = desired_condition($var)) {
    last if !cond_timedwait($var, $abs);
}
# we got it if $ok, otherwise we timed out!
```

- cond_signal VARIABLE

  The `cond_signal` function takes a **locked** variable as a parameter and unblocks one thread that's `cond_waiting` on that variable. If more than one thread is blocked in a `cond_wait` on that variable, only one (and which one is indeterminate) will be unblocked.

  If there are no threads blocked in a `cond_wait` on the variable, the signal is discarded. By always locking before signaling, you can (with care), avoid signaling before another thread has entered cond_wait().

  `cond_signal` will normally generate a warning if you attempt to use it on an unlocked variable. On the rare occasions where doing this may be sensible, you can suppress the warning with:

```
{ no warnings 'threads'; cond_signal($foo); }
```

- cond_broadcast VARIABLE

  The `cond_broadcast` function works similarly to `cond_signal`. `cond_broadcast`, though, will unblock **all** the threads that are blocked in a `cond_wait` on the locked variable, rather than only one.

OBJECTS

threads::shared exports a version of *bless()* that works on shared objects such that *blessings* propagate across threads.

```
# Create a shared 'Foo' object
my $foo :shared = shared_clone({});
bless($foo, 'Foo');

# Create a shared 'Bar' object
my $bar :shared = shared_clone({});
bless($bar, 'Bar');

# Put 'bar' inside 'foo'
$foo->{'bar'} = $bar;

# Rebless the objects via a thread
threads->create(sub {
    # Rebless the outer object
    bless($foo, 'Yin');

    # Cannot directly rebless the inner object
    #bless($foo->{'bar'}, 'Yang');

    # Retrieve and rebless the inner object
    my $obj = $foo->{'bar'};
    bless($obj, 'Yang');
    $foo->{'bar'} = $obj;

})->join();

print(ref($foo),          "\n");    # Prints 'Yin'
print(ref($foo->{'bar'}), "\n");    # Prints 'Yang'
print(ref($bar),          "\n");    # Also prints 'Yang'
```

NOTES

threads::shared is designed to disable itself silently if threads are not available. This allows you to write modules and packages that can be used in both threaded and non-threaded applications.

If you want access to threads, you must `use threads` before you `use threads::shared`. *threads* will emit a warning if you use it after *threads::shared*.

WARNINGS

- cond_broadcast() called on unlocked variable
- cond_signal() called on unlocked variable

  See "cond_signal VARIABLE", above.

BUGS AND LIMITATIONS

When `share` is used on arrays, hashes, array refs or hash refs, any data they contain will be lost.

```
my @arr = qw(foo bar baz);
share(@arr);
# @arr is now empty (i.e., == ());

# Create a 'foo' object
my $foo = { 'data' => 99 };
bless($foo, 'foo');

# Share the object
share($foo);        # Contents are now wiped out
print("ERROR: \$foo is empty\n")
    if (! exists($foo->{'data'}));
```

Therefore, populate such variables **after** declaring them as shared. (Scalar and scalar refs are not affected by this problem.)

Blessing a shared item after it has been nested in another shared item does not propagate the blessing to the shared reference:

```
my $foo = &share({});
my $bar = &share({});
$bar->{foo} = $foo;
bless($foo, 'baz');   # $foo is now of class 'baz',
                      # but $bar->{foo} is unblessed.
```

Therefore, you should bless objects before sharing them.

It is often not wise to share an object unless the class itself has been written to support sharing. For example, a shared object's destructor may get called multiple times, once for each thread's scope exit, or may not get called at all if it is embedded inside another shared object. Another issue is that the contents of hash-based objects will be lost due to the above mentioned limitation. See `examples/class.pl` (in the CPAN distribution of this module) for how to create a class that supports object sharing.

Destructors may not be called on objects if those objects still exist at global destruction time. If the destructors must be called, make sure there are no circular references and that nothing is referencing the objects before the program ends.

Does not support `splice` on arrays. Does not support explicitly changing array lengths via $#array – use `push` and pop instead.

Taking references to the elements of shared arrays and hashes does not autovivify the

elements, and neither does slicing a shared array/hash over non-existent indices/keys autovivify the elements.

`share()` allows you to `share($hashref->{key})` and `share($arrayref->[idx])` without giving any error message. But the `$hashref->{key}` or `$arrayref->[idx]` is **not** shared, causing the error "lock can only be used on shared values" to occur when you attempt to `lock($hashref->{key})` or `lock($arrayref->[idx])` in another thread.

Using `refaddr()` is unreliable for testing whether or not two shared references are equivalent (e.g., when testing for circular references). Use is_shared(), instead:

```
use threads;
use threads::shared;
use Scalar::Util qw(refaddr);

# If ref is shared, use threads::shared's internal ID.
# Otherwise, use refaddr().
my $addr1 = is_shared($ref1) || refaddr($ref1);
my $addr2 = is_shared($ref2) || refaddr($ref2);

if ($addr1 == $addr2) {
    # The refs are equivalent
}
```

each() does not work properly on shared references embedded in shared structures. For example:

```
my %foo :shared;
$foo{'bar'} = shared_clone({'a'=>'x', 'b'=>'y', 'c'=>'z'});

while (my ($key, $val) = each(%{$foo{'bar'}})) {
    ...
}
```

Either of the following will work instead:

```
my $ref = $foo{'bar'};
while (my ($key, $val) = each(%{$ref})) {
    ...
}

foreach my $key (keys(%{$foo{'bar'}})) {
    my $val = $foo{'bar'}{$key};
    ...
}
```

This module supports dual-valued variables created using `dualvar()` from *Scalar::Util*. However, while `$!` acts like a dualvar, it is implemented as a tied SV. To propagate its value, use the follow construct, if needed:

```
my $errno :shared = dualvar($!,$!);
```

View existing bug reports at, and submit any new bugs, problems, patches, etc. to: http:

//rt.cpan.org/Public/Dist/Display.html?Name=threads-shared

SEE ALSO

threads::shared on MetaCPAN: https://metacpan.org/release/threads-shared

Code repository for CPAN distribution: https://github.com/Dual-Life/threads-shared

threads, perlthrtut

http://www.perl.com/pub/a/2002/06/11/threads.html and http://www.perl.com/pub/a/2002/09/04/threads.html

Perl threads mailing list: http://lists.perl.org/list/ithreads.html

Sample code in the *examples* directory of this distribution on CPAN.

AUTHOR

Artur Bergman <sky AT crucially DOT net>

Documentation borrowed from the old Thread.pm.

CPAN version produced by Jerry D. Hedden <jdhedden AT cpan DOT org>.

LICENSE

threads::shared is released under the same license as Perl.

5.2.158 utf8

Perl pragma to enable/disable UTF-8 (or UTF-EBCDIC) in source code

SYNOPSIS

```
use utf8;
no utf8;

# Convert the internal representation of a Perl scalar to/from UTF-
 ↪8.

$num_octets = utf8::upgrade($string);
$success    = utf8::downgrade($string[, $fail_ok]);

# Change each character of a Perl scalar to/from a series of
# characters that represent the UTF-8 bytes of each original␣
 ↪character.

utf8::encode($string);  # "\x{100}"  becomes "\xc4\x80"
utf8::decode($string);  # "\xc4\x80" becomes "\x{100}"

# Convert a code point from the platform native character set to
# Unicode, and vice-versa.
```

(continues on next page)

(continued from previous page)

```
$unicode = utf8::native_to_unicode(ord('A')); # returns 65 on both
                                              # ASCII and EBCDIC
                                              # platforms
$native = utf8::unicode_to_native(65);        # returns 65 on ASCII
                                              # platforms; 193 on
                                              # EBCDIC

$flag = utf8::is_utf8($string); # since Perl 5.8.1
$flag = utf8::valid($string);
```

DESCRIPTION

The `use utf8` pragma tells the Perl parser to allow UTF-8 in the program text in the current lexical scope. The `no utf8` pragma tells Perl to switch back to treating the source text as literal bytes in the current lexical scope. (On EBCDIC platforms, technically it is allowing UTF-EBCDIC, and not UTF-8, but this distinction is academic, so in this document the term UTF-8 is used to mean both).

Do not use this pragma for anything else than telling Perl that your script is written in UTF-8. The utility functions described below are directly usable without `use utf8;`.

Because it is not possible to reliably tell UTF-8 from native 8 bit encodings, you need either a Byte Order Mark at the beginning of your source code, or `use utf8;`, to instruct perl.

When UTF-8 becomes the standard source format, this pragma will effectively become a no-op.

See also the effects of the `-C` switch and its cousin, the `PERL_UNICODE` environment variable, in perlrun.

Enabling the `utf8` pragma has the following effect:

- Bytes in the source text that are not in the ASCII character set will be treated as being part of a literal UTF-8 sequence. This includes most literals such as identifier names, string constants, and constant regular expression patterns.

Note that if you have non-ASCII, non-UTF-8 bytes in your script (for example embedded Latin-1 in your string literals), `use utf8` will be unhappy. If you want to have such bytes under `use utf8`, you can disable this pragma until the end the block (or file, if at top level) by `no utf8;`.

Utility functions

The following functions are defined in the `utf8::` package by the Perl core. You do not need to say `use utf8` to use these and in fact you should not say that unless you really want to have UTF-8 source code.

- `$num_octets = utf8::upgrade($string)`

  (Since Perl v5.8.0) Converts in-place the internal representation of the string from an octet sequence in the native encoding (Latin-1 or EBCDIC) to UTF-8. The logical character sequence itself is unchanged. If *$string* is already upgraded, then this is a no-op. Returns the number of octets necessary to represent the string as UTF-8.

Since Perl v5.38, if `$string` is `undef` no action is taken; prior to that, it would be converted to be defined and zero-length.

If your code needs to be compatible with versions of perl without `use feature 'unicode_strings';`, you can force Unicode semantics on a given string:

```
# force unicode semantics for $string without the
# "unicode_strings" feature
utf8::upgrade($string);
```

For example:

```
# without explicit or implicit use feature 'unicode_strings'
my $x = "\xDF";    # LATIN SMALL LETTER SHARP S
$x =~ /ss/i;       # won't match
my $y = uc($x);    # won't convert
utf8::upgrade($x);
$x =~ /ss/i;       # matches
my $z = uc($x);    # converts to "SS"
```

Note that this function does not handle arbitrary encodings; use *Encode* instead.

- `$success = utf8::downgrade($string[, $fail_ok])`

  (Since Perl v5.8.0) Converts in-place the internal representation of the string from UTF-8 to the equivalent octet sequence in the native encoding (Latin-1 or EBCDIC). The logical character sequence itself is unchanged. If *$string* is already stored as native 8 bit, then this is a no-op. Can be used to make sure that the UTF-8 flag is off, e.g. when you want to make sure that the substr() or length() function works with the usually faster byte algorithm.

  Fails if the original UTF-8 sequence cannot be represented in the native 8 bit encoding. On failure dies or, if the value of *$fail_ok* is true, returns false.

  Returns true on success.

  If your code expects an octet sequence this can be used to validate that you've received one:

```
# throw an exception if not representable as octets
utf8::downgrade($string)

# or do your own error handling
utf8::downgrade($string, 1) or die "string must be octets";
```

  Note that this function does not handle arbitrary encodings; use *Encode* instead.

- `utf8::encode($string)`

  (Since Perl v5.8.0) Converts in-place the character sequence to the corresponding octet sequence in Perl's extended UTF-8. That is, every (possibly wide) character gets replaced with a sequence of one or more characters that represent the individual UTF-8 bytes of the character. The UTF8 flag is turned off. Returns nothing.

```
my $x = "\x{100}"; # $x contains one character, with ord 0x100
utf8::encode($x);  # $x contains two characters, with ords (on
                   # ASCII platforms) 0xc4 and 0x80.  On EBCDIC
                   # 1047, this would instead be 0x8C and 0x41.
```

Similar to:

```
use Encode;
$x = Encode::encode("utf8", $x);
```

Note that this function does not handle arbitrary encodings; use *Encode* instead.

- `$success = utf8::decode($string)`

  (Since Perl v5.8.0) Attempts to convert in-place the octet sequence encoded in Perl's extended UTF-8 to the corresponding character sequence. That is, it replaces each sequence of characters in the string whose ords represent a valid (extended) UTF-8 byte sequence, with the corresponding single character. The UTF-8 flag is turned on only if the source string contains multiple-byte UTF-8 characters. If *$string* is invalid as extended UTF-8, returns false; otherwise returns true.

```
my $x = "\xc4\x80"; # $x contains two characters, with ords
                    # 0xc4 and 0x80
utf8::decode($x);   # On ASCII platforms, $x contains one char,
                    # with ord 0x100.   Since these bytes aren't
                    # legal UTF-EBCDIC, on EBCDIC platforms, $x
 ↪is
                    # unchanged and the function returns FALSE.
my $y = "\xc3\x83\xc2\xab"; This has been encoded twice; this
                    # example is only for ASCII platforms
utf8::decode($y);   # Converts $y to \xc3\xab, returns TRUE;
utf8::decode($y);   # Further converts to \xeb, returns TRUE;
utf8::decode($y);   # Returns FALSE, leaves $y unchanged
```

Note that this function does not handle arbitrary encodings; use *Encode* instead.

- `$unicode = utf8::native_to_unicode($code_point)`

  (Since Perl v5.8.0) This takes an unsigned integer (which represents the ordinal number of a character (or a code point) on the platform the program is being run on) and returns its Unicode equivalent value. Since ASCII platforms natively use the Unicode code points, this function returns its input on them. On EBCDIC platforms it converts from EBCDIC to Unicode.

  A meaningless value will currently be returned if the input is not an unsigned integer.

  Since Perl v5.22.0, calls to this function are optimized out on ASCII platforms, so there is no performance hit in using it there.

- `$native = utf8::unicode_to_native($code_point)`

  (Since Perl v5.8.0) This is the inverse of `utf8::native_to_unicode()`, converting

the other direction. Again, on ASCII platforms, this returns its input, but on EBCDIC platforms it will find the native platform code point, given any Unicode one.

A meaningless value will currently be returned if the input is not an unsigned integer.

Since Perl v5.22.0, calls to this function are optimized out on ASCII platforms, so there is no performance hit in using it there.

- `$flag = utf8::is_utf8($string)`

  (Since Perl 5.8.1) Test whether *$string* is marked internally as encoded in UTF-8. Functionally the same as `Encode::is_utf8($string)`.

  Typically only necessary for debugging and testing, if you need to dump the internals of an SV, *Devel::Peek's* Dump() provides more detail in a compact form.

  If you still think you need this outside of debugging, testing or dealing with filenames, you should probably read perlunitut and "What is "the UTF8 flag"?" in perlunifaq.

  Don't use this flag as a marker to distinguish character and binary data: that should be decided for each variable when you write your code.

  To force unicode semantics in code portable to perl 5.8 and 5.10, call `utf8::upgrade($string)` unconditionally.

- `$flag = utf8::valid($string)`

  [INTERNAL] Test whether *$string* is in a consistent state regarding UTF-8. Will return true if it is well-formed Perl extended UTF-8 and has the UTF-8 flag on **or** if *$string* is held as bytes (both these states are 'consistent'). The main reason for this routine is to allow Perl's test suite to check that operations have left strings in a consistent state.

`utf8::encode` is like `utf8::upgrade`, but the UTF8 flag is cleared. See perlunicode, and the C API functions `sv_utf8_upgrade`, `"sv_utf8_downgrade" in perlapi`, `"sv_utf8_encode" in perlapi`, and `"sv_utf8_decode" in perlapi`, which are wrapped by the Perl functions `utf8::upgrade`, `utf8::downgrade`, `utf8::encode` and `utf8::decode`. Also, the functions `utf8::is_utf8`, `utf8::valid`, `utf8::encode`, `utf8::decode`, `utf8::upgrade`, and `utf8::downgrade` are actually internal, and thus always available, without a `require utf8` statement.

BUGS

Some filesystems may not support UTF-8 file names, or they may be supported incompatibly with Perl. Therefore UTF-8 names that are visible to the filesystem, such as module names may not work.

SEE ALSO

perlunitut, perluniintro, perlrun, *bytes*, perlunicode

5.2.159 vars

Perl pragma to predeclare global variable names

SYNOPSIS

```
use vars qw($frob @mung %seen);
```

DESCRIPTION

NOTE: For use with variables in the current package for a single scope, the functionality provided by this pragma has been superseded by `our` declarations, available in Perl v5.6.0 or later, and use of this pragma is discouraged. See *"our" in perlfunc*.

This pragma will predeclare all the variables whose names are in the list, allowing you to use them under `use strict`, and disabling any typo warnings for them.

Unlike pragmas that affect the `$^H` hints variable, the `use vars` and `use subs` declarations are not lexically scoped to the block they appear in: they affect the entire package in which they appear. It is not possible to rescind these declarations with `no vars` or `no subs`.

Packages such as the **AutoLoader** and **SelfLoader** that delay loading of subroutines within packages can create problems with package lexicals defined using `my()`. While the **vars** pragma cannot duplicate the effect of package lexicals (total transparency outside of the package), it can act as an acceptable substitute by pre-declaring global symbols, ensuring their availability to the later-loaded routines.

See "Pragmatic Modules" in perlmodlib.

5.2.160 version

Turn a version string into an object you can compare, print, and introspect.

Perl has three spellings for a version, and mixing them in one program is normal. `version` gives you a single object type that handles all three so `$a <=> $b` always does the right thing.

- **Decimal** - a plain number like `1.023` or `"1.02_03"`. One integer part, then three-digit chunks after the dot. An underscore marks an alpha release.
- **Dotted-integer** - two or more integers separated by dots, e.g. `1.2.3`. No `v` prefix required, but three or more components are treated as dotted-integer regardless.
- **v-string** - an explicit v prefix, e.g. `v1.23.45`. This form is recommended for `$VERSION` declarations and is required to pass `is_strict`.

Synopsis

```
use version 0.77;
our $VERSION = version->declare("v1.2.3");
my $v = version->parse("1.0203");
if ($v1 == $v2)  { ... }                # numeric equality
@sorted = sort { $a <=> $b } @versions; # ordering
```

`use version` imports qv into the caller's package and installs `version::_VERSION` as `UNIVERSAL::VERSION`, so `$module->VERSION($min)` compares with version-object semantics; `$module->VERSION` itself still returns a plain string.

Operators

version objects are overloaded so you can use the normal Perl comparison operators directly:

- ==, != - numeric equality
- <, <=, >, >=, <=> - numeric ordering
- eq, ne, cmp - same thing, stringwise spelling
- "" - stringifies to original form (dotted) or decimal, whichever the object was created with
- 0+ - numifies to a decimal

The overload fallback is on, so any relational operator not listed above works by auto-generation from <=>.

Functions

Introspection

is_lax

Test whether a string is acceptable to `parse` / `new` as a version.

is_strict

Test whether a string is a strict version suitable for CPAN indexing.

is_lax

Test whether a string is acceptable to `parse` / `new` as a version.

Synopsis

```
version::is_lax($str);
is_lax($str);            # after use version qw(is_lax)
```

Accepts anything the constructor would accept: decimal, dotted-integer, v-string, with or without an underscore for alpha. Use it to validate user input before feeding it to `parse`.

What you get back

A boolean: 1 if the string parses as any valid version, the empty string otherwise.

Examples

```
version::is_lax("1.0203");      # 1
version::is_lax("v1.2.3");      # 1
version::is_lax("1.02_03");     # 1
version::is_lax("");            # ""
version::is_lax("abc");         # ""
```

Differences from upstream

Fully compatible with upstream version 0.9934.

See also

- `is_strict` - the tighter validator, used for CPAN-indexable versions
- new - accepts the same set of strings

is_strict

Test whether a string is a strict version suitable for CPAN indexing.

Synopsis

```
version::is_strict($str);
is_strict($str);         # after use version qw(is_strict)
```

Strict versions rule out alpha underscores and unusual shapes. A decimal version is an integer without leading zeros, optionally followed by one dot and a digit run; a dotted-integer version must start with v and have at least three components. This is the form PAUSE expects for permanent releases.

What you get back

A boolean: 1 if the string matches the strict grammar, the empty string otherwise.

Examples

```
version::is_strict("1.02");         # 1
version::is_strict("v1.2.3");       # 1
version::is_strict("1.2.3");        # ""  - needs the v prefix
version::is_strict("1.02_03");      # ""  - underscores disallowed
version::is_strict("v1.2");         # ""  - needs 3+ components
```

Differences from upstream

Fully compatible with upstream version 0.9934.

See also

- `is_lax` - the permissive counterpart
- `declare` - the recommended way to produce a strict dotted-integer

5.2.161 warnings

Perl pragma to control optional warnings

SYNOPSIS

```
use warnings;
no warnings;

# Standard warnings are enabled by use v5.35 or above
use v5.35;

use warnings "all";
no warnings "uninitialized";

# or equivalent to those last two ...
use warnings qw(all -uninitialized);

use warnings::register;
if (warnings::enabled()) {
    warnings::warn("some warning");
}

if (warnings::enabled("void")) {
    warnings::warn("void", "some warning");
}

if (warnings::enabled($object)) {
    warnings::warn($object, "some warning");
}

warnings::warnif("some warning");
warnings::warnif("void", "some warning");
warnings::warnif($object, "some warning");
```

DESCRIPTION

The `warnings` pragma gives control over which warnings are enabled in which parts of a Perl program. It's a more flexible alternative for both the command line flag **-w** and the equivalent Perl variable, `$^W`.

This pragma works just like the `strict` pragma. This means that the scope of the warning pragma is limited to the enclosing block. It also means that the pragma setting will not leak across files (via `use`, `require` or do). This allows authors to independently define the degree of warning checks that will be applied to their module.

By default, optional warnings are disabled, so any legacy code that doesn't attempt to control the warnings will work unchanged.

All warnings are enabled in a block by either of these:

```
use warnings;
use warnings 'all';
```

Similarly all warnings are disabled in a block by either of these:

```
no warnings;
no warnings 'all';
```

For example, consider the code below:

```
use warnings;
my @x;
{
    no warnings;
    my $y = @x[0];
}
my $z = @x[0];
```

The code in the enclosing block has warnings enabled, but the inner block has them disabled. In this case that means the assignment to the scalar `$z` will trip the `"Scalar value @x[0] better written as $x[0]"` warning, but the assignment to the scalar `$y` will not.

All warnings are enabled automatically within the scope of a `use v5.35` (or higher) declaration.

Default Warnings and Optional Warnings

Before the introduction of lexical warnings, Perl had two classes of warnings: mandatory and optional.

As its name suggests, if your code tripped a mandatory warning, you would get a warning whether you wanted it or not. For example, the code below would always produce an `"isn't numeric"` warning about the "2:".

```
my $x = "2:" + 3;
```

With the introduction of lexical warnings, mandatory warnings now become *default* warnings. The difference is that although the previously mandatory warnings are still enabled by default, they can then be subsequently enabled or disabled with the lexical warning pragma. For example, in the code below, an `"isn't numeric"` warning will only be reported for the `$x` variable.

```
my $x = "2:" + 3;
no warnings;
my $y = "2:" + 3;
```

Note that neither the **-w** flag or the `$^W` can be used to disable/enable default warnings. They are still mandatory in this case.

"Negative warnings"

As a convenience, you can (as of Perl 5.34) pass arguments to the `import()` method both positively and negatively. Negative warnings are those with a `-` sign prepended to their names; positive warnings are anything else. This lets you turn on some warnings and turn off others in one command. So, assuming that you've already turned on a bunch of warnings but want to tweak them a bit in some block, you can do this:

```
{
    use warnings qw(uninitialized -redefine);
    ...
}
```

which is equivalent to:

```
{
    use warnings qw(uninitialized);
    no warnings qw(redefine);
    ...
}
```

The argument list is processed in the order you specify. So, for example, if you don't want to be warned about use of experimental features, except for `somefeature` that you really dislike, you can say this:

```
use warnings qw(all -experimental experimental::somefeature);
```

which is equivalent to:

```
use warnings 'all';
no warnings  'experimental';
use warnings 'experimental::somefeature';
```

As experimental features become regular features of Perl, the corresponding warnings are not printed anymore. They also stop being listed in the "Category Hierarchy" below.

It is still possible to request turning on or off these warnings, but doing so has no effect.

What's wrong with -w and $^W

Although very useful, the big problem with using **-w** on the command line to enable warnings is that it is all or nothing. Take the typical scenario when you are writing a Perl program. Parts of the code you will write yourself, but it's very likely that you will make use of pre-written Perl modules. If you use the **-w** flag in this case, you end up enabling warnings in pieces of code that you haven't written.

Similarly, using `$^W` to either disable or enable blocks of code is fundamentally flawed. For a start, say you want to disable warnings in a block of code. You might expect this to be enough to do the trick:

```
{
    local ($^W) = 0;
    my $x =+ 2;
```

(continues on next page)

(continued from previous page)

```
    my $y; chop $y;
}
```

When this code is run with the **-w** flag, a warning will be produced for the `$x` line: `"Reversed += operator"`.

The problem is that Perl has both compile-time and run-time warnings. To disable compile-time warnings you need to rewrite the code like this:

```
{
    BEGIN { $^W = 0 }
    my $x =+ 2;
    my $y; chop $y;
}
```

And note that unlike the first example, this will permanently set `$^W` since it cannot both run during compile-time and be localized to a run-time block.

The other big problem with `$^W` is the way you can inadvertently change the warning setting in unexpected places in your code. For example, when the code below is run (without the **-w** flag), the second call to `doit` will trip a `"Use of uninitialized value"` warning, whereas the first will not.

```
sub doit
{
    my $y; chop $y;
}

doit();

{
    local ($^W) = 1;
    doit()
}
```

This is a side-effect of `$^W` being dynamically scoped.

Lexical warnings get around these limitations by allowing finer control over where warnings can or can't be tripped.

Controlling Warnings from the Command Line

There are three Command Line flags that can be used to control when warnings are (or aren't) produced:

- **-w**

  This is the existing flag. If the lexical warnings pragma is **not** used in any of your code, or any of the modules that you use, this flag will enable warnings everywhere. See "Backward Compatibility" for details of how this flag interacts with lexical warnings.

- **-W**

If the **-W** flag is used on the command line, it will enable all warnings throughout the program regardless of whether warnings were disabled locally using `no warnings` or `$^W =0`. This includes all files that get included via `use`, `require` or do. Think of it as the Perl equivalent of the "lint" command.

- **-X**

  Does the exact opposite to the **-W** flag, i.e. it disables all warnings.

Backward Compatibility

If you are used to working with a version of Perl prior to the introduction of lexically scoped warnings, or have code that uses both lexical warnings and `$^W`, this section will describe how they interact.

How Lexical Warnings interact with **-w**/`$^W`:

1. If none of the three command line flags (**-w**, **-W** or **-X**) that control warnings is used and neither `$^W` nor the `warnings` pragma are used, then default warnings will be enabled and optional warnings disabled. This means that legacy code that doesn't attempt to control the warnings will work unchanged.
2. The **-w** flag just sets the global `$^W` variable as in 5.005. This means that any legacy code that currently relies on manipulating `$^W` to control warning behavior will still work as is.
3. Apart from now being a boolean, the `$^W` variable operates in exactly the same horrible uncontrolled global way, except that it cannot disable/enable default warnings.
4. If a piece of code is under the control of the `warnings` pragma, both the `$^W` variable and the **-w** flag will be ignored for the scope of the lexical warning.
5. The only way to override a lexical warnings setting is with the **-W** or **-X** command line flags.

The combined effect of 3 & 4 is that it will allow code which uses the `warnings` pragma to control the warning behavior of `$^W`-type code (using a `local $^W=0`) if it really wants to, but not vice-versa.

Category Hierarchy

A hierarchy of "categories" have been defined to allow groups of warnings to be enabled/disabled in isolation.

The current hierarchy is:

```
all -+
     |
     +- closure
     |
     +- deprecated -----+
     |                  |
     |                  +- deprecated::delimiter_will_be_paired
     |                  |
```

(continues on next page)

(continued from previous page)

```
    |                   +- deprecated::dot_in_inc
    |                   |
    |                   +- deprecated::subsequent_use_version
    |                   |
    |                   +- deprecated::unicode_property_name
    |                   |
    |                   +- deprecated::version_downgrade
    |
    +- exiting
    |
    +- experimental ---+
    |                   |
    |                   +- experimental::args_array_with_signatures
    |                   |
    |                   +- experimental::builtin
    |                   |
    |                   +- experimental::class
    |                   |
    |                   +- experimental::declared_refs
    |                   |
    |                   +- experimental::defer
    |                   |
    |                   +- experimental::enhanced_xx
    |                   |
    |                   +- experimental::extra_paired_delimiters
    |                   |
    |                   +- experimental::keyword_all
    |                   |
    |                   +- experimental::keyword_any
    |                   |
    |                   +- experimental::private_use
    |                   |
    |                   +- experimental::re_strict
    |                   |
    |                   +- experimental::refaliasing
    |                   |
    |                   +- experimental::regex_sets
    |                   |
    |                   +- experimental::signature_named_parameters
    |                   |
    |                   +- experimental::try
    |                   |
    |                   +- experimental::uniprop_wildcards
    |                   |
    |                   +- experimental::vlb
    |
    +- glob
    |
    +- imprecision
```

(continues on next page)

(continued from previous page)

```
    |
    +- io -------------+
    |                  |
    |                  +- closed
    |                  |
    |                  +- exec
    |                  |
    |                  +- layer
    |                  |
    |                  +- newline
    |                  |
    |                  +- pipe
    |                  |
    |                  +- syscalls
    |                  |
    |                  +- unopened
    |
    +- locale
    |
    +- misc
    |
    +- missing
    |
    +- missing_import
    |
    +- numeric
    |
    +- once
    |
    +- overflow
    |
    +- pack
    |
    +- portable
    |
    +- recursion
    |
    +- redefine
    |
    +- redundant
    |
    +- regexp
    |
    +- scalar
    |
    +- severe ---------+
    |                  |
    |                  +- debugging
    |                  |
```

(continues on next page)

(continued from previous page)

```
    |                   +- inplace
    |                   |
    |                   +- internal
    |                   |
    |                   +- malloc
    |
    +- shadow
    |
    +- signal
    |
    +- substr
    |
    +- syntax ----------+
    |                   |
    |                   +- ambiguous
    |                   |
    |                   +- bareword
    |                   |
    |                   +- digit
    |                   |
    |                   +- illegalproto
    |                   |
    |                   +- parenthesis
    |                   |
    |                   +- precedence
    |                   |
    |                   +- printf
    |                   |
    |                   +- prototype
    |                   |
    |                   +- qw
    |                   |
    |                   +- reserved
    |                   |
    |                   +- semicolon
    |
    +- taint
    |
    +- threads
    |
    +- uninitialized
    |
    +- unpack
    |
    +- untie
    |
    +- utf8 -----------+
    |                   |
    |                   +- non_unicode
```

(continues on next page)

(continued from previous page)

```
    |                   |
    |                   +- nonchar
    |                   |
    |                   +- surrogate
    |
    +- void
```

Just like the "strict" pragma any of these categories can be combined

```
use warnings qw(void redefine);
no warnings qw(io syntax untie);
```

Also like the "strict" pragma, if there is more than one instance of the `warnings` pragma in a given scope the cumulative effect is additive.

```
use warnings qw(void); # only "void" warnings enabled
...
use warnings qw(io);   # only "void" & "io" warnings enabled
...
no warnings qw(void);  # only "io" warnings enabled
```

To determine which category a specific warning has been assigned to see *perldiag*.

Note: Before Perl 5.8.0, the lexical warnings category "deprecated" was a sub-category of the "syntax" category. It is now a top-level category in its own right.

Note: Before 5.21.0, the "missing" lexical warnings category was internally defined to be the same as the "uninitialized" category. It is now a top-level category in its own right.

Fatal Warnings

The presence of the word "FATAL" in the category list will escalate warnings in those categories into fatal errors in that lexical scope.

NOTE: FATAL warnings should be used with care, particularly `FATAL => 'all'`.

Libraries using warnings::warn for custom warning categories generally don't expect warnings::warn to be fatal and can wind up in an unexpected state as a result.

Moreover, the Perl interpreter itself has had serious bugs involving fatalized warnings. For a summary of resolved and unresolved problems as of January 2015, please see this perl5-porters post.

While some developers find fatalizing some warnings to be a useful defensive programming technique, using `FATAL => 'all'` to fatalize all possible warning categories – including custom ones – is particularly risky. Therefore, the use of `FATAL => 'all'` is discouraged.

The strictures module on CPAN offers one example of a warnings subset that the module's authors believe is relatively safe to fatalize.

NOTE: Users of FATAL warnings, especially those using `FATAL => 'all'`, should be fully aware that they are risking future portability of their programs by doing so. Perl

makes absolutely no commitments to not introduce new warnings or warnings categories in the future; indeed, we explicitly reserve the right to do so. Code that may not warn now may warn in a future release of Perl if the Perl5 development team deems it in the best interests of the community to do so. Should code using FATAL warnings break due to the introduction of a new warning we will NOT consider it an incompatible change. Users of FATAL warnings should take special caution during upgrades to check to see if their code triggers any new warnings and should pay particular attention to the fine print of the documentation of the features they use to ensure they do not exploit features that are documented as risky, deprecated, or unspecified, or where the documentation says "so don't do that", or anything with the same sense and spirit. Use of such features in combination with FATAL warnings is ENTIRELY AT THE USER'S RISK.

The following documentation describes how to use FATAL warnings but the perl5 porters strongly recommend that you understand the risks before doing so, especially for library code intended for use by others, as there is no way for downstream users to change the choice of fatal categories.

In the code below, the use of `time`, `length` and `join` can all produce a `"Useless use of xxx in void context"` warning.

```
use warnings;

time;

{
    use warnings FATAL => qw(void);
    length "abc";
}

join "", 1,2,3;

print "done\n";
```

When run it produces this output

```
Useless use of time in void context at fatal line 3.
Useless use of length in void context at fatal line 7.
```

The scope where `length` is used has escalated the `void` warnings category into a fatal error, so the program terminates immediately when it encounters the warning.

To explicitly turn off a "FATAL" warning you just disable the warning it is associated with. So, for example, to disable the "void" warning in the example above, either of these will do the trick:

```
no warnings qw(void);
no warnings FATAL => qw(void);
```

If you want to downgrade a warning that has been escalated into a fatal error back to a normal warning, you can use the "NONFATAL" keyword. For example, the code below will promote all warnings into fatal errors, except for those in the "syntax" category.

```
use warnings FATAL => 'all', NONFATAL => 'syntax';
```

As of Perl 5.20, instead of `use warnings FATAL => 'all';` you can use:

```
use v5.20;       # Perl 5.20 or greater is required for the␣
 ↪following
use warnings 'FATAL';  # short form of "use warnings FATAL => 'all';
 ↪"
```

However, you should still heed the guidance earlier in this section against using `use warnings FATAL => 'all';`.

If you want your program to be compatible with versions of Perl before 5.20, you must use `use warnings FATAL => 'all';` instead. (In previous versions of Perl, the behavior of the statements `use warnings 'FATAL';`, `use warnings 'NONFATAL';` and `no warnings 'FATAL';` was unspecified; they did not behave as if they included the `=> 'all'` portion. As of 5.20, they do.)

Reporting Warnings from a Module

The `warnings` pragma provides a number of functions that are useful for module authors. These are used when you want to report a module-specific warning to a calling module that has enabled warnings via the `warnings` pragma.

Consider the module `MyMod::Abc` below.

```
package MyMod::Abc;

use warnings::register;

sub open {
    my $path = shift;
    if ($path !~ m#^/#) {
        warnings::warn("changing relative path to /var/abc")
            if warnings::enabled();
        $path = "/var/abc/$path";
    }
}

1;
```

The call to `warnings::register` will create a new warnings category called "MyMod::Abc", i.e. the new category name matches the current package name. The open function in the module will display a warning message if it gets given a relative path as a parameter. This warning will only be displayed if the code that uses `MyMod::Abc` has actually enabled them with the `warnings` pragma as below - note that a plain `use warnings` enables even warnings that have not yet been registered.

```
use warnings;
use MyMod::Abc;
...
abc::open("../fred.txt");
```

The specific warning can be enabled or disabled, but only after the module has been imported:

```
# no warnings 'MyMod::Abc';     # error, unknown category before
                                # the module is loaded
use MyMod::Abc;
no warnings 'MyMod::Abc';       # ok after the module is loaded
...
abc::open("../fred.txt");
```

It is also possible to test whether the pre-defined warnings categories are set in the calling module with the `warnings::enabled` function. Consider this snippet of code:

```
package MyMod::Abc;

sub open2 {
    if (warnings::enabled("deprecated")) {
        warnings::warn("deprecated",
                       "open2 is deprecated, use open instead");
    }
    open(@_);
}

sub open
...
1;
```

The function open2 has been deprecated, so code has been included to display a warning message whenever the calling module has (at least) the "deprecated" warnings category enabled. Something like this, say.

```
use warnings 'deprecated';
use MyMod::Abc;
...
MyMod::Abc::open2($filename);
```

Either the `warnings::warn` or `warnings::warnif` function should be used to actually display the warnings message. This is because they can make use of the feature that allows warnings to be escalated into fatal errors. So in this case

```
use MyMod::Abc;
use warnings FATAL => 'MyMod::Abc';
...
MyMod::Abc::open('../fred.txt');
```

the `warnings::warnif` function will detect this and die after displaying the warning message.

The three warnings functions, `warnings::warn`, `warnings::warnif` and `warnings::enabled` can optionally take an object reference in place of a category name. In this case the functions will use the class name of the object as the warnings category.

Consider this example:

```
package Original;

no warnings;
use warnings::register;

sub new
{
    my $class = shift;
    bless [], $class;
}

sub check
{
    my $self = shift;
    my $value = shift;

    if ($value % 2 && warnings::enabled($self))
      { warnings::warn($self, "Odd numbers are unsafe") }
}

sub doit
{
    my $self = shift;
    my $value = shift;
    $self->check($value);
    # ...
}

1;

package Derived;

use warnings::register;
use Original;
our @ISA = qw( Original );
sub new
{
    my $class = shift;
    bless [], $class;
}


1;
```

The code below makes use of both modules, but it only enables warnings from `Derived`.

```
use Original;
use Derived;
use warnings 'Derived';
my $x = Original->new();
```

(continues on next page)

(continued from previous page)

```
$x->doit(1);
my $y = Derived->new();
$x->doit(1);
```

When this code is run only the `Derived` object, `$y`, will generate a warning.

```
Odd numbers are unsafe at main.pl line 7
```

Notice also that the warning is reported at the line where the object is first used.

When registering new categories of warning, you can supply more names to warnings::register like this:

```
package MyModule;
use warnings::register qw(format precision);

...

warnings::warnif('MyModule::format', '...');
```

FUNCTIONS

Note: The functions with names ending in `_at_level` were added in Perl 5.28.

- use warnings::register

  Creates a new warnings category with the same name as the package where the call to the pragma is used.

- warnings::enabled()

  Use the warnings category with the same name as the current package.

  Return TRUE if that warnings category is enabled in the calling module. Otherwise returns FALSE.

- warnings::enabled($category)

  Return TRUE if the warnings category, `$category`, is enabled in the calling module. Otherwise returns FALSE.

- warnings::enabled($object)

  Use the name of the class for the object reference, `$object`, as the warnings category.

  Return TRUE if that warnings category is enabled in the first scope where the object is used. Otherwise returns FALSE.

- warnings::enabled_at_level($category, $level)

  Like `warnings::enabled`, but $level specifies the exact call frame, 0 being the immediate caller.

- warnings::fatal_enabled()

  Return TRUE if the warnings category with the same name as the current package has been set to FATAL in the calling module. Otherwise returns FALSE.

- warnings::fatal_enabled($category)

  Return TRUE if the warnings category `$category` has been set to FATAL in the calling module. Otherwise returns FALSE.

- warnings::fatal_enabled($object)

  Use the name of the class for the object reference, `$object`, as the warnings category.

  Return TRUE if that warnings category has been set to FATAL in the first scope where the object is used. Otherwise returns FALSE.

- warnings::fatal_enabled_at_level($category, $level)

  Like `warnings::fatal_enabled`, but $level specifies the exact call frame, 0 being the immediate caller.

- warnings::warn($message)

  Print `$message` to STDERR.

  Use the warnings category with the same name as the current package.

  If that warnings category has been set to "FATAL" in the calling module then die. Otherwise return.

- warnings::warn($category, $message)

  Print `$message` to STDERR.

  If the warnings category, `$category`, has been set to "FATAL" in the calling module then die. Otherwise return.

- warnings::warn($object, $message)

  Print `$message` to STDERR.

  Use the name of the class for the object reference, `$object`, as the warnings category.

  If that warnings category has been set to "FATAL" in the scope where `$object` is first used then die. Otherwise return.

- warnings::warn_at_level($category, $level, $message)

  Like `warnings::warn`, but $level specifies the exact call frame, 0 being the immediate caller.

- warnings::warnif($message)

  Equivalent to:

```
if (warnings::enabled())
  { warnings::warn($message) }
```

- warnings::warnif($category, $message)

  Equivalent to:

```
if (warnings::enabled($category))
  { warnings::warn($category, $message) }
```

- warnings::warnif($object, $message)

  Equivalent to:

```
if (warnings::enabled($object))
  { warnings::warn($object, $message) }
```

- warnings::warnif_at_level($category, $level, $message)

  Like `warnings::warnif`, but $level specifies the exact call frame, 0 being the immediate caller.

- warnings::register_categories(@names)

  This registers warning categories for the given names and is primarily for use by the warnings::register pragma.

See also "Pragmatic Modules" in perlmodlib and *perldiag*.

Modules

- `warnings::register` - warnings import function

warnings::register

warnings import function

SYNOPSIS

```
use warnings::register;
```

DESCRIPTION

Creates a warnings category with the same name as the current package.

See *"Reporting Warnings from a Module" in warnings* for more information on this module's usage.

CHAPTER

SIX

CHANGELOG

What each version of pperl changed, one page per version, newest first.

A version is the time the version number was in force: it begins when pperl is set to that number and ends when it is set to the next one. The newest page is the version in development; its changes are added as they are made. Every page accounts for all the work of its version: each change is either described in an entry or was internal (tests, benchmarks, build machinery) with no observable effect. An entry opens with its subject in bold, so a page can be scanned for a module or a feature by name.

Version numbers that have no page, such as 0.6.4, 0.6.7 and 0.6.10, were never set; their changes are in the version before them.

6.1 pperl 0.6.25

In development since 2026-10-25.

6.2 pperl 0.6.24

2026-10-15 to 2026-10-25, 457 commits.

6.2.1 JIT and parallelization

- **Loops over arrays sized with $#a stay compiled**: a loop that stores into an array whose size was set with `$#a = $n`, directly or through a reference, now runs as compiled code. It used to fall back to the interpreter on every element until the JIT gave the loop up, and ran about three times slower than the same loop over an array filled with values first.
- **& | ^ under use v5.28 and later are compiled**: with the `bitwise` feature, which `use v5.28` and later versions turn on, the numeric &, | and ^ are separate operations, and a loop holding one of them ran in the interpreter. Such loops are now compiled, signed under `use integer` and unsigned otherwise, as perl computes them.
- **Index arithmetic no longer decides how a statement computes**: in `$y[$n] = $C[$i] + $C[$i + 64]` over an array of floating-point numbers, the integer `$i + 64` made the compiled loop treat the whole statement as integer arithmetic, and every element fell back to the interpreter until the loop was given up. Arithmetic that only computes an array index is now plain integer arithmetic, its overflow

checked, whatever the elements hold. Loops of this kind that ran about four times slower when every value they computed was 0.0, such as a reverb fed silence, now run as fast as on any other input.

- **Float arithmetic survives a run of zeros**: a loop that computes with arrays of floating-point numbers, such as an audio filter fed silence, kept falling back to the interpreter while every value it saw was zero, and was then replaced by a slower form that also ran the signal after the silence. An array of floats that also holds a value perl keeps as an integer too is now recognised as an array of floats.
- **A copied array element keeps its spelling**: in compiled code, copying an array element that holds a number written as a string, such as `my $x = $list->[$i]` of "-2.0", "0.10" or "1e1" after it had been compared numerically, gave the number instead: "-2", "0.1", "10". The copy is now the string, as in perl. Complete::Sah's completions of floating-point values were affected.
- **A `next` out of a branch that was cold no longer crashes**: a loop nest compiled while an `if` branch holding its own loop and a `next` had never run, as in `if ($four) { while (...) { ... } next }`, could crash pperl once that branch ran, when the enclosing sub returned. It runs as under perl now.
- **A loop that increments the position it reads no longer crashes**: a loop such as `for (my $k = 0; $k < length $s; $k++) { $n += ord substr($s, $k, 1) }` aborted pperl once it ran often enough to be compiled. Such loops now run compiled.
- **Negative array indexes stay compiled**: `$a[-1]`, or an index such as `$n - 100` that goes below zero, counts from the end of the array in compiled code. In a loop over an array of plain numbers, each such access used to fall back to the interpreter, and a loop doing it often was given up.
- **Loops over unsigned 64-bit integers stay compiled**: a loop whose numbers go past 9223372036854775807, perl's unsigned integers as in 64-bit hashes, masks and random numbers, or a `C` function returning `uint64`, fell back to the interpreter on every trip until the JIT gave the loop up, and one naming such a number as a constant, such as `0x9E3779B97F4A7C15`, was never compiled. Such loops now run compiled with perl's exact values, under `use integer` too; a result perl computes as a floating-point number still takes the interpreter's path. A loop that only reads a large integer, such as an argument of -9223372036854775808, no longer leaves the compiled code each time it starts either.

6.2.2 Language and runtime

- **END blocks run after an exit during compilation**: when a program exits or dies while it is still being compiled, in a `BEGIN` block or in the import a `-MModule=...` switch calls, the `END` blocks compiled so far now run and can set the exit status, as in perl. A failing test file that exited this way reported success, which broke the test suite of Test::Needs and every CPAN distribution that needs it. An `END` block that sets `$?` to 0 after `exit 5` now makes the exit status 0.
- **Arguments that are not UTF-8**: an argument that is not valid UTF-8, such as a file name in Latin-1, made pperl stop with an internal error before running anything. Arguments, `-e` code, script names and `-I` directories now reach the program byte for byte, as in perl. And `pperl -e CODE a -x` hands its program `a` and `-x` instead of rejecting `-x` as an unknown option.

- **A regex match creates no variable**: a pattern match no longer creates `${^RE_DEBUG_FLAGS}` in `main::`, and no longer looks that variable up on every match. Code that walks the main symbol table while matching names, such as Devel::Symdump, saw it appear under its iterator and warned "Use of each() on hash after insertion".
- **The built-in library is a directory on disk**: the modules built into pperl are a real directory now, `$Config{privlibexp}` under `~/.cache/pperl`, which `@INC`, `%INC` and `__FILE__` name. It is written the first time a program looks at it - a file test, `opendir`, `chdir`, `glob`, `abs_path` or reading a module as a file - never at startup, and holds the modules as perl installs them, documentation included, with perl's `pod/` files and `Config.pod`. Tools that search `@INC` as a filesystem, such as Pod::Find, Module::Path and Module::List, find pperl's own modules now.
- **@INC hooks see built-in modules**: a code reference at the front of `@INC`, as Test::Without::Module and `use lib sub {...}` put it there, is asked for a built-in module before pperl's own copy is used, as in perl. Such a hook can hide a built-in module or serve its own version of it. A directory in `@INC` still never replaces a built-in module.
- **Modules fall back to Perl when their compiled part is missing**: a module whose compiled part pperl does not have now fails to load with perl's message for that case, "Can't locate loadable object for module ... in @INC", instead of "dynamic loading not available". Modules that try their compiled part and go on in Perl without it look for exactly that message, so DateTime and others now run in their Perl implementation where they died before.
- **A built-in module's file runs however it is reached**: when a built-in module's file reaches perl without a plain `require` - served by an `@INC` hook such as lib::filter's or a FatPacker one, read with `do FILE`, or checked with `perl -c` - its compiled part now loads, as perl finds the module's shared object. Such a file used to die "Can't locate loadable object", which took Moo down with lib::filter.
- **Installed scripts run with pperl**: `$Config{perlpath}` and `$Config{startperl}` name the pperl binary that is running. They named `/usr/bin/perl`, so the commands a distribution installs with `pperl --cpan` started the system's perl, or failed on a machine without one.
- **pperl starts faster**: a short script runs in about a quarter less time, as the binary has less than half the addresses to fix up when it is loaded and a run no longer reads the CPU topology unless it goes parallel. The binary is also 5.5 MB smaller. A panic inside pperl itself, which is always a bug, no longer names a source file and line.
- **Catching errors from built-in modules no longer leaks memory**: a built-in module that died with an error built at run time left a few hundred bytes behind every time, so a program catching such errors in a loop with `eval` grew without bound. *Peta::FFI*, *Storable*, *Opcode*, *POSIX* and the other built-in modules now free everything before they die, and `opset` with an unknown name no longer leaks the set it was building, as it does under perl. The error messages are unchanged.

6.2.3 Modules

- ***DBI* 1.655**: the built-in DBI is 1.655, up from 1.651. `DBD::DBM` accepts only a module name as `dbm_type` and `dbm_mldbm`, so a file path given there is no longer loaded and run, and `DBD::File` no longer reads a table linked into a directory whose name merely begins with that of `f_dir`. These are the fixes for CVE-2026-78030 and CVE-2026-15392.
- **DBI: sql_type_cast and fetchrow_hashref as in *DBI* under perl**: `DBI::sql_type_cast` returns what DBI's compiled version returns, so `SQL_INTEGER` refuses "3.14", "1e3" and "inf", `DBIstcf_DISCARD_STRING` leaves only the number, and the two flags have DBI's values. A plain number with `SQL_INTEGER` and `DBIstcf_DISCARD_STRING` is handled too, where DBI 1.655 aborts perl. `fetchrow_hashref` with a key name that holds no column list, from `FetchHashKeyName` or its argument, sets the handle's error and returns undef instead of dying whatever `RaiseError` says, and a number as that name is taken as a name.
- **PDL: update_data_from reads the value of a tied source**: `$pdl->update_data_from($bytes)` fetches a tied or otherwise magical `$bytes` and copies its current value, as *PDL* does under perl. It used to copy whatever that scalar last held.
- ***PDL*: the operator overloads are PDL's own subs**: `PDL::overload_plus` and the other handlers PDL installs for its operators exist under their names and are what `overload::Method` returns, as with PDL under perl. An error raised through an operator, such as a broadcast mismatch in `$a + $b`, now names PDL's own line and the handler in its backtrace, and so does a warning such as "isn't numeric" for a string operand. A redefined `PDL::plus`, or a subclass's own `inplace`, is now what the operators use.
- **PDL::FFT loads quietly under -w**: `use PDL::FFT` under `-w` no longer warns "Subroutine PDL::realfft redefined", as it doesn't with *PDL* under perl.
- ***Pod::Text* and *Pod::Man* 6.1.1**: the POD formatters are those of podlators 6.1.1, up from the 6.0.2 that perl 5.44.0 ships. Formatting a document whose `=over` nesting brings the margin to the output width no longer loops forever while the output fills memory (CVE-2026-82560). `Pod::Man` lists with tags now use `.TP` and the fixed-width font is `CR`, for groff 1.24.
- ***Text::LineFold* no longer repeats its input**: `fold` on text holding form feeds, vertical tabs or other special breaks returned the whole text once for every piece between them, so the output grew with the square of the input (CVE-2026-8594). Each piece is folded once now; the Text::LineFold of CPAN still has the fault.
- ***Hash::Util::FieldHash* is built in**: perl's field hashes for inside-out classes, whose entries vanish with the object they are keyed by, together with `id`, `id_2obj`, `register` and idhashes. It was the one module of perl's own library that would not load, and `B::Hooks::EndOfScope`, `namespace::clean`, `namespace::autoclean` and Mojolicious, which need it, now install from CPAN and pass their tests.
- **Five more CPAN modules with compiled parts are built in**: `XString` 0.005, `Test::LeakTrace` 0.17, `PerlIO::utf8_strict` 0.010, `Scalar::Util::Numeric` 0.40 and `Sub::Identify` 0.14. Each passes its own test suite with the results it has under perl. Distributions that need one of them, such as DateTime, Specio, PPI and Template Toolkit, get past it when they are installed from CPAN.

- ***Sub::Name*** **is built in**: `Sub::Name` 0.28, which renames a sub for `caller` and stack traces, passes its own tests as under perl. It runs the code of `Sub::Util::set_subname`, which perl ships and which was taken from it, so renaming a sub under the debugger no longer leaks a reference to its `%DB::sub` entry with every call, as `Sub::Name` does under perl.
- ***B::COW*** **is built in**: `B::COW` 0.007 reports whether a string shares its buffer copy-on-write and the share count, with the same answers as under perl, and passes its own tests.
- ***Digest::SHA1*** **is built in**: `Digest::SHA1` 2.13, the SHA-1 module older code uses where newer code uses `Digest::SHA`, passes its own tests with the results it has under perl. Its digests are computed by the built-in `Digest::SHA`, with the processor's SHA instructions where it has them. `sha1_transform` given more than 64 bytes uses the first 64, where perl aborts.
- ***String::CRC32*** **is built in**: `String::CRC32` 2.100 computes the CRC-32 of a string or of what a filehandle reads, with the results it has under perl, and passes its own tests. The CRC is computed by the zlib of the built-in `Compress::Raw::Zlib`. A filehandle whose read fails, such as a closed one, gives the CRC of no data, where perl crashes.
- ***Crypt::OpenSSL::Random*** **is built in**: `Crypt::OpenSSL::Random` 0.17 draws bytes from OpenSSL's random-number generator and seeds it, in the builds that include *Net::SSLeay*, and passes its own tests. It reaches libcrypto through Net::SSLeay, so both draw from one generator. Asking for 0 bytes returns the empty string, where perl returns stray memory; a negative count returns false, and 0 from Net::SSLeay's `RAND_bytes`, where perl exits.
- ***URL::Encode*** **is built in**: `URL::Encode` 0.03 and its fast half `URL::Encode::XS` encode and decode URL-encoded strings and split query strings into their parameters, with the results they have under perl, and pass their own tests. The fast half is used, as under perl when it is installed, and is about 17 times as fast as the pure-Perl one.
- ***Text::LevenshteinXS*** **is built in**: `Text::LevenshteinXS` 0.03 computes the edit distance of two strings, with the results it has under perl, and passes its own tests. Long strings work: perl needs memory for a table of both lengths multiplied, and crashes when that memory is not there (two strings of 20000 bytes need 1.6 GB).
- ***Math::Random::MT*** **is built in**: `Math::Random::MT` 1.17, the Mersenne Twister random-number generator, gives the same sequences as under perl for the same seeds, and passes its own tests.
- ***Data::UUID*** **is built in**: `Data::UUID` 1.227 makes time-based and name-based UUIDs and converts them, passing its own tests with the results it has under perl. Its name-based UUIDs are computed by the built-in `Digest::MD5`. Where perl reads or writes past a string - a binary UUID shorter than 16 bytes, a Base64 UUID that is too short, too long or has bytes above 0x7F, a hex UUID that ends inside a byte - the result is zero-filled, cut to 16 bytes, or an error instead.
- ***Params::Util*** **is built in**: `Params::Util` 1.102, whose checks such as `_STRING`, `_HASH` and `_INSTANCE` many CPAN modules validate their arguments with, passes its own tests with the results it has under perl. It did not load at all before, so PPI, Package::DeprecationManager, Config::MVP and the modules that need them could not be installed from CPAN.

- ***Class::Load::XS* is built in**: the compiled `is_class_loaded` of `Class::Load::XS` 0.10, which Class::Load uses when it is there, passes its own tests with the results it has under perl. Class::Load itself, which loads it, is installed from CPAN as before.
- ***Template::Stash::XS* is built in**: the compiled stash of Template Toolkit 3.106, which the Template Toolkit uses by default, passes the Template Toolkit's own tests with the results they have under perl. The Template Toolkit itself, which did not install without it, now installs from CPAN with `pperl --cpan`. A method that dies "Can't locate object method" no longer overwrites the caller's arguments, and `set` on a value that is not a reference dies with an error where perl crashes.
- ***Package::Stash::XS* is built in**: `Package::Stash::XS` 0.32, the compiled implementation of `Package::Stash` that *Moose*, namespace::clean and many others read and change symbol tables through, passes its own tests with the results it has under perl. `Package::Stash` itself comes from CPAN and now uses it, as it does under perl, instead of its pure-Perl implementation.
- ***List::MoreUtils::XS* is built in**: the compiled backend of `List::MoreUtils` 0.430 passes its own tests and those of the front end with the results they have under perl, and `List::MoreUtils`, installed from CPAN, now uses it instead of its pure-Perl implementation. Where perl crashes or answers wrongly, pperl does not: a block that is a compiled sub, arrays with holes, objects overloading `@{}` over a hash, and `<=>` overloading that gives no number. `reduce_u`, `reduce_0` and `reduce_1` no longer overwrite the caller's `$_`, and `frequency` no longer leaks.
- ***Term::ReadKey* is built in**: `Term::ReadKey` 2.38 sets terminal modes, reads keys with a timeout or without blocking, and reports the terminal's size, speed and control characters as it does under perl; modules that need it, such as `CPAN::Uploader`, now install from CPAN. Where perl aborts, pperl does not: a timed read on a closed handle or on a descriptor of 1024 or above, and `SetTerminalSize` with a width or height ten characters long. `GetSpeed` of a handle that is not a terminal returns 0 for both speeds where perl returns leftover memory.
- ***Filter::Util::Exec*, *Filter::tee* and *Filter::decrypt* are built in**: the source filters of Filter 1.65 that run a program through a command (`Filter::exec`, `Filter::sh`, `Filter::m4`), copy it to a file or decrypt it work as they do under perl and pass the dist's own tests. Where perl crashes, pperl does not: these filters, and `filter_add` of *Filter::Util::Call*, called while nothing is being compiled. A command that stops reading early, such as `head -n 1` on a long source, no longer kills the program without a word or leaves it empty: what the command wrote is compiled. `Filter::cpp` is not available, because pperl's configuration names no C preprocessor.
- ***IO::Tty* and *IO::Pty* are built in**: IO-Tty 1.31 allocates pseudo-terminals, names them, sets their window sizes and defines the terminal constants as it does under perl, and passes the dist's own tests. Where perl crashes or reports a length it read from unrelated memory, pperl does not: `IO::Tty::unpack_winsize` of undef, a number or a reference dies saying the length is 0.
- ***Moose* loads**: the compiled part of Moose 2.4001 is built in, so Moose's own modules load and pass Moose's own tests with the results they have under perl. `pperl --cpan` installs Moose and what is built on it, Catalyst and Dist::Zilla among them: a distribution's check for a working C compiler (*ExtUtils::HasCompiler*) answers yes when the compiled part it needs is built in. Where perl crashes on a malformed

metaclass or method object, pperl dies with an error or leaves it alone.

- ***JSON::XS* is built in**: JSON-XS 4.04 encodes and decodes JSON, parses it incrementally and calls `TO_JSON`, `FREEZE` and `THAW` as it does under perl, and passes its own tests; *Types::Serialiser*, whose booleans JSON `true` and `false` decode to, is built in with it. Where perl crashes, pperl does not: a method called on something blessed into `JSON::XS` that `new` did not make dies with an error, and `JSON::XS::CLONE` called by hand no longer breaks `new`.
- ***YAML::XS* and *YAML::PP* are built in**: YAML-LibYAML 0.910.0 loads and dumps YAML with the settings, objects and output it has under perl and passes its own tests, over YAML::PP 0.43.0, pperl's YAML engine. YAML is read as the YAML 1.2 specification reads it, so the few documents libyaml reads otherwise, rejects or wrongly accepts follow the specification, and `Dump` escapes U+2028 and U+2029 so that what it writes loads back unchanged. Where perl crashes or refuses, pperl does not: the object interface loading a flow mapping inside a flow sequence, and dumping strings of latin-1 bytes. A document that does not parse dies with YAML::PP's account of the problem: nearly always the same document and line, in other words.
- ***FFI::Platypus* is built in**: FFI-Platypus 2.12 calls C functions with every type it offers, from integers, strings and arrays to records by value, closures and custom types, as it does under perl, and no C compiler is needed to install it. Where perl fails, pperl does not: reading an array member of a record no longer leaks memory, an index past an opaque array member warns instead of reading beyond the record, and a hash reference passed for a pointer no longer crashes. A regex capture or a tied scalar passed as an argument passes its value, where perl passes 0, or no string at all, which crashes `strlen($1)`, and a callback that a library calls from a thread of its own stops the program with a message where perl crashes.
- ***FFI::Build* is built in**: a CPAN module that ships C code for *FFI::Platypus* to call, built at install time with FFI::Build, now installs, passes its tests and runs under pperl when a C compiler is on `PATH`, as it does under perl. *Capture::Tiny*, which FFI::Build uses, is built in with it.
- ***DynaLoader* loads shared libraries**: `dl_load_file`, `dl_find_symbol`, `dl_unload_file`, `dl_findfile` and the rest work on any C library as they do under perl, and *FFI::CheckLib*, which finds libraries with them, is built in. A module's compiled part is still never loaded from a file: `XSLoader` and `DynaLoader::bootstrap` load only pperl's built-in modules. Unloading a library that failed to load no longer crashes, as it does under perl when FFI::CheckLib meets a linker script such as `libm.so`.
- ***XSLoader* checks the version a module asks for**: `XSLoader::load($module, $VERSION)` dies "... object version ... does not match bootstrap parameter ..." when the built-in compiled part is of another version, as it does under perl, and loads a built-in module's compiled part even when nothing has required the module yet.
- ***Peta::FFI* reaches more of any C library**: signatures gain fixed-width integers from `i8` to `u64`, `bool`, variadic functions such as `ioctl` and `snprintf`, strings the caller must free, and C objects as Perl objects whose destructor runs with their last reference. `function` and `attach` prepare a C function once and call it as a Perl sub, a function pointer can be called by its address, and `errno` reports what a call left behind. A callback that a library calls from a thread of its own now stops the program with a message instead of corrupting it, and *Peta::FFI::EventSource*

brings such a library's events to Perl through a file descriptor that *AnyEvent*, *EV*, Mojo::IOLoop or a `select` loop watches. `scan` finds the libraries in the directories it is given, or in `PETA_FFI_PATH` and the dynamic linker's own, and lists only the files this program can load.

- ***Peta::FFI*** **passes C structs, arrays and long doubles**: `record` declares a C struct as a Perl class with an accessor per member, and a signature passes and returns it by value or by pointer, callbacks included. Signatures gain pointers to numbers and strings for out-parameters, arrays the function reads and changes in place, `long double` and the complex types, and an output buffer that holds what the function wrote. `symbol`, `fetch`, `store` and `variable` read and write a library's global variables and any typed value in memory. *FFI::Platypus* now runs on the same engine, so whatever it can call, Peta::FFI can.
- ***Peta::FFI*** **reaches C unions**: `union` declares one as `record` declares a struct, with every member at its start, to be passed by pointer and read and written in memory, as `SDL_PollEvent` fills an `SDL_Event`. Passing a union by value is refused, because on some platforms that would call the function wrongly instead of failing. A record member that is itself a record now reads as an object of its class and is set from one.
- ***Peta::FFI*** **takes what C functions hand over**: a handle or a string a function stores through a pointer, as `sqlite3_open` does, becomes the caller's object or string, and arrays of handles, of records and of pointers ended by NULL can be passed. An array the function allocates itself, its length given by the result, by another argument or by a NULL at its end, comes back as a Perl array of objects, strings, records or numbers. Everything is freed with the library's own functions once nothing uses it, so a record taken from such an array still reads its strings for as long as it lives.
- ***Peta::FFI*** **reads arrays C lends and sets strings in records**: an array a function hands out from memory it keeps, as libxkbcommon hands out a keymap's keysyms, is written `&T[r]`, `&T[#k]` or `&T[0]` and is copied without being freed, whether it comes through an argument or, as `&T[#k]` or `&T[0]`, as the result. A record's string member can now be set from Perl: the record keeps its own copy for as long as it lives, `undef` passes NULL, and a string C put there is never freed by Perl. When C moves the strings between records, as sorting an array of them does, each string stays alive with the record that now points to it.
- ***Peta::FFI*** **reads byte buffers C allocates**: a buffer a function allocates and hands over through a pointer, with its length in the result or in another argument, is written `*P[r]` or `*P[#k]`. It arrives as one byte string, NULs and all, and is then freed with the library's own function, which is how librados's `rados_mon_command` returns its reply. Bytes a function only lends, `&P[r]` or `&P[#k]`, are copied and never freed, and a callback handed bytes with their length, `&P[#k]`, gets them as a string.
- ***Peta::FFI::Pool*** **runs C calls that may hang**: a function prepared with *Peta::FFI* runs on one of a pool's threads, with a deadline, so `statfs` on a dead NFS mount no longer needs a child process with a timeout. Each call is reported as an event, through a descriptor any event loop can watch as a *Peta::FFI::EventSource*: done, with its result and what it wrote back; timed out; or busy, when every thread is taken. A call that returns after its deadline changes nothing.
- ***Peta::FFI::Queue*** **takes callbacks from a library's own threads**: a closure made with `queue => $queue` may be called on any thread, as librados calls its monitor log and watch callbacks. It copies its arguments into the queue, returns a value given once, and its sub runs later, when Perl's event loop takes the queue's events

through a *Peta::FFI::EventSource*. A queue holds a bounded number of calls and counts those it had to drop.

- **Loops that call C functions run compiled**: a loop calling a C function prepared with *Peta::FFI* or *FFI::Platypus* used to stay out of the JIT entirely. When the function takes and returns plain integers, pointers or floating-point numbers, the compiled loop now calls it directly; other calls, and any call while C holds a Perl callback, still go through the library as before.
- **C functions called often cost less**: a C function prepared with *Peta::FFI*'s `function` or `attach`, or attached with *FFI::Platypus*, that takes and returns plain numbers or pointers is compiled into machine code of its own once it has been called a few thousand times. A call of `labs` or `ldexp` outside a compiled loop then takes about 30% less time, with the same results and the same errors as before. A function that is seldom called is never compiled, so declaring a library's functions costs no more than it did.
- **Peta::LMDB and Peta::PCRE2 are now Peta::FFI::LMDB and Peta::FFI::PCRE2**: pperl's own modules for C libraries live under *Peta::FFI*, beside *Peta::FFI::UUID*. Programs that load *Peta::FFI::LMDB* or *Peta::FFI::PCRE2* work as before, with the class names, error class and `$Peta::FFI::LMDB::DBM_WRITE` renamed to match; the old names no longer load.
- ***Cairo* survives a libcairo without a function it calls**: when the installed libcairo lacks a function a Cairo method needs, as an older release does, calling the method stopped the whole program. It now dies with a message naming the function, which `eval` catches, as the other library modules already did.
- ***EV* and *Coro* no longer need a system perl**: the built-in `EV` and `Coro` start with `use common::sense`, which pperl did not carry, so they loaded only where a system perl's copy was on `@INC`, and `AnyEvent` otherwise fell back to its pure-Perl event loop without a word. `common::sense` 3.75 is built in now, with the warnings and features it sets under perl 5.44.
- ***Event* runs on libev, and EV and Event share signals**: `Event` now waits for descriptors, signals and time through libev, the engine behind `EV`, and reports the same readiness and conditions as before; its timers and event queues are its own, as they were. A signal that both modules watch now reaches the watchers of both. Under perl the module that installed its handler last took it, and an EV loop reading signals through a signalfd hid them from Event entirely.
- ***XML::Parser*, *HTML::TokeParser* and *PDL::IO::Pic* load without a system perl**: they need `File::ShareDir`, `HTML::Tagset` and `File::Which`, which pperl did not carry, and loaded only where a system perl's copy was on `@INC`. Those three are built in now, with `Class::Inspector`, which `File::ShareDir` needs; `HTML::LinkExtor` is fixed with them.
- **XML::Parser parses with libxml2**: `XML::Parser` and `XML::Parser::Expat` read documents with libxml2 2.15 or newer, and the expat library is no longer needed. Handlers see what they saw over expat: the same calls with the same arguments, character data split where expat splits it, the same positions and source text from `current_byte`, `recognized_string` and `original_string`, and expat's error messages. A document with `standalone="yes"` has the parameter entities of its internal subset read, as XML requires; expat left them unread and then refused the document.

- ***Devel::Cover* measures coverage**: the compiled part of *Devel::Cover* 1.52 is built in, so with the distribution installed by `pperl --cpan Devel::Cover`, `pperl -MDevel::Cover` collects statement, branch, condition and subroutine coverage, and `cover` reports it, with the same counts as perl. Devel::Cover's own test suite passes. While it collects, loops are not compiled, since compiled code would bypass the counting.
- **JSON::PP works in compiled code**: `JSON::PP` encodes and decodes with the same compiled engine as `JSON::XS`, with JSON::PP's own options, results and error messages, about 6 times faster at encoding and 14 times at decoding. CPAN's metadata files and everything else that reads or writes JSON through it gain with it. Where JSON::PP went wrong it no longer does: `decode_prefix` counts characters as documented, what a `FREEZE` method returns is encoded as JSON, the incremental parser keeps Latin-1 text and the rest of its buffer as they are, and bare keys keep their non-ASCII characters.
- ***Text::CSV* is built in**: Text::CSV 2.06 comes with pperl, and its pure-Perl engine `Text::CSV_PP` parses, warns and checks its settings as Text::CSV_XS 1.65 does. A CR at the very end of the input after an empty last field is error 2032, as after any other field; a failed `print` returns "" and warns where it was called; a separator, quote or line end longer than 16 bytes is refused. Installing Text::CSV with `pperl --cpan` gets the same engine, and Text::CSV_XS installs over it with all of its own tests passing.

6.2.4 Command line

- **pperl -S finds the program on PATH**: `pperl -S prove` runs the `prove` installed on the search path, as `perl -S` does; pperl rejected the switch.
- **A program's arguments are its own**: pperl's switches, such as `--parallel` or `--no-jit`, are read only before the program, as perl reads its own. After it, they reach the program in `@ARGV`; pperl used to take them out, so `tool.pl --parallel 4 list` saw only `4 list`.
- **pperl -v names the build and credits perl**: the first line names what kind of binary it is, its native tier and added features such as `dbi`, `pdl` or `shell`, the second is the copyright, and the text says pperl is "compatible to perl 5.44 by Larry Wall and others". `pperl -V` still lists everything. Tests that read `-v` as perl's, such as those of IPC::System::Simple and Expect, pass.
- **pperl –cpan no longer asks you to set PPERL5LIB**: pperl finds what it installs into your own tree without it, so the advice that ended every install is gone.
- **pperl –cpan –system installs where pperl looks**: a system-wide install now goes to `/usr/local/share/pperl/site_perl`, which every pperl searches, with its scripts and manuals beside it; it used to land in a directory pperl never looked in. It now stops before downloading anything when the account may not write there, and says to run it as root.
- **pperl –cpan applies distribution preferences**: a CPAN.pm distroprefs file under `~/.pperl/cpan/prefs` that matches a distribution now takes effect, including its patches, which are looked up beside it. A `.yml` preference used to be read only where the separate YAML distribution happened to be installed, and now uses pperl's built-in *YAML::XS*. Any patch a preference named stopped the install, because no patch command was configured.

- **pperl –cpan installs XS distributions it has a substitute for**: each run first fetches pperl's pure-Perl substitutes for distributions with a compiled part from `https://perl.petamem.com/downloads/cppan`, so a substitute published after your pperl applies to it too. Text::CSV_XS installs and runs on the Perl engine of Text::CSV, true, multidimensional and bareword::filehandles on perl's own features, Scalar::Readonly, Test::Taint, Regexp::Util, next::XS and Socket6 on its built-in modules, and Crypt::SSLeay (Net::SSL) on its OpenSSL binding, where it no longer builds under perl with OpenSSL 3; the few places where their own tests show a difference are marked as known in them. `PPERL_CPPAN_URL` names another address, and set empty it skips the fetch.
- **JavaScript::QuickJS runs on QuickJS-NG**: `pperl --cpan` installs JavaScript::QuickJS as a binding over the QuickJS-NG library (`libqjs.so.0`, on Arch Linux the `quickjs-ng` package), used where it is installed; without it the module installs, its tests skip and say why, and it works once the library is there. The distributions that need it, such as Data::Sah and the Perinci family, install with it. Where QuickJS-NG differs from the QuickJS the CPAN distribution bundles, its own behaviour shows: its error messages, and a Date's `getTime` as a floating-point number.
- **pperl –cpan downloads over https with curl or wget**: like perl's own first-run setup, it now finds `make`, `tar`, `gzip`, `gpg`, `wget` or `curl` and the other tools CPAN.pm uses on your `PATH`. A pperl built without TLS support therefore fetches from CPAN over https through `wget` or `curl` instead of falling back to plain http, and the XS substitutes come the same way.
- **pperl –cpan builds the pure-Perl form of a distribution**: it asks every build for the distribution's Perl-only variant, as `PUREPERL_ONLY=1` and `--pureperl-only` do, so a distribution such as Params::Validate that has a Perl implementation beside its compiled one installs instead of failing for want of a C compiler. A module that does not load without its compiled part is named and kept, not removed: *Params::Util* switches to its Perl half when its tests ask for it, and those tests now run.
- ***Config* names the C compiler**: `$Config{cc}`, its flags, its preprocessor and `gccversion` describe the C compiler found on `PATH` (cc, gcc or clang), so *FFI::Build* compiles the C libraries a distribution bundles and Devel::CheckLib's probes run. `usedl` and `dlsrc` say what perl's say on Linux, as *DynaLoader* loads shared libraries. Modules with a compiled part still load only when they are built in, and *ExtUtils::CBuilder*'s `have_compiler` still says no.
- **@INC includes the architecture directories of PERL5LIB and -I**: as under perl, each entry's `x86_64-linux` subdirectory, and that of each of its version directories, comes ahead of the entry when it exists. An installer puts a distribution's Perl files there whenever the distribution carries anything architecture-dependent, so modules installed into a local::lib by `cpanm -l`, Alien modules among them, are now found. `pperl --cpan` installs into its tree's architecture directory as perl's installers do, and pperl's own tree is searched the same way.
- **ipps runs programs as functions**: with the new `autorun` setting on, calling a sub that does not exist runs the program of that name and returns its output, a list of lines in list context, as in `whoami()` or `my @files = ls('*.pl')`. `:programs ls cat` makes such names subs that need no parentheses, whether `autorun` is on or not, and `:programs` alone lists them. A program that cannot be run is an error,

and the exit status is in `$?` as for backticks.

6.2.5 Documentation

- **Links in the changelog**: the entries of every version link the modules, built-in functions, command-line options, guides and tutorials they name to their pages, in every language of the documentation.

6.3 pperl 0.6.23

2026-10-11 to 2026-10-15, 101 commits.

6.3.1 Modules

- ***MIME::Base64*** **uses vector instructions**: on x86-64, `encode_base64` and `decode_base64` work 12 or 24 bytes at a time where the processor has SSSE3 or AVX2. A large text in lines of equal length is now also decoded on several threads, as large data already was encoded; a text laid out any other way, or with stray characters in it, decodes on one thread as before. The results are unchanged.
- ***Digest::SHA*** **uses the processor's SHA instructions**: the object interface (`Digest::SHA->new`, `add`, `addfile`) computes SHA-1 and SHA-256 with the processor's SHA instructions where it has them, as the functions such as `sha256_hex` already did. SHA-1 and SHA-512 are faster without them too. On x86-64 processors without the SHA instructions, SHA-1, SHA-224 and SHA-256 are faster in every form - functions, objects and HMAC.
- ***Encode*** **converts single-byte encodings faster**: `encode` and `decode` for iso-8859-x, the cp125x code pages, koi8 and the other single-byte encodings no longer search a table for every character, and a string perl holds as bytes is encoded without first being converted to UTF-8. A large `encode` or `decode` is also spread over several threads. The results, the check modes and the error messages are unchanged.
- ***Cwd::getcwd*** **is faster**: `getcwd` and the lexer no longer clear buffers of up to 4 KiB that are written before they are read.
- **%Config entries that are not defined are undef**: `$Config{...}` returned the string "undef", which is true, for 232 keys such as `d_faststdio` and `usedl`, so `if ($Config{d_faststdio})` took the branch for something the build does not have. They are undef now, as in perl; `config_vars` and `config_sh` still print them as `'undef'`.
- ***Compress::Raw::Zlib*** **needs no system zlib**: zlib is part of the binary now, as perl builds its own copy into the module, so `IO::Compress`, `Archive::Tar` and everything else on top of it work on a machine without `libz` and in the static build. Compression is 1.7 to 5.7 times as fast and decompression about twice as fast. A compressed stream is valid for every zlib but is not byte-for-byte the one perl writes, and `ZLIB_VERSION` reads `1.3.0-zlib-rs-0.6.8`.
- **SDL_Quit with a** ***Peta::WebGPU*** **surface still alive no longer crashes**: a program that called `SDL_Quit` while a surface or an unpresented frame was still in scope aborted with "double free or corruption" in about one run in ten. `SDL_Quit` now

releases their swap chains first; using such a surface or frame afterwards dies with a message saying the window system was shut down, and the device keeps working.

- ***Tie::Hash::Indexed*** **is built in**: the ordered hash of CPAN's Tie::Hash::Indexed 0.08 loads without installing anything, as a tied hash and with its object interface, iterators and *Storable* support. Three sequences that crash perl with the original module work: `pop`, `shift` or `clear` in the middle of an `each` loop, and `NEXTKEY` before any `FIRSTKEY`.
- ***PDL*****: broadcast_define and the dataflow inspectors**: `broadcast_define` and `thread_define` now exist, with `broadcastover` and `broadcastover_n` under them. `trans_parent` and `trans_children` return `PDL::Trans` objects with their methods and vtable, so `PDL::Core::pdump`, `pdump_trans` and `pdumphash` work, as do `flags` and the `*_nophys` readers. Also new: `add_threading_magic`, `remove_threading_magic`, `broadcast3`, `tocomplex`, `hdr_copy`, `hcpy` and `nslice_if_pdl`.
- **PDL: an ndarray's data is a Perl scalar**: the scalar that `get_dataref` returns is now the data itself for every ndarray above 32 bytes, so writing into it writes the ndarray, as in *PDL* under perl. That makes `new_around_datasv`, `new_around_pointer` and `set_data_by_file_map` (an ndarray over a memory-mapped file) available. `dump`, `bind` and `dowhenidle` are new too, and `datasv_refcount` of an ndarray without such a scalar dies.
- **PDL: ldouble is the platform's long double**: `ldouble` and `cldouble` now hold the C compiler's `long double`, the 80-bit format on x86-64, as *PDL* under perl does; they were a 128-bit type with different results. A single `ldouble` value no longer reads as 0 in `print`, `sclr` and conversions. The complex functions are exact: `csqrt(-4)` is `2i`, not `1.22464679914735e-16+2i`.
- **PDL: histograms, interpolate and the Image2D filters**: new are `histogram`, `whistogram`, their 2-D forms and `whist`; `wtstat`, `vcos`, `interpolate`, `diff2`, `diffcentred`, `partial`, `caverage` and `cavgover`; and in *PDL::Image2D* `conv2d`, `box2d`, `patch2d`, `patchbad2d`, `bilin2d`, `rescale2d` and `crop`. `hist` without bounds now chooses its bins from the data as *PDL* under perl does. `which` no longer counts bad values, `crossp` and `whichover` broadcast and keep the type, and `interpol` fills an ndarray given as fourth argument.
- ***PDL::ImageND***: `use PDL::ImageND` loads, with `convolveND`, `convolve`, `rebin`, `ninterpol`, `kernctr`, `circ_mean`, `circ_mean_p`, `contour_segments`, `contour_polylines`, `path_join`, `path_segs`, `combcoords`, `attract` and `repulse`. *PDL::FFT* imports `kernctr` from it again, as *PDL* under perl does.
- **PDL: range and indexND as in *PDL* under perl**: `range` accepts an index with fewer dimensions than the source and a size per dimension, honours every boundary condition (`forbid`, `truncate`, `extend`, `periodic`, `mirror`, one for all dimensions or one each), keeps the type, and its result is linked to the source, so a write into it reaches the source. `rangeb` is new. `indexND` gains the same.
- ***PDL::Transform***: `use PDL::Transform` loads: coordinate transformations as objects (`t_linear`, `t_scale`, `t_offset`, `t_rot`, `t_lookup`, `t_fits`, `t_spherical`, `t_radial` and the others), their composition and inversion, `apply` and `invert`, and `map` and `unmap`, which resample an image through a transformation. *PDL::Transform::Cartography* adds the map projections (`t_mercator`,

`t_orthographic`, `t_gnomonic` and the others), `graticule` and `clean_lines`; `earth_coast` and `earth_image` need their data files from an installed *PDL*.

- ***PDL*: headers belong to the ndarray**: an ndarray's header (`sethdr`, `gethdr`, `hdr`) is released with the ndarray; it used to stay in memory, and a later ndarray could come up with a stranger's header. With `hdrcpy` set, every operation now gives its result a deep copy of the header and the flag, where only slices did. `sethdr` refuses anything but a hash reference or undef.
- **PDL: accessors take their argument as *PDL* under perl does**: `nelem`, `dims`, `ndims`, `isnull`, `get_datatype`, `nbytes`, `sever`, `badflag` and the like accept a number or an array reference as a one-off ndarray and die on the class name, where they returned 0 or nothing. After an explicit `$pdl->DESTROY` any use of `$pdl` dies with "Fatal error: ndarray address is NULL". `dimincs` of a view gives the view's increments, and `readonly` refuses a null ndarray.
- **PDL: a negative zero prints as -0 in a matrix**: a matrix whose elements go through `$PDL::doubleformat` or `$PDL::floatformat` now formats each number itself, so `-0` keeps its sign. `$VERSION` is `2.096` in *PDL::Core* and unset in *PDL::Ufunc*, *PDL::Slices* and the other generated modules, as in *PDL*; they all said `2.103`.
- **PDL: set_debugging prints the trace**: with `PDL::Core::set_debugging(1)` *PDL* prints what it does, step by step, to standard output, as PDL under perl does; it printed nothing.
- **PDL: badflag spreads as in *PDL* under perl**: setting or clearing `badflag` now reaches the ndarrays it does in PDL: the parents of a view and their flowing children, not the other views of the same parent. Clearing it on a parent used to leave a view of a view set, and setting it never reached flowing children.
- ***PDL* 2.106**: the embedded PDL is 2.106, up from 2.104. `hclip` and `lclip` work in place, *PDL::Constants* has `DEG2RAD` and `RAD2DEG`, and `avgover`, `minover`, `maxover` and the other "over" synonyms are methods only, no longer imported by `use PDL`. A condition on an ndarray of several elements dies with the full message of PDL, its `PDL::overload_bool` line included. The two functions new in 2.105, `isnan` and `casinh`, are not there yet.
- ***Peta::Shell::Session* evaluates Perl line by line**: a new built-in module that runs pieces of Perl one after another and keeps what each leaves behind: `my` variables, pragmas, the current package. It says when a text is unfinished and returns the value of the last statement. It is the evaluator of ipps, the interactive mode.

6.3.2 Command line

- **pperl -v describes the runtime in one line**: the banner now says "It features JIT, auto-parallelization and GPU acceleration."
- **–shell starts ipps, the interactive mode**: `pperl --shell`, or pperl run under the name `ipps`, gives a prompt that runs Perl, prints the value of the last statement and keeps variables, subs and pragmas from line to line. It has line editing, history, a second prompt for unfinished input and a start file, `~/.pperl/ipps/rc.pl`; without a terminal it reads lines from standard input. A bare `pperl` still reads a program from standard input as perl does. The reserved `--interactive` switch is gone.

- **ipps has commands, settings and macros**: a line that starts with a colon and a word is a command: `:help` lists them. `:vars` shows the session's variables, `:time` and `:jit` run a line and report on it, `:dump` shows a value whole, `:source` a sub's code, `:which` where a module comes from; `:install`, `:load`, `:save`, `:set`, `:history`, `:reset` and `:quit` do what their names say. In `~/.pperl/ipps/rc.pl`, `Peta::Shell->macro`, `->command` and `->set` define words that expand to Perl, commands of your own and settings.
- **ipps runs a command of your shell after !**: a line that starts with `!` and a space is handed to your shell and runs on the terminal, as in `! ls -l *.pl`. It runs under `$SHELL -c`; the new `shell` setting changes that, for example to `bash -ic` for bash's aliases. Without the space, `!` is Perl's negation as before.
- **ipps completes with Tab**: the Tab key completes the word in front of the cursor: a variable of the session, a key of one of its hashes (`$data->{list}[0]{` too), a method of an object or class, a sub, a module after `use`, a file name inside a string, a command, and a file name where none of those fits. A second Tab lists the candidates when there are several, and the prompt and the line are drawn again below the list; that holds for every program that completes through *Term::ReadLine*.
- **ipps shows a value as it is written in Perl**: a number bare, a string in quotes, `undef`, `true` and `false`, an array or hash as its structure, an object with its class in front. A string gets the quotes that need the fewest escapes: `'$5 each'`, `q{it's}`, `"a\tb"`. Large values are cut with a note of what was left out, and the layout follows the width of the terminal. The value last shown is in `$ans`, and `@ans` when there were several.
- **–cpan runs the built-in CPAN client**: `pperl --cpan` installs with the CPAN client that is part of the binary and no longer needs `cpanm` on the machine, so it works where no perl5 is installed. The options after `--cpan` are those of the `cpan` command, `-T` in place of `--notest`. A git repository, a directory or an archive file can still be named in place of a module.
- **–cpan installs only what loads after a failed XS build**: when a distribution's compile fails, `--cpan` installs its Perl files as before, but a module that loads the compiled part is now loaded once first and left out, by name, if it cannot load without it. A module that is only the front of its compiled part used to be installed and then reported as up to date on the next run. A directory named on the command line is covered too.
- **Built-in Perl modules count as installed**: `--cpan` now also treats the Perl modules the binary carries, such as `Test::More`, as installed, at their versions. It used to read a host perl's copies of them, or try to install them where there was none. Asked for a module that is built in, `--cpan` says so and builds nothing, also when CPAN has a newer release.
- **–native-manifest is gone**: `--cpan` no longer writes pperl's module inventory to disk, and the `--native-manifest=DIR` option that wrote it for other installers is removed. The installer asks `Config::lib_file` instead.
- **Modules for other operating systems are no longer built in**: `ExtUtils::MakeMaker`'s and `ExtUtils::CBuilder`'s files for VMS, Windows, OS/2, AIX, macOS and other systems, 26 in all, and the `vmsish` pragma are no longer part of the binary. The `File::Spec` modules for other systems stay.
- **A host perl's core directories are no longer searched**: `@INC` no longer lists the

core library directories of a perl5 installed on the machine, because every module in them is built into pperl and was never loaded from there. The site and vendor directories are searched as before, now under whichever perl version the host has.

- **Makefile.PL builds the Perl part only**: `ExtUtils::MakeMaker` under pperl no longer tries to compile a distribution's C or XS, so a `Makefile.PL` needs no compiler, no perl header files and no perl5 on the machine. A module that cannot load without its compiled part is left out and named, the rest of the distribution is built, and its own tests decide whether it is installed. The "different ideas about the architecture" notice is gone, and `$Config{privlibexp}` and `$Config{archlibexp}` name the built-in library.
- ***Config::lib_files* lists the built-in library**: the new function `Config::lib_files()` returns the names of all library files inside the pperl binary, in the form `require` takes (`File/Temp.pm`).
- ***Config::jit_counts* says what the JIT has compiled**: the new function `Config::jit_counts()` returns two numbers, the loops the JIT has compiled to machine code since the program started and the loops it has seen. It returns the empty list when the JIT is off.
- ***Config::lib_file* says where a library file is**: the new function `Config::lib_file("Foo/Bar.pm")` returns the path under which pperl sees that file, a built-in one first, or `undef`. The path can be opened; a built-in module without Perl source reads as its package and version. `ExtUtils::MakeMaker`, `Module::Metadata` and `CPAN` ask it, so a `Makefile.PL` no longer reports a built-in prerequisite as not found.

6.3.3 Documentation

- ***A tutorial for ipps***: a new tutorial, "ipps, the interactive mode", walks through a first session, the commands, the start file with its settings and macros, and the places where a line at the prompt behaves differently from a line in a file.
- **Changelog translations**: the changelog pages of versions 0.5.6 to 0.6.22 are translated into the 41 languages of the documentation. Entries of earlier versions name the Benchmarks Game programs and group the digits of large numbers the same way on every page, and two entries of 0.6.22 that were cut in two on the page read as one again.
- **Reference sidebar starts collapsed**: the section navigation of the Reference shows one line per namespace when you enter it, and opens only the branch of the page you are reading. It used to open every namespace at once, which made the sidebar several hundred lines long.

6.4 pperl 0.6.22

2026-10-06 to 2026-10-11, 89 commits.

6.4.1 Language and runtime

- **%Config is complete**: it has every key perl 5.44's has, 1,261 of them, where it had 198. The `d_*` and `i_*` answers come from probing the C library and include tree the way Configure does, the type sizes and formats from the build target, the tools from `PATH`, and the version, signal and extension keys from the running interpreter. `ExtUtils::MakeMaker` no longer warns about uninitialized values on every `WriteMakefile`, `DirHandle`, `Net::protoent` and `Net::servent` no longer believe their functions are missing, and `$Config{PERL_VERSION}`, `$Config{d_readdir}`, `$Config{uidsize}` and the rest exist.
- **The library's data files** are in the binary and readable: the files perl's modules open by path beside themselves, such as `ExtUtils::Manifest`'s default `MANIFEST.SKIP` and the `unicore` tables `Unicode::UCD` reads. `make distdir` no longer copies `MYMETA.*` into the distribution directory, and `Unicode::UCD`'s `charblock`, `charscript` and friends no longer depend on a perl installed on the host. `@INC` starts with `./pperl-native`, the embedded tree's own entry.
- **The daemon and duplicated output handles**: a script run through the daemon with `PPERL_DAEMON=2` that duplicates `STDOUT`, `STDERR` or `STDIN` while it compiles - every script that loads `Test::More` does - printed its output only on the first run, and the program reading that first run's output never saw it end. Each run now writes to its own output and reads its own input.
- **$0 and the environment**: assigning a name longer than the command line to `$0` no longer destroys the environment that child processes inherit. A program that renamed itself and then ran `system`, a backtick or `exec` had its children start with almost no environment (no `HOME`, no `PATH` additions), while its own `%ENV` looked fine.

6.4.2 JIT and parallelization

- **Conditional expressions** in compiled loops: a `?:` whose arm reads an element through a reference (`$c ? $x->[$i] : 0`) or from a hash (`$c ? $h{$k} : 0`) no longer makes the loop leave compiled code each time that arm is taken. An arm that is an element of a plain array already stayed compiled.
- **Recursive subs called from compiled loops**: a loop such as `for (1 .. $n) { local $_; $sum += fib($k) }` died with `panic: pp_iter, type=9` once it compiled, when the sub it calls recurses; `Benchmark`'s `timethese` and `countit` of a recursive sub died the same way. The compiled loop mistook a deeper recursion level's frame for its own.
- **A regex run once no longer gets compiled**: the regex compiler decided by the size of the string a pattern was matched against, so a pattern used a single time on a large string, as `charnames::viacode` and `vianame` do for every lookup over the one-megabyte names table, was compiled for that one match and the compilation cost several times the match. A place in the program that builds a new pattern for every lookup and runs each once now compiles none of them after the first; a loop of a thousand charnames lookups runs in a fifth of the instructions, and a pattern that is reused compiles when it did.
- **Integers past 2^53 in compiled loops**: a loop that computes an integer beyond 9007199254740992 and divides by it next, such as the long division inside `Math::BigInt` on numbers held as floats

(`Math::BigFloat->new(2)->bpow(240)->broot(9)` with `Math::BigInt::FastCalc`), could get one quotient digit wrong the first time the loop ran on float-typed values after integer-typed ones; later runs were right. The compiled loop had taken a rounded copy of the exact integer perl kept. It now leaves compiled code for that pass, as it does whenever a value cannot be held exactly.

- **`$_` in a nested statement-modifier loop**: in a compiled loop that contains `EXPR for LO .. HI`, where EXPR calls a function and reads `$_` in its arguments, the first pass of every inner loop saw `$_` undefined: a "Use of uninitialized value `$_`" warning, and 0 where the value was wanted, so a range not starting at 0 gave wrong results.

6.4.3 Modules

- **Native modules in the release build**: the build's profile-guided optimisation now runs *POSIX*, Socket, *Errno*, Config, *Time::Piece*, *Sys::Syslog*, *IPC::SysV*, the DBM modules, *Unicode::Normalize*, *Unicode::UCD*, charnames, the Symbol and EBCDIC encodings, the `:encoding`, `:via` and `:mmap` layers, *File::Map*, *Hash::Util*, *Clone*, *Memoize*, *Math::BigInt::FastCalc*, B, *Opcode*, Safe, *PadWalker*, *Devel::Peek*, mro, attributes, source filters, *EV*, Event, Coro's loop bridges, *Peta::Dict*, *Peta::PCRE2*, *Peta::XS*, *Peta::FFI* and *Peta::LMDB* before it decides what is hot. None of these had been run by the build before, so the release binary carried them compiled for size rather than for speed.

- ***Errno* has every name the host defines**: `%!`, the constants and `@Errno::EXPORT_OK` now come from the errno.h of the machine pperl was built on and the headers it includes, read at build time the way perl's own Errno.pm is generated. `EFTYPE`, `EFSBADCRC` and `EFSCORRUPTED`, which current kernel headers define and the previous fixed table lacked, exist; on this host the list is identical to perl 5.44's, all 137 names.

- **The `-C` switch**: `pperl -CSAD script` and the other forms of perl's Unicode switch work, alone and at the end of a cluster such as `-wCSD`; they were refused with "Unknown option: -C", and only `PERL_UNICODE` in the environment had the effect.

- ***Compress::Raw::Bzip2*** exists, with `Compress::Raw::Bunzip2` and the `BZ_*` constants, over the system's libbz2. With it `IO::Compress::Bzip2`, `IO::Uncompress::Bunzip2`, `IO::Uncompress::AnyUncompress` and `Archive::Tar` read and write bzip2 data, `.tbz` archives among it, and `streamzip` makes bzip2 zip entries; they died with "Can't locate Compress/Raw/Bzip2.pm".

- ***Peta::ReadLine*** is the name of the readline module; `Term::ReadLine::Gnu` is an alias of it, so `use Term::ReadLine` still picks it up, a program that tests `$term->ReadLine eq "Term::ReadLine::Gnu"` (CPAN.pm, the debugger) still passes, and `Term::ReadLine::Gnu::XS::*` calls still work. `Peta::ReadLine->new` reports `Peta::ReadLine`, version 0.2.0; the alias reports the name it was asked by.

- ***Peta::ReadLine*'s function table and keymaps**: `get_all_function_names` lists the bindable commands (it returned one name of garbage), `get_function_name` finds a function's name, and `$attribs->{executing_keymap}` and `$attribs->{binding_keymap}` are Keymap objects that `get_keymap_name` and the

binding methods accept. `import Term::ReadLine::Gnu qw(:keymap_type)` and the other export tags work under the alias as they do under `Peta::ReadLine`.

- ***Peta::ReadLine*'s ignore_some_completions_function**: a completion with `$attribs->{ignore_some_completions_function}` set no longer dies with "panic: attempt to copy freed scalar" in the statement that called `complete_internal` or `call_function`.
- ***Peta::ReadLine* has its own line editor**: no readline, history or curses library is loaded any more, so line editing works where none is installed and in the static build. It behaves as GNU Readline 8.3 does - commands, key bindings, the init file, history, completion, signals, and the same screen after every key. In a UTF-8 locale the commands move, delete, change case and search by characters, not bytes. It identifies itself: `$attribs->{library_version}` is `8.3 (Peta::ReadLine 0.2.0)` and `$attribs->{gnu_readline_p}` is 0.
- ***Imager*** follows Imager 1.037. `getsamples` checks its arguments itself and says in `errstr` why a call failed (a row outside the image, a channel the image does not have); a failed call returns the empty list, or undef in scalar context. An image too large to allocate fails `Imager->new` with a message instead of ending the program, and `set_out_of_memory_handler` exists.
- ***Imager* drawing and images**: `box`, `line` (plain and anti-aliased) and `arc` draw; `getscanline`, `setscanline`, the palette rows and `setcolors` work; images convert between 8-bit, 16-bit and double samples and can be compared. `open_log`, `log` and `close_log` write Imager's log, and colours have `red`, `green`, `blue`, `alpha` and HSV conversion. Three results changed to what Imager gives: a 16-bit image read as 8-bit colours was one too bright for some values, `i_img_info`'s fourth value is the channel mask, and the colour count of a direct image is undef.
- ***Imager* I/O objects**: `Imager::IO` exists: `new_buffer`, `new_fd`, `new_fh`, `new_cb` and `new_bufchain` (and the `io_new_*` functions) make an I/O object over a string, a file descriptor, a file handle, Perl callbacks or a file in memory, with `read`, `write`, `seek`, `getc`, `gets`, `peekn`, `close` and the rest of the class. `io_slurp` of an object that is not a memory file dies with a message; under Imager itself it ends the program silently.
- ***Imager* PNM files**: Imager reads and writes PNM files (PBM, PGM and PPM, ASCII and binary, 8 and 16 bit), `read_multi` included, so `Imager->new(file => 'x.ppm')` and `$img->write(type => 'pnm')` work. `set_file_limits`, `get_file_limits` and `check_file_limits` bound what is read, and the format of a file is detected from its first bytes, with `add_file_magic` for signatures of your own.
- ***Imager* raw files**: `read` and `write` with `type => 'raw'` work: rows of bare 8-bit samples, interleaved by sample or by line, with `raw_datachannels` and `raw_storechannels` choosing which channels of the file are kept.
- ***Imager* BMP files**: Windows BMP files are read at 1, 4, 8, 16, 24 and 32 bits a pixel, run length encoded and with bit-field masks included, and written uncompressed. A grey image with an alpha channel is written with its own grey; under Imager itself each pixel takes its neighbour's value as alpha and half the row is garbage.
- ***Imager* TGA files**: Targa files are read and written: colour-mapped, direct colour and grey, run length encoded or not, at 8, 15, 16, 24 and 32 bits a pixel, with the

`compress`, `wierdpack` and `idstring` options. `%Imager::formats` now lists what Imager lists without any library: raw, pnm, bmp, tga and ifs.

- ***Imager*** **masked images**: `$img->masked` gives a view of part of an image, optionally written through a mask, that can be drawn into, copied and written to a file. `setsamples` is available, with its floating point and 16-bit forms, and so are `getsamples` with `type => 'float'` and `type => '16bit'`. A floating point sample outside 0 to 1 now wraps when stored in an 8- or 16-bit image, as in Imager, and `getmask` on a fresh image is -1.
- ***Imager*** **fills**: `box` and `arc` take a `fill`, and `Imager::Fill` makes solid, hatched, image and opacity fills with the twelve combine modes (`normal`, `multiply`, `add`, `darken`, hue and the rest).
- ***Imager*** **paste and crop**: `$img->paste` copies a rectangle of another image in, converting between sample depths and channel counts, and `$img->crop` cuts one out.
- ***Imager*** **circles and flood fill**: `circle` draws filled anti-aliased circles and, with `filled => 0`, outlines, plain or anti-aliased; `arc` with `filled => 0` draws the outline of an arc. `flood_fill` fills the region around a point with a colour or a fill, optionally up to a `border` colour, and `polybezier` draws a Bezier curve.
- ***Imager*** **polygons**: `polygon` and `polypolygon` fill anti-aliased polygons with a colour or a fill, by the even-odd or the non-zero winding rule, and `arc` with `aa => 1` draws an anti-aliased pie slice.
- ***Imager*** **rubthrough and compose**: `$img->rubthrough` lays another image over the image by its alpha channel, and `$img->compose` does so with a combine mode, an `opacity` and optionally a `mask` image.
- ***Imager*** **fountain fills**: `Imager::Fill->new(fountain => ...)` makes a gradient fill (linear, bilinear, radial, square, revolution or conical, with repeats, segments and super-sampling), and `$img->filter(type => 'fountain')` draws one over a whole image.
- ***Imager*** **channel operations**: `$img->convert` makes a new image by a matrix or a preset (`grey`, `noalpha`, `rgb`, `addalpha` and the single channels), `Imager->combine` makes one from channels of several images, `$img->map` runs channels through lookup tables, and `$img->flip` mirrors the image.
- ***Imager*** **quantisation**: `$img->to_paletted` turns a direct colour image into a paletted one, with the palette from median cut, the `addi` method, a fixed map or your own colours, and the pixels matched by closest colour, perturbation or error diffusion. `Imager->make_palette`, `getcolorcount`, `getcolorusage` and `getcolorusagehash` work.
- ***Imager*** **scaling**: `$img->scale`, `scaleX` and `scaleY` return a scaled copy, with the `normal`, `preview` and `mixing` methods.
- ***Imager*** **rotation**: `$img->rotate` turns an image by quarter turns (`right`) or by any angle (`degrees`, `radians`, with `back`), and `$img->matrix_transform` maps it through a 3 x 3 matrix.
- ***Imager*** **trim**: `$img->trim` and `trim_rect` find and remove the border of an image, by colour ranges (`colors`, `auto`) or by transparency (`alpha`).

- ***Imager*** **transform and transform2**: `Imager::transform2` computes an image from an expression per pixel (`rpnexpr`, or `expr` with Parse::RecDescent), reading up to three input images, and `$img->transform` moves pixels by two stack programs.
- ***Imager*** **filters**: `$img->filter` has every filter Imager has built in: `autolevels`, `autolevels_skew`, `bumpmap`, `bumpmap_complex`, `contrast`, `gaussian`, `gaussian2`, `gradgen`, `hardinvert`, `hardinvertall`, `mosaic`, `nearest_color`, `noise`, `postlevels`, `radnoise`, `turbnoise`, `unsharpmask` and `watermark`, beside `conv` and `fountain`. `$img->difference` and `rgb_difference` show where two images differ.
- ***Imager*** **object destructors**: calling the `DESTROY` of an Imager class on something that is not a reference dies with Imager's message, "is not a reference"; it named an expected type instead. `$Imager::_context` exists.
- ***Imager*** **plugin loader**: `Imager->load_plugin`, which loads filter plugins compiled from C, returns undef and sets `$Imager::ERRSTR` instead of dying with "Undefined subroutine". Such plugins cannot be loaded; Imager has documented that interface as deprecated since 2006.
- ***Peta::WebGPU*** **and fork**: a process that has opened a device can fork. The child used to hang for ever when it exited; now it ends normally. Using an inherited device, buffer or shader in the child dies with a message that says to open a device there, and the child can do so and compute on it. Where the parent has the `gl` backend open, the child's device comes from the other backends.
- ***Peta::WebGPU*** **batches**: `$gpu->batch(sub { ... })`, or `$gpu->begin` and `$gpu->commit`, send any number of dispatches to the card in one go, and `$pipeline->bind(\@buffers)` gives an object `run` takes in place of the pipeline and its buffer list, so a recurring dispatch no longer binds its buffers each time. A `write` inside a batch takes effect in order with the dispatches around it, and `$gpu->timed(sub { ... })` returns the nanoseconds the card spent on a batch. `$gpu->features` and `$gpu->has_feature` name the optional features of the device, which now has half-precision floats in shaders (`enable f16;`) and timestamp queries wherever the adapter does.
- ***Peta::WebGPU*** **buffers**: `$buffer->read($offset, $length)` reads a part of a buffer, so fetching one number no longer copies the whole buffer back from the card. `$buffer->write` takes a *PDL* ndarray of type `float`, `long` or `ulong` as well as a byte string.
- ***Peta::WebGPU*** **memory**: a buffer made with contents no longer takes twice its size on the card until the next dispatch; large contents go up in pieces. `$buffer->destroy` frees a buffer's memory at once, `$gpu->memory` reports what the device has allocated, and `Device->new(memory_limit => 90)` turns running out of card memory into an `Out of Memory` exception at 90 percent of the driver's budget. A program that frees buffers and at once makes new ones is no longer told the card is full while the freed memory is still on its way back, and an error message names the first failure, not one that followed from it.
- ***Peta::WebGPU*** **on OpenGL**: `Device->new(backend => 'gl')` opens on the NVIDIA driver, where it failed with "Parent device is lost". Drawing works there; compute needs OpenGL 4.3 or ES 3.1, and on an older context dies with a message that names the `vulkan` backend.

- ***Net::SSLeay*** **version constants**: `OPENSSL_VERSION_MAJOR`, `OPENSSL_VERSION_MINOR`, `OPENSSL_VERSION_PATCH` and `OPENSSL_VERSION_NUMBER` name the OpenSSL that is loaded. They kept the values of the release pperl was built beside, so after a system update to OpenSSL 3.6.5 `OPENSSL_VERSION_PATCH` still said 4.
- ***GD*** has the rest of GD 2.91: the affine matrices and transforms, `colorReplace` and its variants, `convolution`, `resolution`, the transposed, rotated, scaled and blurred copies, animated GIF, HEIF and AVIF, 48 functions in all. `GD::VERSION_STRING` is now libgd's version string and `GD::LIBGD_VERSION` its number (2.0303 for libgd 2.3.3), as in GD itself. `GD::Image->supported` and the `newFrom<Format>` methods follow what the installed libgd can read.
- ***GD*** **colorsTotal**: `$image->colorsTotal` on a truecolor image returns `undef`, as in GD itself. It returned 0.
- ***Config::config_vars*** and `-V` print the way Perl's `print` does: to the handle `select` chose, through a tied `STDOUT`, and in order with the program's other output. They wrote to the process's standard output directly, so a capture saw nothing and, on a pipe, their lines could come out ahead of text printed earlier. `Config::compile_date` and the matching line of `-V` read `Compiled at Oct 8 2026 20:42:44`, the layout `Config::Perl::V` parses; they named the compiler instead.
- ***Cairo***, ***Cwd***, ***Term::ReadLine::Gnu***: `Cairo::HAS_PS_SURFACE`, `HAS_PDF_SURFACE`, `HAS_SVG_SURFACE`, `HAS_RECORDING_SURFACE` and `HAS_FT_FONT` exist. `Cwd::CLONE` exists. `rl_save_state` returns a `readline_state_tPtr` object, which is freed with its last reference and is the only thing `rl_restore_state` accepts.

6.5 pperl 0.6.21

2026-09-28 to 2026-10-06, 271 commits.

6.5.1 Language and runtime

- **Unicode 17** reaches `\N{...}`, `charnames::viacode` and *Unicode::UCD*: they are perl 5.44's own, built into pperl, where pperl had read them from the host perl (5.42, Unicode 16) and did not know Unicode 17's characters. A program using one `\N{NAME}` no longer spends most of its run compiling the 1.2 MB character-name table (34 ms against a 4 ms baseline); the table is stored in the binary.
- **glob** and `<*>` match files with perl5's own matcher instead of the C library's `glob`: brace alternatives come back in the order written, `GLOB_NOCASE` no longer sorts by case, `^` inside `[...]`, `\*` and an empty `{}` mean what they mean in perl5. A directory error stops a glob as in perl5, and `File::Glob::GLOB_ERROR` reports the last glob's error code instead of always 0.

6.5.2 JIT and parallelization

- **Parallel loops**: compiled loops run on all cores. A counted `foreach`, `while` or C-style `for` loop whose body only computes - integer sums, bitwise reductions, sums of floats whose terms are whole numbers, reads of a hash with integer keys - is split into chunks whose result is bit for bit the sequential one, and a chunk that

reaches a call or an exit resumes sequentially at its first iteration, so perl dies or warns where it would. On 4 physical cores a 400-million-trip addition loop takes 0.13 s instead of 0.58 s, and the Benchmarks Game's mandelbrot written as a loop takes 4.13-4.26 s at N=16,000 instead of 10.95 s. Loops too short to repay the split stay sequential, and `--no-parallel` turns splitting off.

- **Floating-point loops** over arrays, also arrays reached through references, no longer leave compiled code at every float element: a matrix-vector product takes 9.5 ns per multiply-add instead of 248 (79 ns without the JIT), a 4096-element dot product 9.4 ns instead of 70. A `return \@y` elsewhere in the sub, or a `?:` that picks an array element (`$x[$i] > 0 ? $x[$i] : 0.0`), no longer keeps a loop interpreted; a small neural network trains at 16,000 samples per second with the JIT against 2,500 without, where it had run slower with the JIT (1,800 against 4,000). A float loop such as `$sum += ($x > $y)` no longer moves its sum through memory on every trip: 5.2 cycles per trip instead of 10.1.
- **Regular expressions** compiled by ReJIT: a successful match of a pattern without capture groups records its result itself instead of running the interpreter's matcher over the match again, and the Benchmarks Game's regex-redux (single-threaded, 5M) runs 32% fewer instructions and cycles; each compiled pattern reaching its state through its own engine saved a further 3% there. Scans for a literal probe 64 bytes per step and hand long stretches without a candidate to the C library's `memchr`: `/aab./` over 64K of `a` went from 0.14 to 1.02 times perl5's speed, and the other literal and character-class scans measured run 1.7 to 1.9 times as fast.
- **String operations** in compiled loops: 4-argument and lvalue `substr` replacement, `vec` read and assigned, the `x` repetition of a string and `join` over scalars compile, and so do string results stored into package variables and into array or hash elements, and truth tests of strings. A `substr` loop over a 4 MB string runs in 1.49 s instead of 3.65 s (perl5 4.59 s), a `vec` bit sieve in 0.56 s instead of 3.75 s (perl5 4.41 s). Loops that rebuild strings and then test or store them, such as `$s += length $g` over a package variable or `$w = $i & 1 ? "a" : ""`, no longer fall back to the interpreter on every trip.
- **Array references** in compiled loops: element reads, stores, `++` and `+=` through `$r->[...]`, `scalar(@$r)`, `$#$r`, and `push`, `unshift`, `pop` and `shift` through a reference compile, as do `shift @a` and a sub's implicit `shift` of `@_`. Compiled code now honours a tied scalar holding the reference and a class that overloads `@{}`, which it had bypassed, and a loop that reads an array through a reference while updating it by name sees its own updates. An object removed by pop in a compiled loop is destroyed at the end of its statement, as in perl, not after the next one.
- **Hashes, pushes and new arrays** in compiled loops: a plain hash element is looked up inline, an integer key without first being turned into a string, and the Benchmarks Game's k-nucleotide at 25M runs 19.2% fewer instructions. A short-lived array built in a loop (`my $v = [$i, $i + 1]`) compiles, and one that never leaves the loop is kept in registers: such a loop runs 20.3 times as fast. A loop that pushes computed values onto an array, such as a sieve's prime extraction, runs 2.93 times as fast.
- **List foreach loops** such as `for my $v (@a, @b)`, `for ($x, $y)` and `for (1, 5, 9)` compile, also nested in another compiled loop. A benchmark of such loops over two arrays runs in 0.276 s instead of 0.715 s (perl5 0.711 s).

- **Loop conditions** that combine comparisons with &&, || or ?:, and `while ($x)` or `until ($done)` on a plain value, compile; such conditions had kept 133 loops in 14 of 21 real programs interpreted. A sub whose last statement is a compiled `while` loop no longer returns a wrong value in list context.
- **More loops compile.** `defined $v` is tested when the loop runs instead of assumed when it compiles, and `ref $x` as a condition, `<=>` under `use integer`, a C-style `for` that stores into array elements (`for (my $k = 0; $k < $n; $k += $q) { $a[$k] = 1 }`) and a call to a sub defined further down the file no longer keep a loop interpreted.
- **Fewer compiled loops give up.** Loops that compiled but left compiled code at every entry until pperl stopped trying now stay compiled: a `foreach` over constants, a `foreach my $q (@q)`, whose loop variable is a value the array holds as well, a copy of a string or of undef into a `my`, `$b[$i] += 1` over whole-number floats from a division, and a `my $last = ''` written on a rare branch. A modifier `if` as a loop's last statement no longer grows perl's stack: a 10-million-trip loop that took 27.1 MB against the interpreter's 10 MB takes 14.6 MB and 227.8M instructions instead of 507.8M. A loop that keeps leaving compiled code for any other reason is given up after about 4,096 entries instead of retried forever.
- **JIT overhead**: a program that compiles nothing pays less for the JIT; on a module-loading test file the cost over `--no-jit` fell from 32.5M to 14.5M instructions. An `s///` with an expression replacement is no longer taken for a loop to compile, which had made one module's test file 61% slower with the JIT (now 4.3%), and short `foreach` loops over lists and arrays are no longer compiled when they cannot repay the compile.
- **Integers stay integers.** Values written by compiled loops keep the integer or floating-point form perl gives them: an integral result stored into an array element, by +=, or computed through `int()` stays an integer where it had become a float, as *Devel::Peek* and B showed, and later integer reads of it stay compiled. A loop that pperl tried and failed to compile no longer turns an integer variable into a float, which had left 14 string-function benchmarks about 10% slower with the JIT than without.
- **Compiled stores** no longer run a tied variable's `STORE` or an object's `DESTROY` in the middle of compiled code: a store to a tied destination could leave a loop reading a reallocated array or crash it, and an object whose last reference was overwritten was destroyed when the loop ended instead of at the next statement. A lexical written both by a loop and by subs it calls, such as a variable a closure captures, keeps every write.
- ***Memoize***: the cache compiled code uses for a memoized function of numbers no longer makes small integer arguments evict each other. A memoized `fib` over the arguments 0 to 31 made 1.25M avoidable calls in 20M; its loop now also runs in parallel.

6.5.3 Modules

- **The perl 5.44 core library** ships inside pperl: every `.pm` and `.pl` file perl 5.44 installs is embedded, 733 more than before, and pperl no longer takes core modules from the host perl. On a host without perl the previous binary could not load *Encode*, *File::Temp*, *Test::More*, *Math::BigInt*, *JSON::PP*, *Unicode::UCD* or autodie, and

on a host with perl 5.42 it loaded 5.42's versions. `threads` and `threads::shared` ship their 5.44 modules too.

- ***Net::SSLeay*** **and IO::Socket::SSL**: Net::SSLeay 1.96 is native in the full build, over OpenSSL 3 loaded at run time, with the 709 functions its distribution provides for that OpenSSL; a function the loaded library lacks stays undefined, as in a perl5 build against that library, so feature tests such as `defined &Net::SSLeay::X` answer correctly. IO::Socket::SSL 2.099 ships beside it. Both distributions' test suites, and AnyEvent's including its TLS test, match perl 5.44 running the original modules.
- ***DB_File***, ***GDBM_File***, ***NDBM_File*** **and** ***ODBM_File*** are native and keep their databases in LMDB files, through liblmdb loaded at run time, instead of Berkeley DB or gdbm; DB_File supports Hash, Btree and Recno, and a gdbm or Berkeley DB file is recognised and refused, never written. GDBM_File writes and reads gdbm's ASCII and binary dump formats, so gdbm 1.26 loads pperl's dumps and pperl loads gdbm's, which is how existing gdbm files move over. By default writes are held in one transaction until `sync`, `untie` or `close`; `$Peta::LMDB::DBM_WRITE = 'through'`, or GDBM_File's sync mode, commits each write, and a crash loses writes but never the file.
- ***Peta::LMDB*** is new: LMDB's environments, transactions, databases and cursors for Perl, with key comparators written in Perl and errors raised as `Peta::LMDB::Error` objects. The DBM modules above are built on it.
- ***LMDB_File*** 0.14 is native over the same library, and its distribution's tests match perl 5.44 with the original module. Closing an environment explicitly, closing it twice, or using a transaction of a closed environment no longer crashes; perl5 with the original module aborts or segfaults there.
- ***Class::XSAccessor*** 1.19 is native, including the faster call path it gives its accessors. Moo uses it for attribute accessors; before, Moo found the host perl's build, could not load it, and warned "Class::XSAccessor exists but failed to load" before falling back to pure Perl.
- ***XML::Parser*** 2.59 is native in the full build, over libexpat 2.6.0 or newer loaded at run time, so XML::Dumper and the other modules built on XML::Parser work. Its distribution's tests match perl 5.44 with the original module built against the same libexpat.
- ***Unicode::LineBreak*** **and** ***HTML::Parser*** are native. Unicode::LineBreak 2019.001, with *Unicode::GCString* and *Text::LineFold*, breaks lines and grapheme clusters by the Unicode 17 rules and passes Unicode 17's LineBreakTest and GraphemeBreakTest; Thai, Lao, Khmer and Myanmar text is segmented through the system's ICU. HTML::Parser 3.86 matches perl5 with the original module on its whole test suite.
- ***Unicode::Normalize*** 1.32 is native and passes its distribution's tests and Unicode 17's NormalizationTest, 400,680 cases.
- ***Devel::Size*** 0.87 is native, and its distribution's tests match perl 5.44 with the original module.
- ***File::DosGlob***, ***Sys::Syslog*** **and** ***Math::BigInt::FastCalc*** are native: File::DosGlob 1.12, Sys::Syslog 0.36 and Math::BigInt::FastCalc 0.5020.

- ***PDL*** loads as on perl5: `PDL.pm`, *PDL::Lite* and *PDL::LiteF* are upstream's, so `require PDL` loads nothing until `import`, and `use PDL` gives upstream's exports, now including `rcols`, `wcols`, `rfits`, `wfits`, `rpic`, `wpic`, `rim` and `wim`, less 22 functions pperl does not have yet. *PDL::IO::Misc*, *PDL::IO::Pnm*, *PDL::IO::FITS*, *PDL::IO::Pic*, *PDL::ImageRGB*, *PDL::Image2D* and *PDL::Compression* load, *PDL::Types* and *PDL::Constants* are upstream's, and `use PDL::LiteF` no longer leaves every operation dying with "PDL core struct is NULL". A pperl built with PDL loads libquadmath when *PDL::Core* loads, no longer at startup, where it slowed every number-to-string conversion of every program.
- ***PDL*** **computations** that gave wrong answers now match PDL 2.104: `median` and `oddmedian` of a multi-dimensional ndarray, `avg`, sums of byte and 64-bit ndarrays, `stats`, `dim(-1)`, `nbad` and `approx`; seeded `random` numbers equal perl5's bit for bit. In-place operations on a converted type (`$d &= 0; $d |= 1` on doubles) no longer die and leak, `cat` of ndarrays of different types no longer writes garbage, and `clip`, `hclip` and `lclip` take `->inplace`, an output and undef bounds as upstream does (`lclip` died).
- ***PDL*** **errors and operators** follow upstream: errors go through `PDL::barf` with its backtrace, operations upstream does not run in place refuse `->inplace` (a reduction had written its result over its input), and `ne`, `lt`, `gt`, `le`, `ge` and `cmp` on ndarrays die as on perl5 instead of comparing element by element. `matmult` and `x` report upstream's dimension-mismatch message and promote vectors as upstream does, `-$x` keeps an unsigned type, and `abs` of a complex value is its modulus. Loading PDL under `-w` no longer warns about redefined subroutines, and the binary operators read their swap flag as an integer, running get-magic. An out-of-range dimension given to `broadcast` croaks at the first computation, where perl5 reads memory it does not own.
- ***PDL*** **multithreading**: as in PDL 2.104, an operation on an ndarray of at least 1 Mi elements is split across threads by default, with upstream's controls (`set_autopthread_targ`, `set_autopthread_size`, `online_cpus` and the rest); the thread count follows `--threads`, and `--no-parallel` turns the splitting off. Warnings and errors raised in the threads arrive after the join through `PDL::cluck` and `PDL::barf`, as upstream's do. Reading ASCII and PNM data is not split, where upstream's threaded readers crash.
- ***PDL*** **subclasses and redefinitions** are honoured by the functions pperl runs natively for speed - `new`, `null`, the type constructors, `flat`, whole-array reductions such as `sum` and `max`, `at`, `set` and string conversion: each takes its shortcut only for a plain PDL while the subs upstream's Perl would call are unchanged, and runs upstream's Perl otherwise. A list assignment to `flat`'s result no longer clobbers the caller's variable, a format argument to string conversion is no longer ignored, and string conversion clears `$@` as upstream's does.
- ***Imager*** follows Imager 1.036, including its paletted images: every palette has 256 entries, and an index past the colours in use is rejected or read as black. *Imager::Font::FT2* is native, over FreeType loaded at run time, and the Imager and FT2 tests match perl5 with the original Imager 1.036.
- ***Exporter***, ***Carp*** **and** ***File::Temp*** are perl 5.44's own modules (Exporter 5.79, Carp 1.54) instead of pperl's replacements, and the modules built into pperl export through the real Exporter as their originals do: `Errno->can('import')` is `\&Exporter::import`, and Config no longer inherits from Exporter.

- ***List::Util***: `reduce` with a block that branches, such as `reduce { $a > $b ? $a : $b } 1 .. 500`, no longer hangs with the JIT on, and it compiles: 2000 reductions over 2000 elements take 314M cycles against 363M on perl5. `min` and `max` over more than 65,536 numbers return the element itself, as perl5 does, not a new number: `max` of numeric strings returns `"0300000"`, not `300000`.
- ***EV* and Event**: an EV periodic watcher's callback no longer leaks two values per call, which had kept the watcher from ever being freed, and setting its reschedule callback takes effect, where EV 4.37 never calls it or fails every fire. EV's signal names IOT, CLD and POLL mean their signals instead of out-of-range numbers. `Event::queue` with a wrong number of arguments dies with a usage message instead of crashing, and destroying an `Event::generic::Source` that still has watchers no longer frees it twice.
- ***Cairo*** checks its methods' arguments before reading them and reports a wrong call as the Cairo module does (`Usage: Cairo::Matrix::translate(matrix, tx, ty)`), and the `MIME_TYPE_*` and `OUTLINE_ROOT` constants are constants folded at compile time.
- **Native modules in release builds** are part of the profile-guided optimisation training: *List::Util*, *Scalar::Util*, *Encode*, *MIME::Base64*, *Digest::MD5*, *Digest::SHA*, *Storable*, *Data::Dumper*, *Coro* and others had been built as code the training never ran. Digest::SHA's final padding, Encode's single-byte encoder and Cwd's `abs_path` also no longer do work their C originals skip.
- **%Config** describes what pperl provides: `nonxs_ext`, `extensions` and `known_extensions` name its pure-Perl and native extensions as Configure would, and `bin`, `scriptdir`, `sitebin`, `sitescript`, `vendorbin`, `vendorscript` and their `*exp` forms are defined, as in perl 5.44. Test suites that check `$Config{extensions}` no longer skip, and Pod::Find no longer warns about an uninitialised `scriptdir`.
- **Peta modules** are versioned 0.2.0, below 1.0.0 until their interfaces are final; *Peta::XS*, which was 1.0, moves down with them. `Peta::FFI::UUID`'s `uuid_unparse` and `uuid_is_null` die on anything but a 16-byte UUID instead of padding or cutting it.
- **Smaller module changes.** *Sub::Exporter::Progressive* 0.001013 ships, for *File::Map*. *POSIX* functions that return a system call's status give a modifiable undef on failure, `Config::config_re` leaves a bad pattern's error in `$@`, *Digest::SHA* leaves its argument downgraded as perl5 does, `Coro::State::nice` returns the previous priority, `Math::GMP::_gmp_build_version` exists, and *Errno* reports its version as 1.39. *GD* and *Term::ReadLine::Gnu* without their C library fail at `require`, and GD's writers croak on failure, `interlaced(0)` turns interlacing on and `alphaBlending` and `saveAlpha` return nothing, all as the GD module does.

6.5.4 Command line

- **CPU features**: `pperl -V` lists the CPU extensions the binary was compiled for and those the JIT uses on the machine it runs on; `--jit-stats` prints them as a `cpu=` line.

6.5.5 Distribution

- **Minimal builds**: a pperl built without the standard native modules (`native-min`) answers `require Imager`, `GD::Simple`, `Memoize::Expire` or `SDL2::Image` with "Can't locate ..." naming the missing capability, instead of loading the Perl half and failing later.

6.5.6 Documentation

- **Changelog**: this changelog, with one page per version from 0.5.6 on. Every change of a version is accounted for, either in an entry or as internal work.
- **Index**: one page per initial letter, plus one for symbols, under a letter overview. Every entry is the name itself and links to its page; a function also appears under its own name with its module in parentheses, as in `access (POSIX)`, and a pragma as `open (pragma)` beside the built-in `open`. Besides modules and functions it now lists perl's special variables and operators under perl's own index terms, the command-line options and environment variables, the topics of every guide and tutorial, and every version of this changelog, with each module under the versions that changed it.
- **^^ and ^^=**: the operator reference documents perl's high-precedence logical exclusive or; it said there was no ^^.
- **Errata**: the chapter on errors in perl's own documentation is withdrawn. A page removed from the documentation is now also removed from the published site, where it had stayed reachable by its address.
- **Module reference**: the native modules' pages document what a user calls, with synopses whose output is the same on perl 5.44 and pperl, the upstream version implemented, and a section on differences from upstream citing the tests that pin them. Every function is published under its Perl name (`B::CV::GV`, not `cv_gv`); *PDL*, *Net::SSLeay*, *EV*, Event, *Storable*, *POSIX*, *Coro*, *Compress::Raw::Zlib*, *Unicode::Normalize*, *Peta::LMDB* and the DBM modules gained full references, and Cairo's pages moved from `Peta::FFI::Cairo` to `Cairo`.
- **Bundled modules**: the 755 pure-Perl modules embedded in pperl have reference pages made from their own POD, with highlighted code and corrections where the original text does not apply to pperl; text about building XS modules is left out. XSLoader's page describes how it starts native modules under pperl.
- **Graphics guides**: the 2D and 3D graphics guides are translated into all 41 languages of the documentation, and factual errors in their English were corrected.
- **Translations**: the German translation covers the native module reference pages, *SDL2*, *Peta::WebGPU* and the debugging guide, and every language's catalogues follow the reference pages to their new places.

6.6 pperl 0.6.20

2026-09-27, 21 commits.

- **Static binaries** start again. Every statically linked (musl) pperl stopped with a segmentation fault before running a line: the code that records the program's arguments, which an assignment to `$0` rewrites, received none from musl and read

garbage. They also build again: POSIX's six locale categories that only glibc defines (`LC_ADDRESS`, `LC_IDENTIFICATION`, `LC_MEASUREMENT`, `LC_NAME`, `LC_PAPER`, `LC_TELEPHONE`) are reported on musl as perl5 built on musl reports them, "Your vendor has not defined *POSIX* macro".

- ***Peta::Dict*** no longer crashes in a compiled loop that counts numeric-string keys, such as `for my $k (@keys) { $h{$k}++ }`. A key that grew the table during the loop left the compiled code reading the table's freed memory.
- **Regular expressions** compiled by ReJIT search each string once instead of twice. Regex-redux runs 9% fewer instructions; scans for a long literal and for a pattern that starts with `.` take about half and under a third of the cycles they took. No measured pattern got slower.
- **Release archives**: every binary of a release is built with profile-guided optimisation, and a release is one archive holding the binaries, their SHA-256 sums and a build record (commit, target, features). Before it is packed, each binary must print the locale and format numbers under four locale settings exactly as perl5 built against the same C library does.

6.7 pperl 0.6.19

2026-09-17 to 2026-09-27, 526 commits.

6.7.1 Language and runtime

- **Modules compiled into pperl** are loaded the way perl5 loads a file it finds on `@INC`. Such a module no longer sees the lexical variables and pragmas of the code that requires it: `my $VERSION = "mine"; require Guard` overwrote the caller's `$VERSION`. `%INC` and `__FILE__` name the module `./pperl-native/<name>` instead of `native`.

6.7.2 JIT and parallelization

- **mandelbrot**: the Benchmarks Game's mandelbrot compiles its per-pixel loop as one piece. A labelled `next` or `last` out of an inner loop is a jump inside compiled code, an append to a fresh `my $line` no longer leaves it, and the loop's floating-point values stay in registers instead of being reloaded from the stack; at size 16,000 the program runs 362.6G instructions, from 1,160.8G. The range checks on floating-point results are combined into one test per statement, which takes 28.9% of the instructions off mandelbrot at size 2,000.
- **Independent loop iterations**: a counted `foreach` whose iterations do not depend on each other runs four of them side by side in SIMD vectors, and overflow checks that the loop's own tests make unnecessary are proved away and dropped. mandelbrot at size 16,000 takes 19.36 s of CPU time with the vectors, from 31.08 s, and runs 164.3G instructions with the checks dropped, from 217.4G. In the release build it takes 12.13 s.
- **binary-trees**: a recursive subroutine that walks a tree through array references, or builds and returns new arrays, compiles as a function. When one such call builds a tree that the next call in the same statement only walks, the tree is built in a private area that is freed in one step; any failure throws the area away and runs

the statement again interpreted. With the private area, the Benchmarks Game's binary-trees at depth 21 takes 4.29 s of CPU time instead of 26.1 s.

- **fannkuch-redux**: a block inside a compiled loop no longer ends it, so the Benchmarks Game's fannkuch-redux keeps its whole permutation step in compiled code; at n=9 it runs 1,301M instructions, from 2,824M. Statements of its flip loop no longer write perl's stack pointer, `@a = @b` between two integer arrays copies the values directly, a two-index `while ($lo < $hi)` loop checks its bounds once, and loop tests are placed next to their bodies. At n=12 the program takes 53.53 s of CPU time, where Node.js takes 57.42 s.
- **fasta**: a constant that an earlier division left carrying a floating-point value no longer forces its expression into doubles, so the Benchmarks Game's fasta computes `($seed * IA + IC) % IM` in integers, as perl5 does: 7.52 s of CPU time at 25,000,000 becomes 3.64 s. A modulo by a literal compiles to a multiplication (18.09 s to 14.83 s), and short appends and `substr($s, $i, 1)` copy their bytes inline.
- **spectral-norm**: a small numeric function called from a loop, such as `eval_A` in the Benchmarks Game's spectral-norm, no longer pays per call for a signal check, a stack probe and stack resets it cannot need, and a loop counter or read-only integer is checked against the 1e15 limit once when the loop is entered instead of at every use. spectral-norm at N=5,500 takes 2.39 s of CPU time, from 3.04 s; Node.js takes 2.76 s.
- **Memoized functions in compiled code**: the compiled cache probe of a memoized function covers one to four numeric arguments, recursive functions such as `ackermann` and `tak`, and calls in `return f(...)` position. A cache hit is 64.3 times faster than on perl5 with CPAN *Memoize* for `fib`, 51.1 times for a two-argument `dist` and 64.2 times for `ackermann(2, $k)`. Deep memoized recursion stays compiled where the recursion warning is off, to a depth sized from the C stack that is actually left. A memoized function whose argument is a string no longer warns `Argument "..." isn't numeric`.
- **Nested loops** compile as one unit in more shapes: a `foreach` over an array or an array reference (`for (@a)`, `for my $x (@$ref)`) inside a compiled loop keeps its state in registers, as do inner loops under a `do {} while` loop, inside one branch of an `if`, or three levels deep. Whether a nest is worth compiling is judged by the work of all its trips, not by its outer body alone. fannkuch-redux at n=12 runs 25.1% fewer instructions.
- **Recursive subroutines** compiled by the JIT no longer fall back to the interpreter beyond 90 levels: they recurse in compiled code as deep as the C stack allows and issue perl's "Deep recursion" warning at level 100 themselves, naming the line perl names. Where a `__WARN__` handler, a fatal warning or a tied `STDERR` could see the call frames, the call is run again interpreted.
- **Subroutines that update outer variables**: a loop that calls a short subroutine writing a variable of an enclosing scope or a package variable, such as `sub add { my ($d) = @_; $acc += $d; 0 }`, compiles together with the call. The writes are made when the call returns, after every check, so a call that has to run again interpreted never writes twice.
- **JIT compile time**: a loop the JIT would reject after compiling it is rejected before its machine code is generated. Short runs gain most: mandelbrot at size 120 runs 11.5% fewer instructions and spectral-norm at N=500 8.1% fewer.

- **Regular expressions**: ReJIT compiles a pattern anchored at the start of the string, such as `/^bankrupt[a-z]+/`, once its interpreted matches have cost as much as a compile, instead of as soon as the subject is long. Such a pattern matched 500 times against a 71 KB string runs 3.52M instructions instead of 5.31M.
- **Number flags after compiled loops**: a compiled loop that copies integers (`my $t = $p[$i]`, `@p[$i, $j] = @p[$j, $i]`) leaves them integers instead of floating-point numbers, and `--` of an integral float keeps it a float. The values were right before; the flags, visible through `Devel::Peek`, were not.
- **Compiled loops** no longer give wrong answers in these cases: `@c = ($c[0] + 1, $c[1])` died with "Can't coerce ARRAY to integer in foreach loop iterator"; `$a[$i] += 1` past the largest integer wrapped to a negative number; `defined(wantarray)` was true in void context; and a second loop over the same non-`my` variable, `for $i (-3 .. -1)` after `for $i (0 .. 3)`, read the first loop's counter. A loop whose computed bound is NaN dies with "Range iterator outside integer range" instead of aborting pperl, and an array a compiled loop iterates is no longer kept alive past its scope, which had stopped a blessed array's `DESTROY` from running.
- **Compiled subroutines** no longer lose an update to an array element, as in `sub bump { my $i = shift; $acc[$i % 4] += $i; $i }`, or an assignment whose operator writes straight into a lexical, as in `$t = sin($i)`.
- **Warnings from compiled loops**: under `use warnings`, `$x--` from -1e20 in a compiled loop warns "Lost precision when decrementing" on every iteration as perl does, where it warned 65 times in 50,000; the same holds for `++` and for array and hash elements. A warning raised in a loop's condition names the line perl 5.44 names.

6.7.3 Modules

- ***Coro*** 6.57 runs, with *Coro::State* native and the distribution's own Perl modules; its 20 upstream test files pass on the assembler and the `setjmp` context-switch backends alike. Hot loops inside a coroutine compile, because the JIT now compiles on a stack of its own instead of the coroutine's 128 KiB one. Coro is part of builds with the `native-max` feature. Two cases in which perl5 crashes - a Coro::State call on a blessed hash that is not a coroutine, and the `:cede` layer without an interval - croak or fall back to the default interval instead.
- ***EV*** **and Event**: EV 4.37 with libev 4.33 and Event 1.28 are native, with *Coro::EV* and *Coro::Event* on top, in `native-max` builds; EV's 11 upstream test files pass. With EV present, *AnyEvent* picks it as its event loop, as on a perl5 that has EV installed. `EV::Periodic`'s `reschedule_cb` keeps its callback alive, where perl5 frees it after the first call of the getter.
- ***AnyEvent*** **and** ***Guard***: AnyEvent 7.17 ships with pperl and Guard is native, both in default builds. `Guard::cancel` on a reference to a plain value croaks, where perl5 sometimes crashes.
- **Native modules**: each module that is XS on perl5 and native in pperl ships its distribution's own `.pm` unchanged, among them B, *POSIX*, Socket, *Storable*, *Encode*, IO, *Data::Dumper*, *Scalar::Util*, *List::Util*, *Time::HiRes*, *Fcntl*, *File::Glob*, *Clone* and *PadWalker*, and none of them is loaded from the host perl's library any more. The native part is installed when the `.pm` calls `XSLoader::load`,

as in perl5, so `ref \$Socket::{AF_INET}` is `GLOB` and `B::svref_2object(\&List::Util::sum)->FILE` is `ListUtil.c`. A module left out of a build fails with perl5's own "Can't locate ..." message.

- ***Peta::Dict*** is new: `tie my %h, 'Peta::Dict'` binds a hash to an open-addressed table in which an integer and its decimal string are one key and `"05"` is another, as in a Perl hash; on perl5 the same source runs on a pure-Perl Peta::Dict with identical output. A compiled loop reads and writes such a hash through an in-register probe of the table instead of a method call per access, and `$h{$k}++` increments an integer stored in the table in place. The result of a bitwise operator is stored as an integer, as in perl, so a key built with `<<`, `|` and `&` stays one. The Benchmarks Game's k-nucleotide, counting packed keys in a Peta::Dict, takes 42.86 s of CPU time at 25,000,000 with the in-place increment, from 51.05 s.
- ***Peta::WebGPU*** draws as well as computes: surfaces, render pipelines and render passes with vertex and index buffers, uniforms, depth buffers and face culling, into an *SDL2* window or into a texture whose pixels can be read back without a display. A vertex layout written as a list of formats, `['float32x3', 'float32x3']`, gets its offsets and stride computed, and enum names are WebGPU's own spellings. SDL2 windows report the window-system handles this needs through `$win->wm_info`.
- ***Encode***'s Perl half is upstream Encode 3.24's, and its encoding tables are generated at build time from the distribution's `.ucm` files, matching upstream's for all 103 encodings. `encode_utf8(undef)` returns undef and `encode_utf8(42)` returns `"42"`, where both returned an empty string, and Encode's functions have perl5's prototypes, so `encode_utf8 $a, "\n"` is no longer one call with two arguments.
- **Crashes in native modules**: error messages containing a `%` sign no longer crash the native modules that raised them. *Digest::SHA* no longer crashes on a foreign object or computes a wrong digest past 512 MiB of input, *Storable* no longer passes a class name to a string `eval`, and *GD* and *SDL2* no longer free an image twice or crash on surfaces.
- ***Memoize***: a memoized recursive function no longer crashes pperl at depth; without the JIT it segfaulted at 20,000 levels, where perl5 runs 500,000. A numeric function such as `eval_A` memoized before it was ever called no longer crashes pperl on its first call.
- ***PDL*** **operators**: `+=`, `-=`, `*=` and the other assignment operators modify an ndarray in place, so aliases and slices see the change, where they built a new ndarray; `PDL::mult($a, $b, $c, 0)` writes its result into `$c`. `.=` broadcasts over dimensions, keeps 64-bit values exact, wraps bytes as C does and refuses a read-only target. `xchg`, `transpose` and `dummy` return views that write through to their parent and keep its type, where they returned double-precision copies.
- ***PDL*** **exports and loading**: `use PDL` exports what upstream exports, each name from the sub-module that owns it - 228 names, where a hand-written list exported about 58 that perl5 does not and lacked about 122 that it does; `glue` and `topdl` are no longer exported. *PDL::Exporter* and *PDL::IO::Storable* ship with pperl instead of being loaded from the host perl. `use PDL; use PDL::Lite` under `-w` no longer warns that `main::pdl` was redefined, a regression of 0.6.18, and `\&PDL::Ufunc::sumover` is the same subroutine as `\&PDL::sumover`.
- ***PDL*** **broadcasting**: `broadcast`, `broadcast1`, `broadcast2` and `broadcastI` re-

order dimensions as upstream does instead of returning the same ndarray, and `unbroadcast` puts them back, where it used to drop trailing size-1 dimensions. A reduction given an output ndarray, `maximum($src, $out)`, writes into it, and PDL error messages no longer print garbage.

- ***PDL***: methods called on a result not yet computed, such as `isempty`, `copy` or `transpose` on `$a + $b`, no longer treat it as empty, and `dims` in scalar context returns the number of dimensions. `svd` and `ylinvals` work on slices, `tritosquare` fills the lower triangle, and `pctover`, `oddpctover`, `pct`, `oddpct` and `inner` compute as upstream, broadcasting and bad values included. Element reads and writes of 64-bit, unsigned and complex ndarrays no longer pass through a double.
- ***List::Util*** **and** ***Scalar::Util***: `sum`, `sum0` and `product` switch to floating point when an integer total would overflow, where `sum(IV_MAX, 1)` wrapped to a negative number, and they honour tied and overloaded arguments. Scalar::Util's `isvstring`, `openhandle` and `looks_like_number`, and List::Util's `pairs` and `mesh`, behave as upstream's own tests expect.
- ***Math::GMP*** functions called with a plain number as the first argument, such as `Math::GMP::bsqrt(16)` or `stringify(42)`, return the answer instead of crashing, and every function checks its argument count as perl5's does. `2 ** Math::GMP->new(3)` is 8, not 9.
- ***File::Glob***: under `-T`, every name returned by `glob`, `<*.c>`, `bsd_glob` and `csh_glob` is tainted, as in perl5; all of them came back untainted.
- ***GD***: `copyRotate90`, `copyRotate180`, `copyRotate270`, the `copyFlip` methods, `crop`, `contrast` and `smooth` work as in the GD distribution.
- ***Cwd***: `fast_abs_path` follows Cwd.pm, so for an empty string or a symbolic-link loop it returns undef with `$!` set to ENOENT.

6.7.4 Command line

- **The debugger** starts with `pperl -d`, `-de0` and `-d:Module` as with perl5, where `-d` was accepted and ignored and the other forms were rejected. The default *Term::ReadLine::Gnu* front end no longer aborts at the first prompt, and a piped debugger session matches perl 5.44's byte for byte.
- **Command-line switches**: `-T` and `-t` combine with other switches (`-Tw`), `-W` and `-X` are perl5's and take effect in command-line order, and `-s` is perl5's switch parsing again; pperl's own statistics are `--stats` only. A program given with `-e` no longer gains an empty last line, so a warning at its end names line 1, as in perl5.

6.7.5 Documentation

- **Translations**: the guides, tutorials and errata, and the module reference pages, were brought to complete translation in all 41 languages, except the pages held for rewrite. Shipped translations were reviewed in every language, and wrong terms, headings and link labels left in English, double-encoded text and bold text that rendered as literal asterisks were repaired.
- ***Coroutines guide***: a new chapter of the concurrency guide, Coroutines and event loops, covers *Coro*, *EV*, Event and AnyEvent: starting and switching corou-

tines, channels, semaphores and signals, blocking I/O through *Coro::AnyEvent* and *Coro::Handle*, and how *AnyEvent* picks its loop. It is translated into all 41 languages.

- **Graphics guides**: the 2D guide has a new chapter on raster drawing with *GD*, and examples in its other chapters were corrected after running them. The 3D guide gains its GPU part on *Peta::WebGPU* - a device and window, the render pipeline, a 3D scene with depth buffer, culling, instancing and lighting - and two capstones, a modeller and a 3D game.
- **Corrections**: about 250 statements in the guides about what pperl supports were run against pperl and perl5, and the false ones corrected: the one-liner switches `-n`, `-p`, `-a`, `-F`, `-i` and `-l` work, threads are not available, and `-d` starts the debugger. Errors that translators reported in the regular-expression guide, the tutorials and the perlfunc pages were checked on perl 5.44 and fixed, among them the bound of `{n,m}` and the version that introduced `any` and `all`. The introduction's architecture diagram shows the runtime pperl has.
- **Module reference**: the module reference pages state what each function does, checked against perl 5.44, with the differences from upstream listed where there are any, and no longer carry implementation notes. Mistakes that translators reported were fixed, among them the precedence in the *Fcntl* `S_IS*` examples and the behaviour of *Hash::Util*'s `lock_hash`.

6.8 pperl 0.6.18

2026-09-10 to 2026-09-16, 78 commits.

- ***Memoize*** is native and no longer loaded from the host perl's library; its `Memoize::*` helper modules ship with pperl. `memoize`, `unmemoize` and `flush_cache` support `NORMALIZER`, `INSTALL` and the `MEMORY`, `HASH`, `TIE`, `FAULT` and `MERGE` caches, checked against the Memoize 1.17 distribution's own tests. A memoized function of one or two numeric arguments keeps its results in a cache that compiled code reads directly, and a loop calling it compiles: a memoized `fib(30)` called in a loop costs 43 instructions per call, against 2858 on perl5 with Memoize and 1682 on pperl without the JIT.
- **Foreach loops** over an array or an array reference (`for my $e (@$table)`) compile when nested in another compiled loop, and a `last` that leaves one compiles to a jump, also from inside an `if (...) { ...; last }` block. Appending an array element to a string (`$line .= $e->[0]`, `$s .= $a[$i]`) compiles too. fasta, whose character-selection loop combines all three, runs 49.3% fewer instructions at 250,000.
- **Integer array loops**: a compiled loop that copies scalars and swaps or stores elements of an integer array (`@p[$i,$j] = @p[$j,$i]`, `$p[$k] = $k`) keeps both integers and runs without leaving compiled code. fannkuch-redux at n=9, whose flip loop has this shape, runs 38.6% fewer instructions.
- **Loop unrolling**: the JIT compiles a `for` loop over a constant range of at most eight iterations, such as `for my $i (0 .. 4)`, as straight-line copies of its body, and does the same for a loop nested in it whose bounds become constant in each copy, such as `for my $j ($i + 1 .. 4)`. nbody's pair loop becomes ten independent chains of `sqrt` and division, and nbody at 50,000,000 steps runs 21.9% fewer instructions.

- **Substring counting**: a loop that counts substrings in a hash, `$h{substr($seq, $i, $k)}++`, compiles when the offset is a lexical loop variable (`for my $i (...)`) rather than `$_`. k-nucleotide at 250,000 runs 11.0% fewer instructions.
- ***SDBM_File*** blesses the tied object into the class `tie` was given, so a subclass works: after `tie %h, 'My::DBM', ...`, `ref tied %h` is `My::DBM`, not `SDBM_File`. `Memoize::AnyDBM_File` relies on this.
- ***PDL***: `nan` and `inf` build an ndarray of the dimensions they are given (`nan(4, 4)`) instead of ignoring their arguments; without arguments they still return a 0-dimensional value. `PDL::zeroes(4,4)` takes its first argument as the class and fails, as on perl5, while the `zeroes` and `zeros` that `use PDL` imports still take dimensions alone.

6.9 pperl 0.6.17

2026-09-06 to 2026-09-09, 96 commits.

6.9.1 Language and runtime

- **`__END__` in bundled modules** no longer opens a `DATA` handle. A module shipped inside pperl that ends in `__END__` followed by POD, `strict` and `warnings` among them, had a `DATA` handle reading that POD; as in perl5, only `__DATA__` gives a module a `DATA` handle.

6.9.2 JIT and parallelization

- **nbody**: the Benchmarks Game's nbody at 200,000 steps runs 293M instructions, from 2,778M. At its official 50,000,000 steps it runs 47.65G instructions in 6.36 s of CPU time, where Node.js takes 49.66G and 6.0 s. Besides the call and array work below, compiled loops keep floating-point values in registers instead of reloading them from memory, check a loop's constant bound once instead of on every iteration, and skip checks that cannot fail, such as overflow on `$j + 1` of a loop counter and the domain test on `sqrt` of a sum of squares. Placing the arrays' buffers so that loads and stores to different arrays no longer stall the processor cut nbody's CPU cycles by 18%.
- **Calls to subroutines that loop**: a compiled loop that calls a subroutine with loops of its own, such as nbody's `advance()`, now compiles together with the called body, where before each call left compiled code and entered it again twice. The subroutine's `my` variables are held in registers, and its call frame is pushed and popped inside compiled code. `$_[0]` in a subroutine called from one place with a numeric literal argument reads the literal directly.
- **Arrays of numbers** whose elements are all floating-point numbers, or all integers, when a compiled loop starts are read and written as flat memory, without a type check per element. For the loop's duration such an array's values are kept in a buffer and written back when the loop is left, and an index that is a loop counter is checked against the array's size once, at the loop's start. On nbody the buffer alone saves 15.3% of the instructions.
- **Calls to small numeric subroutines** no longer leave compiled code: a loop calling a function such as `eval_A` in the Benchmarks Game's spectral-norm, a `use inte-`

ger body that reads `$_[0]` and `$_[1]` directly, passes the arguments in registers, and the function's own `my` variables are registers too. spectral-norm at N=500 runs 440.0M instructions, from 776.5M, and at its official N=5,500 takes 3.25 s of CPU time, from 5.33 s.

- **Nested `while` and C-style `for` loops** inside a compiled loop compile as part of it, and an unlabelled `last` or `next` out of them is a jump. The Benchmarks Game's fannkuch-redux at n=9 runs 30.7% fewer instructions (6,650M to 4,609M).
- **Number types after compiled loops** match perl5's. `$v = $i * 0.5` and `sqrt(4)` stay floating-point numbers even when the result is integral, an integer literal or a `use integer` result stays an integer, and an integer variable such as `$t = ($i <=> 250)` no longer carries a floating-point copy perl5 would not have. The values were right before; the flags, visible through `Devel::Peek`, were not.
- **Loops left half-compiled by 0.6.16** compile fully again: a loop whose variables are all assigned before they are read (`$t = ($i <=> 250)`), one that updates a package variable (`$G = $G + 1`), and one whose counter was used as a float during its first iterations (`sin($i)`). They had fallen back to a slower, checked mode; for the package-variable loop that was 2.2 times the instructions.
- **JIT compile time**: a subroutine the JIT would compile and then reject as a numeric function is rejected before machine code is generated. nbody's fixed compile cost falls from 157M to 107M instructions, and the Benchmarks Game's mandelbrot at size 120, a run that is mostly compilation, executes 9.5% fewer instructions.

6.9.3 Modules

- ***Peta::WebGPU*** is new: GPU compute on the wgpu library, in builds with the `webgpu` feature. `Peta::WebGPU::Device->new` opens a device with the adapter's own limits, compiles WGSL shaders, creates storage and uniform buffers and pipelines, and runs compute passes; `have_gpu()` and `adapters()` report what the machine has. A shader or validation error is a Perl exception with wgpu's message, and a method with no natural result returns its object, so `or die` works and calls chain. A *PDL* ndarray uploads to a buffer directly (`$gpu->storage_buffer(float(1 .. 8))`); a double-typed one is refused with the conversion to use, since WGSL has only 32-bit numbers.
- ***SDL2***: 25 object methods that die on failure, among them `clear`, `copy`, `draw_line`, `fill_rect`, `present`, the `save_*` methods and every `set_*`, returned nothing on success, so `$ren->clear or die` died when it worked. They now return a true value, the object where there is no natural result, and calls chain: `$ren->set_draw_color(30, 30, 40, 255)->clear`. The direct `SDL_*` functions still return what the C functions do.
- **Loading modules under `-w`** prints no warnings perl5 does not print. `use POSIX ()`, and with it `use Imager`, printed 27: constant subroutines shared with *Fcntl* reported as redefined, and two "Ambiguous call resolved as CORE::open()". `use PDL; use PDL::MatrixOps` printed 48 redefinitions, `use PDL::Lite; use PDL` 532 and `use PDL::LiteF; use PDL` 631, because PDL's sub-modules were loaded twice.
- ***PDL*** **printing** follows upstream PDL's rules. Elements print as perl prints numbers, so `pdl(1.0) - pdl(0.88096)` shows `0.11904`, not `0.11904000000000003`; `$PDL::doubleformat` and `$PDL::floatformat` exist and take effect, and

`$PDL::toolongtoprint` is honoured. A 3-D ndarray prints as nested blocks instead of one flat list, `PDL->null` as `Null`, an empty ndarray as `Empty[0]`, and a 2-D ndarray ends with its newline.

- ***PDL***: append on a slice no longer aborts pperl and `glue` no longer fails with "Not physical but doesn't have parent"; neither needs `->sever` first. `-=`, `+=`, `*=`, `/=`, `**=` and `%=` on an ndarray no longer warn "Use of uninitialized value in subroutine entry". `pdl('BAD')` is a 0-dimensional ndarray, `pdl('[[5]]')` stays 2-dimensional, and `norm` normalises a vector as upstream's does instead of dividing the whole ndarray by its Frobenius norm. `use PDL ()` no longer defines `main::pdl`.

6.9.4 Documentation

- **Module reference**: the *Imager* and *GD* pages include documentation that was written but never published: Imager's colour classes, its kinds of image and the channel mask, and `GD::Image` and `GD::Font`.
- **Graphics guides**: the 2D graphics guide is fully translated again in all 41 languages after its move to *SDL2*. The 3D guide's chapter on opening a window with an OpenGL context is withdrawn, since pperl's GPU binding is *Peta::WebGPU*, and the guides' examples save their images with `$ren->save_png`.

6.10 pperl 0.6.16

2026-08-29 to 2026-09-06, 134 commits.

6.10.1 JIT and parallelization

- **Regular expressions** are compiled to machine code once a pattern has searched 64 KiB of text. The compiler covers literals, character and *POSIX* classes, alternations, captures, greedy and non-greedy quantifiers, `\b` and `\B`, and case-insensitive matching under `/aa`; a pattern it cannot compile runs on the regular engine and, once refused, costs next to nothing per match. It is on by default, and `--no-jit` turns it off together with the JIT for Perl code. Its results matched perl5's on all 29 files of PCRE2's regression test corpus.
- **Regex search**: a compiled pattern finds candidate start positions 16 bytes at a time, testing up to three of its leading characters at once, also when it starts with an alternation, a character class or a case-insensitive literal. An anchored pattern is tried only where it can match. The Benchmarks Game's regex-redux at 250,000 runs in 1.74 billion instructions, against 9.80 billion in 0.6.15.
- **Numeric loops** keep their variables in machine registers also when the loop body reads and writes array elements, and their intermediate results no longer pass through Perl values; each array is checked once when the loop is entered instead of at every statement. The Benchmarks Game's nbody at 200,000 steps runs 35.8% fewer instructions, and the Benchmarks Game's mandelbrot 14.1% fewer at size 400 and 7.7% fewer at size 120.
- **Nested counted loops**: a `for` loop over a range nested in a compiled loop, such as `for my $j ($i + 1 .. $n)`, with a lexical loop variable or without one, no longer pushes a Perl loop context on every pass. Its counter lives in a register, and

the context is built only if control leaves compiled code while the loop is open. nbody runs another 34.8% fewer instructions, 2.80 billion against 4.29 billion.

- **Compound assignment** to array and hash elements compiles: `$a[$i] += $x`, `-=` and `*=` no longer keep their whole loop from compiling, and nbody, which updates its velocities this way, runs 18.25% fewer instructions. `|=`, `&=`, `^=`, `<<=` and `>>=` compile on scalar variables, and `||=`, `&&=` and `//=` on lexical and package scalars.
- **Hash elements** with a constant or lexical key, `$h{$k}` and `$h{'name'}` on a lexical hash, compile in a loop both as values and as assignment targets, as in `$sum += $h{$k}` or `$h{$k} += 1`; before, only a computed key compiled. Appending one to a string, `$s .= $h{'name'}`, compiles too.
- **Package variables** no longer keep a loop from compiling when they are incremented or assigned: `$count++` and `$total = $total + $x` on a package variable compile, as does assigning a package variable or a conditional expression to an array or hash element.
- **Increments in expressions**: `++` and `--` on an array or hash element compile when their result is used, as in `my $old = $a[$i]++`, and so do `$x = $g++` on a package variable and `($c ? $x : $y)++`.
- **Many lexical variables**: loops in a sub or file with many lexical variables compile as they do in small ones. Variables and temporaries beyond the 64th of their sub or file were never kept in registers, and some statements using them, such as `$t{"k$i"} += 1`, kept their loop from compiling.
- **Short loops** that cannot repay the cost of compiling them stay interpreted: a counted `for` loop is compiled only once it has run, or its bounds show it will run, enough iterations for its size, counted across all its runs. Loops that call a sub, and `while` loops, compile as before. Short programs whose loops ran a few hundred times used 4 to 8 times the instructions they need with `--no-jit`.
- **Subroutine calls with computed arguments** in a compiled loop, such as `f($i, $i + 1, $i + 2)`, are compiled into the loop again; in 0.6.15 such a loop ran no faster than without the JIT. A loop making a three-argument call runs a 15.6th of the instructions, and every Benchmarks Game program measured ran fewer as well.
- **Numeric subroutine calls** from a compiled loop pass their arguments in registers when the sub takes two numbers through `my ($a, $b) = @_` or reads `$_[0]` and `$_[1]` directly. A loop calling a sub that reads `$_[0]` directly could crash or die with “Modification of a read-only value”; it now gives perl5’s results.
- **Conditional assignments** such as `my $v = $c ? 1 : $i / 7;` followed by arithmetic on `$v` give perl5’s result in a compiled loop. If only the integer branch had run when the loop was compiled, the fraction was later read as an integer, and the sum was wrong but plausible.
- **Element stores** in a compiled loop keep a value’s string: with `$x` holding `"0.0"` after a numeric read, `$a[$i] = $x` stored `0` and now stores `"0.0"`, as perl5 does. The same applied to hash elements and to slice assignments.
- **Crashes in compiled loops**: some nested loops, among them an inner `for (1 .. $n)` whose body never reads `$_`, ended in a segmentation fault in compiled code. They now run as on perl5.

6.10.2 Modules

- ***SDL2*** is a new native binding to libSDL2: windows, renderers, textures, OpenGL contexts, events and timers, both as SDL's own C functions one for one and as objects (`SDL2::Window`, `SDL2::Renderer`, `SDL2::Texture`, `SDL2::GLContext`) that free their handles when they go out of scope. `SDL2::Surface`, `SDL2::Image` and `$renderer->save_png`, `save_jpg` and `save_bmp` load and save images, and `$renderer->set_target($texture)` draws into a texture. Its 709 constants are taken from the installed SDL2 headers, not typed in.
- ***GD*** loads: `use GD` reaches the system's libgd through a native binding, and GD's own Perl modules, including `GD::Polygon`, `GD::Polyline` and `GD::Simple`, run unchanged on top of it. Images can be drawn, compared and written as PNG and JPEG. `$im->clone` after `filledPolygon` returns a correct copy, where perl5's GD crashes on a libgd defect.
- ***Imager*** loads, as a first part: images of all four kinds can be created, described, and have their pixels written and read back, with `Imager.pm` and its Perl modules running unchanged. Drawing, file formats, transforms, filters and fonts are not yet available.
- ***Peta::PCRE2*** is a new native module that binds the system's libpcre2-8: `new`, `match`, `count`, `substitute`, `capture_count`, `jit`, `version` and `jit_available`, and 12 `PCRE2_*` constants. Without libpcre2 installed, its functions croak.
- ***Cairo*** without libcairo installed croaks with the name of the package to install on every call; before, the second call hung.

6.10.3 Command line

- **`--no-regex-jit`** turns off the regex compiler and leaves the JIT for Perl code running; `PPERL_RXJIT=0` does the same from the environment, and `--help` describes both. `PPERL_RXJIT_STATS=1` reports for each pattern whether it was compiled or refused, and why.

6.10.4 Distribution

- **Profile-guided builds** of the release binary now train on programs that exercise the JIT and the regex compiler; before, those parts were barely run during training and were optimised for size. On the first build with the extended training, every Benchmarks Game program measured ran fewer instructions, the Benchmarks Game's revcomp 6.2% fewer.

6.10.5 Documentation

- ***3D graphics guide***: a new guide with chapters on the mathematical foundations, a software rasteriser, shading, meshes, model formats (OBJ and STL readers and writers), modelling operations, scene graphs, and opening a window with an OpenGL context through *SDL2*. Every example was run, and the outputs shown are what pperl printed.
- ***2D graphics guide***: the GUI chapter and both capstone projects, a paint program and a game, are rewritten on the native *SDL2* binding and run on pperl. The chap-

ters on `Peta::FFI` now cover callbacks and its memory primitives, which they had called out of reach.

- **Module reference**: 49 new pages, among them `Peta::PCRE2`, 18 for *Cairo* and 14 for *PDL*, and the catalogues of all 41 translation languages were updated to match.

6.11 pperl 0.6.15

2026-08-21 to 2026-08-29, 125 commits.

6.11.1 JIT and parallelization

- **The previous JIT engine is removed.** It had been off by default and was chosen with `PPERL_JIT2=0`; that setting now runs pperl without a JIT, and the engine that was already the default is the only one. `--jit-stats` no longer prints the old engine's counters, which were always zero, and reports how many compiled regions were installed and how many loops were tracked.
- **List assignments in compiled loops** keep all their elements. `@w = ($i, "$str$i")` in a compiled loop assigned only the string, and `@out = ($c ? $x : $y, $i)` in a rarely taken branch of a compiled loop lost its first element.
- **Compiled loops** give perl5's results in three more cases. A sub call that is the last statement of a sub could lose its return value when compiled into a loop; `$x = chr($n)` into a tied variable did not call the tie's `STORE`; and `pop(@$r) + $#$r` in one statement read the array's last index from before the `pop`.
- **Calls to two-argument subs**: a sub that reads `$_[0]` and `$_[1]` directly and keeps a `my` variable, as `eval_A` in the Benchmarks Game's spectral-norm does, is compiled and called from the compiled loop that uses it, where before the loop made a regular Perl call. spectral-norm at N=500 runs 11.7% fewer instructions.
- **Long-running loops**: a counted `for` loop that averages at least 256 iterations each time it is entered is compiled a second time. The new version checks the types of the loop's floating-point variables and arrays in a first iteration, then runs the remaining iterations in a copy of the body without those checks and with the loop counter in a register; a loop that leaves this copy too often goes back to the first version. spectral-norm at N=500 runs 17.9% fewer instructions from this, and with the previous entry's change 792.4 million in all, against 1,099.9 million before.
- **Array slices and element stores**: swapping two elements with a slice, `@p[$i, $j] = @p[$j, $i]`, and storing into an element, `$p[$j] = $p[$j + 1]` or `$count[$k]++`, compile in a loop over a lexical array. The Benchmarks Game's fannkuch-redux at n=9 runs 22.5% fewer instructions from the slice alone. It needed more instructions than perl5 before; now perl5 needs 1.12 times as many.
- **Compound assignment to array elements**, such as `$vxs[$i] -= $dx * $mm2`, no longer keeps a loop from compiling: the element is fetched by the interpreter and the arithmetic around it is compiled. The Benchmarks Game's nbody at 200,000 steps runs 8.61 billion instructions instead of 13.46 billion. It needed more instructions than perl5 before; now perl5 needs 1.31 times as many.
- **Counted loops without a loop variable**, such as `for (1 .. 50)`, keep their counter in a register, and a compiled loop updates less interpreter bookkeeping at

each statement, while `die` and `warn` still report the right line. Further specialised versions of a loop are compiled only once the loop has run long enough to repay them. The Benchmarks Game's mandelbrot at its official size of 16,000 runs 25.8% fewer instructions, 1,541 billion against 2,077 billion.

- **Compiled loops that keep falling back** to the interpreter are noticed in every case and handed back to it; before, a loop could take its slow path on 99,936 of 100,000 iterations indefinitely. A loop whose called sub is redefined is compiled again instead of staying interpreted for the rest of the run.
- **`$#array` and `$#_`** compile inside a loop, as `$#$ref` already did; before, either kept the statement using it from compiling.

6.11.2 Modules

- ***List::Util***: `first`, `any`, `all`, none and `notall` with a block, and `reduce`, run their block as compiled code when the list has at least 64 elements. Against perl5 with its own List::Util, `first` is 1.61 times as fast, `any`, `all` and none 1.88 to 1.90 times and `reduce` 1.10 times, where before they ran at 0.82 to 0.86 times perl5's speed. `min` and `max` return the element itself, call overloaded `>` and tied values' `FETCH`, and compare numbers above 2**53 as perl5 does.

6.12 pperl 0.6.14

2026-08-18 to 2026-08-21, 45 commits.

- **Recursive subroutines** compile to machine code. A subroutine that takes one argument (`$_[0]`, `my ($n) = @_` or `shift`), returns a number and calls itself, such as `fib` or `check_tree` in the Benchmarks Game's binary-trees, runs compiled, its recursive calls included. Its body may branch, return through `return EXPR` or a conditional expression, and test `ref $_[0]`; a loop that calls it calls the compiled code directly.
- **binary-trees**: a call whose argument is another call, as in `$check += check_tree(bottomup_tree($d))`, reaches the compiled `check_tree` while `bottomup_tree` still runs in the interpreter. At depth 16 binary-trees takes 2.76 s instead of 4.07 s, 1.5 times as fast as perl5, where it ran at parity before.
- **Subroutine calls in compiled loops**: a loop that calls an ordinary subroutine, one with a real `@_` that may write package variables, compiles together with the called body, with the call frame perl5 would build. Arguments may be computed expressions such as `tally($i % 97)`, the sub's return value can be used, and several call sites of one sub in a loop all compile. A sub that was redefined or re-aliased, a call resolved through `AUTOLOAD`, and a call into a sub that is already running take the regular call path.
- **Subroutine call overhead** in compiled loops is lower: compiled code sets up and removes the sub's frame itself, without pushing the argument mark or the sub onto the stack. Temporary arguments are not copied when the sub cannot alias or capture them, and a scalar return value is not copied when the sub has nothing to clean up on return. A loop calling `sub tally { $SUM += $_[0]; $CNT += 1 }` two million times runs 1.84 times as fast as on perl5, and a loop adding up the return value of a sub without `my` variables 1.74 times as fast.

- **`wantarray` and `caller`** compile in a sub called from a compiled loop when they are used as a truth value, as in `wantarray ? 2 : 1` or `run() unless caller`.
- **Array references in compiled loops**: `$m[$i][$j]`, `$_[0][0]`, `$x->[$i]`, `$$y[$i]`, `$#$x`, pop `@$x` and slices such as `my ($a, $b, $c) = @$x[-3..-1]` compile, as do `unshift` and blocks that open their own scope. Negative indexes such as `$x->[-1]` no longer leave compiled code on every iteration. In the Benchmarks Game's pidigits, the inner loop of *Math::BigInt::Calc*'s arithmetic now compiles in full.
- **`push` in compiled loops**: `$n = push @a, $x;` assigns the array's new length on every iteration. `$n` kept the value it had when the loop was compiled, so after 200,000 pushes it read 64.

6.13 pperl 0.6.13

2026-08-15 to 2026-08-18, 35 commits.

6.13.1 JIT and parallelization

- **Element increments** compile: `$h{$key}++` and `$a[$i]--` in a loop no longer keep it from compiling, also when the hash key is a computed string, as in `$h{substr($seq, $i, $k)}++`. The first increment of a new element creates it as an integer, as perl5 does. The Benchmarks Game's k-nucleotide at its official input runs in 74.1 s instead of 99.2 s, 1.30 times as fast as perl5.
- **`@_` element reads** compile: `$_[$j]` and `$_[0]` inside a sub no longer keep its loops from compiling. The Benchmarks Game's spectral-norm, whose inner loops read `$_[$j]`, compiles them and at N=5,500 runs in 17.8 s instead of 61.8 s; perl5 takes 64.6 s.
- **Inlined subroutines**: a loop that calls a short sub by name, one without branches or recursion that takes its arguments through `my (...) = @_`, `my $x = shift` or `$_[k]`, compiles with the sub's body in place of the call. A sub that only reads a lexical of its file, such as `sub f { $g }`, is inlined too, and redefining an inlined sub takes effect as on perl5. A loop calling a three-argument sub runs 18.1 times as fast as on perl5, and one using the return value of a two-argument sub 17.7 times.
- **Loops that call subroutines** compile when the calls are the only part of the loop the JIT cannot compile: the calls run in the interpreter between stretches of compiled code, and hot loops inside the called sub still run compiled. A `die`, a `last` that leaves the sub, and recursion behave as on perl5.
- **`use integer`**: integer addition, subtraction, multiplication, division and modulus, the six numeric comparisons and the shift operators compile under the pragma, with perl5's wrapping integer results; before, they were left to the interpreter. Benchmark loops under `use integer` run 3.7 to 4.4 times as fast as on perl5.
- **Foreach over arrays**: `for my $x (@a)` and `for my $x (@$ref)` compile, where only loops over a range did before; reversed and indexed forms stay in the interpreter. A benchmark loop over an array runs 4.94 times as fast as on perl5, against 0.74 times before.

- **Lexical array reads** with a variable or constant index, `$a[$j]` and `$a[3]`, compile in a loop, also inside an inlined sub reading an array of its file.

6.13.2 Modules

- ***Storable*** freezes faster: a benchmark freezing an array, which ran at 0.895 times perl5's speed, runs about 1.2 times as fast as perl5.
- ***Digest::MD5*** computes digests in 2.2% more cycles than perl5's Digest::MD5, down from 13.4% more.
- ***Math::GMP*** reads a tied or otherwise magical argument before converting it, as perl5 does, and `_gmp_lib_version` returns a v-string.

6.13.3 Documentation

- **Peta::FFI::Cairo reference**: its pages are translated into 41 languages, where their prose was untranslated before. They were corrected against the module, among them the description of reference counting, and `Context::get_matrix` and `get_font_matrix` are documented for the first time.
- **Command-line options page**: the page built from `pperl --help` lost every wrapped line of an option's description, 11 lines across `--cpan`, `--cpan --system`, `--via-daemon`, `--via-daemon=trunk` and `--interactive`, and did not list `--native-manifest=DIR` at all. It now shows the complete text.

6.14 pperl 0.6.12

2026-08-07 to 2026-08-14, 94 commits.

6.14.1 JIT and parallelization

- **The new JIT engine is the default.** The rebuilt JIT replaces the previous engine. It compiles the hot path of a loop to machine code and leaves the code around the loop, and the branches the loop rarely takes, to the interpreter. A loop whose compiled code keeps falling back to the interpreter is handed back to the interpreter entirely.
- **The previous JIT engine** remains available with `PPERL_JIT2=0`. It is still faster on loops whose bodies hold many statements: the Benchmarks Game's mandelbrot at size 16,000 takes 36.8 s on it and 204.7 s on the new engine. It is also the only one of the two that runs loops in parallel, so under the new default, auto-parallelization no longer applies to loops.
- **Arithmetic in compiled loops**: integer and floating-point `+`, `-`, `*`, `/`, `%` and `**`, the assignments `+=`, `-=`, `*=`, `/=` and `%=`, numeric comparisons including `<=>`, the bitwise and shift operators, `int`, `abs`, unary minus, `!`, `sqrt`, `sin`, `cos`, `exp`, `log`, `++` and `--` compile, on lexical and package variables. Integer values stay integers in compiled code, as they do in perl5. Benchmark loops of addition, multiplication, division and modulus run 4.27 to 5.06 times as fast as on perl5.
- **Strings in compiled loops**: `length`, `eq`, `ne`, `lt`, `gt`, `le`, `ge`, `cmp`, concatenation, interpolation, `.=`, `uc`, `lc`, `ucfirst`, scalar `reverse`, `ord`, `chr`, `substr` and `index`

compile, as do assignments of strings to variables. Loops that concatenate run 2.09 to 2.48 times as fast as on perl5, and loops using `substr` or `index` 2.90 and 2.99 times.

- **Arrays and hashes in compiled loops**: reading and assigning elements of lexical arrays and hashes, `$a[$i]`, `$a[3]` and `$h{$k}`, compiles, as do `push`, `pop` and list assignments such as `@a = (...)`, `%h = (...)`, `($x, $y) = ($y, $x)` and `my ($p, $q) = (...)`. A benchmark loop writing array elements runs 4.57 times as fast as on perl5, one reading them 3.10 times.
- **Conditionals in compiled loops**: `if`/`else` blocks, the conditional operator `?:`, `my` declarations and `defined` inside a loop body compile with the rest of the loop. A loop assigning a conditional expression runs 2.89 times as fast as on perl5.

6.14.2 Modules

- ***Encode*** encodes into the single-byte encodings, such as Latin-1, without searching a table for every character: a 1 KB string encodes at 5.8 times perl5's speed, where it ran at 0.69 times. Strings of 64 KiB and more are encoded on several cores unless `--no-parallel` is given.
- ***List::Util***: `sum`, `min`, `max` and `product` over 65,536 or more integers are computed on several cores unless `--no-parallel` is given.

6.14.3 Documentation

- **Module reference translations**: the pages of `B`, `IO`, `Data::Dumper`, `PerlIO::mmap` and `Time::HiRes` in all 41 translation languages follow their corrected English text. Changed descriptions read in English until they are translated again.

6.15 pperl 0.6.11

2026-08-05 to 2026-08-07, 21 commits.

- **JIT compilation** takes less time in the release binary, which no longer runs a consistency check over every piece of machine code the JIT generates. The check is kept in debug builds.
- ***File::Temp***: the documentation of `safe_level` describes what it does. `STANDARD`, `MEDIUM` and `HIGH` are all honoured, and a parent directory that fails the check is rejected with `Parent directory (...) is not safe`; the page still said that the setter had no effect and that higher levels were lowered to `STANDARD`.
- **Translations**: the description of `B`'s `children` method read as the description of `length` in 37 of the 41 languages and was empty in two; it is correct in all 41. The Estonian and French reference pages had translated argument placeholders such as `LENGTH` and `SCALAR` inside code, so the text explained arguments the English signature above it does not have; they match the signature again.

6.16 pperl 0.6.9

2026-07-28 to 2026-08-05, 211 commits.

6.16.1 Language and runtime

- **SIGPIPE** stops a program whose output pipe was closed, as on perl5: `pperl generate.pl | head` ran forever. pperl keeps the `SIGPIPE` disposition it was started with instead of ignoring the signal, so the exit status is 141, and `$SIG{PIPE}` handlers and `$SIG{PIPE} = 'IGNORE'` behave as on perl5.
- **`$0`**: assigning to it, as cpanm and Starman do to label their processes, no longer crashes pperl or writes into unrelated memory. The assignment changes the process's command line as `ps` and `/proc/self/cmdline` show it, also for names longer than the original command line, so Starman's renamed workers can be told apart.

6.16.2 JIT and parallelization

- **mandelbrot**: the Benchmarks Game's mandelbrot, with its thread pool replaced by a plain loop over scanlines and its kernel unchanged, now runs its kernel compiled; before, the kernel never compiled. At size 2000 it takes 0.62 s of CPU time on one core, against 4.96 s wall-clock and 19.7 s CPU time for the published four-thread entry on perl5.
- **Loops that now compile**: `for (LIST)` with the implicit `$_` compiles, as do `next LABEL` and `last LABEL` aimed at an enclosing compiled loop, and a `for` loop over an array nested in another compiled loop, which now compiles together with it. A loop body may begin with `my $x;` or `my ($a, $b);`, assign a list such as `my ($a, $b) = (0, 0)`, and append with `.=` to a variable declared without a value; each of these kept the loop from compiling.
- **Element updates**: the assignment operators `+=`, `-=`, `*=`, `/=`, `|=`, `&=`, `^=`, `<<=` and `>>=` and the operators `++` and `--` compile on array and hash elements and on elements reached through references, such as `$m[$i][$j] += $x`, also when the index is computed, as in `$a[$i + 1]` or `$a[$i % 2]`. `sqrt` compiles too.
- **Subroutines called from compiled loops** compile together with the loop also when they use arrays, hashes or strings, use package variables, or assign to a `my` variable they close over, as in `my $acc; sub add { $acc += $_[0] }`. Arguments read through `@_` or as `$_[0]` compile, also in a sub that never copies them, such as `sub f { $_[0] * $_[1] }`, and a call with literal arguments such as `f(3, 5)` no longer keeps its loop from compiling.
- **Wrong results in compiled loops**: a nested range loop whose body assigns its loop variable or whose lower bound is an expression, a sub assigning to `$_[0]`, and a sub adding to a variable it closes over could abort or give wrong results in compiled code. They now give perl5's results.

6.16.3 Modules

- ***Encode*** provides all 124 encodings of the upstream distribution, up from 21: the single-byte tables generated from the Unicode mapping data, UTF-16, UTF-32 and UCS-2, the 21 multi-byte CJK encodings, ISO-2022, gsm0338, hz, UTF-7 and the MIME encoded-word codecs. Each was checked against perl5's Encode, the CJK tables over every two-byte sequence. Names resolve through upstream's alias rules, so `windows-1252`, `latin2`, `sjis` or `macintosh` work without a perl5 installation, and `Encode->encodings` lists the loaded encodings until `":all"` is asked for.
- ***Encode*** **error handling**: the `CHECK` argument of `encode`, `decode` and `from_to` is honoured. `FB_CROAK` dies, `FB_PERLQQ`, `FB_HTMLCREF` and `FB_XMLCREF` substitute escapes, and `FB_QUIET` and `FB_WARN` return what was converted and leave the rest in the source variable, as on perl5; before, every mode behaved as the default. The UTF-16 and UTF-32 encodings report surrogates, noncharacters and partial characters with perl5's messages.
- **`:encoding` layers** no longer replace a character that straddles a read buffer boundary with U+FFFD; a 6 KiB file of three-byte characters lost characters at three places. `:encoding(UTF-16LE)`, `:encoding(koi8-r)` and the other new encodings work for reading and writing, `$PerlIO::encoding::fallback` is reset on the layer's first use as on perl5, and `require PerlIO::encoding` at run time no longer dies with "Out of memory during stack extend".
- ***Storable*** writes and reads perl5's format byte for byte: an image stored by perl5 retrieves on pperl and the other way round, and with `$Storable::canonical` both write identical bytes. Hooks (`STORABLE_freeze`, `STORABLE_thaw`, `STORABLE_attach`), code references, regexps, tied and restricted hashes, weak references and images written by Storable versions before 0.7 are handled, and the module's Perl half is upstream's `Storable.pm`. Storable's own test suite passes all 2,728 assertions it passes on perl5. Freezing a tied hash stored none of its keys; it now stores the tie object, as perl5 does.
- **Thawing untrusted data**: a *Storable* image whose class name was crafted Perl code ran that code when it was thawed; class methods are now looked up without compiling anything. A truncated image returns undef without setting `$@`, a corrupt one croaks with perl5's message, and an image that blesses a file name into `CGITempFile` warns as perl5 does (CVE-2015-1592).
- ***POSIX***: `use POSIX;` imports POSIX's default export list; it imported nothing, so `floor`, `INT_MAX` or `WIFEXITED` had to be fully qualified, and Test::TCP did not install. The functions perl5 implements in Perl, such as `POSIX::localtime` and `POSIX::stat`, exist, C-only names such as `POSIX::malloc` croak with perl5's advice, `use POSIX qw(locale_h)` works, and a call with the wrong number of arguments croaks with perl5's usage message where `POSIX::abort(1,2,3)` aborted and `POSIX::_exit(1,2,3)` exited. perl5's own POSIX test suite gives the same results on pperl, all 4,514 assertions.
- ***POSIX*** **time functions**: `POSIX::strftime` derives the weekday, the day of the year, daylight saving time and the UTC offset from the date instead of trusting optional arguments; called with six fields, `%a`, `%j`, `%V`, `%z` and `%Z` were wrong. POSIX::strftime::Compiler, which Plack and Starman depend on, now installs with `--cpan`. `strftime` no longer cuts its result at 1024 bytes, `asctime` accepts six arguments, `ctime` honours `$ENV{TZ}`, and `difftime` truncates its arguments to integers as perl5 does.

- ***POSIX* signals**: `POSIX::SigAction` and `POSIX::SigRt` exist, so `sigaction` works in its documented form, with a code reference as handler, a signal name such as `"INT"`, and the `SAFE` flag as perl5 reports it. `sigaction` given an array or scalar reference instead of a `POSIX::SigAction` crashed the interpreter, and one that croaked left signals blocked; both now behave as on perl5. `POSIX::SigSet` objects check their arguments, `addset(-1)` croaks, and can be subclassed.
- ***POSIX* constants**: the 16 terminal speed constants such as `B9600` had wrong values (13 instead of 9600), and `FP_NAN`, `FP_INFINITE`, `FP_ZERO` and `FP_NORMAL` were permuted, so `fpclassify(0.0)` returned 4 instead of 2. `ULONG_MAX` is exact, `MB_CUR_MAX` follows the locale, `FLT_ROUNDS` follows `fesetround`, and `ARG_MAX`, `LDBL_EPSILON`, `LDBL_MAX` and `LDBL_MIN` croak as on perl5 instead of returning approximations. POSIX gained 104 constants and re-exports Fcntl's record-locking constants `F_GETLK`, `F_SETLK` and `F_SETLKW`.
- ***POSIX* locale**: under a locale such as `de_DE.UTF-8`, `POSIX::strtod("3,5")` returns 3.5, `localeconv()` reports the locale's separators, and `setlocale(LC_ALL)` returns the locale's name, as on perl5. `localeconv` returns all 24 keys, with -1 for fields the locale leaves unspecified, and `mblen`, `mbtowc` and `wctomb` answer as perl5's do at the edges, `mblen("")` being 0.
- ***POSIX* arguments and results**: POSIX functions read arguments that come from a regex capture or a tied variable; `POSIX::floor($1)` returned 0. Functions perl5 declares as returning a system result, such as `sysconf`, `mkfifo` and `setuid`, give `undef` on failure and "0 but true" for zero, so `defined(mkfifo(...)) or die` works. `strtol` and `strtoul` accept the `0b` prefix, `strtoul` returns values above the signed range, `strtod` in scalar context returns the number, and the `POSIX::Termios` speed setters accept a baud rate.
- **DBD::SQLite** is a new driver over the system's libsqlite3. It passes 98 of the 103 test files of DBD::SQLite 1.78's own suite and supports user-defined functions, aggregates and collations, the commit, rollback, update and authorizer hooks, online backup, the catalogue methods and the fts3 Perl tokenizer. Values bind and fetch with the storage class and bytes upstream's driver gives them, strings with embedded NUL bytes included, and SQL that issues its own `BEGIN` works.
- **DBD::Pg** supports PostgreSQL arrays in both directions, `COPY` through `pg_putcopydata` and `pg_getcopydata` as well as `pg_putline` and `pg_getline`, savepoints, asynchronous queries (`pg_async`, `pg_ready`, `pg_result`, `pg_cancel`), `cancel`, `bind_param_inout` and `pg_enable_utf8`. It exports the `PG_*` type constants, where `use DBD::Pg ':pg_types'` died, and reports `PRECISION`, `SCALE`, `NULLABLE`, `ParamValues`, `ParamTypes`, `pg_lib_version` and the type names in `pg_type` as upstream's driver does.
- **DBD::Pg placeholders and quoting**: `quote` honours its type argument, so `quote(3.14, SQL_FLOAT)` is unquoted and an invalid float dies, and identifiers that are reserved words are quoted. A placeholder written `\?` stays a literal `?` operator, `:1` and binding by name (`bind_param('$1', ...)`) work, `pg_placeholder_dollaronly` and `pg_placeholder_nocolons` are honoured, and an array slice `arr[1:2]` is not taken for a placeholder. `do` without bind values sends the statement unchanged, statements such as `SET search_path TO ?` run with placeholders, and only `bytea` parameters are sent in binary form; an integer parameter beside a `bytea` one made the server reject the statement.
- **DBD::MariaDB** is a new driver over libmariadb, and DBD::mysql is now

a compatibility layer over it that keeps the `dbi:mysql:` names and the `mysql_*` attributes. Both gained asynchronous queries (`async => 1`, `mysql_async_ready`, `mysql_async_result`, `mysql_fd`), `table_info`, `column_info`, `primary_key_info`, `foreign_key_info` and `statistics_info`, the full `get_info` table, `ChopBlanks`, implicit reconnect and the numeric `mysql_serverversion` and `mysql_clientversion`. Placeholders inside comments and after `LIMIT` are handled as upstream's driver handles them.

- ***DBI***: with `InactiveDestroy` set, DBD::Pg and DBD::MariaDB leave the connection open when a handle is destroyed, so a forked child no longer closes its parent's connection. A true `Taint` attribute no longer croaks, `NAME_hash` is defined for a statement without columns, and the drivers' `$VERSION` strings use upstream's `_01` form instead of a `-pperl1` suffix that version checks could not parse.
- ***Peta::FFI***: `Peta::FFI::closure` turns a Perl code reference into a C function pointer that a C library can call, and `closure_free` releases it. A `die` in the Perl code does not unwind through the C caller.
- ***HTTP::Parser::XS*** is native, with the same results as the 0.17 module on 26,000 differential cases; it is the parser Plack, Starman and Twiggy use. On input where scanning dominates it is 6.0 times as fast as the module on perl5, and as fast on a typical ten-header request.
- ***Time::Piece***: `use Time::Piece` works, where it died with a dynamic-loading error. Time::Piece 1.41's Perl modules run on a native implementation of its compiled part, and the distribution's own 15 test files pass, 1,142 assertions.
- ***Term::ReadLine::Gnu*** provides all 160 functions of the 1.47 module's compiled part and runs the distribution's own `Gnu.pm`, embedded in pperl rather than taken from a perl5 installation. Perl callbacks such as `completion_entry_function` and those added with `add_defun` are called; before, they were stored and ignored.
- **`glob`**: the built-in `glob` and `<*.c>` use *File::Glob*, as on perl5, instead of starting a shell for every call. Results no longer depend on the locale's collation order, `$File::Glob::DEFAULT_FLAGS` is honoured, and 200 globs take about 10 ms instead of 520 ms. File::Glob's constants have perl5's values, `bsd_glob` sorts names differing only in case as perl5 does, and `use File::Glob ':bsd_glob'` replaces `glob`.
- **Socket** has the 118 constants and seven functions it lacked, among them `pack_ip_mreq` and its relatives, without which joining a multicast group was impossible, and `sockaddr_un`. `unpack_sockaddr_un` validates its argument, the deprecated `AI_IDN_*` and `NI_IDN_*` constants croak as on perl5, and the `:crlf` tag exports `$CR`, `$LF` and `$CRLF`.
- ***Fcntl*** has the record-locking constants `F_GETLK`, `F_SETLK`, `F_SETLKW`, `F_RDLCK`, `F_WRLCK` and `F_UNLCK`, and 48 more. A plain `use Fcntl` exports upstream's 68 names, `:Fcompat` and `:mode` hold the right names, `S_ENFMT` and `S_ISTXT` are absent as on perl5 on Linux, and the `S_IS*` functions return perl5's true and false values.
- ***File::Temp***: `unlink0` compares the file handle with the path before deleting, so a file swapped in after creation is no longer removed, and the `MEDIUM` and `HIGH` safety levels are enforced instead of being treated as `STANDARD`. A File::Temp object keeps its file name and unlink flag in its glob, so `${*$fh}` gives the name, and `File::Temp::cleanup()` no longer deletes the file of an object that is still alive.

- **Imports from native modules**: overwriting a sub imported from a native module, as *Net::Ping* does with Socket's `SO_BINDTODEVICE`, freed the module's own sub and could crash the program later; imports now hold a reference, as on perl5. Import lists accept `/pattern/` entries (`use POSIX qw(/^S_I/)`) and variables, and twelve native modules, among them Socket, *Cwd* and *Scalar::Util*, report perl 5.44's module versions, so `use Socket 2.041` works.
- **B** has every method perl5's B has, among them the `B::REGEXP` methods *B::Deparse* needs for a sub that uses a regexp, and `B::COP::label`, without which deparsed code lost its loop labels. `B::OP->size` returns the op's size in bytes instead of its class number, `B::hash` and `B::cstring` give perl5's results, and four list-returning methods such as `B::AV::ARRAY` no longer write past the stack and corrupt memory. B's own test suite passes 535 of the 539 assertions it passes on perl5.
- **mro**: an `@ISA` chain deeper than 100 classes no longer dies with "Recursive inheritance detected"; linearisations are cached as in perl5, and every depth measured, up to 500, behaves as on perl5. The C3 "Inconsistent hierarchy" error carries perl5's merge trace, `mro::get_linear_isa($class, 'dfs')` honours its second argument and croaks on an unknown name, and a failing `next::method` names the method and the class.
- **Config**: `%Config` has the `extensions` and `static_ext` keys, listing the native modules the binary contains, and answers 174 of perl5's keys, up from 107. Test suites that check `$Config{extensions}` before running, as perl5's own do, no longer skip themselves, and `Config::config_re` matches as perl5's does.
- ***Devel::Peek***: `DumpProg` and `runops_debug` work, `use Devel::Peek ':opd=stP'` sets the debugging flags, and the module's exports and variables, such as `$D_flags` and `%EXPORT_TAGS`, match perl5's.
- ***IO::Handle***: `ungetc` accepts every code point perl5 accepts, including those above U+10FFFF, and treats negative and very large arguments as perl5 does; `setbuf` croaks only on output handles.
- ***Data::Dumper***, ***List::Util***, ***Time::HiRes***: `print Dumper($x)` printed its arguments before the dump, and `Time::HiRes::stat` in scalar context returned a wrong value; both behave as on perl5. `List::Util::shuffle` and `sample` follow `srand`, so `srand(42)` gives perl5's permutation; `shuffle` returned the same order on every call, and `sample` could return its count argument.
- **Other native modules**: `Cwd::fast_realpath` exists, `Hash::Util::hash_seed` returns the full 32-byte seed, and `Hash::Util::bucket_ratio` of an empty hash is 0. SDBM_File's methods called with too few arguments croak with a usage message instead of crashing, and `attributes::get` finds the handler of a glob aliased from another package.

6.16.4 Command line

- **Programs on standard input**: pperl reads the program from standard input the three ways perl5 does, `pperl < prog.pl`, `pperl - < prog.pl` and `pperl /dev/stdin < prog.pl`, with `$0` set as perl5 sets it. All three were refused.
- **`--cpan`**: `pperl --cpan MODULE...` runs cpanm under pperl, installing into the library tree pperl searches, and `--cpan --system` installs into `/usr/local/share/pperl`. When building a distribution's C part fails, it installs the pure-Perl

files, which many such distributions fall back to, and names each distribution it did this for; Params::Validate installs and runs this way. Modules installed into `~/.pperl` (or `$PPERL_HOME`) are found without setting `PPERL5LIB`.

- **Command-line switches**: `--` ends switch processing and is removed from the arguments, as on perl5; pperl listed it in its usage text but rejected it. Several `-e` or `-E` switches form one program, joined by newlines, where only the last one used to run, and the Makefiles *ExtUtils::MakeMaker* writes rely on both. With `-e`, a program argument that looks like a switch reaches `@ARGV`.
- **`-I` in option clusters**: `pperl -wIlib` is `-w` plus `-Ilib`, as on perl5; it died with "Unknown option: -I".

6.16.5 Documentation

- **Translations**: the catalogues of all 41 translation languages were brought up to date with the English pages for the first time since April, and the pages that had none, among them the CPAN tutorial, *HTTP::Parser::XS* and *Time::Piece*, are translated in every language. Slovak gained the complete perlfunc reference, and Hebrew and Greek the module reference, which raised them from 79.3% and 76.2% to 93.0% and 92.9%.
- **Translation fixes**: 665 translated headings in all 41 languages that begin with a number, a dash, # or >, among them the numbered headings of the licence page, were displayed in English and now appear translated. Links to the perlfunc categories carry the translated category name, about 9,500 labels, and the line "Fully compatible with upstream Perl 5.44." is translated again on the 217 perlfunc pages of 40 languages where it had fallen back to English.
- ***CPAN tutorial***: a new tutorial covers installing CPAN modules with `pperl --cpan`, the modules already built into pperl, how modules that need a C compiler are handled, and troubleshooting.
- **Module reference**: 14 module descriptions on the index pages that were cut off mid-sentence, among them *HTTP::Parser::XS*, are complete, and the pages for *IO::Handle*, *Time::HiRes*, *PerlIO::mmap* and *Data::Dumper* show descriptions instead of implementation notes. `List::Util::shuffle` has its own page.
- **perlfunc reference**: factual errors were corrected on the pages for `abs`, `semop`, `shmget`, `send`, `crypt`, `getnetbyaddr`, `getservbyport`, `qq` and `exists`; `shmget` now states that a key of 0 is `IPC_PRIVATE`.
- ***How-to page***: guides and tutorials are separate sections of cards, and the entry for programmers coming from other languages names every guide that exists, PHP, Ruby, Lua, sed, awk, shell and Tcl included.

6.17 pperl 0.6.8

2026-07-27 to 2026-07-28, 3 commits.

- **`--cpan`** installs pure-Perl CPAN distributions for pperl. Modules built into pperl count as installed at pperl's own version, not at the host perl's, so they are never rebuilt.

- ***ExtUtils::MakeMaker***: a distribution's `Makefile.PL` runs and writes the same Makefile as on perl5. `B::svref_2object(\$v)->MAGIC` died for lack of a `B::MAGIC` class, which stopped every `Makefile.PL`; `B::MAGIC` now exists with its accessors. `$Config{eunicefix}` is `:` as on perl5, so MakeMaker no longer executes the Makefile it has just written.
- **The daemon**: `pperl --via-daemon script.pl` runs the script from its cached compiled image, as `PPERL_DAEMON=2` does; the reduced mode that a bare `--via-daemon` used to select is now `--via-daemon=trunk`, and any other value is refused with the accepted forms. `--via-daemon=script` and `PPERL_DAEMON=2`, documented as the same request, kept separate caches, so a script run both ways was never served from its cached image again; they now share one.
- **Safe**: a subroutine signature inside a `Safe` compartment was denied, because `Opcode` took its op tags from the host perl's `Opcode.pm`, which lacks the signature ops of perl 5.44. The tag table is now part of pperl, and the host perl's `Opcode.pm` is no longer loaded.
- ***Encode***: an encoding alias registered at run time works with `encode` and `decode`, not only with `resolve_alias`.
- ***PDL***: slicing a slice no longer checks bounds against an unset shape, `sever` detaches an ndarray from its parent, and operations with a scalar on a `flowing` ndarray no longer leave empty results or crash.
- ***Daemon guide***: the chapter "Fast start-up: the daemon" gains a worked example (about 24 ms per cold run against about 4 ms from the cached image, on the machine it was written on), how to check whether a run was served, how to opt in from a shebang line or a shell function, a systemd user unit, and an exact description of `--daemon-idle`.

6.18 pperl 0.6.6

2026-07-26 to 2026-07-27, 12 commits.

6.18.1 Modules

- ***Compress::Raw::Zlib*** is native and uses the system's zlib, so `Compress::Zlib`, the pure-Perl `IO::Compress::Gzip` and `DBM_Filter::compress` work. Its version, `ZLIB_VERNUM`, compressed lengths, CRC-32 and Adler-32 checksums and status codes match perl5's. bzip2 is not covered.
- **Import lists**: `use` of a native module accepts Exporter's full grammar: `!name` and `!:tag` remove names, a list that begins with a removal starts from the default exports, and `:DEFAULT` is a tag. `use Compress::Raw::Zlib qw(!crc32 !adler32)`, as *IO::Compress* writes it, imported nothing and left the module unusable. Unknown names are reported together in one error, as *Exporter* does.
- ***Storable*** no longer warns "Attempt to free unreferenced scalar" when a stored structure refers to the same value more than once. Weak references come back from `thaw` and `dclone` as weak references, without the "panic: del_backref" error.
- ***PDL***: assigning through a slice, as in `$a->slice("1:3") .= pdl(10,20,30)`, works; it died with "dimension mismatch" or set the parent's elements to zero.

- **`use re`**: `use re 'taint'`, `use re 'eval'` and `use re '/aa'` take effect; they were silently ignored.
- ***POSIX***, **Socket and** ***Encode*** gain export tags: POSIX `:math_h`, Socket `:crlf`, `:addrinfo` and `:all`, Encode `:all`, `:default`, `:fallbacks` and `:fallback_all`. POSIX also has `strxfrm`.
- ***Devel::Peek***: `SvREFCNT($x)` works as documented; every call died with a usage message.
- ***Hash::Util***: `hash_traversal_mask` reads and sets a hash's traversal mask, and returns `undef` for a hash that has none, as on perl5; it always returned 0.

6.18.2 Command line

- **The daemon** can keep a compiled image of a script: `--via-daemon=script` or `PPERL_DAEMON=2` runs the script from it, so a repeated run skips compiling the script and the modules it loads as well as starting the interpreter. `--daemon-max-scripts=N` limits how many scripts are kept (100 by default; the least recently used goes first), and `--daemon-reset` drops one script's image, or with no path all of them, and exits.
- **`--interactive`**, in builds that offer it, says on standard error that it is not implemented and exits with status 2. It printed a banner and reported success without running anything.

6.18.3 Distribution

- **Benchmark suite**: pperl's benchmark suite ships as a separate archive with its documentation and runs outside a source tree. It finds pperl through `--pperl=PATH`, `$PPERL`, a `pperl` beside it or on the `PATH`, and `--display` shows saved results without a pperl binary.

6.19 pperl 0.6.5

2026-07-23 to 2026-07-26, 16 commits.

6.19.1 JIT and parallelization

- **`next` and `last` in compiled loops**: a `while` loop whose body uses `next`, such as the statement parser of DBD::mysql, crashed the JIT compiler and left the JIT disabled for the rest of the run; such loops now compile, as do `foreach` loops with `next`. A `last` inside `do {...} while` leaves the enclosing loop, as in perl, where the compiled code ended only the `do` block.

6.19.2 Modules

- **DBD::Pg and DBD::mysql** are part of pperl, written in Perl over PostgreSQL's libpq and the MySQL client library, so `DBI->connect("dbi:Pg:...")` and `dbi:mysql:` work with no driver to build. DBD::Pg takes `?`, `$1` and `:name` placeholders and binary `bytea` parameters (`{pg_type => 17}`) and reports the SQLSTATE;

DBD::mysql fetches binary data intact. Both are tested against live PostgreSQL and MariaDB servers.

- **DBD::Sybase** is included, written in Perl over FreeTDS's ct-lib: it connects by server name or by `host=` and `port=`, runs queries with ? placeholders and transactions, and reports errors through `err`, `errstr` and `state`. It has not yet been tested against a live server.
- ***Peta::FFI*** 0.02 passes C pointers as opaque values (signature type `o`, with `undef` as NULL) and reads, writes, allocates and frees C memory with `peek`, `peek_cstr`, `poke`, `alloc`, `free`, `pack_ptr` and `unpack_ptr`. That is enough to drive a C library such as SQLite from pure Perl.
- ***DBI***: in a pperl built without database support, `use DBI` stops with a message naming the missing capability instead of loading the host perl's DBI and failing inside *XSLoader*.
- **Magical arguments to native modules** are read correctly: `md5_hex($1)`, `sha1_hex(substr(...))`, `encode_base64(substr(...))` and `$md5->add(substr($d, $o, $n))` all processed an empty string.
- **Default imports**: `use Socket;` imports Socket's default list again, and *File::Path* no longer dies calling `getcwd`; eleven native modules had an `import` that did nothing. A native module's own `import` is no longer replaced by the generic one.
- ***Time::HiRes***: importing `stat`, `sleep` or `time` from Time::HiRes overrides the builtin only in the importing package, as on perl5. It overrode it for the whole program, which broke `stat(_)` everywhere else; *File::Copy* imports it, and `require CPAN` died with `Bareword "_" not allowed while "strict subs" in use`.
- ***File::Temp*** exports what perl5's does, with the `:POSIX`, `:mktemp` and `:seekable` tags and the `STANDARD`, `MEDIUM`, `HIGH` and `SEEK_*` constants, and its objects can `seek`. `tempfile()` without `DIR` creates the file in the current directory instead of dying, and a forked child that exits no longer removes the parent's temporary files.
- ***Storable*** keeps booleans: a boolean value (perl 5.36 and later) comes back from `freeze` and `thaw` or `dclone` as a boolean.
- ***Opcode*** is native, providing the operator masks that `Safe` compartments are built from.
- ***IPC::SysV*** is native: its constants, `ftok`, `shmat`, `shmdt`, `memread` and `memwrite`, and what the `stat` methods of *IPC::Msg*, *IPC::Semaphore* and *IPC::SharedMem* need.
- **Warnings when loading modules**: loading `re`, `mro`, `version`, `Encode`, `Math::GMP`, `Config`, `attributes` or `Time::HiRes` no longer warns that a subroutine was redefined, and `use Config`, `B`, `Storable` and `PerlIO::encoding` no longer cause "used only once: possible typo" warnings that perl5 does not give. `version` objects use perl's built-in implementation for all their methods, as on perl5, instead of a mix of two.
- ***Scalar::Util***: `readonly` returns the same value as on perl5, the flag word (such as 134283264) rather than 1. `readonly`, `tainted` and `dualvar` are faster; a benchmark of `readonly` went from 0.66 to about 0.95 times perl5's speed.
- **Config**: `$Config::VERSION` is set and equals `$]`, as on perl5, and the `d_pw*` keys that *User::pwent* requires are present, so `User::pwent` loads.

- ***POSIX***: `strtol` and `strtoul` with one argument use base 0 instead of warning `Argument "*main::strtol" isn't numeric`, and `POSIX::sprintf` no longer warns "Use of uninitialized value in subroutine entry".

6.19.3 Command line

- **The daemon**: `pperl --daemon` runs a background daemon that holds a ready interpreter (`--daemon=start`, `stop` and `status` manage it), and `pperl --via-daemon script.pl` or `PPERL_DAEMON=1` runs a script in a process forked from it, with the caller's standard streams and working directory. When no daemon answers, the script runs normally. `--build-id` prints the binary identity the daemon is keyed on. On its own this measured about 0.6 ms slower than a normal start, so it is opt-in.

6.19.4 Documentation

- **Translations**: the "*Graphics with Perl*" guide is translated into the remaining 31 languages and so into all 41. In the Arabic, Persian and Hebrew translations, 72 arrows that pointed the wrong way are corrected.
- ***Daemon guide***: a new chapter of the performance guide, "Fast start-up: the daemon", describes the daemon, in English and in all 41 translations.
- **perl 5.44 in the documentation**: the introduction, the page of differences from Perl 5, the coming-from guides, the errata and the perlfunc pages state compatibility with perl 5.44 instead of 5.42.
- **Getting started and errata**: Getting started lists `-E`, `-I` and `-V`, drops `--cache`, and describes the module search path as it behaves: `-I` works alongside `PERL5LIB` and `PPERL5LIB`. The errata page on identifier length gives perl 5.44's limit of 1019 bytes, which pperl matches, and withdraws the claim that pperl differs above 251 bytes.

6.20 pperl 0.6.3

2026-07-18 to 2026-07-23, 35 commits.

6.20.1 Language and runtime

- **Start-up**: pperl starts no worker threads until a loop first runs in parallel, so a program that never does starts none; creating them took about 1 ms of every start. In the benchmark suite's start-up rows pperl went from 1.277 to 1.136 times perl5's start-up time (geometric mean).

6.20.2 JIT and parallelization

- **Subroutines called from compiled loops** compile together with the loop when their body contains loops of its own (`while`, C-style `for`, ranges, nested loops) or reads a lexical variable of the enclosing scope, a closure.

- **Array references in compiled loops**: `for (@$A)`, `$A->[$i][$j]` and loop bounds such as `scalar(@$A)` compile, so matrix code written over arrays of arrays, such as a matrix difference or a matrix product's triple loop, compiles as one nest of loops.
- **Hashes in compiled loops**: hash elements with computed keys, such as `$h{$i & 1023}`, can be read and written in a compiled loop, and `%h = (key => value, ...)` with constant keys and numeric values compiles too.
- **Strings in compiled loops**: `substr`, `uc`, `lc`, `ucfirst`, `lcfirst` and `reverse` compile as parts of larger expressions, as in `length(substr($s, 0, $n))`, and so do string comparisons and concatenations of array elements. Loops whose condition is a pattern match without `/g` or a `length`, such as `while ($s =~ /x/)` or `while (length $s)`, compile.
- **Nested loops**: C-style `for (;;)` loops nested in other loops compile. Whether a loop is worth compiling is judged by the work of its body, inner loops included, so `for (0..10) { for (0..9999) {...} }` compiles although its outer loop runs only 11 times.
- **`do {...} while` and `do {...} until`** loops compile.
- **`map` and `grep`** statements of the form `@dst = map {...} @src` compile on their own, outside any loop, when the source array is large enough to repay the compilation.
- **More loops compile**: whether a loop compiled could depend on the code around it, for example on a preceding statement with a `?:` conditional; such loops now compile. So do loops that use the result of a comparison as a number, as in `$n += ($x > $y)` or `$a[$i == 0]`.
- **Parallel reductions over arrays**: a loop that adds up values read from arrays it does not write, such as `$sum += $a[$i]`, runs in parallel on several cores.
- **JIT compilation cost**: finding and compiling a program's loops takes less time. A program of 600 short loops ran in 0.371 s instead of 0.577 s, and the search for loops now grows with the size of the program rather than with its square.

6.20.3 Modules

- ***MIME::Base64***: `encode_base64` encodes inputs of 1 MiB and more on several cores, with output identical to the serial encoder; on 32 MiB it is 2.85 times as fast as perl5's MIME::Base64.
- ***Digest::SHA***: methods that return their object, such as `add`, kept an extra reference to it, so a Digest::SHA object was never freed.
- ***POSIX***: `POSIX::read` accepts a reference to its buffer as well as the buffer itself, as on perl5, and the data it reads is tainted under `-T`.
- ***PDL***: `lu_backsub` given its right-hand side as a plain Perl value no longer reads freed memory.

6.20.4 Command line

- **Version reporting**: `pperl -v` and `pperl -V` name perl 5.44 as the version pperl is compatible with, and `$Config{version}` is 5.44.99, matching `$^V`; all three said 5.42.0.

6.20.5 Documentation

- **Translations**: the "*Graphics with Perl*" guide is translated into German, French, Italian, Czech, Greek, Danish, Hebrew, Ukrainian, Croatian and Spanish.

6.21 pperl 0.6.2

2026-07-16 to 2026-07-18, 35 commits.

6.21.1 Language and runtime

- **Pragmas built into the binary**: `strict`, `warnings`, `feature`, `builtin`, `lib`, `0` and `DynaLoader` are part of the pperl binary, in the versions that match its interpreter, instead of being read from disk. pperl looked for its own `warnings.pm` next to the binary, and where it did not find it, `no warnings 'experimental::class'` could switch off the wrong warning.
- **`PPERL5LIB`** works like `PERL5LIB` for pperl alone and is searched before it. The trees `/usr/local/share/pperl/site_perl`, `/usr/share/pperl/site_perl` and `/usr/share/pperl/vendor_perl` are searched before the host perl's library trees, so a module can be installed for pperl without touching the trees perl5 uses. Neither can replace a module built into pperl.

6.21.2 JIT and parallelization

- **Grids in compiled loops**: reads and stores through nested array references, such as `$grid->[$y][$x]` and `$a[$i][$j]`, compile, at any depth from two to eight levels, and mix with plain array and hash elements in the same loop. A rectangular grid of numbers is copied into a flat buffer once when the loop starts. An output grid that the loop fills from an empty `[]` is built as perl5 builds it, with undefined rows and short rows left as they are. A Game of Life program runs 10.4 times as fast as with `--no-jit`.
- **Conditions and loop control in compiled loops**: `next`, and `next if` and `last if` with compound conditions such as `A || B`, `A && B` or `!$a && !$b`, compile inside loop bodies, as do `foreach` loops over integer ranges nested to any depth, the conditional operator `?:` used as a value, and `&&` and `||` over plain values.
- **Nested `for my` loops**: a `for my $j (...)` loop inside another `for my $i (...)` loop compiled nothing; such nests now compile. A matrix difference written as `$C[$i][$j] = $A->[$i][$j] - $B->[$i][$j]` over two nested loops, as in Math::Matrix, runs compiled.
- **Loops the JIT cannot speed up** no longer make a program slower than `--no-jit`. A loop whose compiled runs keep being abandoned, a loop of fewer than 64 iterations that is entered again and again, and a parallel attempt that keeps failing are

given up after a few tries instead of paying their set-up cost on every entry. Loops the JIT does speed up are unaffected.

- **Parallel grid loops**: a loop that reads one grid and writes each cell of another grid once runs on several cores, with the same output for any thread count. Input grids are shared between the threads instead of being copied for each of them, and the work is cut into fewer pieces when copying the output grid would cost more than the work. A grid map ran 1.51 times as fast on four threads as on one.
- **Parallelization threshold**: whether a loop runs in parallel depends on an estimate of its total work, its iterations times the measured cost of its body, instead of a fixed minimum number of iterations. A nested loop of 501 outer iterations runs in parallel, one of 101 stays sequential.
- **`blessed` in compiled loops**: a compiled loop calling *Scalar::Util*'s `blessed` leaked memory on every iteration until the system killed the process. 50 million calls used 12.7 GB; they now use 31.6 MB.

6.21.3 Modules

- **B**: `B::HV::ARRAY` and the `B::GV` methods `FORM`, `IO` and `CVGEN` exist, so code that walks a package's symbol table through B works.
- ***Digest::SHA***: `sha256`, `sha1` and `sha512` on short inputs run 1.9 to 3.0 times as fast as on perl5, where they ran at about half its speed, and the object interface 1.96 times as fast.
- ***MIME::Base64*** **and** ***MIME::QuotedPrint***: `MIME::Base64::encode` and `decode`, and `MIME::QuotedPrint::encode` and `decode`, exist as on perl5, as aliases that are not exported. Encoding or decoding a string with characters above 255 dies with "Wide character in subroutine entry", as on perl5.
- ***Storable***: `freeze` of an array runs at 1.05 times perl5's speed, where it ran at 0.52.
- ***Scalar::Util*** loads *List::Util*, as it does on perl5. Code that calls `List::Util::first { ... } @list` with only Scalar::Util loaded, as *DBD::File* does, died with "Can't call method first".

6.21.4 Command line

- **`-M` and `-m`** load a module before the program runs, as on perl5, with import arguments as in `-Mconstant=ANSWER,42`; pperl rejected both switches before. `pperl -MO=Deparse` prints the program back as source and `pperl -MO=Concise` prints its op tree.
- **`--threads`** accepts `phys` (all physical cores), `log` (all logical cores, hyperthreads included), `phys-N` and `log-N` (N fewer, at least one), `P%` of the logical cores, or a number, which is taken as given even beyond the number of cores. Without `--threads`, pperl uses all physical cores; it used all logical cores before.

6.21.5 Documentation

- **Differences from Perl 5**: the page is rewritten to list the deliberate differences that remain (no loading of compiled extensions, the module search path, no interpreter threads, Linux only, the version pperl reports) and the current known issues. Limitations that no longer exist are removed from it.

6.22 pperl 0.6.1

2026-07-16, 1 commit.

- **`@INC`** is built by probing the host's library trees in the Arch, Debian and Fedora layouts, instead of holding one distribution's fixed paths. A directory from `-I` or `PERL5LIB` that holds versioned subdirectories, as `local::lib` and Carton trees built by perl5 do, has them added, nearest perl version first; architecture subdirectories are never added. A module built into pperl is always loaded in preference to a `.pm` file of the same name on disk.
- **`-E`, `-V` and `-V:key`**: `-E` enables `say` and the other optional features, as on perl5; it ran as `-e` before. `-V` prints perl5's configuration summary with the real `@INC`, including the effect of `-I` and `PERL5LIB`, and `-V:key` prints one configuration value, as in `-V:archname`.
- ***Peta::XS*** is a new module that lets pure Perl read facts about a value, such as its reference count, and attach magic: Perl callbacks that run when a variable is read, written or freed, as Variable::Magic offers on perl5. A program asks for its capabilities by name at `use` time, and one pperl does not provide is refused at compile time.

6.23 pperl 0.6.0

2026-07-11 to 2026-07-15, 54 commits.

6.23.1 Language and runtime

- **The interpreter is faster**: with the JIT off, pperl averages 108.6 on the perlbench suite, where perl5 scores 100, up from 85. Every benchmark of the suite scores at least 84, and the `while` loop and sub call benchmarks run faster than on perl5.

6.23.2 JIT and parallelization

- **The JIT compiler and auto-parallelization** work in pperl's default runtime; in 0.5.6 they worked only under `--pp`. Numeric `for`, `while` and `until` loops, nested loops and `if`/`else` in loop bodies compile to machine code, with integer arithmetic checked for overflow and floating-point expressions evaluated in perl's order, so results are those of the interpreter. A sum over 100 million integers takes 0.14 s against perl5's 2.76 s, and a 1000x1000 Mandelbrot render 214 ms against 12.5 s.
- **Parallel loops**: a counting loop whose iterations only add to sums runs on several cores, with the result of a sequential run. A floating-point sum is split across cores only when every partial sum is exact; other floating-point sums run compiled on one core. Loops too short to pay for the threads stay sequential: 100,000 iterations

for a flat loop, 512 for one with a nested loop. The 1000x1000 Mandelbrot render takes 51 ms in parallel.

- **Strings in compiled loops**: concatenation and `.=`, `sprintf` with `%d` formats, `length`, `index` with a constant string, string comparisons, `uc`, `lc`, `ucfirst`, `lcfirst`, `chomp`, `reverse`, `x` and `substr` compile. A loop calling `index` with a constant string makes 253 million calls per second, perl5 26 million.
- **Arrays and hashes in compiled loops**: array elements with constant or computed indexes, `push`, `pop`, list assignment, `$#a`, `foreach` over an array or over the default `$_`, `while (@a)` and hash elements compile, as do `defined`, `ref` and `++` or `--` used as a value.
- **`map`, `grep`, `sort`, `join` and `split` in compiled loops**: `map` and `grep` blocks that compute a value, `sort { $a <=> $b }` and its descending form, `join` over an array and `split` on a literal pattern compile. `map` takes 0.13 and `grep` 0.17 of the time perl5 takes.
- **Subroutine calls in compiled loops**: calls to named subs compile with the loop, including subs that take their arguments with `shift` or a signature, declare `my` variables, return early, or call themselves or each other recursively. A signature sub with optional parameters compiles once per argument count, and a branch that calls `die`, `warn`, `croak` or `confess` no longer keeps a sub from compiling. A recursive sub such as `fib` compiles on its own as well: `fib(30)` takes 17 ms instead of 538 ms, and the "Deep recursion" warning comes at the same depth as on perl5.
- **Regular expressions in compiled loops**: matches, captures and substitutions with a constant replacement, `s/PAT/TEXT/` and `s/PAT/TEXT/g`, compile and run perl's regex engine, so `$&`, `$1` and `@-` are set as in the interpreter. Patterns with `(?{ })`, `m//g`, `s///e` and `s///r` stay interpreted.
- ***Scalar::Util* and *List::Util* in compiled loops**: `blessed`, `reftype`, `refaddr`, `weaken`, `unweaken`, `isweak`, `looks_like_number` and `readonly` from Scalar::Util and `sum`, `sum0`, `product`, `min` and `max` from List::Util can be called in a compiled loop.

6.23.3 Modules

- ***B::Deparse* and *B::Concise* work**: the `B` module was revised against perl5's, so `B::Deparse` turns code references back into source and `B::Concise` prints op trees. Deparsing interpolated strings and chained element access such as `$h->{a}{b}` works, where it died. `B::CV::XSUB` and `B::CV::XSUBANY` exist, and loading `B` no longer warns about a missing `B::OBJECT` package.
- ***Data::Dumper*** supports `Freezer` and `Toaster`, `Deparse`, `Sortkeys` with a code reference, `Purity` and `Deepcopy`, a `Seen` hash given by the caller, globs and regexps, with perl5's rules for quoting numbers. With `Sortkeys` under `use locale`, keys sort in the locale's collation order. It runs at about perl5's speed.
- ***XSLoader***: `XSLoader::load` succeeds for every module built into pperl, as for a statically linked module on perl5. A module that loads its compiled part in an `eval` and falls back to pure Perl on failure no longer takes the slower fallback: *Data::Dumper* did, and ran 2 to 5 times slower.
- ***List::Util***: `first`, `any` and the other functions that take a block call it as perl5 does, without a full sub call per element; `first` and `any` took 6 to 7 times perl5's time and

take 1.2 times. `uniq` keeps `undef` as `undef`, `uniqint` returns integers and honours an overloaded `int`, and `uniqnum` keeps integers above 2**53 apart.

- ***Sub::Util***: calling `set_subname` again and again on the same sub no longer gets slower with every call or leaks memory.
- ***Scalar::Util***: `blessed`, `refaddr`, `reftype`, `weaken`, `unweaken` and `isweak` are the `builtin::` functions, as in Scalar::Util on perl5.
- **Function prototypes**: the functions of *Time::HiRes*, *Scalar::Util*, *MIME::Base64* and *MIME::QuotedPrint* carry the prototypes perl5 gives them. With Time::HiRes's `time` imported, `time - $t0` subtracts instead of calling `time(-$t0)`.
- ***File::Spec***: the Unix functions that are XS on perl5 are native in pperl; `canonpath` runs at perl5's speed, where it took 12.8 times as long.

6.23.4 Command line

- **The `--pp` runtime is removed.** pperl has one runtime; `--pp` and the options only it supported, `--cache`, `--flush`, `--daemon`, `--client`, `--stop`, `--optree`, `--from-json`, `--trace`, `--timeout`, `--dump-optree` and `--compare-bytecode`, are gone. `-c` checks syntax with the parser that runs the program, so its messages and exit codes are perl5's; it used the other runtime's parser before.

6.23.5 Distribution

- **Static and cross-architecture builds**: pperl builds as a static x86-64 binary against musl, and as static AArch64 and RISC-V binaries with the JIT and auto-parallelization. Release binaries come with SHA-256 sums, and each must pass a smoke test, run under emulation for AArch64 and RISC-V.

6.23.6 Documentation

- **One runtime in the documentation**: the "Runtimes" page, the reference pages of the `--pp` runtime's modules in every language and every mention of `--pp` are removed, and the reference section is titled "Reference".
- ***JIT guide***: the architecture guide's JIT chapter is rewritten for the compiler of this release, covering what it compiles, how it keeps results exact and how to watch it work. The pages on writing fast pperl code, on measuring and on concurrent execution are brought in line with it.
- **Module reference**: the pages of *Devel::Peek*, *Hash::Util*, *List::Util*, *Math::GMP*, the *PDL* modules, *Peta::FFI*, *SDBM_File* and mro are expanded and corrected, and their code examples are highlighted as Perl.
- **Serbian translation**: links in ten perlfunc pages showed control characters instead of their labels; they are repaired, and the Serbian PDF builds.

6.24 pperl 0.5.6

2026-04-18 to 2026-07-08, 226 commits.

6.24.1 Language and runtime

- **Exit status**: pperl exits with the status the program ends with, wrapped to 8 bits as in perl5. It exited with 0 whatever the program did, so `exit 3` or an uncaught `die` reported success to the shell, and a parent's `$?` read 0 for such a child.
- **Taint mode**: `-T` turns on taint mode and `-t` its warn-only form, as in perl5; both switches were ignored, and `-t` switched on pperl's execution tracing, which is now `--trace`. *Scalar::Util*'s `tainted` reports whether a value is tainted, where it always returned false.
- **Experimental warnings**: `no warnings 'experimental::class'` silences the warnings that `class`, `field` and `method` raise, as in perl5; they were printed regardless.

6.24.2 Modules

- ***Cairo*** is native: `use Cairo` provides the interface of the CPAN Cairo module 1.109 over the system's libcairo, with no distribution to install and nothing to compile. Contexts, surfaces, patterns, matrices, fonts, regions and paths are covered, including the PNG, PDF, PostScript and SVG stream callbacks, MIME data, the tied `Cairo::Path` view and glyph arrays. `Cairo::FtFontFace->create` is not available, because it needs Font::FreeType.
- ***File::Map*** 0.71 is native: `map_file`, `map_handle`, `map_anonymous`, `sys_map`, `remap`, `protect`, `sync`, `advise` and `unmap` map files and anonymous memory into Perl strings.
- **Startup** no longer sets up every native module: a native module is set up when the program first loads it. Setting all of them up took about 8 ms on every run, loaded libgmp, libreadline and three other libraries, and left modules the program never asked for in `%INC`.
- ***PDL***: `matmult` no longer reads memory it has already freed, `interpol` takes its arguments in upstream's order, and `lgamma` in scalar context returns the logarithm instead of the sign. `at` on an empty or null ndarray croaks "Position out of range" instead of returning undef, and arithmetic between a float ndarray and a fractional number gives a double result, as upstream. `setvaltobad` and `setbadtonan` work, `inplace` included, where they croaked.
- ***PDL* input and output**: ndarrays can be stored and retrieved with *Storable* through *PDL::IO::Storable*, `dclone` included. PDL::IO::FlexRaw no longer dies with "Operation '!=': no method found", `mapfraw` maps a file into an ndarray through *File::Map*, and `hdr` creates an empty header hash on first use, as upstream.
- ***Storable***: `dclone` copies through an object's `STORABLE_freeze` and `STORABLE_thaw` hooks, as perl5's does, and an object frozen by a hook is thawed into a scalar or an array again, where it always came back as a hash.
- ***File::Glob***: `bsd_glob` and `csh_glob` no longer corrupt memory and crash when a pattern matches more than about 128 names. `bsd_glob` sorts case-insensitively by default, as in perl5, and loading File::Glob no longer warns that `import` was redefined or that `DEFAULT_FLAGS` was used only once.
- **Long result lists**: *List::Util*'s `zip`, `mesh` and `unpairs` and their variants, *Hash::Util*'s `legal_keys`, `hidden_keys`, `bucket_info` and `bucket_stats`, `re::regnames`,

Data::Dumper and PDL's `list` no longer corrupt memory when they return more than about 128 values.

- **Imported functions**: a function imported from a native module, such as *List::Util*'s `first` in `first { ... } @list`, no longer draws a "used only once: possible typo" warning.
- ***File::Temp***: files and directories created with `CLEANUP => 1` are removed when the program ends; every such program left an empty directory behind in the temporary directory.
- ***POSIX* and Socket**: `use POSIX ':sys_wait_h', ':errno_h'` and POSIX's other export tags import their names, where only importing the names one by one worked, and Socket exports `SOMAXCONN`.

6.24.3 Command line

- **One-liner switches**: `-n`, `-p`, `-a`, `-F`, `-l`, `-0` and `-i` work as in perl5, where pperl rejected them; `-l` and `-0` take an optional octal value and `-i` edits files in place. pperl's interactive shell, which `-i` used to start, is reached with `--interactive` only.

6.24.4 Documentation

- **Documentation site**: the documentation moved to a new site with a PDF edition, a language picker, a license page, a page of command-line options taken from `pperl --help`, a glossary and an index. Every page links to its Markdown source, in the reader's language on translated pages. The module reference pages were rewritten from implementation notes into descriptions for Perl programmers.
- **Language reference**: every built-in function has its own page, 234 in all, indexed alphabetically and by category. perlop, perlvar, perldata and perlsub are split into topic pages, such as the operator families and the groups of special variables, and perldiag (the diagnostics pperl emits), perllocale, perlsec and a page on `BEGIN`, `END` and the other phase blocks are new. An errata chapter records where perl's own documentation is wrong or stale.
- ***Guides and tutorials***: new guides cover regular expressions in twelve chapters, among them a comparison with PCRE2, *POSIX* and RE2 syntax, as well as object-oriented Perl, debugging, concurrent execution, performance, writing tie classes, pperl's architecture and Perl one-liners. New tutorials cover `pack` and `unpack`, I/O streams, references, Unicode with a collection of recipes, boolean logic and interprocess communication.
- ***Coming from other languages***: a new series teaches Perl by comparison with Python, PHP, Ruby, Lua, sed, awk, the shell and Tcl, with every Perl example run on pperl.
- ***Graphics guide***: "Graphics with Perl" covers 2D graphics in twelve chapters: raster and vector foundations and the classic drawing algorithms in plain Perl, the toolkits from *Imager*, *GD* and ImageMagick to *Cairo*, SVG, charts and GUI toolkits, binding a C library through *Peta::FFI* with Cairo as the example, and two Prima projects, a drawing program and a 2D game.

- **Translations**: the documentation is translated into 41 languages, among them German, French, Spanish, Chinese, Japanese, Russian, Arabic and Hebrew, and right-to-left languages are laid out right to left. perlop, perlvar, perldata, perlsub, perldiag and the "Coming from" series are translated completely in all 41; text not yet translated reads in English. The language picker shows how much of each language is translated.

APPENDIX

A

DISCLAIMER & BUG REPORTS

A.1 PetaPerl is not upstream Perl

PetaPerl is an independent, from-scratch Perl 5 runtime written in Rust. It is **not** a build of the CPAN reference perl interpreter. Many modules you are used to from CPAN - `PDL`, `Scalar::Util`, `Storable`, `POSIX`, and dozens more - ship with PetaPerl as **native Rust implementations** built into the interpreter. No XS, no C compiler required, no CPAN install step.

This has one consequence that matters to every user:

Important

Any bug, missing feature, or behavioral difference you observe when using a module name that exists on CPAN (`PDL::Ufunc`, `Scalar::Util`, `IO::File`, ...) while running under PetaPerl is **our bug, not upstream's.**

Do **not** file these issues against the upstream CPAN project. Upstream maintainers are volunteers working on a codebase that is not the one you are running.

A.2 Bug reports

PetaPerl is a **research preview under high-velocity change**. A public bug-reporting mechanism is not yet in place.

Note

What we ask you to do today:

- **Do not file issues against upstream CPAN projects** for behaviour you observe in pperl's native re-implementations. Upstream maintainers work on a different codebase and can't fix ours. This is the one message this page exists to deliver.
- **Hold on to your findings.** A minimal reproducing script, the output of `pperl -V`, and a short description of expected vs. actual behaviour are the three things any future tracker will ask for. Saving them now means less work later.
- **Check whether it's already known.** Every module page carries a *Differences from upstream* section listing deliberate divergences. If your observation matches a bullet there, it's not a bug - it's documented behaviour.

When a public tracker opens, this page will be updated with the submission URL.

A.3 Intent of compatibility

PetaPerl aims at ~100% behavioral compatibility with **Perl 5.44** and with the specific versions of the CPAN modules we ship (see `pperl -V`, `PPERL_MODS`). Where we diverge, we diverge deliberately and document it.

Older language forms (e.g. `use v5.18;` features that differed from later versions) are not a compatibility target. PetaPerl follows current upstream Perl, not historical upstream Perl.

A.4 Upstream's contributions

The CPAN modules whose names we carry represent years of work by volunteer maintainers who deserve credit for the interface design and original implementation even when our code shares no lines with theirs. We thank them, and we do not want their inboxes flooded with issues produced by a runtime they did not write. That is the single most important reason this page exists.

A.5 License

See the *License* page for the full terms of the PetaMem Research Preview License that governs your use of this Software.

APPENDIX

B

LICENSE

B.1 PetaMem Research Preview License

Copyright © 2026 PetaMem s.r.o. (IČ: 26512521), Prague, Czech Republic. All rights reserved.

B.1.1 1. Definitions

“Software” means pperl (PetaPerl), a Perl 5-compatible interpreter implemented in Rust, distributed in binary form under this License, including any accompanying documentation. “Licensor” means PetaMem s.r.o. “You” means the individual or entity exercising permissions granted by this License.

B.1.2 2. Grant

Subject to the terms below, Licensor grants You a worldwide, royalty-free, non-exclusive, non-transferable license to use, copy, run, benchmark, evaluate, and redistribute the Software in unmodified binary form, for any purpose, commercial or non-commercial, for so long as the Software’s version number is below 1.0.

B.1.3 3. Attribution and Non-Appropriation

You shall not represent the Software, in whole or in part, as Your own work, nor remove, obscure, or alter any copyright notices, version identifiers, author attributions, or license texts contained in or accompanying the Software. Any redistribution must preserve this License and all original notices intact.

B.1.4 4. Reverse Engineering

Reverse engineering, disassembly, and decompilation of the Software for the purposes of interoperability, security research, academic study, and personal curiosity are expressly permitted. Publication or redistribution of source code, derived source, or modified binaries obtained through such activity is not granted under this License and is prohibited.

B.1.5 5. Research Preview Status

The Software is a research preview. Its interfaces, behavior, file formats, performance characteristics, and licensing terms may change without notice in any subsequent release. This License applies only to the version of the Software with which it was distributed.

B.1.6 6. No Warranty

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, AND ACCURACY. YOU ASSUME ALL RISK ARISING FROM YOUR USE OF THE SOFTWARE.

B.1.7 7. Limitation of Liability

TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL LICENSOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, OR EXEMPLARY DAMAGES ARISING OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THIS LICENSE, REGARDLESS OF THE LEGAL THEORY ASSERTED, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

B.1.8 8. Termination

This License and all rights granted hereunder terminate automatically upon Your breach of any term. Upon the public release of version 1.0 or later of the Software, this License ceases to apply to subsequent versions, which will be governed by their own license terms.

B.1.9 9. Reservation of Rights

All rights not expressly granted are reserved by Licensor. No rights in source code, trademarks, patents, or trade secrets are conveyed by this License except as explicitly stated. “PetaMem”, “PetaPerl”, and “pperl” are used here as identifiers of the Software and its origin; no trademark license is granted.

B.1.10 10. Governing Law and Venue

This License shall be governed by and construed in accordance with the laws of the Czech Republic, without regard to conflict of law provisions. The exclusive venue for any dispute arising hereunder shall be the competent courts of Prague, Czech Republic.

APPENDIX

C

GLOSSARY

Terms, acronyms, and project-specific vocabulary.

pperl
: The PetaPerl runtime - a Rust reimplementation of Perl 5, binary name `pperl`. Reads the same Perl source code the stock `perl` interpreter does. A behavioural mirror of `perl5`: semantics, edge cases, magic. Where a test disagrees with upstream perl5, pperl is the one that's wrong. Adds a JIT and auto-parallelization on top.

SV
: Scalar Value. A Perl scalar as represented in pperl's runtime - the enum lives in `src/runtime/sv.rs` and mirrors `perl5`'s SV layout where the semantics depend on it.

JIT
: Just-In-Time compilation. pperl compiles hot loops to machine code via Cranelift; triggered automatically on eligible `for`/`while` bodies.

named unary operator
: A Perl built-in that takes **exactly one argument** at a specific precedence level - tighter than comparison operators, looser than the arithmetic and shift operators. Unlike a **list operator** (which greedily slurps the rest of the expression), a named unary takes its one argument and stops there. Examples: `defined`, `exists`, `ref`, `scalar`, `length`, `uc`/`lc`, `chr`/`ord`, `int`, `abs`, `sqrt`, `sin`/`cos`, `log`, `exp`, `-e`/`-r`/`-f`/... file tests. See `perlop` precedence row 10.

list operator
: A Perl built-in that takes a **list of arguments** and, at the right-hand side of the operator, greedily consumes the rest of the expression. Examples: `print`, `sort`, `push`, `unshift`, `join`, `split`. Contrast with **named unary operator**.

This is an initial seed. Terms referenced by `{term}` roles elsewhere in the docs should be added here when they appear.

Non-alphabetical

??, 80
  PHP, 303
..., 1462
  flip-flop, 241, 343, 352, 360
  Lua, 328
..., 1462
  Lua, 333
--, 1482
{...}, 579
  vs block, 579
. (*dot*)
  in patterns, 64
  under /s, 102
!, 1458
  return value, 605
#
  Lua, 329
#{}
  Ruby, 314
$
  end of string, 69
  under /m, 69
$?, 560, 566, 670
  decoding, 205, 562
  exit status, 364
$!, 565, 667
  volatility, 205, 566
$., 239, 342, 348, 357, 373, 676
  across files, 266, 345
  selecting lines, 241
$', 93, 203, 679
${...}, 569
$", 239, 675, 1521
  joining fields, 243
$#, 1489
  shell, 365
$#{...}, 572
$#_
  pperl 0.6.15, 5499
$#array, 369
  pperl 0.6.15, 5499
$$, 664
$&, 93, 248, 679
  performance cost, 203
$(, 666
$(...)
  shell, 360
$), 666
$+, 94, 679
$,, 239, 349, 674
$/, 238, 266, 556, 673
  slurp, 557
$0, 663
  awk, 348
  pperl 0.6.9, 5503
$1, 85, 678
  after a failed match, 86
  awk, 349
  in replacement, 108
$2, 678
$3, 678
$;, 676
$<, 664
$=
  sed, 342
$>, 665
$@, 204, 295, 310, 324, 339, 565, 669
  clobbering, 205
  shell, 365
$\, 236, 674
$], 682
  in pperl, 33
$^E, 205, 671
${^GLOBAL_PHASE}, 685
$^I, 683
${^MATCH}, 93, 203, 679
$^N, 94, 679
  in code blocks, 130
$^O, 682
  in pperl, 33
$^P, 195, 684

${^POSTMATCH}, 93, 203, 679
${^PREMATCH}, 93, 203, 679
$^R, 130, 684
$^S, 200, 208, 684
$^T, 683
  with the daemon, 163
$^V, 682
  in pperl, 33
$^W, 683
${^WARNING_BITS}, 683
$^X, 683
$_, 660
  as current record, 348
  as pattern space, 342
  implicit match target, 58
  in a block, 315
  in one-liners, 239
$_GET, 308
$`, 93, 203, 679
$|, 199, 675
$ARG, 660
$ARGV, 239, 266, 687
$AUTOLOAD, 178
$BASETIME, 683
$CHILD_ERROR, 670
$Coro::current, 37
$Coro::main, 37
$DB::IN, 223
$DB::OUT, 223
$DB::single, 215, 225
$DB::trace, 225
$EFFECTIVE_GROUP_ID, 666
$EFFECTIVE_USER_ID, 665
$EGID, 666
$ERRNO, 667
$EUID, 665
$EVAL_ERROR, 669
$EXCEPTIONS_BEING_CAUGHT, 684
$EXECUTABLE_NAME, 683
$EXTENDED_OS_ERROR, 671
$GID, 666
$INC, 691
$INPLACE_EDIT, 683
$INPUT_LINE_NUMBER, 676
$INPUT_RECORD_SEPARATOR, 673
$LAST_PAREN_MATCH, 679
$LAST_REGEXP_CODE_RESULT, 684
$LAST_SUBMATCH_RESULT, 679
$LIST_SEPARATOR, 675
$LOAD_PATH, 322
$MATCH, 679
$NEGATIVE_INDICES, 278
$NF, 349
$NR, 676
$OFS, 674
$OLD_PERL_VERSION, 682
$ORS, 674
$OS_ERROR, 667
$OSNAME, 682
$OUTPUT_AUTOFLUSH, 675
$OUTPUT_FIELD_SEPARATOR, 674
$OUTPUT_RECORD_SEPARATOR, 674
$PERL_VERSION, 682
$PERLDB, 684
$PID, 664
$POSTMATCH, 679
$PREMATCH, 679
$PROCESS_ID, 664
$PROGRAM_NAME, 663
$REAL_GROUP_ID, 666
$REAL_USER_ID, 664
$REGERROR, 131
$REGMARK, 131
$RS, 673
$self, 178, 294
  in method, 172
$SIG{__DIE__}, 208
  chaining handlers, 208
  stack trace, 200
$SIG{__WARN__}, 208
  promoting warnings to fatal, 197
$SIG{CHLD}
  reaping children, 626
$SUBSCRIPT_SEPARATOR, 676
$SUBSEP, 676
$this, 308
$UID, 664
${var#prefix}, 359
${var%suffix}, 359
${var/a/b}, 359
$VERSION
  installer probe, 638
$WARNING, 683
%, 1449
  negative operand, 524
%!, 668
%-, 678
%{...}, 569
%+, 86, 248, 678
%=, 1468
%{^CAPTURE}, 678
%{^CAPTURE_ALL}, 678

%Config
  useithreads, 42
%DB::alias, 218
%ENV, 687
  passing values to one-liners, 259
%ERRNO, 668
%INC, 690
  native module, 638
%LAST_PAREN_MATCH, 678
%OS_ERROR, 668
%seen idiom, 244
%SIG, 691
&, 1459, 1524, 1538
  bitwise AND, 615
&., 1461
&{...}, 581
&&, 1457
  return value, 604
&&=, 1468
&=, 615, 1468
(?#...), 60
(?&NAME), 89
(?(cond)yes|no), 91
(?(DEFINE)...), 91
(??{...}), 130
(?!)
  always fails, 133
(?!...), 72, 617
(?{...}), 130
(*{...}), 130
(*:NAME), 132
(*ACCEPT), 133
(*asr:...), 119
(*atomic:...), 88
(*atomic_script_run:...), 119
(*COMMIT), 133
(*F), 133
(*FAIL), 133
(*MARK:NAME), 132
(*nla:...), 74
(*nlb:...), 74
(*pla:...), 74
(*plb:...), 74
(*PRUNE), 132
(*script_run:...), 119
(*SKIP), 132
(*sr:...), 119
(*THEN), 133
(?1), 89
(?:...), 86
(?<!...), 74
(?<=...), 74
(?<name>...), 86
(?=...), 72, 617
(?>...), 88
(?[...]), 66
  with Unicode properties, 117
(?^...), 106
(?|...), 98
(?flags), 105
(?flags:...), 105
(?P<name>...), 87
(?R), 89, 251
*, 1448, 1512
  quantifier, 77
*?, 80
.*?, 66
**, 1449
**=, 1468
*+, 81
*=, 1468
*args, 290
+, 1448, 1449
  numeric only, 300
  quantifier, 77
+?, 80
+{...}, 579
?+, 81
++, 81, 1482
+=, 1468
,, 1478
-, 1448, 1449
., 1450
  string concatenation, 286, 328
/, 1448
//, 604, 1457
  defined-or in reductions, 252
//=, 604, 618, 1468
/=, 1468
/a, 105, 593
  ASCII-only classes, 117
/aa, 105, 593
  case folding, 117
/bin/sh, 561
/c, 103
/d, 105
  dual-mode semantics, 118
/e, 104, 248
  in substitution, 110
/ee, 104, 248
  security, 111

/g, 103
  in list context, 103, 247, 292, 306, 321, 335
  in substitution, 109
/gc, 103
/i, 101
  Unicode, 119
  Unicode case folding, 594
/l, 105, 593
  locale semantics, 118
/m, 101, 249
  anchors, 69
/n, 104
/o, 105
  versus qr//, 128
/p, 105, 203
/r, 104, 248
/s, 102, 249
/u, 105, 593
  Unicode semantics, 117
/x, 102
  and interpolated whitespace, 106
/xx, 103
0, 1486
-0, 10, 238
  input record separator, 238
0 but true, 601
0.5.6, 5518
0.6.0, 5516
0.6.1, 5516
0.6.2, 5514
0.6.3, 5512
0.6.5, 5510
0.6.6, 5509
0.6.8, 5508
0.6.9, 5502
0.6.11, 5502
0.6.12, 5501
0.6.13, 5500
0.6.14, 5499
0.6.15, 5498
0.6.16, 5495
0.6.17, 5493
0.6.18, 5492
0.6.19, 5487
0.6.20, 5486
0.6.21, 5480
0.6.22, 5474
0.6.23, 5470
0.6.24, 5459
0.6.25, 5459
1 while, 110
2>&1, 360
2D graphics, 377
3D graphics, 432, 433
-0777, 238
?:, 605, 1470
  chaining, 290, 318, 332
  instead of if/else, 618
  nesting, 605
::
  package separator, 293, 322
:isa, 173, 294, 308, 323, 337
:param, 170
:reader, 170
:shared, 46
:writer, 170
<, 1452
<<, 615, 1460
<<=, 1468
<=, 1452
<=>, 1453
  return value, 602
  versus cmp, 157
<>, 266
=, 1467
!=, 1452
.=, 1468
-=, 1468
==, 1452
  on strings, 300, 602
=>
  auto-quoting, 287, 316, 330, 370
  quoting barewords, 514
=~, 57, 292, 306, 321, 335, 371, 1465
>, 1452
->, 1475
  calling a code reference, 581
  element access, 570
  optional between subscripts, 291, 571
->&, 172
>=, 1452
>>, 615, 1460
>>=, 1468
?
  quantifier, 77
@-, 92, 679
@{...}, 569
@+, 92, 679
@{[ ]}
  expression interpolation, 299,

314
interpolating expressions, 243
@^CAPTURE, 681
@{^CAPTURE}, 93, 681
@_, 661, 1526
unpacking arguments, 370
@ARG, 661
@ARGV, 365, 686
decoding, 591, 600
@F, 236, 242, 349
@INC, 293, 322, 337, 689
in pperl, 32
order, 636
pperl 0.6.1, 5516
system trees, 637
@ISA, 177
@LAST_MATCH_END, 679
@LAST_MATCH_START, 679
[:^digit:], 65
[:alpha:], 65
[\s\S], 64
[...], 579
[^...], 62
[incr Tcl], 376
\
taking a reference, 568
\&, 581
\1, 87
in replacement, 109
sed, 342
\1 better written as $1, 109
\A, 69, 249
\B, 70
\b, 70
inside a character class, 63
with interpolated non-word text, 71
\b{...}, 119
\b{gcb}, 71
\b{lb}, 71
\b{sb}, 71
\b{wb}, 71
\cX, 59
\d, 63
Unicode, 592
\E, 60
\G, 71
synchronised matching, 72
\g{...}, 87
\g{-1}, 88
\h, 63
\K, 74
in substitution, 112
\k<name>, 86
\N, 63
\N{...}, 115, 594
\N{U+...}, 115, 594
\o{...}, 59
\P{...}, 65, 593
\p{...}, 65, 593
single-letter form, 115
user-defined, 600
\pL, 115
\p{Property=Value}, 116
\Q, 60
\R, 63, 598
\s, 63
Unicode, 592
\U
in patterns, 60
in replacement, 109
\u
in replacement, 109
\v, 63
\w, 63
Unicode, 592
\X, 116, 598
\x{...}, 59
Unicode code point, 115
\Z, 69
\z, 69, 249
^, 1459
bitwise XOR, 615
start of string, 69
under /m, 69
^., 1461
^=, 615, 1468
^^, 1457
pperl 0.6.21, 5486
^^=, 1468
pperl 0.6.21, 5486
__CLASS__, 174, 719
__FILE__, 200, 721
__index, 337
__init__, 294
__LINE__, 200, 724
__PACKAGE__, 727
__SUB__, 730
_charnames, 5225
_exit (*POSIX*), 3941
_gmp_build_version (*Math::GMP*), 3676
_gmp_lib_version (*Math::GMP*), 3676

|, 1459
  bitwise OR, 615
  in patterns, 95
|., 1461
|-, 561
-|, 561
|=, 615, 1468
||, 1457
  default value, 604
  return value, 604
||=, 604, 1468
~, 1459
  awk, 351
  bitwise NOT, 615
~., 1461
!~, 57, 1465
  double negation, 611

A

-a, 10, 349, 363
  autosplit, 236
AABB, 450
abs, 733
abs_path (*Cwd*), 2143
accept, 736
access (*POSIX*), 3935
accessor, 176
  read/write, 176
adapters (*Peta::WebGPU*), 4069
add (*Digest::MD5*), 2436
add (*Digest::SHA*), 2458
add (*Digest::SHA1*), 2472
add_bits (*Digest::MD5*), 2448
add_bits (*Digest::SHA*), 2462
add_history (*Peta::ReadLine::XS*), 4046
add_outline (*Cairo::PdfSurface*), 1998
add_symbol (*Package::Stash::XS*), 3949
add_ui_gmp (*Math::GMP*), 3677
addfile (*Digest::MD5*), 2437
addfile (*Digest::SHA*), 2464
addfile (*Digest::SHA1*), 2472
address
  sed, 343
address range
  sed, 343
ADJUST, 172
adler32 (*Compress::Raw::Zlib*), 2056
Advanced Regular Expressions, 371
advise (*File::Map*), 2853
AE, 1590
AF_UNIX, 629
again (*EV::Timer*), 2504
alarm, 738
algorithm (*Digest::SHA*), 2458
all, 742
all (*List::Util*), 3508
all_keys (*Hash::Util*), 3041
alloc (*Peta::FFI*), 4033
allow_blessed (*JSON::XS*), 3475
alpha channel, 381
  in GD, 391
alternation, 95, 617
  captures, 98
  empty alternative, 97
  leftmost alternative wins, 95
  of many strings, 136
  order of alternatives, 96
  performance, 98
  precedence, 96
  versus character class, 97
ambient term, 445
anchor, 69
and, 1458
  vs &&, 603
anonymous array, 291, 305, 320, 334, 579
anonymous hash, 291, 305, 320, 334, 579
anonymous reference, 578
anonymous subroutine, 320, 581
anti-aliasing, 385
  in GD, 394
any, 744
any (*List::Util*), 3507
AnyDBM_File, 1593
AnyEvent, 1594
  loop selection, 39
  pperl 0.6.19, 5489
AnyEvent::Debug, 1642
AnyEvent::DNS, 1633
AnyEvent::Handle, 1646
AnyEvent::Impl::Cocoa, 1683
AnyEvent::Impl::EV, 1684
AnyEvent::Impl::Event, 1684
AnyEvent::Impl::EventLib, 1685
AnyEvent::Impl::FLTK, 1686
AnyEvent::Impl::Glib, 1687
AnyEvent::Impl::IOAsync, 1688
AnyEvent::Impl::Irssi, 1690
AnyEvent::Impl::Perl, 1696
AnyEvent::Impl::POE, 1691
AnyEvent::Impl::Qt, 1696
AnyEvent::Impl::Tk, 1697
AnyEvent::Impl::UV, 1698

AnyEvent::IO, 1670
AnyEvent::IO::IOAIO, 1681
AnyEvent::IO::Perl, 1682
AnyEvent::Log, 1699
AnyEvent::Loop, 1714
AnyEvent::Socket, 1716
AnyEvent::Strict, 1725
AnyEvent::TLS, 1725
AnyEvent::Util, 1736
App::Cpan, 1741
App::Prove, 1747
App::Prove::State, 1752
App::Prove::State::Result, 1754
App::Prove::State::Result::Test, 1756
append mode, 556, 557
append_path (*Cairo::Context*), 1988
architecture directories, 32
architecture directory, 637, 643
Archive::Tar, 1758
Archive::Tar::File, 1774
argument, 1526
ARGV, 266, 555, 687
arithmetic
    one-liners, 252
Array
    Ruby, 315
array, 1488
    0-based index, 329
    interpolation, 1521
    length, 329, 1489
    PHP, 300
    slice, 1490
array names, 370
array of arrays, 571
    iterating, 572
    shared inner array, 573
array of hashes, 575
array_combine, 311
array_filter, 300
array_key_exists, 301
array_map, 300
array_pop, 300
array_push, 300
arrow, 1475
ascii (*JSON::PP*), 3468
asctime (*POSIX*), 3940
assignment, 1467
associative array
    awk, 355
    PHP, 301
    Tcl, 370
associativity, 1481
at (*PDL*), 3901
atan2, 747
atomic append, 557
atomic group, 88
    nested, 125
    performance, 125
atomic rename, 558
attach (*Peta::FFI*), 4026
attr (*EV::Stat*), 2503
attr_accessor, 323
attribute, 1553
Attribute::Handlers, 1778
attributes (*pragma*), 5225
attrs, 1553
auto-decrement, 1482
autodie, 198, 566
    lexical scope, 198, 566
autodie (*pragma*), 5226
autodie::exception (*pragma*), 5236
autodie::exception::system (*pragma*), 5240
autodie::hints (*pragma*), 5241
autodie::Scope::Guard (*pragma*), 5232
autodie::Scope::GuardStack (*pragma*), 5233
autodie::skip (*pragma*), 5247
autodie::Util (*pragma*), 5234
Auto-FFI, 19
autoflush, 199, 675
    on pipes, 624
    on sockets, 626
auto-increment, 1482
AUTOLOAD, 178
    DESTROY guard, 178
    in class, 193
    in method lookup, 181
AutoLoader, 1790
auto-parallelization, 3, 34
    and the JIT, 18
    command-line control, 35
    determinism, 36
    eligible loops, 35
    floating-point reductions, 35
    how it works, 34
    limitations, 36
    loop shape, 159
    reduction detection, 36
    replacing a work crew, 51
    scaling, 36

string variables, 153, 159
AutoSplit, 1795
autosplit, 236, 242
field separator, 237
autouse (*pragma*), 5248
autovivification, 572
lvalue vs rvalue, 578
membership test, 355
with push, 574
average, 252
awk
common mistakes in Perl, 356
moving to Perl, 347
regular expressions, 140
axis-aligned bounding box, 450
overlap test, 520

B

B, 1798
pperl 0.6.2, 5515
pperl 0.6.9, 5507
b64digest (*Digest::MD5*), 2441
b64digest (*Digest::SHA1*), 2472
B::AV, 1804
B::BINOP, 1805
B::BM, 1805
B::cchar, 1856
B::comppadlist, 1856
B::Concise, 198, 1811
pperl 0.6.0, 5517
B::COP, 1805
B::COW, 1806
B::COW::cowrefcnt, 1808
B::COW::cowrefcnt_max, 1808
B::COW::is_cow, 1808
B::cstring, 1856
B::CV, 1808
B::CV::GV, 1810
B::CV::NAME_HEK, 1811
B::Deparse, 198, 1824
pperl 0.6.0, 5517
signatures, 33
B::FM, 1832
B::GV, 1832
B::GV::NAME, 1834
B::GV::SAFENAME, 1857
B::GV::SV, 1834
B::hash, 1856
B::HE, 1834
B::HV, 1835
B::HV::NAME, 1835
B::IO, 1836
B::IO::IsSTD, 1837
B::IV, 1837
B::IV::packiv, 1838
B::LISTOP, 1838
B::LOGOP, 1838
B::LOOP, 1838
B::MAGIC, 1839
B::main_root, 1855
B::METHOP, 1840
B::NV, 1840
B::OP, 1840
B::OP::oplist, 1841
B::Op_private, 1842
B::opnumber, 1855
B::PADLIST, 1844
B::PADNAME, 1844
B::PADNAMELIST, 1845
B::PADOP, 1845
B::PMOP, 1845
B::PV, 1846
B::PV::PV, 1847
B::PVLV, 1847
B::PVMG, 1847
B::RHE, 1848
B::safename, 1857
B::Showlex, 1849
B::SV, 1848
B::SV::object_2svref, 1849
B::SV::REFCNT, 1848
B::SV::SvTYPE, 1849
B::SVOP, 1849
B::svref_2object, 1854
B::Terse, 1851
B::UNOP, 1852
B::UNOP_AUX, 1852
B::walkoptree, 1855
B::walkoptree_debug, 1855
B::Xref, 1852
back-face culling, 441
on the GPU, 480
backreference, 87
portability, 94
relative, 88
versus octal escape, 88
backslash
in patterns, 58
making a reference, 568
backspace
in a character class, 63
backtick, 360

backtracking, 122
  greedy and non-greedy examples, 83
  saved states, 123
backtracking control verb, 131
balanced parentheses
  matching, 89
bareword
  constant not yet imported, 465
barycentric coordinates, 442
base
  declaring a superclass, 177
base (*pragma*), 5249
Base64, 257
basename
  with s///, 359
bdiv (*Math::GMP*), 3679
BEGIN
  awk, 353
  in one-liners, 235, 346, 353
  variable scope, 268
begin (*Peta::FFI::LMDB::Env*), 4005
begin/rescue/ensure, 324
bench/run-perlbench, 149
Benchmark, 149, 1857
  comparing patterns, 136
benchmarking, 149
Bezier curve, 410
bfac (*Math::GMP*), 3681
bgcd (*Math::GMP*), 3680
bidirectional pipe, 562, 621
big-endian, 535
bigfloat (*pragma*), 5251
bigint (*pragma*), 5258
bignum (*pragma*), 5266
bigrat (*pragma*), 5276
binary data, 532
binary file format, 550
binary files, 560, 591
binary string, 587
bincompat_options (*Config*), 2083
bind, 749
binding, 1465
binding operator, 57
binmode, 558, 752
  push layer, 560
  strip layers, 560
bit fields
  decoding with shifts and masks, 549, 552
  encoding, 553
bit flags, 615
bit string, 539
bit tricks, 616
bit-exactness, 35
bitmap font, 395
bitwise and, 1459
bitwise operations, 615
bitwise or, 1459
bitwise shift, 1460
bitwise xor, 1459
blcm (*Math::GMP*), 3680
bless, 755
  inheritance-safe constructor, 175
  object model, 175
blessed, 206
blib (*pragma*), 5283
Blinn-Phong, 445
  in WGSL, 481
block
  Ruby, 320
  versus Tcl braces, 372
blshift (*Math::GMP*), 3681
blur, 386
bmodinv (*Math::GMP*), 3679
BMP header, 537
bmulf (*Math::GMP*), 3678
bnok (*Math::GMP*), 3681
bool, 1486
boolean, 1486
boolean context, 601
boolean logic, 601
boolean operations
  on solids, 458
boolean_values (*JSON::PP*), 3469
boss/worker pattern, 44
bounding sphere, 450
  against a box, 520
  computing, 503
bounding volume, 450
bounding volume hierarchy, 461
Boyer-Moore, 135
bracketed character class, 62
branch reset, 98
  capture numbering, 92
break, 759
  awk, 354
  Lua, 332
  shell, 364
breakpoint, 214
  conditional, 214

in BEGIN blocks, 215
listing and disabling, 216
on file load, 218
one-shot, 218
postponed, 218
Bresenham's algorithm, 382
broot (*Math::GMP*), 3682
brootrem (*Math::GMP*), 3682
brshift (*Math::GMP*), 3681
bsd_glob (*File::Glob*), 2836
bsqrt (*Math::GMP*), 3683
bucket_array (*Hash::Util*), 3044
bucket_info (*Hash::Util*), 3044
bucket_ratio (*Hash::Util*), 3044
bucket_stats (*Hash::Util*), 3045
bucket_stats_formatted (*Hash::Util*), 3045
buffered I/O, 199, 557
--build-id, 12
daemon identity, 166
built-in, 1526
builtin (*pragma*), 5284
bump-along, 60
disabling with \G, 72
BVH, 461
byte order, 535
bytes (*pragma*), 5292

C

-C, 591
not supported, 33
Unicode on standard streams, 238, 264
-c, 10
syntax check, 198
C comment
matching, 127
C struct
padding, 545
C3 linearisation, 181
Cairo, 1981
drawing model, 410
native binding, 28, 420
paths, 410
pperl 0.5.6, 5519
pperl 0.6.16, 5497
pperl 0.6.21, 5485
pperl 0.6.22, 5480
premultiplied alpha, 411
versus SVG, 412
Cairo::Context, 1984
Cairo::Context::append_path, 1988
Cairo::Context::copy_clip_rectangle_list, 1989
Cairo::Context::copy_path, 1988
Cairo::Context::select_font_face, 1987
Cairo::Context::show_glyphs, 1988
Cairo::Context::show_text, 1987
Cairo::Context::show_text_glyphs, 1988
Cairo::Context::tag_begin, 1987
Cairo::Font, 1989
Cairo::FontFace, 1990
Cairo::FontOptions, 1990
Cairo::Format, 1991
Cairo::Format::stride_for_width, 1992
Cairo::FtFontFace, 1992
Cairo::FtFontFace::create, 1992
Cairo::Gradient, 1992
Cairo::ImageSurface, 1993
Cairo::ImageSurface::create_for_data, 1993
Cairo::ImageSurface::create_from_png, 1994
Cairo::ImageSurface::create_from_png_stream, 1994
Cairo::LIB_VERSION_ENCODE, 2009
Cairo::LinearGradient, 1994
Cairo::Matrix, 1995
Cairo::Pattern, 1996
Cairo::PdfSurface, 1998
Cairo::PdfSurface::add_outline, 1998
Cairo::PdfSurface::set_metadata, 1998
Cairo::PsSurface, 1999
Cairo::RadialGradient, 1999
Cairo::RecordingSurface, 1999
Cairo::RecordingSurface::create, 2000
Cairo::Region, 2000
Cairo::Region::create, 2002
Cairo::ScaledFont, 2002
Cairo::ScaledFont::text_to_glyphs, 2003
Cairo::SolidPattern, 2004
Cairo::Surface, 2004
Cairo::Surface::create_for_rectangle, 2007
Cairo::Surface::create_similar,

2007
Cairo::Surface::set_mime_data, 2008
Cairo::Surface::supports_mime_type, 2007
Cairo::Surface::write_to_png_stream, 2007
Cairo::SurfacePattern, 2008
Cairo::SvgSurface, 2008
Cairo::ToyFontFace, 2008
Cairo::VERSION_ENCODE, 2009
call, 1524
call (*Peta::FFI*), 4023
callback, 582
call-by-reference, 1526
call-by-value, 1526
caller, 762
  pperl 0.6.14, 5500
callout, 139
camera
  look-at, 438
can, 183
Can't locate ... in @INC, 643
Can't locate loadable object for module, 640, 644
capture group, 84
  cost, 129
  in substitution, 247
  numbering, 85
  undefined versus empty, 85
Capture::Tiny, 2009
Carp, 200, 2017
  pperl 0.6.21, 5484
carp, 200
Carp::Always, 200
case
  shell, 364
case conversion, 241
case folding, 119, 594
  fc vs lc, 598
  locale-dependent, 597
case modification escape, 109
case-insensitive hash, 273
case-insensitive matching, 101
case-insensitive sort, 598
cast (*Peta::XS*), 4076
cat, 1450
catastrophic backtracking, 123
catch, 204, 765
Catmull-Clark subdivision, 457
cchar (*B*), 1856
central differences, 492
Changelog, 5457
character class, 62
  combining negated shorthands, 64
  metacharacters inside, 63
  negated, 62, 617
  negated, versus non-greedy dot, 66
  range, 62
  set operations, 66
character set modifiers, 593
characters vs bytes, 558, 587
  in patterns, 121
charnames, 594
  redefinition warnings, 33
charnames (*pragma*), 5294
charset modifier, 105
  mutual exclusion, 106
  resolution order, 118
  Unicode, 117
chart, 413
  bar chart, 413
  SVG, 414
Chart::Clicker, 415
Chart::Gnuplot, 415
chase camera, 523
chdir, 769
chdir (*Cwd*), 2147
checksum, 546
child process
  pipe, 560
chmod, 772
chomp, 293, 307, 321, 336, 348, 357, 373, 556, 775
chop, 777
chown, 780
chr, 782
chroot, 785
chunked file format, 554
circle drawing, 383
circular reference, 584
class, 787
  coming from Lua, 337
  coming from PHP, 308
  coming from Python, 294
  coming from Ruby, 323
  enabling the feature, 169
  equivalents of classical constructs, 190
  interoperation with bless, 174, 184
  migrating from bless, 190

minimal example, 169
restrictions, 174
Class::Accessor, 180
Class::Inspector, 2022
Class::Inspector::Functions, 2027
Class::Load::XS, 2029
Class::Load::XS::is_class_loaded, 2029
Class::Method::Delegate, 188
Class::MOP, 2030
Class::MOP::get_code_info, 2031
Class::MOP::Method, 2030
Class::MOP::Method::is_stub, 2030
Class::Struct, 2031
Class::Tiny, 180
Class::XSAccessor, 2038
pperl 0.6.21, 5483
Class::XSAccessor::Array, 2041
Class::XSAccessor::Heavy, 2045
CLEAR
tied array, 278
tied hash, 274
clip space, 437
WebGPU versus OpenGL, 477
clipping
near plane, 443
Clone, 2046
clone (*Clone*), 2046
clone (*Digest::MD5*), 2435
Clone::clone, 2046
close, 556, 793
error checking, 557, 566
pipe, 561
closed_over (*PadWalker*), 3954
closedir, 796
closure, 291, 305, 320, 334, 583, 1536
in loops, 583
closure (*Peta::FFI*), 4021
cluck, 200
cmp, 1455
return value, 602
cmpthese, 149
CNF, 614
code point, 588
code reference, 581
collating element, 65
collision
bounce, 428
collision detection, 520
colour
sRGB decoding, 467
colour model, 380
column-major order, 436
WGSL, 476
combining character, 598
comma, 1478
command substitution, 360
command-line flags
pperl support, 33
command-line options
summary, 7
common-prefix factoring, 98
compare (*Data::UUID*), 2418
comparison
numeric vs string, 286, 300, 314, 329, 354, 368, 602
compatibility
with perl5, 31
compile_date (*Config*), 2083
Composer, 308
compositing, 381
composition, 186
comppadlist (*B*), 1856
Compress::Raw::Bunzip2, 2048
Compress::Raw::Bzip2, 2049
pperl 0.6.22, 5476
Compress::Raw::Zlib, 2052
pperl 0.6.6, 5509
Compress::Raw::Zlib::adler32, 2056
Compress::Zlib, 2057
compute shader, 487
bounds check, 488
concat, 1450
concatenate, 1450
concatenation, 1450
text and bytes, 596
concurrency, 34
choosing a model, 34, 51
cond_signal, 49
cond_wait, 49
condition
simplifying, 612
condition variable, 49
conditional expression
Python, 290
conditional pattern, 91
confess, 200
Config, 2071
pperl 0.6.5, 5511
pperl 0.6.9, 5507
Config::bincompat_options, 2083
Config::compile_date, 2083

Config::config_re, 2080
Config::config_sh, 2078
Config::config_vars, 2079
    pperl 0.6.22, 5480
Config::Extensions, 2074
Config::header_files, 2084
Config::jit_counts, 2082
Config::lib_file, 639, 2080
    troubleshooting, 643
Config::lib_files, 2081
Config::local_patches, 2083
Config::myconfig, 2077
Config::non_bincompat_options, 2084
Config::Perl::V, 2074
config_re (*Config*), 2080
config_sh (*Config*), 2078
config_vars (*Config*), 2079
conjunctive normal form, 614
connect, 799
    TCP client, 626
constant, 1539
constant (*pragma*), 5301
constant folding, 152, 202
    built-in functions, 153
constructive solid geometry, 458
constructor
    generated, 169
    sub new, 175
constructor property promotion, 308
context, 1498
    coming from Lua, 331
    coming from PHP, 302
    coming from Python, 288
    coming from Ruby, 316
    why it matters, 284
context (*Digest::MD5*), 2442
continue, 802
    awk, 354
    shell, 364
contradiction, 607
convolution, 386
    in GD, 404
    kernel, 386
cooperative multitasking, 37
coordinate system
    graphics, 379
    handedness, 435
co-process, 562, 620
    deadlock, 621
copy_clip_rectangle_list (*Cairo::Context*), 1989
copy_path (*Cairo::Context*), 1988
copy-on-write, 52
CORE::GLOBAL, 1526
core_bools (*JSON::PP*), 3470
Corinna, 167
Coro, 37, 2085
    async, 37
    cede, 37
    join, 37
    pperl 0.6.19, 5489
    priority, 37
    rouse callbacks, 38
Coro::AIO, 2105
Coro::AnyEvent, 38, 2106
Coro::BDB, 2112
Coro::Channel, 38, 2112
Coro::Debug, 2114
Coro::EV, 39, 2117
Coro::Event, 39, 2118
Coro::Handle, 38, 2119
Coro::LWP, 2121
Coro::RWLock, 2122
Coro::Select, 2124
Coro::Semaphore, 38, 2125
Coro::SemaphoreSet, 2127
Coro::Signal, 38, 2128
Coro::Socket, 38, 2129
Coro::Specific, 2131
Coro::State, 2132
    cctx_stacksize, 40
Coro::Storable, 2135
Coro::Timer, 2136
Coro::Util, 2137
coroutine, 37
    blocking I/O, 38
    C stack, 40
    per-coroutine state, 40
    replacing threads, 54
    when to use, 37
cos, 805
count
    PHP, 302
count (*Peta::FFI::PCRE2*), 4015
counter generator, 583
countof, 288, 316
cover, 231
coverage, 231
    anti-aliasing, 385
cowrefcnt (*B::COW*), 1808
cowrefcnt_max (*B::COW*), 1808
CPAN, 1865

installing modules, 634
troubleshooting, 643
cpan
built-in client, 635
--cpan, 12, 634
arguments, 635
curl, 637, 644
distribution preferences, 637
external tools, 637
git repository, 635
https, 637
install location, 635
offline install, 637
patches, 637
Perl part of a failed XS build, 642
plain http fallback, 644
pperl 0.6.8, 5508
pperl 0.6.9, 5507
skipping tests, 645
--system, 635
-T, 645
wget, 637, 644
XS substitutes, 638
XS substitutes not refreshed, 644
CPAN::Debug, 1910
CPAN::Distroprefs, 1910
CPAN::FirstTime, 1913
CPAN::HandleConfig, 1923
CPAN::Kwalify, 1924
CPAN::Meta, 1924
CPAN::Meta::Converter, 1932
CPAN::Meta::Feature, 1934
CPAN::Meta::History, 1935
CPAN::Meta::Merge, 1938
CPAN::Meta::Prereqs, 1940
CPAN::Meta::Requirements, 1942
CPAN::Meta::Requirements::Range, 1949
CPAN::Meta::Spec, 1952
CPAN::Meta::Validator, 1972
CPAN::Meta::YAML, 1973
CPAN::Mirrors, 1974
CPAN::Nox, 1977
CPAN::Plugin, 1978
CPAN::Plugin::Specfile, 1979
CPAN::Queue, 1980
CPAN::Tarzip, 1980
CPAN::Version, 1980
CPAN.pm, 635
Cranelift, 3, 16
create (*Cairo::FtFontFace*), 1992
create (*Cairo::RecordingSurface*), 2000
create (*Cairo::Region*), 2002
create (*Data::UUID*), 2417
create_for_data (*Cairo::ImageSurface*), 1993
create_for_rectangle (*Cairo::Surface*), 2007
create_from_name (*Data::UUID*), 2417
create_from_png (*Cairo::ImageSurface*), 1994
create_from_png_stream (*Cairo::ImageSurface*), 1994
create_similar (*Cairo::Surface*), 2007
croak, 200
cross product, 435
crypt, 807
Crypt::OpenSSL::Random, 2139
CSG, 458
csh_glob (*File::Glob*), 2837
cstring (*B*), 1856
cstring (*XString*), 5156
CSV, 256
quoted fields, 250
ctime (*POSIX*), 3941
cut, 358
Cwd, 2140
pperl 0.6.19, 5491
pperl 0.6.22, 5480
cwd (*Cwd*), 2144
Cwd::abs_path, 2143
Cwd::chdir, 2147
Cwd::cwd, 2144
Cwd::fast_abs_path, 2145
Cwd::getcwd, 2142

D

d
sed, 344
-d
starting the debugger, 195, 209
daemon, 161
fallback to a cold run, 165
in the shebang line, 165
limitations, 166
preload-safety contract, 163
security, 166
status, 163
systemd service, 165
template, 163

template cache, 164
--daemon, 12
starting and stopping, 161
--daemon-idle, 12
idle timeout, 164
--daemon-max-scripts, 12
cache size, 164
--daemon-reset, 12
dropping templates, 164
DAP, 227
data structure
dumping, 201, 219
data symbols, 24
data type, 1485
Data::Dump, 202
Data::Dumper, 201, 2411
configuration variables, 201
pperl 0.6.0, 5517
pperl 0.6.9, 5507
Data::Dumper::Concise, 201
Data::Dumper::Dumpxs, 2413
Data::Printer, 201
Data::UUID, 2415
Data::UUID::compare, 2418
Data::UUID::create, 2417
Data::UUID::create_b64, 2417
Data::UUID::create_bin, 2417
Data::UUID::create_from_name, 2417
Data::UUID::create_from_name_b64, 2417
Data::UUID::create_from_name_bin, 2417
Data::UUID::create_from_name_hex, 2417
Data::UUID::create_from_name_str, 2417
Data::UUID::create_hex, 2417
Data::UUID::create_str, 2417
Data::UUID::from_b64string, 2418
Data::UUID::from_hexstring, 2418
Data::UUID::from_string, 2418
Data::UUID::new, 2417
Data::UUID::to_b64string, 2418
Data::UUID::to_hexstring, 2418
Data::UUID::to_string, 2418
datagram, 630
date arithmetic, 255
DateTime, 255
DB, 2148
DB::DB, 195
DB::sub, 195
DB_File, 2394
pperl 0.6.21, 5483
DB_File::del, 2403
DB_File::fd, 2406
DB_File::filter_fetch_key, 2408
DB_File::filter_fetch_value, 2410
DB_File::filter_store_key, 2409
DB_File::filter_store_value, 2411
DB_File::get, 2404
DB_File::pop, 2401
DB_File::push, 2402
DB_File::put, 2404
DB_File::seq, 2407
DB_File::shift, 2401
DB_File::sync, 2406
DB_File::unshift, 2400
DBD::DBM, 2152
DBD::File, 2167
DBD::Gofer, 2175
DBD::Gofer::Policy::Base, 2185
DBD::Gofer::Policy::classic, 2186
DBD::Gofer::Policy::pedantic, 2187
DBD::Gofer::Policy::rush, 2187
DBD::Gofer::Transport::Base, 2188
DBD::Gofer::Transport::corostream, 2191
DBD::Gofer::Transport::null, 2193
DBD::Gofer::Transport::pipeone, 2194
DBD::Gofer::Transport::stream, 2195
DBD::Mem, 2196
DBD::Proxy, 2198
DBD::Sponge, 2210
DBD::SQLite
pperl 0.6.9, 5505
DBD::SQLite::Constants, 2202
DBGp, 227
DBI, 2212
pperl 0.6.5, 5511
pperl 0.6.9, 5506
DBI::Const::GetInfo::ANSI, 2325
DBI::Const::GetInfo::ODBC, 2326
DBI::Const::GetInfoReturn, 2327
DBI::Const::GetInfoType, 2327
DBI::DBD::Metadata, 2328
DBI::DBD::SqlEngine, 2330
DBI::Gofer::Execute, 2340
DBI::Gofer::Request, 2344
DBI::Gofer::Response, 2344
DBI::Gofer::Serializer::Base, 2345
DBI::Gofer::Serializer::DataD-

umper, 2345
DBI::Gofer::Serializer::Storable, 2345
DBI::Gofer::Transport::Base, 2346
DBI::Gofer::Transport::pipeone, 2347
DBI::Gofer::Transport::stream, 2347
DBI::Profile, 2347
DBI::ProfileData, 2359
DBI::ProfileDumper, 2364
DBI::ProfileSubs, 2367
DBI::ProxyServer, 2367
DBI::PurePerl, 2377
DBI::SQL::Nano, 2380
DBI::Util::CacheMemory, 2384
DBM_Filter, 2385
DBM_Filter::compress, 2391
DBM_Filter::encode, 2392
DBM_Filter::int32, 2393
DBM_Filter::null, 2393
DBM_Filter::utf8, 2394
dbmclose, 810
dbmopen, 812
DDP, 201
De Morgan's laws, 610
  unless, 610
-de0, 209
dead code elimination, 153
DeadCode (*Devel::Peek*), 2423
deadlock, 48
  co-process, 621
debug output
  conditional, 202
debugger
  actions, 216
  backtrace, 212
  command aliases, 218
  commands, 213
  continue, 211
  dumping structures, 219
  evaluating code, 212
  frame option, 224
  inheritance tree, 221
  lexical variables, 220
  LineInfo option, 224
  listing methods, 220
  listing source, 210
  listing subroutines, 221
  loaded modules, 221
  options, 222
  package variables, 220
  printing values, 211
  prompt, 209
  prompt hooks, 217
  redefining a subroutine, 213
  restarting, 213
  starting, 195, 209
  stepping, 211
  tracing, 224
debugging, 195
  in containers, 229
  remote, 229
declare -A, 365
decode, 589
decode (*Encode::Unicode*), 2533
decode (*JSON::PP*), 3467
decode (*JSON::XS*), 3475
decode_base64 (*MIME::Base64*), 3539
decode_base64url (*MIME::Base64*), 3544
decode_json (*JSON::PP*), 3468
decode_json (*JSON::XS*), 3476
decode_prefix (*JSON::PP*), 3467
decode_prefix (*JSON::XS*), 3476
decode_qp (*MIME::QuotedPrint*), 3554
decoded_base64_length (*MIME::Base64*), 3542
decrement, 1482
deep copy, 580
def
  Python, 290
  Ruby, 319
defer, 815
defined, 820, 1486
  versus exists, 301
defined-or, 289, 303, 317, 332, 604
del (*DB_File*), 2403
delegation, 188
  versus inheritance, 189
DELETE
  tied array, 278
  tied hash, 274
delete, 824
deprecate (*pragma*), 5305
depth buffer, 442
  on the GPU, 479
depth range, 477
dereference, 569, 1475
  arrow form, 570
  curly-brace form, 569
  postfix form, 291, 305, 334
DESTROY, 178
  freeing C objects, 420, 422

in class, 193
tied variable, 272
destructor, 178
detached thread, 43
deterministic replay, 530
Devel::Camelcadedb, 230
Devel::Confess, 200
Devel::Cover, 231, 2419
-ignore, 232
-select, 232
Devel::Debug::DBGp, 227
Devel::DumpTrace, 225
Devel::NYTProf, 148
Devel::Peek, 222, 2419
pperl 0.6.6, 5510
pperl 0.6.9, 5507
Devel::Peek::DeadCode, 2423
Devel::Peek::Dump, 2422
Devel::Peek::DumpArray, 2423
Devel::Peek::mstat, 2424
Devel::Peek::SvREFCNT, 2423
Devel::SelfStubber, 2424
Devel::Size, 2425
pperl 0.6.21, 5483
Devel::Trace, 225
Devel::Trace::Subs, 225
Devel::TraceINC, 225
Devel::TraceUse, 226
diagnostic messages
common, 199
diagnostics, 197
diagnostics (*pragma*), 5306
diamond inheritance, 181
diamond operator, 266, 345, 352, 556
dict, 287
Tcl, 370
dict comprehension, 296
die, 565, 827
throwing an exception, 204
trailing newline, 200, 204, 269, 295, 310, 324, 339, 567
with a hash reference, 207
diffuse lighting, 445
Digest, 2426
digest (*Digest::MD5*), 2439
Digest::base, 2473
Digest::file, 2474
Digest::MD5, 2430
pperl 0.6.13, 5501
Digest::MD5::add, 2436
Digest::MD5::add_bits, 2448
Digest::MD5::addfile, 2437
Digest::MD5::b64digest, 2441
Digest::MD5::clone, 2435
Digest::MD5::context, 2442
Digest::MD5::digest, 2439
Digest::MD5::hexdigest, 2440
Digest::MD5::md5, 2444
Digest::MD5::md5_base64, 2446
Digest::MD5::md5_hex, 2445
Digest::MD5::new, 2434
Digest::MD5::reset, 2447
Digest::SHA, 2450
pperl 0.6.2, 5515
pperl 0.6.3, 5513
Digest::SHA1, 2470
Digest::SHA1::add, 2472
Digest::SHA1::addfile, 2472
Digest::SHA1::b64digest, 2472
Digest::SHA1::digest, 2472
Digest::SHA1::hexdigest, 2472
Digest::SHA1::new, 2471
Digest::SHA1::sha1, 2473
Digest::SHA1::sha1_base64, 2473
Digest::SHA1::sha1_hex, 2473
Digest::SHA1::sha1_transform, 2473
Digest::SHA::add, 2458
Digest::SHA::add_bits, 2462
Digest::SHA::addfile, 2464
Digest::SHA::algorithm, 2458
Digest::SHA::dump, 2467
Digest::SHA::getstate, 2465
Digest::SHA::hashsize, 2459
Digest::SHA::hmac_sha1, 2456
Digest::SHA::hmac_sha1_base64, 2456
Digest::SHA::hmac_sha1_hex, 2456
Digest::SHA::hmac_sha224, 2456
Digest::SHA::hmac_sha224_base64, 2456
Digest::SHA::hmac_sha224_hex, 2456
Digest::SHA::hmac_sha256, 2456
Digest::SHA::hmac_sha256_base64, 2456
Digest::SHA::hmac_sha256_hex, 2456
Digest::SHA::hmac_sha384, 2456
Digest::SHA::hmac_sha384_base64, 2456
Digest::SHA::hmac_sha384_hex, 2456
Digest::SHA::hmac_sha512, 2456
Digest::SHA::hmac_sha512_base64, 2456
Digest::SHA::hmac_sha512_hex, 2456

Digest::SHA::hmac_sha512224, 2456
Digest::SHA::hmac_sha512224_base64, 2456
Digest::SHA::hmac_sha512224_hex, 2456
Digest::SHA::hmac_sha512256, 2456
Digest::SHA::hmac_sha512256_base64, 2456
Digest::SHA::hmac_sha512256_hex, 2456
Digest::SHA::load, 2468
Digest::SHA::new, 2460
Digest::SHA::newSHA, 2457
Digest::SHA::putstate, 2466
Digest::SHA::reset, 2461
Digest::SHA::sha1, 2454
Digest::SHA::sha1_base64, 2454
Digest::SHA::sha1_hex, 2454
Digest::SHA::sha224, 2454
Digest::SHA::sha224_base64, 2454
Digest::SHA::sha224_hex, 2454
Digest::SHA::sha256, 2454
Digest::SHA::sha256_base64, 2454
Digest::SHA::sha256_hex, 2454
Digest::SHA::sha384, 2454
Digest::SHA::sha384_base64, 2454
Digest::SHA::sha384_hex, 2454
Digest::SHA::sha512, 2454
Digest::SHA::sha512_base64, 2454
Digest::SHA::sha512_hex, 2454
Digest::SHA::sha512224, 2454
Digest::SHA::sha512224_base64, 2454
Digest::SHA::sha512224_hex, 2454
Digest::SHA::sha512256, 2454
Digest::SHA::sha512256_base64, 2454
Digest::SHA::sha512256_hex, 2454
directed edge, 449
DirHandle, 2475
dirname
  with s///, 359
disjunctive normal form, 614
dispatch table, 303, 372, 514, 582
dispell (*Peta::XS*), 4077
distroprefs, 637
dlclose (*Peta::FFI*), 4031
dlopen (*Peta::FFI*), 4022
DNF, 614
DNS
  name encoding, 547
  query packet, 546
do, 832
do {...} until
  pperl 0.6.3, 5513
do {...} while
  pperl 0.6.3, 5513
Docker
  debugging, 229
documentation
  guides, 14
  tutorials, 14
DOES, 184
dot product, 435
double encoding, 595
double fork, 54
double negation, 612
draw call, 473
draw order, 382, 442
drawing application, 424
-dt
  not supported, 33
dump, 837
Dump (*Devel::Peek*), 2422
dump (*Digest::SHA*), 2467
dump (*GDBM_File*), 2973
DumpArray (*Devel::Peek*), 2423
Dumpvalue, 2475
Dumpxs (*Data::Dumper*), 2413
dup2 (*POSIX*), 3936
duplicate lines
  removing, 244
DynaLoader, 2478
dynamic scope, 1535

E

-E, 10
  enabled features, 235
  pperl 0.6.1, 5516
-e, 10, 346
  multiple chunks, 235
each, 839
  walking a hash, 301
each_line, 321
ear clipping, 385, 455
ECMAScript
  regular expressions, 138
edge function, 440
elif, 290
elseif
  Lua, 332
  PHP, 303
  Tcl, 372
elsif, 290

Emacs
    debugging, 230
    regular expressions, 139
embedded code, 130
    execution frequency, 131
embedding
    C hosts, 33
empty match
    in substitution, 113
    unintended, 61
encapsulation, 176
Encode, 2512
    pperl 0.6.6, 5510
    pperl 0.6.8, 5509
    pperl 0.6.9, 5504
    pperl 0.6.12, 5502
    pperl 0.6.19, 5490
encode, 589
encode (*Encode::Unicode*), 2533
encode (*JSON::PP*), 3466
encode (*JSON::XS*), 3475
Encode module, 559, 589
Encode::Alias, 2514
Encode::Byte, 2516
Encode::CJKConstants, 2517
Encode::CN, 2517
Encode::CN::HZ, 2517
Encode::Config, 2517
Encode::EBCDIC, 2517
Encode::Encoder, 2518
Encode::Encoding, 2520
Encode::encodings, 2535
Encode::FB_*, 589
Encode::FB_CROAK, 600
Encode::find_encoding, 2535
Encode::GSM0338, 2524
Encode::Guess, 2525
Encode::JP, 2528
Encode::JP::H2Z, 2528
Encode::JP::JIS7, 2528
Encode::KR, 2528
Encode::KR::2022_KR, 2529
Encode::MIME::Header, 2529
Encode::MIME::Name, 2531
Encode::Symbol, 2531
Encode::TW, 2531
Encode::Unicode, 2532
Encode::Unicode::decode, 2533
Encode::Unicode::encode, 2533
Encode::Unicode::UTF7, 2532
Encode::utf8, 2535
Encode::XS, 2534
encode_base64 (*MIME::Base64*), 3538
encode_base64url (*MIME::Base64*), 3543
encode_json (*JSON::PP*), 3467
encode_qp (*MIME::QuotedPrint*), 3553
encoded_base64_length
        (*MIME::Base64*), 3541
encoding, 558, 588
    filehandle, 558, 590
    identifying the source, 596
    in one-liners, 264
    lossy, 589
encoding (*pragma*), 5309
encoding::warnings (*pragma*), 5317
encodings (*Encode*), 2535
END
    awk, 353
    in one-liners, 235, 268, 346, 353
endgrent, 843
endhostent, 845
endianness, 535
    detecting the host's, 537
    floating point, 537
    modifiers, 536
    native, 536
    on a group, 536
endnetent, 848
endprotoent, 850
endpwent, 851
endservent, 853
English, 2537
Env, 2537
eof, 855
    last line, 343
    resetting line numbers, 266
ephemeral port, 627
epoch time, 255
eq, 1452, 1454
    vs ==, 602
equal, 1452
equality, 1452
equals, 1452
equivalence, 609
equivalence class, 65
Errno, 2539
errno, 566
errno (*Peta::FFI*), 4026
error, 667
    Lua, 339
error handling, 565
    checklist, 567

escape sequence
    in patterns, 59
Euler angles, 439
EV, 2485
    pperl 0.6.19, 5489
    pperl 0.6.21, 5485
    timers, 39
EV::Async, 2493
EV::Child, 2494
EV::Embed, 2494
EV::IO, 2495
EV::io, 2508
EV::io_ns, 2508
EV::Loop, 2496
EV::loop, 2507
EV::once, 2511
EV::Periodic, 2500
EV::periodic, 2510
EV::periodic_ns, 2510
EV::run, 2507
EV::Signal, 2501
EV::Stat, 2502
EV::Stat::attr, 2503
EV::Stat::prev, 2503
EV::Stat::stat, 2503
EV::Timer, 2503
EV::timer, 2509
EV::Timer::again, 2504
EV::timer_ns, 2509
EV::Watcher, 2505
EV::Watcher::keepalive, 2507
eval, 858
    exception catching, 204, 295, 310, 324, 339, 567
    scope of, 1534
evalbytes, 864
even-odd rule, 384
Event, 2539
    pperl 0.6.19, 5489
    pperl 0.6.21, 5485
event loop, 37
event queue
    SDL2, 416
Event::Event, 2543
Event::Event::Dataful, 2544
Event::Event::Io, 2545
Event::generic, 2549
Event::generic::Source, 2550
Event::group, 2551
Event::idle, 2552
Event::io, 2553
Event::one_event, 2557
Event::signal, 2554
Event::timer, 2555
Event::var, 2556
Event::Watcher, 2545
Event::Watcher::Tied, 2548
exception, 204, 566, 667
exception class, 206
    requirements, 207
exception objects, 206, 567
exclusive or, 605
exec, 561, 868
    Tcl, 375
EXIF, 412
    writing tags, 413
EXISTS
    tied array, 278
    tied hash, 274
exists, 871
    avoiding autovivification, 572
    membership test, 287, 301, 316, 330, 355, 370
    nested data structure, 578
exit, 875
exit status, 205, 562
exp, 878
experimental (*pragma*), 5320
explode, 306
exponentiation, 1449
Exporter, 2557
    pperl 0.6.21, 5484
Exporter::Heavy, 2566
expr
    Tcl, 368
EXTEND, 278
extended bracketed character class, 66
extends, 308
extrusion, 455
    of a face, 509
ExtUtils::CBuilder, 2566
ExtUtils::Command, 2570
ExtUtils::Command::MM, 2573
ExtUtils::Embed, 2575
ExtUtils::HasCompiler, 2578
ExtUtils::Install, 2579
ExtUtils::Installed, 2583
ExtUtils::Liblist, 2586
ExtUtils::MakeMaker, 2616
    pperl 0.6.8, 5509
ExtUtils::MakeMaker::Config, 2649

ExtUtils::MakeMaker::Locale, 2649
ExtUtils::Manifest, 2652
ExtUtils::Miniperl, 2658
ExtUtils::MM, 2590
ExtUtils::MM_Any, 2590
ExtUtils::MM_Unix, 2606
ExtUtils::MY, 2616
ExtUtils::Packlist, 2660
ExtUtils::PL2Bat, 2658
ExtUtils::testlib, 2662

F

-F, 10, 349, 363
    delimiter forms, 267
    split pattern, 237
factorial, 254
false, 1486
false values, 289, 317, 332, 601
falsehood, 1486
fan triangulation, 451
fast_abs_path (*Cwd*), 2145
Fatal, 2811
fc, 598, 880
Fcntl, 2813
    pperl 0.6.9, 5506
fcntl, 883
Fcntl::S_IFMT, 2815
Fcntl::S_IMODE, 2815
Fcntl::S_ISBLK, 2816
Fcntl::S_ISCHR, 2816
Fcntl::S_ISDIR, 2816
Fcntl::S_ISFIFO, 2816
Fcntl::S_ISLNK, 2816
Fcntl::S_ISREG, 2816
Fcntl::S_ISSOCK, 2816
fd (*DB_File*), 2406
feature
    class, 169
    try, 204
feature (*pragma*), 5323
Feature::Compat::Try, 204
fesetround (*POSIX*), 3931
FETCH
    tied array, 278
    tied hash, 274
    tied scalar, 272
fetch (*Peta::FFI*), 4030
FETCHSIZE, 278
FFI, 19
    graphics libraries, 418
FFI::Build, 2663
FFI::Build::File::Base, 2668
FFI::Build::File::C, 2672
FFI::Build::File::CXX, 2673
FFI::Build::File::Library, 2674
FFI::Build::File::Object, 2676
FFI::Build::MM, 2677
FFI::Build::Platform, 2680
FFI::Build::Plugin, 2684
FFI::Build::PluginData, 2685
FFI::CheckLib, 2687
FFI::Platypus, 643, 2693
FFI::Platypus::API, 2695
FFI::Platypus::Buffer, 2700
FFI::Platypus::Bundle, 2706
FFI::Platypus::Closure, 2718
FFI::Platypus::Constant, 2720
FFI::Platypus::DL, 2724
FFI::Platypus::Function, 2727
FFI::Platypus::Internal, 2729
FFI::Platypus::Lang, 2731
FFI::Platypus::Lang::ASM, 2733
FFI::Platypus::Lang::C, 2734
FFI::Platypus::Lang::Win32, 2736
FFI::Platypus::Legacy, 2741
FFI::Platypus::Memory, 2742
FFI::Platypus::Record, 2745
FFI::Platypus::Record::Meta, 2752
FFI::Platypus::Record::TieArray, 2753
FFI::Platypus::ShareConfig, 2755
FFI::Platypus::Type, 2756
FFI::Platypus::Type::PointerSizeBuffer, 2780
FFI::Platypus::Type::StringArray, 2781
FFI::Platypus::Type::StringPointer, 2784
FFI::Platypus::Type::WideString, 2786
FFI::Platypus::TypeParser, 2792
FFI::Platypus::TypeParser::Version0, 2794
FFI::Platypus::TypeParser::Version1, 2796
FFI::Platypus::TypeParser::Version2, 2797
FFI::Probe, 2799
FFI::Probe::Runner, 2802
FFI::Probe::Runner::Builder, 2805
FFI::Probe::Runner::Result, 2808
FFI::Temp, 2810

fgets, 307
fibonacci (*Math::GMP*), 3680
field, 886
    :param, 294, 308, 323, 337
    :reader, 323
    attributes, 170
    default values, 171
    per-instance storage, 170
    scope rules, 171
fields (*pragma*), 5337
FIFO, 623
    blocking open, 624
file
    PHP, 307
File::Basename, 2817
File::Compare, 2820
File::Copy, 2820
File::DosGlob, 2823
    pperl 0.6.21, 5483
File::Fetch, 2824
File::Find, 2829
File::Glob, 2835
    pperl 0.5.6, 5519
    pperl 0.6.19, 5491
File::Glob::bsd_glob, 2836
File::Glob::csh_glob, 2837
File::Glob::GLOB_ERROR, 2838
File::GlobMapper, 2839
File::Map, 2844
    pperl 0.5.6, 5519
File::Map::advise, 2853
File::Map::map_anonymous, 2849
File::Map::map_file, 2847
File::Map::map_handle, 2848
File::Map::protect, 2854
File::Map::remap, 2852
File::Map::sync, 2851
File::Map::sys_map, 2850
File::Map::unmap, 2852
File::Path, 2855
File::ShareDir, 2865
File::Spec, 2871
    pperl 0.6.0, 5518
File::Spec::AmigaOS, 2876
File::Spec::Cygwin, 2877
File::Spec::Epoc, 2878
File::Spec::Functions, 2878
File::Spec::Mac, 2880
File::Spec::OS2, 2886
File::Spec::Unix, 2887
File::Spec::VMS, 2891
File::Spec::Win32, 2893
File::stat, 2915
File::Temp, 2895
    pperl 0.5.6, 5520
    pperl 0.6.5, 5511
    pperl 0.6.9, 5506
    pperl 0.6.11, 5502
    pperl 0.6.21, 5484
File::Which, 2911
FileCache, 2917
FileHandle, 2918
filehandle, 555, 1514
    encoding, 558
    scalar, 563
fileno, 892
File.open, 321
filetest, 364, 1483
filetest (*pragma*), 5339
Filter::cpp, 2940
Filter::decrypt, 2941
Filter::exec, 2942
Filter::m4, 2943
Filter::sh, 2944
Filter::Simple, 2921
Filter::tee, 2945
Filter::Util::Call, 2931
Filter::Util::Call::filter_add, 2937
Filter::Util::Call::filter_del, 2935
Filter::Util::Call::filter_read, 2933
Filter::Util::Call::filter_read_exact, 2936
Filter::Util::Call::real_import, 2934
Filter::Util::Exec, 2939
Filter::Util::Exec::filter_add, 2940
filter_add (*Filter::Util::Call*), 2937
filter_add (*Filter::Util::Exec*), 2940
filter_del (*Filter::Util::Call*), 2935
filter_fetch_key (*DB_File*), 2408
filter_fetch_value (*DB_File*), 2410
filter_json_object (*JSON::PP*), 3470
filter_json_single_key_object (*JSON::PP*), 3470
filter_read (*Filter::Util::Call*), 2933
filter_read_exact (*Filter::Util::Call*), 2936
filter_store_key (*DB_File*), 2409

filter_store_value (*DB_File*), 2411
finally, 204, 324, 895
find_encoding (*Encode*), 2535
FindBin, 2946
first (*List::Util*), 3507
FIRSTKEY, 275
fixed timestep, 468
    in a game, 529
    step cap, 468
fixed-width records, 532
    parsing with unpack, 538
flat shading, 446
    unshared vertices, 502
flip-flop operator, 241, 343, 352, 360
float, 1453
floating point, 1453
floating-point, 1453
floating-point render target, 481
flock, 556, 900
flood fill, 387
    in GD, 392
    mesh faces, 509
floor division, 286
flowing (*PDL*), 3902
FLT_ROUNDS (*POSIX*), 3930
flush, 557, 675
flush_cache (*Memoize*), 3702
fly camera, 460
fma (*POSIX*), 3928
fn
    PHP, 305
FNR, 348
fog, 526
fopen, 307
for, 1536
for (*k in arr*), 355
for f in *, 364
foreach, 1536
    PHP, 303
    Tcl, 369
foreshortening, 439
fork, 561, 565, 903
    and auto-parallelism, 622
    process-level concurrency, 52
    with pipe, 622
    work crew, 53
forking server, 626
formal parameter, 1541
format, 907
formline, 911
FourCC, 554
fpathconf (*POSIX*), 3943
fpclassify (*POSIX*), 3930
fractal Brownian motion, 488
fragment shader, 471
frame loop, 416, 466
    frame rate, 430
    responsiveness, 517
free (*Peta::FFI*), 4034
freeflowing regex, 127
FreeType, 395
frexp (*POSIX*), 3927
from_b64string (*Data::UUID*), 2418
front face, 480
frustum culling, 461
FS, 349
f-string, 286
function, 1523
    Lua, 333
    PHP, 304
function (*Peta::FFI*), 4025
functional completeness, 613

G

game
    3D, 518
game loop, 427
gather and scatter, 492
Gaussian blur, 389
GD, 2947
    alpha, 391
    alphaBlending, 391
    anti-aliasing, 394
    colorAllocate, 390
    colorResolve, 390
    colours, 390
    copyMerge, 400
    copyResampled, 400
    cropping, 402
    drawing primitives, 392
    filters, 404
    flipping, 402
    pperl 0.6.16, 5497
    pperl 0.6.19, 5491
    pperl 0.6.22, 5480
    raster drawing, 390
    rotating, 402
    saving and loading, 397
    scaling, 400
    stringFT, 395
    text, 395
    transparency, 399

versus Imager, 408
GD::Graph, 413
GD::Group, 2950
GD::Image, 2951
GD::Polygon, 392, 2952
GD::Polyline, 2952
GD::Simple, 406, 2958
GD::supportsFileType, 2966
GDBM_File, 2966
pperl 0.6.21, 5483
GDBM_File::dump, 2973
GDBM_File::load, 2974
GDBM_File::recover, 2972
GDBM_File::reorganize, 2971
ge, 1454
gensym, 621
get (*DB_File*), 2404
get (*Peta::FFI::LMDB::Txn*), 4008
get (*Template::Stash::XS*), 4375
get_code_info (*Class::MOP*), 2031
get_code_info (*Sub::Identify*), 4265
get_code_location (*Sub::Identify*), 4265
get_dataref (*PDL*), 3900
get_isarev (*mro*), 5352
get_linear_isa (*mro*), 5351
get_mro (*mro*), 5351
get_pkg_gen (*mro*), 5353
get_str_gmp (*Math::GMP*), 3676
get_view (*Peta::FFI::LMDB::Txn*), 4009
getaddrinfo (*Socket*), 4254
getattr (*POSIX::Termios*), 3925
GETC, 281
getc, 915
getcc (*POSIX::Termios*), 3926
GetControlChars (*Term::ReadKey*), 4398
getcwd (*Cwd*), 2142
getdata (*Peta::XS*), 4078
getgrent, 917
getgrgid, 920
getgrnam, 923
gethostbyaddr, 926
gethostbyname, 929
gethostent, 932
getline, 352
getlogin, 935
getnameinfo (*Socket*), 4255
getnetbyaddr, 937
getnetbyname, 939
getnetent, 942
Getopt::Long, 2975
Getopt::Long::Parser, 2995
Getopt::Std, 2996
getpeername, 944
getpgrp, 947
getppid, 950
getpriority, 952
getprotobyname, 954
getprotobynumber, 957
getprotoent, 959
getpwent, 962
getpwnam, 965
getpwuid, 968
gets, 373
getservbyname, 972
getservbyport, 974
getservent, 978
getsockname, 980
discovering the port, 627
getsockopt, 983
GetSpeed (*Term::ReadKey*), 4398
getstate (*Digest::SHA*), 2465
GetTermSizeGWINSZ (*Term::ReadKey*), 4397
GIF header, 550
gimbal lock, 439
GLB, 455
glob, 364, 561, 985
pperl 0.6.9, 5506
GLOB_ERROR (*File::Glob*), 2838
global destruction, 178
global variable
Ruby, 313
glTF, 455
accessors, 455
gmp_copy (*Math::GMP*), 3682
gmp_tstbit (*Math::GMP*), 3682
gmtime, 989
gnuplot, 415
Go regexp, 140
goto, 992
Gouraud shading, 446
GPU, 463
detecting, 463
when it pays, 494
GPU adapter, 463
grapheme cluster, 116, 598
counting, 599
reversing a string, 599
graphics
2D, 377
3D, 433
greatest common divisor, 254

greedy quantifier, 78
grep, 996, 1538
  as array_filter, 300
  as filtering comprehension, 287
  compiled, 52
  JIT compilation, 160
  pperl 0.6.3, 5513
  regular expressions, 140
grep -n, 360
grouping, 84
gsub
  awk, 351
  Ruby, 321
gt, 1454
Gtk3, 417
Guard, 2997
  pperl 0.6.19, 5489
Guard::scope_guard, 2999
Guttman-Rosler transform, 156
GV (*B::CV*), 1810

H

-h, 10
half-edge structure, 451
half-pixel offset, 379
  rasterisation, 440
halfway vector, 445
handedness, 435
handle_type (*Peta::FFI*), 4027
hard edge, 450
HARNESS_PERL_SWITCHES, 231
Hash
  Ruby, 316
hash, 1493
  order, 301, 330
  scalar context, 1499
  slice, 1497
hash (*B*), 1856
hash of arrays, 574
hash of hashes, 576
Hash::Util, 3031
  pperl 0.6.6, 5510
Hash::Util::all_keys, 3041
Hash::Util::bucket_array, 3044
Hash::Util::bucket_info, 3044
Hash::Util::bucket_ratio, 3044
Hash::Util::bucket_stats, 3045
Hash::Util::bucket_stats_formatted, 3045
Hash::Util::FieldHash, 3039
Hash::Util::FieldHash::id, 3040
Hash::Util::FieldHash::id_2obj, 3040
Hash::Util::FieldHash::register, 3041
Hash::Util::hash_locked, 3049
Hash::Util::hash_seed, 3042
Hash::Util::hash_traversal_mask, 3043
Hash::Util::hash_unlocked, 3049
Hash::Util::hash_value, 3043
Hash::Util::hidden_keys, 3049
Hash::Util::hidden_ref_keys, 3042
Hash::Util::hv_store, 3042
Hash::Util::legal_keys, 3049
Hash::Util::lock_hash, 3048
Hash::Util::lock_hash_recurse, 3048
Hash::Util::lock_keys, 3046
Hash::Util::lock_ref_keys, 3047
Hash::Util::lock_ref_keys_plus, 3047
Hash::Util::lock_value, 3047
Hash::Util::num_buckets, 3045
Hash::Util::unlock_keys, 3046
Hash::Util::unlock_ref_keys, 3047
Hash::Util::unlock_value, 3048
Hash::Util::used_buckets, 3045
hash_locked (*Hash::Util*), 3049
hash_seed (*Hash::Util*), 3042
hash_traversal_mask (*Hash::Util*), 3043
hash_unlocked (*Hash::Util*), 3049
hash_value (*Hash::Util*), 3043
hashsize (*Digest::SHA*), 2459
have_gpu (*Peta::WebGPU*), 4069
hdrcpy (*PDL*), 3904
head (*List::Util*), 3490
header_files (*Config*), 2084
headless rendering, 464
  testing a game, 530
heightfield, 492
here-doc, 360
hex, 999
hex dump, 257
hex string, 540
hexdigest (*Digest::MD5*), 2440
hidden_keys (*Hash::Util*), 3049
hidden_ref_keys (*Hash::Util*), 3042
history expansion, 263
hmac_sha1 (*Digest::SHA*), 2456
hold space, 345
homogeneous coordinates, 436
homograph attack, 119

hot loop
  checklist, 161
HSV, 380
  conversion to RGB, 380
HTML::Entities, 2999
HTML::Filter, 3002
HTML::HeadParser, 3003
HTML::LinkExtor, 3005
HTML::Parser, 3007
  pperl 0.6.21, 5483
HTML::PullParser, 3009
HTML::Tagset, 3011
HTML::TokeParser, 3014
HTTP::Parser::XS, 3018
  pperl 0.6.9, 5506
HTTP::Tiny, 3021
hv_store (*Hash::Util*), 3042

I

-I, 10
-i, 10, 352
  in-place editing, 237
  pitfalls, 265
  versus sed, 345
I/O
  errors, 565
  opening files, 555
I18N::Collate, 3050
I18N::Langinfo, 3083
I18N::Langinfo::langinfo, 3084
I18N::LangTags, 3051
I18N::LangTags::Detect, 3060
I18N::LangTags::List, 3061
id (*Hash::Util::FieldHash*), 3040
id_2obj (*Hash::Util::FieldHash*), 3040
IEEE 754, 537
if (*pragma*), 5341
IFS, 358
image formats
  lossy and lossless, 397
image metadata, 412
  writing, 413
Image::ExifTool, 413
Image::Info, 412
Image::Magick, 409
ImageMagick, 409
Imager, 3406
  colours, 387
  drawing primitives, 387
  filters, 389
  loading and saving, 388
  pperl 0.6.16, 5497
  pperl 0.6.21, 5484
  pperl 0.6.22, 5477
  raster drawing, 387
  scaling, 388
Imager::Color, 387, 3414
Imager::Color::Float, 3418
Imager::Color::Table, 3420
Imager::Expr, 3421
Imager::Expr::Assem, 3423
Imager::Fill, 3425
Imager::Font, 3429
Imager::Font::BBox, 3439
Imager::Font::FreeType2, 3442
Imager::Font::FT2, 3441
Imager::Font::Test, 3443
Imager::Font::Wrap, 3443
Imager::Fountain, 3446
Imager::Matrix2d, 3449
Imager::Regops, 3452
Imager::Test, 3452
Imager::Transform, 3455
Imager::TrimColorList, 3457
immediate mode, 416
implication, 608
implode, 308
import, 1001
  compile time versus runtime, 465
  installing methods, 186
  Python, 293
in
  awk, 355
  Python, 287
in_array, 308
INADDR_ANY, 625
INADDR_LOOPBACK, 625
include_path, 308
incr, 372
incr_parse (*JSON::PP*), 3471
incr_parse (*JSON::XS*), 3476
increment, 1482
index, 1005
index buffer, 474
  16-bit indices, 474
indexed triangle mesh, 448
inet_aton, 254, 625
inet_aton (*Socket*), 4245
inet_ntoa, 254, 625
inet_ntoa (*Socket*), 4246
inet_ntop (*Socket*), 4248
inet_pton (*Socket*), 4247

inheritance
  @ISA, 177
  class feature, 173
  mixed class and bless, 184
INIT
  with the daemon, 163
initialize
  Ruby, 323
inline modifier, 105
in-memory filehandle, 563
  layers, 564
  limitations, 565
  reading, 564
  writing, 563
inner (*PDL*), 3899
in-place editing, 237, 245, 345, 352, 360, 375
  backups, 265
input_line_number, 676
input_record_separator, 673
inside test
  triangle, 440
install (*re*), 5396
instance variable
  Ruby, 313
instancing, 481
  repeated mesh, 502
  whole frame in one draw, 524
int, 1007
  integer division, 286
intdiv, 300
integer (*pragma*), 5342
integer division
  Ruby, 314
--interactive
  pperl 0.6.6, 5510
interactive mode, 645
interpolation, 1516, 1517
  array, 1521
  barycentric, 442
  coming from Lua, 328
  coming from PHP, 299
  coming from Python, 286
  coming from Ruby, 314
  coming from Tcl, 368
  in patterns, 60
  perspective-correct, 443
interpreter threads, 41
interprocess communication, 619
  choosing a channel, 620
intify (*Math::GMP*), 3677
invalidate_all_method_caches (*mro*), 5352
inverse transpose matrix, 482
invocant, 178
IO, 3084
io (*EV*), 2508
IO::Compress, 3085
IO::Compress::Base, 3086
IO::Compress::Bzip2, 3087
IO::Compress::Deflate, 3100
IO::Compress::Gzip, 3115
IO::Compress::RawDeflate, 3132
IO::Compress::Zip, 3147
IO::Dir, 3170
IO::File, 3171
IO::Handle, 3173
  pperl 0.6.9, 5507
IO::Pipe, 3179
IO::Poll, 3180
IO::Pty, 3182
IO::Pty::pty_allocate, 3185
IO::Seekable, 3185
IO::Select, 3187
IO::Socket, 3189
IO::Socket::INET, 3200
IO::Socket::IP, 3202
IO::Socket::SSL, 3212
  pperl 0.6.21, 5483
IO::Socket::SSL::Intercept, 3250
IO::Socket::SSL::PublicSuffix, 3253
IO::Socket::SSL::Utils, 3255
IO::Socket::UNIX, 3260
IO::Tty, 3262
IO::Tty::Constant, 3263
IO::Tty::pack_winsize, 3280
IO::Tty::ttyname, 3279
IO::Uncompress::AnyInflate, 3280
IO::Uncompress::AnyUncompress, 3295
IO::Uncompress::Base, 3309
IO::Uncompress::Bunzip2, 3310
IO::Uncompress::Gunzip, 3323
IO::Uncompress::Inflate, 3337
IO::Uncompress::RawInflate, 3351
IO::Uncompress::Unzip, 3365
IO::Zlib, 3380
ioctl, 1010
io.open, 336
IP address
  converting to integer, 254
  matching, 67
ipairs, 332

IPC, 619
IPC::Cmd, 3385
IPC::Msg, 3393
IPC::Open2, 562, 621, 3395
IPC::Open3, 562, 621, 3397
IPC::Semaphore, 3400
IPC::SharedMem, 3402
IPC::SysV, 3404
  pperl 0.6.5, 5511
IPC::SysV::shmat, 3406
IPC_CREAT, 631
IPC_EXCL, 631
IPC_PRIVATE, 631
IPC_RMID, 631
ipcs, 632
ipps, 645
  $ans, 646
  :jit, 649
  :programs, 649, 652
  :time, 649
  :vars, 649
  autorun, 652
  commands, 649
  history, 646
  history file, 652
  limits, 655
  local, 655
  macros, 652
  match variables, 655
  output without a newline, 647
  rc.pl, 652
  settings, 652
  Tab completion, 646
  values, 646
is_bool (*JSON::PP*), 3471
is_class_loaded (*Class::Load::XS*), 2029
is_cow (*B::COW*), 1808
is_lax (*version*), 5440
is_perfect_power (*Math::GMP*), 3683
is_regexp (*re*), 5398
is_strict (*version*), 5441
is_stub (*Class::MOP::Method*), 2030
is_sub_constant (*Sub::Identify*), 4265
is_universal (*mro*), 5352
isa, 1014
  class membership, 183
  exception dispatch, 206
isbig (*Scalar::Util::Numeric*), 4234
isfloat (*Scalar::Util::Numeric*), 4235
isinf (*Scalar::Util::Numeric*), 4235
isint (*Scalar::Util::Numeric*), 4234
isnum (*Scalar::Util::Numeric*), 4233
isset, 301
IsSTD (*B::IO*), 1837
isuv (*Scalar::Util::Numeric*), 4234
isweak, 586
ithreads, 41
  alternatives, 50
  porting to coroutines, 40
  porting to pperl, 50
  pperl status, 41
  shared-nothing model, 42

J

java.util.regex, 138
JIT, 3, **5527**
  checking at the prompt, 649
  declined loops, 158
  log, 149
  loop shape, 158
  string operations, 159
  subroutine calls in loops, 154
JIT compilation, 16
  abort and replay, 18
  covered constructs, 17
  diagnostics, 18
  guards, 18
  integer and float modes, 18
  limitations, 19
  performance, 19
jit_counts (*Config*), 2082
--jit-stats, 11, 18
  JIT counters, 149
jn (*POSIX*), 3929
join, 358, 1017
  Tcl, 374
JSON, 257
JSON::PP, 3460
  in one-liners, 246, 257
JSON::PP::ascii, 3468
JSON::PP::Boolean, 3466
JSON::PP::boolean_values, 3469
JSON::PP::core_bools, 3470
JSON::PP::decode, 3467
JSON::PP::decode_json, 3468
JSON::PP::decode_prefix, 3467
JSON::PP::encode, 3466
JSON::PP::encode_json, 3467
JSON::PP::filter_json_object, 3470
JSON::PP::filter_json_single_key_object, 3470
JSON::PP::incr_parse, 3471

JSON::PP::is_bool, 3471
JSON::PP::max_depth, 3469
JSON::PP::new, 3468
JSON::PP::PP_encode_json, 3466
JSON::PP::sort_by, 3471
JSON::XS, 3472
JSON::XS::allow_blessed, 3475
JSON::XS::allow_nonref, 3475
JSON::XS::allow_tags, 3475
JSON::XS::allow_unknown, 3475
JSON::XS::ascii, 3475
JSON::XS::Boolean, 3474
JSON::XS::canonical, 3475
JSON::XS::convert_blessed, 3475
JSON::XS::decode, 3475
JSON::XS::decode_json, 3476
JSON::XS::decode_prefix, 3476
JSON::XS::encode, 3475
JSON::XS::incr_parse, 3476
JSON::XS::indent, 3475
JSON::XS::latin1, 3475
JSON::XS::pretty, 3475
JSON::XS::relaxed, 3475
JSON::XS::shrink, 3475
JSON::XS::space_after, 3475
JSON::XS::space_before, 3475
JSON::XS::utf8, 3475

K

keepalive (*EV::Watcher*), 2507
keep-left, 74
KELVIN SIGN, 117
key?, 316
key repeat, 429
keyboard state, 417, 429, 522
keys, 1019
keysym, 429
keyword argument
    Python, 290
    Ruby, 319
kill, 1022
kubectl
    debugging, 229

L

-l, 10, 353
    line-end handling, 236
    with -0, 266
lambda
    Python, 291
    Ruby, 320
Lambert term, 445
langinfo (*I18N::Langinfo*), 3084
lappend, 369
last, 332, 1025
    stopping input, 344
launch.json, 228
layered I/O, 558
lazy initialisation, 189, 618
lc, 1029
lcfirst, 1032
le, 1454
leaning toothpick syndrome, 56
left shift, 1460
leftmost match, 60
    and greediness, 79
legal_keys (*Hash::Util*), 3049
legendre (*Math::GMP*), 3679
len, 287
length, 1035
    characters vs bytes, 587
    undecoded input, 596
length-prefixed data, 542
    labels, 547
less (*pragma*), 5344
lexer
    with /gc, 103
lexical, 1534
lexical filehandle, 555
lexical scope, 1534
    autodie, 566
lexical variable, 1534
lib (*pragma*), 5345
lib_file (*Config*), 2080
lib_files (*Config*), 2081
libcairo, 420
libffi, 19, 418
libgd, 390
lighting, 444
    terrain relief, 493
lindex, 369
line drawing, 382
line endings
    converting, 242
    normalising, 598
line number, 676
linear light, 467
    colours picked by eye, 483
linear-time matching, 140
lines
    Lua, 336
link, 1038
list, 1499

Python, 287
Tcl, 369
list comprehension, 287
list context, 288, 302, 316, 331, 1498
list operator, 1481, **5527**
List::MoreUtils::XS, 3480
List::Util, 3481
in one-liners, 244
pperl 0.6.0, 5517
pperl 0.6.9, 5507
pperl 0.6.12, 5502
pperl 0.6.15, 5499
pperl 0.6.19, 5491
pperl 0.6.21, 5485
shuffle, 253
sum, min, max, 252
List::Util::all, 3508
List::Util::any, 3507
List::Util::first, 3507
List::Util::head, 3490
List::Util::max, 3489
List::Util::maxstr, 3496
List::Util::mesh, 3502
List::Util::mesh_shortest, 3503
List::Util::min, 3488
List::Util::minstr, 3496
List::Util::none, 3509
List::Util::notall, 3509
List::Util::pairfirst, 3501
List::Util::pairgrep, 3501
List::Util::pairkeys, 3499
List::Util::pairmap, 3500
List::Util::pairs, 3497
List::Util::pairvalues, 3499
List::Util::product, 3490
List::Util::reduce, 3505
List::Util::reductions, 3506
List::Util::sample, 3492
List::Util::shuffle, 3492
List::Util::sum, 3486
List::Util::sum0, 3487
List::Util::tail, 3491
List::Util::uniq, 3493
List::Util::uniqint, 3495
List::Util::uniqnum, 3494
List::Util::uniqstr, 3494
List::Util::unpairs, 3498
List::Util::XS, 3486
List::Util::zip, 3504
List::Util::zip_shortest, 3504
listen, 1040
little-endian, 535
llength, 369
LMDB_File, 3477
pperl 0.6.21, 5483
load (*Digest::SHA*), 2468
load (*GDBM_File*), 2974
local, 557, 1042, 1535
in coroutines, 40
Lua, 328
shell, 365
special variable, 1535
local transform, 459
local_patches (*Config*), 2083
locale
in regular expressions, 118
locale (*pragma*), 5347
Locale::Maketext, 3510
Locale::Maketext::Guts, 3532
Locale::Maketext::GutsLoader, 3533
Locale::Maketext::Simple, 3533
localeconv (*POSIX*), 3943
localtime, 1047
list context, 255
lock, 1050
advisory mutex, 47
without thread support, 41
lock order, 48
lock_hash (*Hash::Util*), 3048
lock_hash_recurse (*Hash::Util*), 3048
lock_keys (*Hash::Util*), 3046
lock_ref_keys (*Hash::Util*), 3047
lock_ref_keys_plus (*Hash::Util*), 3047
lock_value (*Hash::Util*), 3047
log, 1053
Log::Any, 203
Log::Dispatch, 203
Log::Log4perl, 203
logging, 203
logical and, 1457
longest-leftmost match, 94
lookahead assertion, 72, 249, 617
defeated by backtracking, 73
lookaround
combining assertions, 75, 617
portability, 75
verb-style aliases, 74
lookbehind assertion, 74, 249
variable-length, 74
loop (*EV*), 2507
loop unrolling, 152
loop-invariant hoisting, 151

lsearch, 369
lseek (*POSIX*), 3936
lsort, 369
lstat, 1056
lt, 1454
Lua
    common mistakes in Perl, 340
    moving to Perl, 327
Lua pattern, 335
LuaRocks, 337
lvalue, 1551

M

-M, 10
    loading modules, 237
    pperl 0.6.2, 5515
m, 1466
-m, 10
    loading modules without imports, 237
    pperl 0.6.2, 5515
m//, 1059
    delimiters, 56
    single-quote delimiters, 56
magic (*Peta::XS*), 4073
magic number, 552
main_root (*B*), 1855
manifold mesh, 449
map, 1063
    as array_map, 300
    as list comprehension, 287
    building a hash, 296, 311, 325, 339
    compiled, 52
    JIT compilation, 160
    pperl 0.6.3, 5513
map_anonymous (*File::Map*), 2849
map_file (*File::Map*), 2847
map_handle (*File::Map*), 2848
match
    awk, 351
    in list context, 85
match (*Peta::FFI::PCRE2*), 4015
match offsets, 92
matches null string many times, 83
material implication, 608
Math::BigFloat, 3555
Math::BigInt, 3573
    in one-liners, 252
Math::BigInt::Calc, 3629
Math::BigInt::FastCalc, 3630
    pperl 0.6.21, 5483
Math::BigInt::Lib, 3631
Math::BigRat, 3641
Math::Complex, 3660
Math::GMP, 3669
    pperl 0.6.13, 5501
    pperl 0.6.19, 5491
    via libgmp, 27
Math::GMP::_gmp_build_version, 3676
Math::GMP::_gmp_lib_version, 3676
Math::GMP::add_ui_gmp, 3677
Math::GMP::bdiv, 3679
Math::GMP::bfac, 3681
Math::GMP::bgcd, 3680
Math::GMP::blcm, 3680
Math::GMP::blshift, 3681
Math::GMP::bmodinv, 3679
Math::GMP::bmulf, 3678
Math::GMP::bnok, 3681
Math::GMP::broot, 3682
Math::GMP::brootrem, 3682
Math::GMP::brshift, 3681
Math::GMP::bsqrt, 3683
Math::GMP::fibonacci, 3680
Math::GMP::get_str_gmp, 3676
Math::GMP::gmp_copy, 3682
Math::GMP::gmp_tstbit, 3682
Math::GMP::intify, 3677
Math::GMP::is_perfect_power, 3683
Math::GMP::legendre, 3679
Math::GMP::mmod_gmp, 3678
Math::GMP::new, 3683
Math::GMP::op_numify, 3678
Math::GMP::powm_gmp, 3677
Math::GMP::probab_prime, 3679
Math::GMP::sizeinbase_gmp, 3677
Math::Prime::Util, 254
Math::Random::MT, 3684
Math::Trig, 3685
math.floor, 329
matrix, 571
    4x4, 436
matrix multiplication
    order, 437
max (*List::Util*), 3489
max_depth (*JSON::PP*), 3469
maxstr (*List::Util*), 3496
maybe::next::method, 5354
md5 (*Digest::MD5*), 2444
md5_base64 (*Digest::MD5*), 2446
md5_hex (*Digest::MD5*), 2445
Memoize, 3695

pperl 0.6.18, 5492
pperl 0.6.19, 5490
pperl 0.6.21, 5482
memoize (*Memoize*), 3702
Memoize::AnyDBM_File, 3696
Memoize::Expire, 3697
Memoize::flush_cache, 3702
Memoize::memoize, 3702
Memoize::NDBM_File, 3701
Memoize::SDBM_File, 3701
Memoize::Storable, 3701
Memoize::unmemoize, 3702
memory leak
reference cycle, 584
mesh, 448
holes, 449
mesh (*List::Util*), 3502
mesh_shortest (*List::Util*), 3503
metacharacter, 58
escaping, 58
metamethod, 337
metatable, 337
method, 1066
signatures, 172
method modifier, 187
method resolution, 181
method resolution order, 181
midpoint circle algorithm, 383
migrating to Perl, 283
MIME::Base64, 257, 3536
pperl 0.6.2, 5515
pperl 0.6.3, 5513
MIME::Base64::decode_base64, 3539
MIME::Base64::decode_base64url, 3544
MIME::Base64::decoded_base64_length, 3542
MIME::Base64::encode_base64, 3538
MIME::Base64::encode_base64url, 3543
MIME::Base64::encoded_base64_length, 3541
MIME::Charset, 3545
MIME::QuotedPrint, 3552
pperl 0.6.2, 5515
MIME::QuotedPrint::decode_qp, 3554
MIME::QuotedPrint::encode_qp, 3553
min (*List::Util*), 3488
minstr (*List::Util*), 3496
missing decode, 596
mixin, 186
mkdir, 1070
mkfifo, 623
mktime, 255
mktime (*POSIX*), 3939
mmod_gmp (*Math::GMP*), 3678
mod, 1449
model file format, 451
model matrix
order of operations, 503
model space, 437
modelling operation, 455
modelling tool, 496
modf (*POSIX*), 3927
modifier
regular expression, 100
module loading
tracing, 226
Module::CoreList, 3703
Module::CoreList::Utils, 3707
Module::Load, 3709
Module::Load::Conditional, 3713
Module::Loaded, 3717
Module::Metadata, 3718
modulo, 1449
mojibake, 264, 558, 596
Möller-Trumbore algorithm, 458, 507
Moo, 180
migrating to class, 194
Moo::Role, 187
Moose, 180, 3724
handles, 188
migrating to class, 194
Moose::Role, 187
mouse
click versus drag, 514
mouse drag, 426
-Mre=debug, 226
mro
dfs and c3, 181
pperl 0.6.9, 5507
mro (*pragma*), 5348
mro::get_isarev, 5352
mro::get_linear_isa, 5351
mro::get_mro, 5351
mro::get_pkg_gen, 5353
mro::invalidate_all_method_caches, 5352
mro::is_universal, 5352
mro::set_mro, 5351
msgctl, 1073
msgget, 1075

msgrcv, 1078
msgsnd, 1081
mstat (*Devel::Peek*), 2424
MTL, 451
MULTICALL, 31
multidimensional array, 571
multi-line matching, 101
multiple inheritance, 181
  drawbacks, 185
multiple return values, 331
mutex, 47
my, 1084, 1534
  fresh variable per iteration, 574
  versus implicit globals, 328
  versus shell variables, 365
my method, 172
my sub, 1525
myconfig (*Config*), 2077

N

-n, 10
  expanded loop, 346, 363
  implicit read loop, 235
  versus awk, 348
  versus sed, 342
{n,m}, 77
NAME (*B::GV*), 1834
NAME (*B::HV*), 1835
NAME_HEK (*B::CV*), 1811
named argument
  PHP, 304
named capture, 86, 248, 292, 306, 321, 371
  Python syntax, 87
named character, 594
named pipe, 623
named unary operator, **5527**
namespace eval, 376
nan (*POSIX*), 3928
NAND, 607
  De Morgan, 612
  functional completeness, 613
native binding, 27, 420
  methods, 423
  object lifetime, 422
  opaque handle, 422
  symbol resolution, 421
native module, 638
  lib_file, 639
  version mismatch, 640
  version probe, 638
native modules, 3, 29
  list, 29
  load priority, 32
  loading, 29
  performance, 31
NDBM_File, 3724
  pperl 0.6.21, 5483
NDJSON, 257
ne, 1454
near plane, 443
nearest-neighbour scaling, 400
negation, 605
  arithmetic, 1449
  binary, 1459
Neovim
  debugging, 230
nested data structure, 568
  literal, 580
  mixed, 576
nested quantifiers, 123
Net::Cmd, 3733
Net::Config, 3738
Net::Domain, 3742
Net::FTP, 3743
Net::FTP::dataconn, 3753
Net::hostent, 3795
Net::netent, 3796
Net::Netrc, 3763
Net::NNTP, 3755
Net::Ping, 3770
Net::POP3, 3766
Net::protoent, 3798
Net::servent, 3799
Net::SMTP, 3779
Net::SSLeay, 3787
  pperl 0.6.21, 5483
Net::SSLeay::Handle, 3789
Net::Time, 3793
network byte order, 535
new (*Data::UUID*), 2417
new (*Digest::MD5*), 2434
new (*Digest::SHA*), 2460
new (*Digest::SHA1*), 2471
new (*JSON::PP*), 3468
new (*Math::GMP*), 3683
new (*Package::Stash::XS*), 3948
new (*Peta::FFI::LMDB::Env*), 4004
new (*Peta::FFI::PCRE2*), 4015
new (*POSIX::SigSet*), 3923
new (*POSIX::Termios*), 3925
newline translation, 559

newSHA (*Digest::SHA*), 2457
NEXT, 3726
next, 332, 1089
    awk, 354
    skipping a line, 344
next::can, 5353
next::method, 182, 5353
NEXTKEY, 275
NF, 349
NFA versus DFA, 145
nice (*POSIX*), 3941
nil
    Lua, 332
    Ruby, 317
no, 1092
--no-jit, 11, 18
    isolating the JIT, 150
non_bincompat_options (*Config*), 2084
non-capturing group, 86
non-destructive substitution, 104, 248
None, 289
none (*List::Util*), 3509
non-greedy quantifier, 80, 247
    versus negated character class, 81
NONINTERACTIVE_TESTING, 637
--no-parallel, 11, 35
    isolating the parallelizer, 150
NOR, 607
    De Morgan, 612
    functional completeness, 613
--no-regex-jit, 11
    pperl 0.6.16, 5497
normal, 444
    area and angle weighting, 449
    averaging, 449
    heightfield, 492
    non-uniform scale, 482
    per face, 444
    per vertex, 444
normal map, 447
normalised device coordinates, 437
normalization, 600
not, 1458
    return value, 605
    vs !, 603
notall (*List::Util*), 3509
NR, 348
NR==FNR, 355
NUL-delimited records, 245
null, 1486
    PHP, 303
NUL-terminated string, 538
num_buckets (*Hash::Util*), 3045
number, 1486
    literal, 1502
nybble order, 540

O

O, 3801
O_EXCL, 556
O_NONBLOCK, 556
OBJ, 451
    1-based and negative indices, 451
    index offsets across objects, 512
    writing a scene, 512
object
    blessed reference, 175
    hash-based, 176
object recycling, 518
Object::Pad, 180
object_2svref (*B::SV*), 1849
object-oriented programming, 167
    when to use, 168
observer pattern, 585
oct, 1095
ODBM_File, 3802
    pperl 0.6.21, 5483
offscreen canvas, 424
offscreen drawing, 416
OFS, 349
ok (*pragma*), 5354
once (*EV*), 2511
one_event (*Event*), 2557
one-liner, 233, 296, 311, 325, 339
    as shell alias, 258
    awk equivalents, 353
    context lines, 244
    converting to a script, 262
    counting lines, 243
    filtering lines, 240
    in a shell pipeline, 363
    passing arguments, 259
    quoting, 263
    reductions, 243
    sed equivalents, 346
    switches, 235
op tree, 3
op_numify (*Math::GMP*), 3678

opaque pointer, 21, 419
    as blessed integer, 422
Opcode, 3803
    pperl 0.6.5, 5511
Opcode::deny, 3806
Opcode::deny_only, 3806
Opcode::permit, 3806
Opcode::permit_only, 3806
open, 555, 1098
    2-arg form, 556
    3-arg form, 293, 307, 321, 336, 555, 556
    modes, 556
    scalar reference, 563
    security, 556
    to process, 560
open (*POSIX*), 3935
open (*pragma*), 5355
open pragma, 559
open2
    argument order, 621
open3
    argument order, 621
open_db (*Peta::FFI::LMDB::Txn*), 4010
opendir, 556, 1101
operator, 1447
    additive, 1448
    assignment, 1467
    binding, 1465
    bitwise and, 1459
    bitwise or, 1459
    bitwise string, 1461
    bitwise xor, 1459
    comma, 1478
    conditional, 1470
    equality, 1452, 1454
    filetest, 1483
    list, 1481
    list rightward, 1481
    logical and, 1457, 1458
    logical defined-or, 1457
    logical dor, 1457
    logical exclusive or, 1457
    logical not, 1458
    logical or, 1457, 1458
    logical xor, 1457
    match, 1466
    multiplicative, 1448
    named unary, 1482
    precedence, 1481
    range, 1462
    regexp, 1466
    relational, 1452, 1454
    shift, 1460
    ternary, 1470
    unary, 1449
operator precedence
    or vs ||, 565, 605
    print, 268
    word forms, 605
oplist (*B::OP*), 1841
opnumber (*B*), 1855
ops (*pragma*), 5357
optimization (*re*), 5397
or, 1458
    vs ||, 603
or die idiom, 293, 307, 321, 336, 375, 565, 604
orbit camera, 460
ord, 1104
ORS, 353
our, 1107
our sub, 1525
output redirection
    awk, 352
output_field_separator, 674
output_record_separator, 674
overload
    operator metamethods, 337
overload (*pragma*), 5357
overloading (*pragma*), 5381
override, 1526

P

p
    sed, 344
-p, 10
    expanded loop, 346, 363
    implicit print loop, 236
    versus sed, 342
pack, 1112
    . directive, 545
    ! modifier, 533
    * count, 542
    / directive, 542
    < and > modifiers, 536
    @ directive, 545
    [template] count, 541
    a, A and Z directives, 538
    absolute position, 545
    alignment, 545
    b and B directives, 539

binary serialisation, 532
C directive, 533
fixed width, 533
fixed-size binary records, 453
groups, 541
h and H directives, 540
i and I directives, 533
integer directives, 533
N template, 254
n, N, v and V directives, 535
native width, 533
nested groups, 543
out-of-range values, 534
padding, 544
repeat count, 533, 541
round-trip testing, 553
round-trip with unpack, 555
signed vs unsigned, 534
string directives, 538
template length, 534
use cases, 532
vertex data, 472
W directive, 533
X directive, 544
x directive, 544
pack_ptr (*Peta::FFI*), 4034
pack_sockaddr_in (*Socket*), 4249
pack_sockaddr_in6 (*Socket*), 4250
pack_sockaddr_un (*Socket*), 4252
pack_winsize (*IO::Tty*), 3280
package, 1119
Package::Stash::XS, 3946
Package::Stash::XS::add_symbol, 3949
Package::Stash::XS::new, 3948
package.path, 337
packiv (*B::IV*), 1838
PadWalker, 220, 3949
PadWalker::closed_over, 3954
PadWalker::peek_my, 3951
PadWalker::peek_our, 3952
PadWalker::peek_sub, 3953
PadWalker::set_closed_over, 3955
PadWalker::var_name, 3957
painter's algorithm, 442
pairfirst (*List::Util*), 3501
pairgrep (*List::Util*), 3501
pairkeys (*List::Util*), 3499
pairmap (*List::Util*), 3500
pairs, 330
pairs (*List::Util*), 3497
pairvalues (*List::Util*), 3499
palette image, 390
paragraph mode, 238, 245, 266
--parallel, 35
--parallel-threshold, 11, 35
parameter, 1526
formal, 1541
parameter expansion, 359
Params::Check, 3958
Params::Util, 3964
Params::Util::PP, 3966
parent
declaring a superclass, 177
parent (*pragma*), 5382
Parse::CPAN::Meta, 3967
pass by reference, 1529, 1531
pass-by-reference, 1529, 1531
path
stroke and fill, 410
pathMap, 229
pattern matching, 56
pattern space, 342
pattern-action rule
awk, 352
PBR, 447
pcall, 339
PCRE2, 139
differences from Perl, 139
PDL, 3806
get_dataref, 493
pperl 0.5.6, 5519
pperl 0.6.3, 5513
pperl 0.6.6, 5509
pperl 0.6.8, 5509
pperl 0.6.17, 5495
pperl 0.6.18, 5493
pperl 0.6.19, 5491
pperl 0.6.21, 5484
slices for neighbour lookup, 490
versus GPU, 494
PDL::at, 3901
PDL::Bad, 3839
PDL::Basic, 3839
PDL::Complex::Overloads, 3840
PDL::Compression, 3840
PDL::Constants, 3841
PDL::Core, 3842
PDL::Exporter, 3844
PDL::FFT, 3845
PDL::flowing, 3902
PDL::get_dataref, 3900

PDL::hdrcpy, 3904
PDL::Image2D, 3862
PDL::ImageND, 3863
  pperl 0.6.23, 5471
PDL::ImageRGB, 3863
PDL::inner, 3899
PDL::IO::FITS, 3846
PDL::IO::Misc, 3853
PDL::IO::Pic, 3854
PDL::IO::Pnm, 3861
PDL::IO::Storable, 3861
PDL::Lite, 3864
PDL::LiteF, 3864
PDL::Math, 3864
PDL::MatrixOps, 3865
PDL::new_from_specification, 3900
PDL::Ops, 3865
PDL::Options, 3866
PDL::PP::Signature, 3873
PDL::Primitive, 3873
PDL::set_autopthread_targ, 3902
PDL::set_datatype, 3903
PDL::set_donttouchdata, 3903
PDL::set_sv_to_null_pdl, 3903
PDL::setdims, 3903
PDL::sever, 3902
PDL::Slices, 3874
PDL::Trans, 3874
PDL::Trans::VTable, 3876
PDL::Transform, 3876
  pperl 0.6.23, 5471
PDL::Transform::Cartography, 3877
PDL::Types, 3895
PDL::Ufunc, 3899
PDL::upd_data, 3900
PDL::zeroes, 3900
peek (*Peta::FFI*), 4032
peek_cstr (*Peta::FFI*), 4033
peek_my (*PadWalker*), 3951
peek_our (*PadWalker*), 3952
peek_sub (*PadWalker*), 3953
perf
  profiling pperl, 149
performance, 148
  idioms, 151
periodic (*EV*), 2510
Perl exited with active threads, 44
perl5db.pl, 195
PERL5LIB, 7
  in pperl, 32
  vs PPERL5LIB, 636
Perl::LanguageServer, 227
Perl::OSType, 3971
PERL_MM_USE_DEFAULT, 637
PERL_ONLY, 641
PERL_UNICODE, 591
.perldb, 218, 222
PERLDB_OPTS, 222
  NonStop and AutoTrace, 225
perlfaq (*pragma*), 5384
PerlIO, 3973
PerlIO layers, 558, 590
  :crlf, 559, 590
  :encoding, 559, 590
  :raw, 559, 560, 590
  :std, 559, 591
  :utf8, 559, 590
  at open, 558, 590
  in-memory, 564
  via binmode, 558, 591
PerlIO::encoding, 3979
PerlIO::get_layers, 592
PerlIO::mmap, 3980
PerlIO::scalar, 3980
PerlIO::utf8_strict, 3981
PerlIO::via, 3981
PerlIO::via::QuotedPrint, 3982
PerlMagick, 409
perlstring (*XString*), 5156
permit_only (*Opcode*), 3806
per-pixel shading, 446
  on the GPU, 481
perspective divide, 438
perspective projection, 438
perspective-correct interpolation, 443
Peta::Dict, 3985
  pperl 0.6.19, 5490
  pperl 0.6.20, 5487
Peta::FFI, 19, 643, 3986
  alloc, 24
  arrays, 22
  attach, 20
  call, 20
  callbacks, 25, 419
  closure, 25
  dlclose, 20
  dlopen, 20
  errno, 24
  events from library threads, 25
  fetch, 24
  for graphics, 418

function, 20
handle_type, 22
header files, 28
limitations, 28, 419, 432
native binding, 19, 418
output buffers, 22
pack_ptr, 24
peek, 24
pointers to scalars, 22
poke, 24
pperl 0.6.5, 5511
pperl 0.6.9, 5506
prepared calls, 20
raw calls, 19, 418
record, 23
records, 432
release, 22
scan, 26
SDL2 raw calls, 431
store, 24
structs, 23, 419
symbol, 24
typed handles, 22
type-signature codes, 21, 418
union, 24, 431
variable, 24
variadic calls, 21
versus native modules, 28
Peta::FFI::alloc, 4033
Peta::FFI::attach, 4026
Peta::FFI::call, 4023
Peta::FFI::closure, 4021
Peta::FFI::dlclose, 4031
Peta::FFI::dlopen, 4022
Peta::FFI::errno, 4026
Peta::FFI::EventSource, 25, 3993
Peta::FFI::fetch, 4030
Peta::FFI::free, 4034
Peta::FFI::function, 4025
Peta::FFI::handle_type, 4027
Peta::FFI::Libc, 27, 4011
Peta::FFI::LMDB, 3995
Peta::FFI::LMDB::Cursor, 3998
Peta::FFI::LMDB::DB, 4001
Peta::FFI::LMDB::DB::set_compare, 4001
Peta::FFI::LMDB::Env, 4002
Peta::FFI::LMDB::Env::begin, 4005
Peta::FFI::LMDB::Env::new, 4004
Peta::FFI::LMDB::Txn, 4006
Peta::FFI::LMDB::Txn::get, 4008
Peta::FFI::LMDB::Txn::get_view, 4009
Peta::FFI::LMDB::Txn::open_db, 4010
Peta::FFI::LMDB::Txn::put, 4010
Peta::FFI::pack_ptr, 4034
Peta::FFI::PCRE2, 4012
Peta::FFI::PCRE2::count, 4015
Peta::FFI::PCRE2::match, 4015
Peta::FFI::PCRE2::new, 4015
Peta::FFI::PCRE2::substitute, 4016
Peta::FFI::peek, 4032
Peta::FFI::peek_cstr, 4033
Peta::FFI::poke, 4033
Peta::FFI::Pool, 4016
Peta::FFI::Queue, 4018
Peta::FFI::record, 4028
Peta::FFI::release, 4027
Peta::FFI::scan, 4031
Peta::FFI::sizeof, 4029
Peta::FFI::store, 4030
Peta::FFI::symbol, 4030
Peta::FFI::union, 4029
Peta::FFI::unpack_ptr, 4035
Peta::FFI::UUID, 27, 4020
Peta::FFI::variable, 4030
Peta::ReadLine, 4035
pperl 0.6.22, 5476
Peta::ReadLine::XS, 4036
Peta::ReadLine::XS::add_history, 4046
Peta::ReadLine::XS::read_history_range, 4046
Peta::ReadLine::XS::rl_completion_matches, 4046
Peta::ReadLine::XS::rl_parse_and_bind, 4047
Peta::ReadLine::XS::rl_readline, 4046
Peta::Shell, 4047
Peta::Shell::Complete, 4053
Peta::Shell::Printer, 4055
Peta::Shell::Session, 4057
Peta::WebGPU, 4060
device and surface, 463
pperl 0.6.17, 5494
pperl 0.6.19, 5490
Peta::WebGPU::adapters, 4069
Peta::WebGPU::have_gpu, 4069
Peta::XS, 4070
pperl 0.6.1, 5516
Peta::XS::cast, 4076

Peta::XS::dispell, 4077
Peta::XS::getdata, 4078
Peta::XS::magic, 4073
Peta::XS::refcount, 4071
Peta::XS::value_slots, 4072
Peta::XS::wizard, 4074
PETA_FFI_PATH, 26
PetaPerl, 5526
Phong shading, 446
PHP
    common mistakes in Perl, 312
    moving to Perl, 297
physically based rendering, 447
picking, 460
    triangle-level, 507
ping-pong buffers, 490
pipe, 1124
    child process, 560
    closing unused ends, 622
    with fork, 622
pipe (*POSIX*), 3937
pipe open, 345, 352, 375, 560
    list form, 53, 561
    read from command, 360, 561
    string form, 561
    write to command, 360, 561
PIPE_BUF, 557
pipeline pattern, 44
    with pipe open, 53
platform support, 33
PLY, 454
PNG
    from GPU pixels, 464
Pod::Checker, 4079
Pod::Escapes, 4087
Pod::Functions, 4090
Pod::Html, 4090
Pod::Html::Util, 4094
Pod::Man, 4095
Pod::ParseLink, 4108
Pod::Perldoc, 4109
Pod::Perldoc::BaseTo, 4110
Pod::Perldoc::GetOptsOO, 4111
Pod::Perldoc::ToANSI, 4112
Pod::Perldoc::ToChecker, 4113
Pod::Perldoc::ToMan, 4114
Pod::Perldoc::ToNroff, 4115
Pod::Perldoc::ToPod, 4116
Pod::Perldoc::ToRtf, 4117
Pod::Perldoc::ToTerm, 4118
Pod::Perldoc::ToText, 4119
Pod::Perldoc::ToTk, 4120
Pod::Perldoc::ToXml, 4120
Pod::Simple, 4121
Pod::Simple::Checker, 4129
Pod::Simple::Debug, 4130
Pod::Simple::DumpAsText, 4132
Pod::Simple::DumpAsXML, 4133
Pod::Simple::HTML, 4134
Pod::Simple::HTMLBatch, 4139
Pod::Simple::JustPod, 4144
Pod::Simple::LinkSection, 4146
Pod::Simple::Methody, 4148
Pod::Simple::PullParser, 4149
Pod::Simple::PullParserEndToken, 4154
Pod::Simple::PullParserStartToken, 4155
Pod::Simple::PullParserTextToken, 4157
Pod::Simple::PullParserToken, 4159
Pod::Simple::RTF, 4161
Pod::Simple::Search, 4163
Pod::Simple::SimpleTree, 4169
Pod::Simple::Text, 4171
Pod::Simple::TextContent, 4172
Pod::Simple::XHTML, 4173
Pod::Simple::XMLOutStream, 4180
Pod::Text, 4181
Pod::Text::Color, 4188
Pod::Text::Overstrike, 4190
Pod::Text::Termcap, 4191
Pod::Usage, 4193
point in mesh test, 458
poke (*Peta::FFI*), 4033
polygon fill, 384
POP, 278
pop, 1128
pop (*DB_File*), 2401
pos, 1130
    after s///g, 112
    and \G, 71
    with /g, 103
positional parameter
    shell, 365
POSIX, 3904
    pperl 0.5.6, 5520
    pperl 0.6.3, 5513
    pperl 0.6.5, 5512
    pperl 0.6.6, 5510
    pperl 0.6.9, 5504
    subexpression maximisation, 94

POSIX BRE, 140
POSIX character class, 65
POSIX ERE, 140
POSIX::_exit, 3941
POSIX::access, 3935
POSIX::asctime, 3940
POSIX::ctime, 3941
POSIX::dup2, 3936
POSIX::fesetround, 3931
POSIX::FLT_ROUNDS, 3930
POSIX::fma, 3928
POSIX::fpathconf, 3943
POSIX::fpclassify, 3930
POSIX::frexp, 3927
POSIX::jn, 3929
POSIX::localeconv, 3943
POSIX::lseek, 3936
POSIX::mkfifo, 623
POSIX::mktime, 3939
POSIX::modf, 3927
POSIX::nan, 3928
POSIX::nice, 3941
POSIX::open, 3935
POSIX::pipe, 3937
POSIX::read, 3937
POSIX::remquo, 3929
POSIX::setlocale, 3942
POSIX::setpgid, 3934
POSIX::sigaction, 3945
POSIX::sigprocmask, 3944
POSIX::SigSet, 3922
POSIX::SigSet::new, 3923
POSIX::strftime, 3938
POSIX::strtod, 3931
POSIX::strtol, 3932
POSIX::strtoul, 3933
POSIX::strxfrm, 3933
POSIX::sysconf, 3935
POSIX::tcflow, 3944
POSIX::tcflush, 3944
POSIX::Termios, 3923
POSIX::Termios::getattr, 3925
POSIX::Termios::getcc, 3926
POSIX::Termios::new, 3925
POSIX::Termios::setattr, 3926
POSIX::times, 3940
POSIX::tzname, 3939
POSIX::uname, 3934
POSIX::WEXITSTATUS, 3931
POSIX::write, 3938
POSIX::yn, 3929
possessive quantifier, 81
  as atomic group, 82
  performance, 124
power, 1449
power of two
  testing, 616
powm_gmp (*Math::GMP*), 3677
pperl, 5526, **5527**
  architecture, 3, 16
  differences from perl5, 31
  goals, 3
  installation, 7
  supported platforms, 5
pperl 0.5.6, 5518
pperl 0.6.0, 5516
pperl 0.6.1, 5516
pperl 0.6.2, 5514
pperl 0.6.3, 5512
pperl 0.6.5, 5510
pperl 0.6.6, 5509
pperl 0.6.8, 5508
pperl 0.6.9, 5502
pperl 0.6.11, 5502
pperl 0.6.12, 5501
pperl 0.6.13, 5500
pperl 0.6.14, 5499
pperl 0.6.15, 5498
pperl 0.6.16, 5495
pperl 0.6.17, 5493
pperl 0.6.18, 5492
pperl 0.6.19, 5487
pperl 0.6.20, 5486
pperl 0.6.21, 5480
pperl 0.6.22, 5474
pperl 0.6.23, 5470
pperl 0.6.24, 5459
pperl 0.6.25, 5459
PPERL5LIB, 7, 32, 636
  pperl 0.6.2, 5514
PPERL_CPPAN_URL, 638
  troubleshooting, 644
PPERL_DAEMON, 161
PPERL_HOME, 635, 652
PPERL_JIT_LOG, 18, 149, 158
PPERL_RXJIT, 11
PPERL_RXJIT_STATS, 11
precedence, 1481
preg_match, 306
preg_match_all, 306
preg_replace, 306
premultiplied alpha, 381

in Cairo, 411
pre-opened handles, 555
present, 466
present mode, 467
Prima, 417
prime numbers, 254
PRINT, 281
print, 1132
precedence, 268
to a string via tie, 280
to buffer, 563
to filehandle, 352, 557
print debugging, 200
PRINTF, 281
printf, 563, 1135
number formatting, 252
private method, 172
probab_prime (*Math::GMP*), 3679
proc, 370
Ruby, 320
procedural generation, 487
producer and consumer, 624
product (*List::Util*), 3490
profiling, 148
projection matrix, 438
WebGPU depth range, 477
protect (*File::Map*), 2854
prototype, 1138, 1537
pty_allocate (*IO::Pty*), 3185
pure Perl, 634
pure-Perl fallback, 32
PUSH, 278
push, 1141
onto a dereferenced array, 572
push (*DB_File*), 2402
push/pull, 455, 509
put (*DB_File*), 2404
put (*Peta::FFI::LMDB::Txn*), 4010
puts
Tcl, 375
putstate (*Digest::SHA*), 2466
PV (*B::PV*), 1847
Python
common mistakes in Perl, 297
moving to Perl, 284
Python re
differences from Perl, 138
Perl equivalents, 292

Q

q
sed, 344
q//, 1144
qq//, 1147
qr
precompiling patterns, 151
qr//, 1152
interpolating into patterns, 57
pattern reuse, 57
performance, 128
stringification, 106
quantifier, 77
greedy, 78
non-greedy, 80
on zero-width assertion, 83
portability, 84
possessive, 81
upper bound, 77
quaternion, 439
queue
between threads, 48
quoted string
matching, 125
quotemeta, 1156
building alternations, 136
in patterns, 60
quoting
versus Tcl, 368
qw//, 1158
qx, 360, 375
qx//, 1161

R

race condition, 47
raise
Python, 295
Ruby, 324
rand, 1164
in one-liners, 253
random numbers, 253
range, 1462
range pattern
awk, 352
raster image, 379
as nested array, 379
raster library
choosing, 410
rasterisation, 379
ray
through a pixel, 460
Rayon, 34
ray-sphere intersection, 460

ray-triangle intersection, 507
re
    debug, 226
    strict mode, 59
re (*pragma*), 5394
RE2, 140
    backreferences, 94
re::install, 5396
re::is_regexp, 5398
re::optimization, 5397
re::regexp_pattern, 5398
re::regmust, 5396
re::regname, 5399
re::regnames, 5399
re::regnames_count, 5400
READ, 281
read, 1166
    buffered, 560
    shell, 357
read (*POSIX*), 3937
read mode, 556
read_history_range (*Peta::ReadLine::XS*), 4046
readback, 464
    cost, 493
readdir, 556, 1171
READLINE, 281
readline, 293, 307, 321, 336, 357, 373, 556, 1173
readlink, 1177
read-modify-write, 47
readonly (*Scalar::Util*), 4235
readpipe, 1180
read-write mode, 558
real, 1453
real_import (*Filter::Util::Call*), 2934
record, 575
record (*Peta::FFI*), 4028
recover (*GDBM_File*), 2972
recursion, 1545
recursive pattern, 251
recursive subpattern, 89
    captures, 90
    depth limit, 129
recv, 1182
    sender address, 630
redirection
    shell, 360
redo, 1185
reduce (*List::Util*), 3505
reduction, 154, 159
reduction variable, 36
reductions (*List::Util*), 3506
ref, 1188
    CODE, 583
    testing for a reference, 570
REFCNT (*B::SV*), 1848
refcount (*Peta::XS*), 4071
reference, 568, 1486, 1529, 1531
    common mistakes, 571
    copying, 568
    nested data, 291, 305, 320, 334
    passing to a subroutine, 370
    stringification, 569
    to a reused array, 574
reference counting, 584
re.findall, 292
regex, 55
regex engine
    choosing, 146
    comparison, 137
regex optimiser, 135
    first-character discrimination, 135
    fixed-string check, 135
    implicit anchoring, 136
    minimum length, 136
    simple repetition, 135
regexp, 55
    replace, 1466
    substitute, 1466
    Tcl, 371
regexp_pattern (*re*), 5398
register (*Hash::Util::FieldHash*), 3041
regmust (*re*), 5396
regname (*re*), 5399
regnames (*re*), 5399
regnames_count (*re*), 5400
regsub, 371
regular expression, 55
    as set of strings, 617
    basics, 56
    comments, 102
    compile caching, 136
    debugging, 226
    extracting all matches, 247
    grammar, 91
    match preference rules, 134
    matching principles, 79
    modifiers, 100
    multi-line matching, 249
    optimisation tips, 128

other engines, 137
parsing phases, 60, 106
performance, 122
Unicode, 592
whitespace, 102
relational operator, 1452, 1454
release (*Peta::FFI*), 4027
remainder, 1449
remap (*File::Map*), 2852
remquo (*POSIX*), 3929
rename, 1192
render pass, 464
render pipeline, 472
render target, 416, 440
offscreen canvas, 424
reorganize (*GDBM_File*), 2971
REPL, 209, 645
replace, 1466
require, 1194
Lua, 337
PHP, 308
Ruby, 322
resampling, 400
re.search, 292
reservoir sampling, 253
reset, 1200
reset (*Digest::MD5*), 2447
reset (*Digest::SHA*), 2461
re.sub, 292
return, 1203, 1530
return value, 1530
reverse, 1207
revolve, 456
rewinddir, 1210
RGB, 380
hex notation, 380
RGBA, 381
ridged noise, 488
right shift, 1460
right-handed coordinate system, 435
rindex, 1212
rl_completion_matches (*Peta::ReadLine::XS*), 4046
rl_parse_and_bind (*Peta::ReadLine::XS*), 4047
rl_readline (*Peta::ReadLine::XS*), 4046
rmdir, 1215
role, 186
requirements, 187
Role::Tiny, 180, 187
Role::Tiny::With, 187
ROT13, 250
rotation
Euler angles versus quaternions, 439
Ruby
common mistakes in Perl, 326
moving to Perl, 312
RubyGems, 322

S

s, 1466
sed, 342
s'''', 113
-s, 10
s///, 108, 1217
delimiters, 112, 342
in one-liners, 241
replacement string, 108
return value, 108, 292, 321, 342, 351, 359, 371
s///e, 110, 342, 351, 359
s///ee, 111
s///r, 111, 292, 306, 321, 335, 342, 359
chaining, 111
S_IFMT (*Fcntl*), 2815
S_IMODE (*Fcntl*), 2815
S_ISREG (*Fcntl*), 2816
Safe, 4222
pperl 0.6.8, 5509
safename (*B*), 1857
sample (*List::Util*), 3492
say, 1221
on a tied filehandle, 282
SCALAR, 274
scalar, 1224, 1486
constant, 1502
filehandle, 563
forcing scalar context, 288
literal, 1502
scalar context, 288, 302, 316, 331, 1498
array in, 573
scalar reference
as filehandle target, 563
Scalar::List::Utils, 4228
Scalar::Util, 4229
isweak, 586
pperl 0.6.0, 5518
pperl 0.6.2, 5515
pperl 0.6.5, 5511
pperl 0.6.19, 5491
weaken, 585

Scalar::Util::Numeric, 4231
Scalar::Util::Numeric::isbig, 4234
Scalar::Util::Numeric::isfloat, 4235
Scalar::Util::Numeric::isinf, 4235
Scalar::Util::Numeric::isint, 4234
Scalar::Util::Numeric::isnum, 4233
Scalar::Util::Numeric::isuv, 4234
Scalar::Util::readonly, 4235
Scalar::Util::set_prototype, 4236
Scalar::Util::tainted, 4235
scan (*Peta::FFI*), 4031
scancode, 429
scanline fill, 384
scene graph, 459
Schwartzian transform, 155, 296, 311, 325, 339
scope
  dynamic, 1535
  lexical, 1534
scope_guard (*Guard*), 2999
Scoping Rules of Regex Variables, 680
screenshot, 513
script run, 119
  rules, 120
Script_Extensions, 116
SDBM_File, 4202
  pperl 0.6.18, 5493
SDL2, 4204
  2D game, 427
  drawing application, 424
  drawing primitives, 416
  headless, 427
  import trap, 465
  input events, 417
  mouse events, 426
  pperl 0.6.16, 5497
  pperl 0.6.17, 5494
  renderer, 415
  save_png, 427
  textures, 416
  window, 415
  window events, 468
  with WebGPU, 465
SDL2::Events, 4214
SDL2::Image, 4215
  saving PNG, 513
SDL2::Render, 4217
SDL2::SDL_CreateRenderer, 4221
SDL2::SDL_CreateWindow, 4221
SDL2::SDL_GetKeyboardState, 4221
SDL2::SDL_GetWindowWMInfo, 4222
SDL2::SDL_PushEvent, 4221
SDL2::Surface, 4218
SDL2::Video, 4219
SDL_CreateRenderer (*SDL2*), 4221
SDL_CreateWindow (*SDL2*), 4221
SDL_GetKeyboardState (*SDL2*), 4221
SDL_GetWindowWMInfo (*SDL2*), 4222
SDL_PushEvent, 468
  does not change key state, 522
SDL_PushEvent (*SDL2*), 4221
SDL_VIDEODRIVER, 427
Search::Dict, 4236
security
  open, 556
sed
  common mistakes in Perl, 347
  moving to Perl, 341
  regular expressions, 140
seek, 1227
  on in-memory handle, 565
seekdir, 1230
select, 1232
  Ruby, 315
select_font_face (*Cairo::Context*), 1987
selectfile (*Term::ReadKey*), 4396
SelectSaver, 4237
self
  Python, 294
  Ruby, 323
SelfLoader, 4237
semaphore, 49, 633
semctl, 1235
  SETVAL and GETVAL, 633
semget, 1238
  creating a set, 633
semop, 1241
  operation string, 633
send, 1244
  datagram, 630
sentence boundary, 119
seq (*DB_File*), 2407
sequence
  Lua, 329
set
  Tcl, 368
set (*Template::Stash::XS*), 4375
set bits
  counting, 539
set_autopthread_targ (*PDL*), 3902

set_closed_over (*PadWalker*), 3955
set_compare (*Peta::FFI::LMDB::DB*), 4001
set_datatype (*PDL*), 3903
set_donttouchdata (*PDL*), 3903
set_metadata (*Cairo::PdfSurface*), 1998
set_mime_data (*Cairo::Surface*), 2008
set_mro (*mro*), 5351
set_prototype (*Scalar::Util*), 4236
set_prototype (*Sub::Util*), 4269
set_subname (*Sub::Util*), 4269
set_sv_to_null_pdl (*PDL*), 3903
setattr (*POSIX::Termios*), 3926
SetControlChars (*Term::ReadKey*), 4399
setdims (*PDL*), 3903
setgrent, 1246
sethostent, 1249
setlocale (*POSIX*), 3942
setmetatable, 337
setnetent, 1251
setnodelay (*Term::ReadKey*), 4397
setpgid (*POSIX*), 3934
setpgrp, 1253
setpriority, 1256
setprotoent, 1258
setpwent, 1260
SetReadMode (*Term::ReadKey*), 4397
setservent, 1262
setsockopt, 1265
SetTerminalSize (*Term::ReadKey*), 4398
sever (*PDL*), 3902
sha1 (*Digest::SHA*), 2454
sha1 (*Digest::SHA1*), 2473
sha1_transform (*Digest::SHA1*), 2473
shader, 471
  compilation errors, 474
shading, 444
shallow copy, 580
  shared referents, 577
shared library
  discovery, 26
shared memory, 632
shared variable, 46
sharpen, 386
Sheffer stroke, 613
--shell, 10, 645
shell alias, 258
shell function, 258
shell injection, 360, 561
shell quoting, 259, 263
shell script
  common mistakes in Perl, 366
  moving to Perl, 356
shell variable, 365
SHIFT, 278
shift, 1268
  left, 1460
  right, 1460
shift (*DB_File*), 2401
shift operator, 1460
shininess, 445
shl, 1460
shmat (*IPC::SysV*), 3406
shmctl, 1270
  removing a segment, 632
shmget, 1272
  creating a segment, 632
shmread, 1276
  explicit length, 632
shmwrite, 1278
  explicit length, 632
short-circuit evaluation, 604
  as control flow, 618
shorthand character class, 63
  portability, 68
  Unicode, 592
show_glyphs (*Cairo::Context*), 1988
show_text (*Cairo::Context*), 1987
show_text_glyphs (*Cairo::Context*), 1988
shr, 1460
shuffle (*List::Util*), 3492
shutdown, 1280
side-effect analysis, 36
sigaction (*POSIX*), 3945
sigil, 1522
  coming from Lua, 328
  coming from PHP, 298
  coming from Python, 285
  coming from Ruby, 313
  coming from Tcl, 369
signal
  child process, 562
signed area, 440
signed volume, 448
  as a test, 455
sigprocmask (*POSIX*), 3944
sigtrap (*pragma*), 5400
sin, 1283
sizeinbase_gmp (*Math::GMP*), 3677
sizeof (*Peta::FFI*), 4029
slab method, 461
slash, 1448
sleep, 1286

slice, 1490
slug
  generating, 113
slurp
  list context, 293
slurp mode, 238, 245, 557
Smart::Comments, 203
smoothstep, 488
  material blending, 493
SO_REUSEADDR, 626
SOCK_DGRAM, 630
sockaddr_family (*Socket*), 4253
sockaddr_in, 625
sockaddr_in (*Socket*), 4257
sockaddr_in6 (*Socket*), 4257
Socket, 4242
  address helpers, 625
  pperl 0.5.6, 5520
  pperl 0.6.6, 5510
  pperl 0.6.9, 5506
socket, 1289
  TCP, 625
Socket::getaddrinfo, 4254
Socket::getnameinfo, 4255
Socket::inet_aton, 4245
Socket::inet_ntoa, 4246
Socket::inet_ntop, 4248
Socket::inet_pton, 4247
Socket::pack_sockaddr_in, 4249
Socket::pack_sockaddr_in6, 4250
Socket::pack_sockaddr_un, 4252
Socket::sockaddr_family, 4253
Socket::sockaddr_in, 4257
Socket::sockaddr_in6, 4257
Socket::unpack_sockaddr_in, 4249
Socket::unpack_sockaddr_in6, 4251
Socket::unpack_sockaddr_un, 4253
socketpair, 629, 1292
  vs pipe, 629
sort, 1295
  by a hash field, 575
  comparator cost, 155
  multi-key comparator, 157
  numeric comparator, 287, 300, 315, 329, 369
sort (*pragma*), 5402
sort_by, 325
sort_by (*JSON::PP*), 3471
sorted, 287
  key function, 296
source code
  encoding, 588
source filter, 203
source::encoding (*pragma*), 5404
source-over compositing, 381
spaceship, 1453
specular highlight, 445
splain, 197
SPLICE, 278
splice, 1298
split, 1301
  ' ' versus / /, 349, 358, 374
  awk default mode, 349
  capturing separators, 99
  on text strings, 595
  on zero-width matches, 75
  Tcl, 374
  trailing empty fields, 358
  zero-length matches, 133
splitOnLastStarter (*Unicode::Normalize*), 5131
sprintf, 1306
  encoding, 597
sqrt, 1311
srand, 1314
  reproducible noise, 488
  reproducible runs, 518
sRGB, 467
stable (*pragma*), 5404
stack trace, 200, 212
StackTrace::Auto, 207
standard deviation, 253
start-up time, 161
stat, 1317
state, 1320, 1535
state sub, 1525
state variable, 1535
statement modifier, 290, 303, 318, 332, 372
static, 1535
Statistics::Descriptive, 253
--stats, 10
STDERR, 555
  capturing from a child, 621
  tied, 282
STDIN, 555
  encoding, 592
STDOUT, 555
  encoding, 591
STL, 453
  binary and ASCII variants, 453
Storable, 4258

dclone, 580
pperl 0.5.6, 5519
pperl 0.6.2, 5515
pperl 0.6.5, 5511
pperl 0.6.6, 5509
pperl 0.6.9, 5504
pperl 0.6.13, 5501
storage buffer, 481
STORE
tied array, 278
tied hash, 274
tied scalar, 272
store (*Peta::FFI*), 4030
STORESIZE, 278
str_repeat, 299
strftime, 255
strftime (*POSIX*), 3938
strict, 197
barewords, 465
strict (*pragma*), 5406
stride, 472
stride_for_width (*Cairo::Format*), 1992
string, 1486
concatenation, 1450
literal, 1516
null, 1486
string map, 371
string match, 371
String#scan, 321
String::CRC32, 4261
string.gmatch, 335
string.gsub, 335
string.match, 335
string.rep, 328
strtod (*POSIX*), 3931
strtol (*POSIX*), 3932
strtoul (*POSIX*), 3933
struct by value, 23
strxfrm (*POSIX*), 3933
study, 1324
sub, 1326, 1523
anonymous, 581
awk, 351
Sub::Exporter::Progressive, 4262
Sub::Identify, 4263
Sub::Identify::get_code_info, 4265
Sub::Identify::get_code_location, 4265
Sub::Identify::is_sub_constant, 4265
Sub::Name, 4266
Sub::Name::subname, 4266
Sub::Util, 4267
pperl 0.6.0, 5518
Sub::Util::set_prototype, 4269
Sub::Util::set_subname, 4269
Sub::Util::subname, 4268
subdivision, 457
midpoint cache, 498
subdivision surface, 457
subname (*Sub::Name*), 4266
subname (*Sub::Util*), 4268
subprocess, 620
subroutine, 1523
anonymous, 1524
argument, 1526
attribute, 1553
call, 1524
declaration, 1523
implicit return value, 319
lexical, 1525
lvalue, 1551
parameter, 1526
prototype, 1537
return value, 290, 304, 333, 1530
signature, 290, 304, 319, 333, 370
subroutine reference, 581
subs (*pragma*), 5408
substitute, 1466
substitute (*Peta::FFI::PCRE2*), 4016
substitution, 108, 1466
substr, 1332
characters vs bytes, 595
versus awk, 351
sum (*List::Util*), 3486
sum0 (*List::Util*), 3487
SUPER::, 177
compile-time package, 182
in class, 173
superglobal
PHP, 308
super-linear cache, 123
supersampling, 385
supports_mime_type (*Cairo::Surface*), 2007
supportsFileType (*GD*), 2966
surface
WebGPU, 465
surface of revolution, 456
SV, **5527**
internals, 222
SV (*B::GV*), 1834

SVG
    charts, 414
    generating markup, 411
    groups and style, 411
svref_2object (*B*), 1854
SvREFCNT (*Devel::Peek*), 2423
SvTYPE (*B::SV*), 1849
switch
    PHP, 303
    Tcl, 372
Symbol, 4270
symbol
    Ruby, 316
symbol (*Peta::FFI*), 4030
symlink, 1336
sync (*DB_File*), 2406
sync (*File::Map*), 2851
syntax check, 198
syntax class
    Emacs, 139
Syntax::Keyword::Try, 204
Sys::Hostname, 4272
Sys::Syslog, 4272
    pperl 0.6.21, 5483
sys_map (*File::Map*), 2850
syscall, 1339
sysconf (*POSIX*), 3935
sysopen, 556, 558, 1342
sys.path, 293
sysread, 558, 1345
sysseek, 1349
system, 1353
    list form, 360
System V IPC, 631
syswrite, 558, 1356

T

-T, 10
-t, 10
table
    Lua, 327
table.insert, 329
table.remove, 329
table.sort, 339
tac, 345
tag_begin (*Cairo::Context*), 1987
tail (*List::Util*), 3491
tainted (*Scalar::Util*), 4235
TAP::Base, 4274
TAP::Formatter::Base, 4275
TAP::Formatter::Color, 4278
TAP::Formatter::Console, 4279
TAP::Formatter::Console::ParallelSession, 4279
TAP::Formatter::Console::Session, 4280
TAP::Formatter::File, 4280
TAP::Formatter::File::Session, 4281
TAP::Formatter::Session, 4281
TAP::Harness, 4282
TAP::Harness::Env, 4291
TAP::Object, 4293
TAP::Parser, 4295
TAP::Parser::Aggregator, 4317
TAP::Parser::Grammar, 4320
TAP::Parser::Iterator, 4324
TAP::Parser::Iterator::Array, 4326
TAP::Parser::Iterator::Process, 4327
TAP::Parser::Iterator::Stream, 4329
TAP::Parser::IteratorFactory, 4330
TAP::Parser::Multiplexer, 4333
TAP::Parser::Result, 4334
TAP::Parser::Result::Bailout, 4338
TAP::Parser::Result::Comment, 4339
TAP::Parser::Result::Plan, 4340
TAP::Parser::Result::Pragma, 4341
TAP::Parser::Result::Test, 4342
TAP::Parser::Result::Unknown, 4345
TAP::Parser::Result::Version, 4345
TAP::Parser::Result::YAML, 4346
TAP::Parser::ResultFactory, 4347
TAP::Parser::Scheduler, 4349
TAP::Parser::Scheduler::Job, 4352
TAP::Parser::Scheduler::Spinner, 4354
TAP::Parser::Source, 4354
TAP::Parser::SourceHandler, 4358
TAP::Parser::SourceHandler::Executable, 4361
TAP::Parser::SourceHandler::File, 4363
TAP::Parser::SourceHandler::Handle, 4365
TAP::Parser::SourceHandler::Perl, 4366
TAP::Parser::SourceHandler::RawTAP, 4369
TAP::Parser::YAMLish::Reader, 4370
TAP::Parser::YAMLish::Writer, 4371
tautology, 607
tcflow (*POSIX*), 3944

tcflush (*POSIX*), 3944
Tcl
  common mistakes in Perl, 376
  moving to Perl, 367
TCP, 625
  client, 626
  server, 626
tell, 1359
telldir, 1361
Template::Stash::XS, 4374
Template::Stash::XS::get, 4375
Template::Stash::XS::set, 4375
Term::ANSIColor, 4376
Term::Cap, 4389
Term::Complete, 4393
Term::ReadKey, 4394
Term::ReadKey::GetControlChars, 4398
Term::ReadKey::GetSpeed, 4398
Term::ReadKey::GetTermSizeGWINSZ, 4397
Term::ReadKey::selectfile, 4396
Term::ReadKey::SetControlChars, 4399
Term::ReadKey::setnodelay, 4397
Term::ReadKey::SetReadMode, 4397
Term::ReadKey::SetTerminalSize, 4398
Term::ReadLine, 4399
Term::ReadLine::Gnu, 4403
  pperl 0.6.9, 5506
  pperl 0.6.22, 5480
Term::Table, 4403
Term::Table::Cell, 4406
Term::Table::CellStack, 4407
Term::Table::HashBase, 4408
Term::Table::LineBreak, 4413
Term::Table::Util, 4414
terminal encoding, 592
ternary, 605, 1470
terrain
  materials, 493
terrain generation, 487
tessellation, 385, 440
Test, 4415
test
  shell, 364
test fixtures, 564
Test2, 4483
Test2::API, 4487
Test2::API::Breakage, 4503
Test2::API::Context, 4503
Test2::API::Instance, 4513
Test2::API::InterceptResult, 4517
Test2::API::InterceptResult::Event, 4525
Test2::API::InterceptResult::Hub, 4534
Test2::API::InterceptResult::Squasher, 4535
Test2::API::Stack, 4536
Test2::AsyncSubtest, 4537
Test2::AsyncSubtest::Event::Attach, 4542
Test2::AsyncSubtest::Event::Detach, 4543
Test2::AsyncSubtest::Hub, 4543
Test2::Bundle, 4544
Test2::Bundle::Extended, 4546
Test2::Bundle::More, 4553
Test2::Bundle::Simple, 4555
Test2::Compare, 4557
Test2::Compare::Array, 4562
Test2::Compare::Bag, 4564
Test2::Compare::Base, 4565
Test2::Compare::Bool, 4567
Test2::Compare::Custom, 4568
Test2::Compare::DeepRef, 4570
Test2::Compare::Delta, 4571
Test2::Compare::Event, 4574
Test2::Compare::EventMeta, 4575
Test2::Compare::Float, 4576
Test2::Compare::Hash, 4577
Test2::Compare::Isa, 4577
Test2::Compare::Meta, 4578
Test2::Compare::Negatable, 4579
Test2::Compare::Number, 4580
Test2::Compare::Object, 4581
Test2::Compare::OrderedSubset, 4582
Test2::Compare::Pattern, 4583
Test2::Compare::Ref, 4584
Test2::Compare::Regex, 4585
Test2::Compare::Scalar, 4585
Test2::Compare::Set, 4586
Test2::Compare::String, 4587
Test2::Compare::Undef, 4587
Test2::Compare::Wildcard, 4588
Test2::Env, 4589
Test2::Event, 4591
Test2::Event::Bail, 4598
Test2::Event::Diag, 4599
Test2::Event::Encoding, 4600

Test2::Event::Exception, 4601
Test2::Event::Fail, 4602
Test2::Event::Generic, 4603
Test2::Event::Note, 4605
Test2::Event::Ok, 4606
Test2::Event::Pass, 4608
Test2::Event::Plan, 4608
Test2::Event::Skip, 4610
Test2::Event::Subtest, 4611
Test2::Event::TAP::Version, 4612
Test2::Event::V2, 4613
Test2::Event::Waiting, 4615
Test2::EventFacet, 4616
Test2::EventFacet::About, 4617
Test2::EventFacet::Amnesty, 4618
Test2::EventFacet::Assert, 4619
Test2::EventFacet::Control, 4620
Test2::EventFacet::Error, 4622
Test2::EventFacet::Hub, 4623
Test2::EventFacet::Info, 4624
Test2::EventFacet::Info::Table, 4626
Test2::EventFacet::Meta, 4628
Test2::EventFacet::Parent, 4629
Test2::EventFacet::Plan, 4630
Test2::EventFacet::Render, 4631
Test2::EventFacet::Trace, 4632
Test2::Formatter, 4636
Test2::Formatter::TAP, 4638
Test2::Handle, 4639
Test2::Hub, 4640
Test2::Hub::Interceptor, 4647
Test2::Hub::Interceptor::Terminator, 4648
Test2::Hub::Subtest, 4648
Test2::IPC, 4649
Test2::IPC::Driver, 4650
Test2::IPC::Driver::Files, 4654
Test2::Manual, 4655
Test2::Manual::Anatomy, 4657
Test2::Manual::Anatomy::API, 4658
Test2::Manual::Anatomy::Context, 4659
Test2::Manual::Anatomy::EndToEnd, 4661
Test2::Manual::Anatomy::Event, 4667
Test2::Manual::Anatomy::Hubs, 4673
Test2::Manual::Anatomy::IPC, 4675
Test2::Manual::Anatomy::Utilities, 4677
Test2::Manual::Concurrency, 4678
Test2::Manual::Contributing, 4680
Test2::Manual::Testing, 4682
Test2::Manual::Testing::Introduction, 4687
Test2::Manual::Testing::Migrating, 4692
Test2::Manual::Testing::Planning, 4703
Test2::Manual::Testing::Todo, 4705
Test2::Manual::Tooling, 4707
Test2::Manual::Tooling::FirstTool, 4709
Test2::Manual::Tooling::Formatter, 4711
Test2::Manual::Tooling::Nesting, 4713
Test2::Manual::Tooling::Plugin::TestExit, 4716
Test2::Manual::Tooling::Plugin::TestingDone, 4718
Test2::Manual::Tooling::Plugin::ToolCompletes, 4720
Test2::Manual::Tooling::Plugin::ToolStarts, 4721
Test2::Manual::Tooling::Subtest, 4723
Test2::Manual::Tooling::TestBuilder, 4726
Test2::Manual::Tooling::Testing, 4728
Test2::Mock, 4731
Test2::Plugin, 4738
Test2::Plugin::BailOnFail, 4739
Test2::Plugin::DieOnFail, 4740
Test2::Plugin::ExitSummary, 4741
Test2::Plugin::SRand, 4741
Test2::Plugin::Times, 4743
Test2::Plugin::UTF8, 4744
Test2::Require, 4745
Test2::Require::AuthorTesting, 4748
Test2::Require::AutomatedTesting, 4748
Test2::Require::EnvVar, 4749
Test2::Require::ExtendedTesting, 4750
Test2::Require::Fork, 4751
Test2::Require::Module, 4752
Test2::Require::NonInteractiveTesting, 4754
Test2::Require::Perl, 4754
Test2::Require::RealFork, 4755

Test2::Require::ReleaseTesting, 4756
Test2::Require::Threads, 4757
Test2::Suite, 4758
Test2::Todo, 4764
Test2::Tools, 4766
Test2::Tools::AsyncSubtest, 4768
Test2::Tools::Basic, 4770
Test2::Tools::Class, 4774
Test2::Tools::ClassicCompare, 4776
Test2::Tools::Compare, 4779
Test2::Tools::Defer, 4799
Test2::Tools::Encoding, 4800
Test2::Tools::Event, 4801
Test2::Tools::Exception, 4802
Test2::Tools::Exports, 4803
Test2::Tools::GenTemp, 4805
Test2::Tools::Grab, 4806
Test2::Tools::Mock, 4808
Test2::Tools::Ref, 4812
Test2::Tools::Refcount, 4813
Test2::Tools::Spec, 4817
Test2::Tools::Subtest, 4824
Test2::Tools::Target, 4826
Test2::Tools::Tester, 4826
Test2::Tools::Tiny, 4829
Test2::Tools::Warnings, 4831
Test2::Util, 4833
Test2::Util::ExternalMeta, 4835
Test2::Util::Facets2Legacy, 4837
Test2::Util::Grabber, 4840
Test2::Util::Guard, 4842
Test2::Util::HashBase, 4843
Test2::Util::Importer, 4848
Test2::Util::Ref, 4848
Test2::Util::Sig, 4849
Test2::Util::Stash, 4850
Test2::Util::Sub, 4852
Test2::Util::Table, 4854
Test2::Util::Table::LineBreak, 4857
Test2::Util::Times, 4858
Test2::Util::Trace, 4859
Test2::V0, 4859
Test2::V1, 4867
Test2::V1::Base, 4881
Test2::V1::Handle, 4882
Test2::Workflow, 4883
Test2::Workflow::BlockBase, 4885
Test2::Workflow::Build, 4886
Test2::Workflow::Runner, 4886
Test2::Workflow::Task, 4887
Test2::Workflow::Task::Action, 4888
Test2::Workflow::Task::Group, 4888
Test::Builder, 4423
Test::Builder::Formatter, 4437
Test::Builder::Module, 4438
Test::Builder::Tester, 4440
Test::Builder::Tester::Color, 4444
Test::Builder::TodoDiag, 4444
Test::Harness, 4445
Test::LeakTrace, 4449
Test::LeakTrace::Script, 4450
Test::More, 4451
Test::Simple, 4471
Test::Tester, 4474
Test::Tester::Capture, 4481
Test::Tester::CaptureRunner, 4481
Test::use::ok, 4482
text string, 587
Text::Abbrev, 4889
Text::Balanced, 4889
Text::CSV, 256, 4911
Text::CSV_PP, 4963
Text::CSV_XS, 638
Text::LevenshteinXS, 5015
Text::LineFold, 5016
Text::ParseWords, 5018
Text::Tabs, 5021
Text::Wrap, 5022
text_to_glyphs (*Cairo::ScaledFont*), 2003
texture mapping, 447
thermal erosion, 490
thousands separator
  inserting, 110
Thread, 5025
thread, 42
  arguments, 42
  process-wide state, 50
  return value, 43
  structural patterns, 44
  termination, 44
thread ID, 45
thread safety, 45
Thread::Queue, 48, 5029
Thread::Semaphore, 49, 5035
threads
  create, 42
  detach, 43
  join, 43
  list, 45
  not implemented in pperl, 33, 41
  tid, 45

yield, 45
--threads, 11, 35
pperl 0.6.2, 5515
threads (*pragma*), 5408
threads::shared, 46
in pperl, 50
threads::shared (*pragma*), 5425
three-valued logic, 601
throw
PHP, 310
Throwable, 206
Throwable::SugarFactory, 206
thumbnail, 388
tie, 1364
on a glob, 281
writing a class, 269
Tie::Array, 279, 5038
Tie::File, 5040
Tie::Handle, 282, 5051
Tie::Hash, 276, 5052
Tie::Hash::Indexed, 5056
Tie::Hash::Indexed::Iterator, 5059
Tie::Hash::NamedCapture, 5059
Tie::Memoize, 5060
Tie::RefHash, 5061
Tie::Scalar, 272, 5063
Tie::StdHandle, 282, 5065
Tie::StdHash, 276
Tie::StdScalar, 272
Tie::SubstrHash, 5065
Tie::Watch, 217
TIEARRAY, 278
tied, 1368
calling class methods, 280
tied array, 276
call timing, 279
minimum methods, 279
size protocol, 278
tied filehandle, 280
call timing, 282
lexical handle, 281
minimum methods, 282
tied hash, 273
boolean context, 275
call timing, 275
iteration, 275
minimum methods, 276
tied scalar, 271
call timing, 272
read-only, 272
tied variable, 269
performance, 270
TIEHANDLE, 281
TIEHASH, 274
TIESCALAR, 272
time, 1370
Time::gmtime, 5076
Time::HiRes, 5066
gettimeofday, 203
pperl 0.6.5, 5511
pperl 0.6.9, 5507
timing a block, 149
Time::Local, 5069
Time::localtime, 5077
Time::Piece, 5073
pperl 0.6.9, 5506
Time::Seconds, 5074
Time::tm, 5078
timer (*EV*), 2509
times, 1372
times (*POSIX*), 3940
Tk, 417
to_b64string (*Data::UUID*), 2418
to_h, 325
tokenisation
with \G, 71
toupper, 351
tr, 1466
flags, 250
versus s///, 268
tr///, 1375
counting characters, 343
squeezing repeats, 343
versus regular expression, 56, 343, 371
versus s///, 114
tracing, 224
non-interactive, 225
Traditional NFA, 95
worst case, 122
trampoline, 18
transform hierarchy, 459
in a game, 522
transform pipeline, 437
transformation matrix, 436
transliterate, 1466
transliteration, 250
transparency
GIF, 399
PNG, 399
triangle
as the only primitive, 440

triangulation, 385
true, 1486
true-colour image, 390
TrueType font, 395
truncate, 1379
    on open, 557
truth, 1486
truth function, 606
    Perl spellings, 607
truth table, 606
    debugging a condition, 610
truthiness, 601
    coming from Lua, 332
    coming from PHP, 303
    coming from Python, 289
    coming from Ruby, 317
try, 204, 1382
try/catch, 295, 310, 324, 339
try/except, 295
Try::Tiny, 205
TSV, 256
ttyname (*IO::Tty*), 3279
turtle graphics, 406
type, 1485
typeglob, 1512
    installing subroutines, 186
Types::Serialiser, 5078
Types::Serialiser::Error, 5082
tzname (*POSIX*), 3939

U

uc, 1387
ucfirst, 1390
UDP, 629
    datagram exchange, 630
umask, 1393
uname (*POSIX*), 3934
unary operator, 1449
undef, 1396, 1486
    as unknown, 601
    truthiness, 601
undefined, 1486
Unicode, 587
    character names, 115
    command-line switch, 238
    decode, process, encode, 587
    I/O, 590
    in regular expressions, 114
    pitfalls, 595
    round-trip, 589
    script preamble, 597
Unicode property, 65, 593
    compound form, 116
    general category, 115
    portability, 121
    script vs block, 593
Unicode script
    matching, 116
Unicode::Collate, 5091
    not available, 600
Unicode::Collate::CJK::Big5, 5110
Unicode::Collate::CJK::GB2312, 5110
Unicode::Collate::CJK::JISX0208, 5111
Unicode::Collate::CJK::Korean, 5112
Unicode::Collate::CJK::Pinyin, 5112
Unicode::Collate::CJK::Stroke, 5113
Unicode::Collate::CJK::Zhuyin, 5114
Unicode::Collate::Locale, 5115
Unicode::GCString, 5122
Unicode::LineBreak, 5124
    pperl 0.6.21, 5483
Unicode::LineBreak::SouthEastAsian, 5127
Unicode::Normalize, 600, 5127
    pperl 0.6.21, 5483
Unicode::Normalize::splitOnLastStarter, 5131
Unicode::UCD, 5131
unicode_strings, 118
uniform, 476
uniform buffer, 476
    alignment, 478
    updating, 478
union (*Peta::FFI*), 4029
uniq (*List::Util*), 3493
uniqint (*List::Util*), 3495
uniqnum (*List::Util*), 3494
uniqstr (*List::Util*), 3494
UNIVERSAL, 5082
UNIVERSAL::can, 183
UNIVERSAL::isa, 183
UNIX-domain socket, 629
unless, 290, 318, 332
    rewriting as if, 610
unlink, 1400
unlock_keys (*Hash::Util*), 3046
unlock_ref_keys (*Hash::Util*), 3047
unlock_value (*Hash::Util*), 3048
unmap (*File::Map*), 2852
unmemoize (*Memoize*), 3702
unpack, 1403

%32b*, 539
/ directive, 542
binary serialisation, 532
bulk transfer, 493
checksum, 546
computed template, 550
H* template, 257
in stages, 546
unpack_ptr (*Peta::FFI*), 4035
unpack_sockaddr_in (*Socket*), 4249
unpack_sockaddr_in6 (*Socket*), 4251
unpack_sockaddr_un (*Socket*), 4253
unpairs (*List::Util*), 3498
unrolling the loop, 126
unset, 301
Tcl, 370
unsharp mask, 389
UNSHIFT, 278
unshift, 1409
reversing lines, 345
unshift (*DB_File*), 2400
UNTIE, 274
untie, 1412
until, 290, 318, 332
unwrapping, 447
upd_data (*PDL*), 3900
upvalue, 334
upvar, 370
URI::Escape, 257
URL encoding, 257
URL::Encode, 5085
URL::Encode::XS, 5089
use, 1414
compile time, 293, 308, 322, 337
import list, 293
use constant
debug flag, 152, 202
use integer
pperl 0.6.13, 5500
use locale, 597
use open, 559, 591
use re
pperl 0.6.6, 5510
use re '/u', 118
use re 'eval', 130
use re 'strict', 59, 107
use utf8, 588
non-ASCII source, 297
vs I/O layers, 597
use v5.12
Unicode semantics, 118
use v5.36
strict and warnings, 197
used_buckets (*Hash::Util*), 3045
useithreads, 42
User::grent, 5132
User::pwent, 5133
user-defined property, 600
usort, 311
UTF-8, 264, 558
utf8 (*pragma*), 5434
UTF-8 flag, 588
and matching, 118
utf8::, 588
utime, 1419
UV coordinate, 447
UV seam, 450

V

-V, 10
pperl 0.6.1, 5516
-v, 10
-V:key
pperl 0.6.1, 5516
validation error, 474
value noise, 488
value_slots (*Peta::XS*), 4072
values, 1422
var_name (*PadWalker*), 3957
variable
lexical, 1534
local, 1535
persistent, 1535
static, 1535
temporary, 1535
variable (*Peta::FFI*), 4030
variadic parameter
PHP, 304
vars (*pragma*), 5438
vec, 1425
vector, 435
normalising, 435
vector graphics, 410
vector image, 379
version
pperl, 33
version (*pragma*), 5439
version::is_lax, 5440
version::is_strict, 5441
VERSION_ENCODE (*Cairo*), 2009
vertex
attributes, 450

shared, 448
welding, 450
vertex buffer, 472
vertex layout, 472
silent mismatch, 476
vertex shader, 471
vertical sync, 467
--via-daemon, 12
opting in, 161
view matrix, 438
view space, 437
VS Code
debugging, 228
vsync, 415, 430

W

-W, 10
-w, 10
wait, 1428
wait(2), 562
waitpid, 1430
walkoptree (*B*), 1855
walkoptree_debug (*B*), 1855
wantarray, 288, 302, 316, 331, 1434
pperl 0.6.14, 5500
warn, 1437
for diagnostics, 200
trailing newline, 200
warnings, 197
fatal, 197, 208
non_unicode, 599
nonchar, 599
surrogate, 599
warnings (*pragma*), 5442
warnings::register (*pragma*), 5457
watcher, 39
watchpoint, 217
Wavefront OBJ, 451
weak reference, 584
becoming undef, 585
use cases, 585
weaken, 585
common mistakes, 586
WebGPU, 463
error messages, 474
welding, 450
WEXITSTATUS (*POSIX*), 3931
wgpu, 463
WGSL, 471
while read, 357
whitespace
trimming, 242
Wide character in print, 595
widget toolkit, 417
hand-drawn controls, 425
winding order, 441
consistency check, 448
reversed by reflection, 480, 498
screen y flip, 496
window resize, 468
wire protocol, 532
with pack, 546
with statement, 293
wizard (*Peta::XS*), 4074
WNOHANG, 626
word boundary, 70
Unicode, 71
word frequency, 262
word splitting
shell, 358
work crew pattern, 44
without threads, 51
work stealing, 34
world space, 437
world transform, 459
WRITE, 281
write, 1442
write (*POSIX*), 3938
write mode, 556
write_to_png_stream (*Cairo::Surface*), 2007
Wx, 417

X

-X, 10, 1483
x, 1451
string repetition, 286, 299, 314, 328
-x
not supported, 33
x=, 1468
XML::Parser, 5135
pperl 0.6.21, 5483
XML::Parser::Expat, 5147
XML::Parser::Style::Debug, 5150
XML::Parser::Style::Objects, 5151
XML::Parser::Style::Stream, 5151
XML::Parser::Style::Subs, 5153
XML::Parser::Style::Tree, 5153
XNOR, 609
XOR, 609
xor, 605, 1457

XOR swap, 616
XS
    alternatives, 643
    GUI toolkits, 417
    mandatory build, 641
    native replacement, 29
    native replacements, 7
    not loadable, 634, 640
    not loaded by pperl, 32
    optional accelerator, 640
    pure Perl instead of, 3
    pure-Perl fallback, 641
XSLoader, 5154
    pperl 0.6.0, 5517
XSLoader::load
    under pperl, 640
XString, 5155
XString::cstring, 5156
XString::perlstring, 5156

Y

y, 1466
    sed, 343
y///, 114, 343, 1446
YAML::LibYAML, 5157
YAML::PP, 5157
YAML::PP::Common, 5177
YAML::PP::Constructor, 5178
YAML::PP::Emitter, 5179
YAML::PP::Grammar, 5180
YAML::PP::Highlight, 5196
YAML::PP::Perl, 5196
YAML::PP::Schema, 5197
YAML::PP::Schema::Binary, 5198
YAML::PP::Schema::Catchall, 5198
YAML::PP::Schema::Core, 5199
YAML::PP::Schema::Failsafe, 5199
YAML::PP::Schema::Include, 5200
YAML::PP::Schema::JSON, 5202
YAML::PP::Schema::Merge, 5203
YAML::PP::Schema::Perl, 5204
YAML::PP::Schema::Tie::IxHash, 5212
YAML::PP::Schema::YAML1_1, 5214
YAML::PP::Type::MergeKey, 5214
YAML::PP::Writer, 5215
YAML::PP::Writer::File, 5216
YAML::XS, 5216
YAML::XS::LibYAML, 5224
yield
    Ruby, 320
yn (*POSIX*), 3929

Z

-z
    shell, 364
Z-buffer, 382, 442
zeroes (*PDL*), 3900
zero-length match, 133
    loop termination, 133
    under /g, 110
zero-width assertion, 69, 249
zip (*List::Util*), 3504
zip_shortest (*List::Util*), 3504
zombie process, 54, 626